

Игорь Блинов
Валерий Романчик

JAVA

FROM EPAM

2-е издание, исправленное



TRAINING
CENTER



Игорь Николаевич Блинов

Доцент ММФ БГУ, кандидат физ-мат наук, тренер EPAM, Sun Certified Java Programmer и Sun Certified Web Component Developer. Разработчик очных и онлайн курсов по Java.



Валерий Станиславович Романчик

Профессор кафедры Веб-технологий и компьютерного моделирования БГУ. Около сорока пяти лет преподает на ММФ курс “Методы программирования”, а последние 15 лет – курс “Веб-программирование”.



Основанная в 1993 году компания EPAM – ведущий мировой разработчик цифровых платформ, продуктов и сложных дизайн и программных решений. Благодаря своей инженерной ДНК, многолетнему технологическому опыту и компетенциям в сфере консалтинга, дизайна и инновационных стратегий, EPAM тесно сотрудничает со своими клиентами для создания передовых решений, которые превращают сложные бизнес-задачи в реальные бизнес-возможности. Команды EPAM работают с заказчиками в более чем 30 странах в Северной Америке, Европе, Азии и Австралии.

EPAM признана одной из четырех технологических компаний, входящих в список Forbes «25 самых быстрорастущих публичных технологических компаний» каждый раз начиная с 2013 года, а также единственным представителем индустрии ИТ сервисов в списке Fortune 2019 «100 самых быстрорастущих компаний».

Компания активно инвестирует в подготовку будущих профессионалов EPAM и оказывает особую поддержку тем, кто только начинает карьеру. EPAM Training Center предлагает бесплатные тренинги по самым трендовым технологиям и направлениям для студентов старших курсов и специалистов, которые хотят сменить профессию.

Информацию о программах обучения можно найти на портале training.by, с готовыми курсами можно познакомиться на learn.epam.com.

И. Н. Блинов В. С. Романчик

Java

from EPAM

Учебно-методическое пособие

2-е издание, исправленное

2021

УДК 004.434
ББК 32.973.26-018.2Б69

Рецензенты:

кандидат технических наук, доцент *В. Д. Левчук*,
кандидат технических наук, доцент *О. Г. Смолякова*

Рекомендовано

Ученым Советом механико-математического факультета Белорусского государственного университета в качестве пособия для студентов высших учебных заведений, обучающихся по специальностям

*1-31 03 08 «Математика и информационные технологии (по направлениям)»,
1-31 03 01 «Математика (по направлениям)»*

Блинов, И. Н., Романчик, В. С.

Б69 Java from EPAM : учеб.-метод. пособие / И. Н. Блинов, В. С. Романчик. — 2-е издание
Минск: Четыре четверти, 2021. — 560 с.
ISBN 978-985-581-391-1.

Второе издание пособия предназначено для программистов, начинающих и продолжающих изучение технологий Java SE. В книге рассматриваются основы языка Java и концепции объектно-ориентированного функционального программирования. Также изложены аспекты применения библиотек классов языка Java, включая файлы, коллекции, Stream API, сетевые и многопоточные приложения, а также взаимодействие с СУБД и XML.

В конце каждой главы даются теоретические вопросы по изученной главе, тестовые вопросы по материалу главы и задания для выполнения. В приложениях приведены дополнительные материалы с кратким описанием технологий Log4J2 и TestNG.

**УДК 004.434
ББК 32.973.26-018.2**

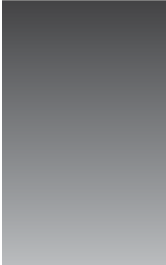
ISBN 978-985-581-391-1

© Блинов И. Н., Романчик В. С., 2021



ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ	10
Глава 1. ВВЕДЕНИЕ В ООП И ФП	11
Глава 2. ТИПЫ ДАННЫХ И ОПЕРАТОРЫ.....	30
Глава 3. КЛАССЫ И МЕТОДЫ.....	65
Глава 4. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ	121
Глава 5. ВНУТРЕННИЕ КЛАССЫ.....	159
Глава 6. ИНТЕРФЕЙСЫ И АННОТАЦИИ	177
Глава 7. ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ	198
Глава 8. СТРОКИ	225
Глава 9. ИСКЛЮЧЕНИЯ И ОШИБКИ.....	264
Глава 10. ПОТОКИ ВВОДА/ВЫВОДА	290
Глава 11. КОЛЛЕКЦИИ И STREAM API	320
Глава 12. ПОТОКИ ВЫПОЛНЕНИЯ.....	369
Глава 13. JAVA DATABASE CONNECTIVITY	428
Глава 14. СЕТЕВЫЕ ПРОГРАММЫ.....	461
Глава 15. JAVA API FOR XML PROCESSING	483
ОТВЕТЫ НА ТЕСТОВЫЕ ВОПРОСЫ	515
Приложение 1. Log4J2.....	516
Приложение 2. TestNG	534



СОДЕРЖАНИЕ ГЛАВ

ПРЕДИСЛОВИЕ	10
Глава 1. ВВЕДЕНИЕ В ООП И ФП.....	11
Базовые понятия ООП.....	11
Базовые понятия ФП	12
Базовые понятия Java	12
Простое приложение.....	14
Установка JDK и IDE	16
Компиляция и запуск приложения.....	17
Основы классов и объектов.....	18
Объектные ссылки.....	23
Консольный ввод\вывод.....	24
Base code conventions	25
Вопросы к главе 1.....	26
Задания к главе 1	27
Тестовые задания к главе 1	27
Глава 2. ТИПЫ ДАННЫХ И ОПЕРАТОРЫ	30
Базовые типы данных и литералы	30
Документирование кода	35
Операторы	37
Классы-оболочки.....	44
Оператор условного перехода.....	49
Оператор выбора	51
Циклы.....	54
Массивы	57
Вопросы к главе 2.....	59
Задания к главе 2	60
Тестовые задания к главе 2	62
Глава 3. КЛАССЫ И МЕТОДЫ.....	65
Переменные класса, экземпляра класса и константы	67
Ограничение доступа	68
Конструкторы.....	69
Методы.....	71
Проектирование методов	72
Использование параметров метода	72

Использование параметра метода для получения результата	75
Использование возвращаемого значения.....	77
Оболочка Optional	78
Статические методы и поля	81
Модификатор final и неизменяемость.....	83
Абстрактные методы.....	84
Модификатор native	84
Модификатор synchronized	85
Логические блоки.....	85
Перегрузка методов.....	86
Параметризованные классы	88
Параметризованные методы	93
Методы с переменным числом параметров.....	94
Перечисления	96
Immutable и record	101
Декомпозиция	102
Рекомендации при проектировании классов	109
Вопросы к главе 3	110
Задания к главе 3	112
Тестовые задания к главе 3	117
Глава 4. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ	121
Наследование	121
Классы и методы final	125
Использование super и this.....	127
Переопределение методов и полиморфизм	129
Методы подставки.....	132
«Переопределение» статических методов	133
Абстракция.....	134
Полиморфизм и расширение функциональности.....	136
Класс Object.....	138
Клонирование объектов	141
«Сборка мусора» и освобождение ресурсов.....	145
Пакеты	146
Статический импорт.....	149
Рекомендации при проектировании иерархии.....	149
Вопросы к главе 4	150
Задания к главе 4.....	152
Тестовые задания к главе 4	156
Глава 5. ВНУТРЕННИЕ КЛАССЫ	159
Внутренние (inner) классы	160
Вложенные (nested) классы.....	166
Анонимные (anonymous) классы.....	169
Вопросы к главе 5	172

Задания к главе 5	173
Тестовые задания к главе 5	174
Глава 6. ИНТЕРФЕЙСЫ И АННОТАЦИИ	177
Интерфейсы	177
Параметризация интерфейсов	184
Аннотации	186
Вопросы к главе 6	192
Задания к главе 6	192
Тестовые задания к главе 6	196
Глава 7. ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ	198
Методы default и static в интерфейсах	198
Функциональные интерфейсы	201
Интерфейс Predicate	203
Интерфейс Function	206
Интерфейс Consumer	209
Интерфейс Supplier	211
Интерфейс Comparator	213
Замыкания	216
Ссылки на методы	218
Вопросы к главе 7	220
Задания к главе 7	220
Тестовые задания к главе 7	223
Глава 8. СТРОКИ	225
Класс String	225
StringBuilder и StringBuffer	231
Регулярные выражения	234
Интернационализация приложения	240
Интернационализация чисел	243
Интернационализация дат	244
API Date\Time	246
Форматирование информации	250
Шифрование и кодирование строк	254
Вопросы к главе 8	257
Задания к главе 8	258
Тестовые задания к главе 8	262
Глава 9. ИСКЛЮЧЕНИЯ И ОШИБКИ	264
Иерархия исключений и ошибок	264
Способы обработки исключений	267
Обработка нескольких исключений	269
Оператор throw	272
Собственные исключения	274
Генерация непроверяемых исключений	276

Блок finally	277
Наследование и исключения.....	278
Ошибки статической инициализации	280
Рекомендации по обработке исключений.....	281
Отладочный механизм assertion.....	284
Вопросы к главе 9	286
Задания к главе 9.....	287
Тестовые задания к главе 9	287
 Глава 10. ПОТОКИ ВВОДА/ВЫВОДА	290
Байтовые и символьные потоки ввода/вывода	290
File, Path и Files	295
Чтение из потока	298
Предопределенные стандартные потоки	301
Сериализация объектов.....	302
Сериализация в XML	306
Класс Scanner.....	308
Архивация	311
Вопросы к главе 10.....	316
Задания к главе 10	316
Тестовые задания к главе 10	318
 Глава 11. КОЛЛЕКЦИИ И STREAM API	320
Общие определения	320
ArrayList	323
Итераторы	328
Stream API.....	331
Алгоритмы сведения Collectors.....	337
Метасимвол в коллекциях.....	340
Класс LinkedList и интерфейс Queue	341
Интерфейс Deque и класс ArrayDeque	345
Множества.....	346
EnumSet	349
Карты отображений.....	350
Унаследованные коллекции.....	356
Алгоритмы класса Collections.....	360
Вопросы к главе 11.....	362
Задания к главе 11	362
Тестовые задания к главе 11	365
 Глава 12. ПОТОКИ ВЫПОЛНЕНИЯ	369
Класс Thread и интерфейс Runnable.....	369
Жизненный цикл потока	371
Перечисление TimeUnit.....	372
Интерфейс Callable.....	373

Механизм ForkJoin	375
Параллелизм	379
Timer и поток TimerTask	381
Управление приоритетами	382
Управление потоками	382
Потоки-демоны	385
Потоки и исключения	386
Атомарные типы и модификатор volatile	387
Методы synchronized	390
Инструкция synchronized	392
Монитор	393
Механизм wait\notify	394
Интерфейс Lock как альтернатива synchronized	397
Семафор	403
Барьер	408
CountDownLatch или «защелка»	411
Deadlock	415
Обмен блокировками	416
Блокирующие очереди	418
CopyOnWriteArrayList	420
Phaser	422
Вопросы к главе 12	423
Задания к главе 12	424
Тестовые задания к главе 12	425
Глава 13. Java DataBase Connectivity	428
Драйверы, соединения и запросы	428
СУБД MySQL	431
Соединение и запрос	431
Добавление и изменение записи	435
Метаданные	435
Подготовленные запросы и хранимые процедуры	437
Транзакции	441
Точки сохранения	443
Data Access Object	444
DAO. Уровень метода	446
DAO. Уровень слоя	448
Вопросы к главе 13	453
Задания к главе 13	454
Глава 14. СЕТЕВЫЕ ПРОГРАММЫ	461
Компьютерные сети	461
Адресация в глобальных сетях	461
Доменные имена	462
URL адреса	463
Сетевые программы	464

Сокетные соединения по TCP/IP	468
Многопоточность.....	470
Датаграммы и протокол UDP.....	473
Электронная почта	476
Простой сервлет.....	478
Вопросы к главе 14.....	481
Задания к главе 14	481
Глава 15. Java API for XML Processing	483
Древовидная и псевдособытийная модели	483
Валидация.....	484
Псевдособытийная модель.....	486
SAX-анализаторы.....	487
Древовидная модель.....	495
DOM	496
Создание XML-документа	499
StAX	500
Схема XSD	508
Вопросы к главе 15.....	510
Задания к главе 15	510
ОТВЕТЫ НА ТЕСТОВЫЕ ВОПРОСЫ	515
Приложение 1. Log4J2.....	516
Приложение 2. TestNG	534
СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ.....	553

ПРЕДИСЛОВИЕ

Второе издание «Java from EPAM» включает исправления предыдущей книги авторов «Java from EPAM».

Книга создавалась в процессе обучения языку Java и технологиям студентов механико-математического факультета и факультета прикладной математики и информатики Белорусского государственного университета, а также слушателей очных и онлайн-тренингов EPAM Systems по ряду направлений технологий Java. При изучении Java в рамках данного пособия знание других языков необязательно, книгу можно использовать для обучения программированию на языке Java «с нуля».

Интересы авторов, направленные на обучение, определили структуру этой книги. Она предназначена как для начинающих изучение Java-технологий, так и для студентов и программистов, переходящих на Java с другого языка программирования. Авторы считают, что профессионала «под ключ» обучить нельзя, им становятся только после участия в разработке нескольких серьезных Java-проектов. В то же время данный курс может служить ступенькой мастерству. Прошедшие обучение по этому курсу успешно сдают различные экзамены, получают международные сертификаты и участвуют в командной разработке промышленных программных проектов.

Книга разделена на две логические части. В первой даны фундаментальные основы языка Java и концепции объектно-ориентированного программирования. Во второй изложены наиболее важные аспекты применения языка, в частности коллекции и базы данных, многопоточность и взаимодействие с XML. В конце каждой главы приводятся вопросы для закрепления теоретических знаний, тестовые вопросы по материалам данной главы и задания для выполнения по рассмотренной теме. Ответы к тестовым вопросам сгруппированы в отдельный блок.

В приложениях предложены дополнительные материалы, относящиеся к использованию в информационных системах технологий Log4J и TestNG.

Авторы выражают благодарность компании EPAM Systems и ее сотрудникам, принимавшим участие в подготовке материалов этой книги и в ее издании.

ВВЕДЕНИЕ В ООП И ФП

Каждый может написать программу, которую может понять компьютер. Хороший программист пишет программу, которую может понять человек.

Мартин Фаулер

В связи с проникновением компьютеров во все сферы информационного общества программные системы становятся более простыми для пользователя и более сложными по внутренней архитектуре. Программирование стало делом команды, где маленьким проектом считается тот, который выполняет команда из 5–10 специалистов за время от полугода, а большим, который длится годами и исполняется сотнями программистов в разных странах. Основными способами создания сложных программных продуктов стали современные информационные технологии и такие методологии, как объектно-ориентированное программирование (ООП) и функциональное программирование (ФП).

Базовые понятия ООП

Java возникла на пике популярности технологий объектно-ориентированного программирования (ООП) и включает все основные ее парадигмы.

ООП — методология программирования, основанная на функционировании программного продукта как результата взаимодействия совокупности объектов, каждый из которых является экземпляром конкретного класса.

Объект — именованная модель реальной сущности, обладающая конкретными значениями свойств и проявляющая свое поведение.

Класс — модель информационной сущности, представляющая универсальный тип данных, состоящая из набора полей данных и методов их обработки.

В применении к объектно-ориентированным языкам программирования понятия объекта и класса конкретизируются. Во всех языках классы и объекты обладают рядом общих свойств, таких как инкапсуляция (объединение открытых данных и закрытых методов), наследование (заимствование функциональности базовых классов производными), полиморфизм (возможность использования объектов с одинаковым интерфейсом при наследовании).

Базовые понятия ФП

Функциональное программирование (ФП) было добавлено в очередную версию Java и ориентировано на вычисления и обработку информации. Этот подход предполагает формирование функции как объекта и передачу этого объекта в метод для использования или, наоборот, сама функция есть возвращаемое методом значение. Функциональный стиль программирования не заменит другие подходы к созданию кода. Многие действительно последовательные процессы, такие как поведение программных моделей в реальном времени, игровые и другие программы, организующие взаимодействие компьютера с человеком, не могут быть выражены в функциональном стиле. Функциональное программирование позволяет по-иному взглянуть на процесс программирования, а некоторые приемы программирования, которые предназначены для написания программ в чисто функциональном стиле, могут с успехом использоваться и в традиционном программировании вместе с ООП.

Базовые понятия Java

Объектно-ориентированный язык Java был разработан в компании Sun Microsystems в 1995 году для программирования небольших устройств и введения динамики на сайтах в виде апплетов. Немного позже язык Java нашел широкое применение в интернет-приложениях, добавив на клиентские веб-страницы динамический интерфейс, улучшив вычислительные возможности. Однако уже буквально через несколько лет после создания язык практически покинул клиентские страницы и перебрался на серверы. На стороне клиента его место заняли языки JavaScript и его производные.

При создании язык Java предполагался более простым, чем его синтаксический предок C++. Сегодня с появлением новых версий возможности языка Java существенно расширились и во многом перекрывают функциональность C++. Java уже не уступает по сложности предшественникам и называть его простым нельзя.

Отсутствие указателей, как наиболее опасного средства языка C++, нельзя считать сужением возможностей, а тем более — недостатком, это просто требование безопасности. Возможность работы с произвольными адресами памяти через безтиповые указатели позволяет игнорировать защиту памяти. Вместо указателей в Java широко используются ссылки. Отсутствие в Java множественного наследования состояния легко заменяется на более понятные конструкции с применением интерфейсов.

Системная библиотека классов языка Java содержит классы и пакеты, реализующие и расширяющие базовые возможности языка, а также сетевые средства, взаимодействие с базами данных, многопоточность и многое другое. Методы классов, включенные в эти библиотеки, вызываются JVM (Java Virtual Machine) во время интерпретации программы.

В Java все объекты программы расположены в динамической памяти — куче данных (*heap*) — и доступны по объектным ссылкам, которые хранятся в стеке (*stack*). Это решение исключило непосредственный доступ к памяти, но усложнило работу с элементами массивов и сделало ее менее эффективной по сравнению с программами на C++. В свою очередь, в Java предложен усовершенствованный механизм работы с коллекциями, реализующими основные динамические структуры данных. Необходимо отметить, что объектная ссылка языка Java содержит информацию о классе объекта, на который она ссылается, так что объектная ссылка — это не только ссылка на объект, размещенный в динамической памяти, но и дескриптор (описание) объекта. Наличие дескрипторов позволяет JVM выполнять проверку совместимости типов на фазе интерпретации кода, генерируя исключение в случае ошибки. В Java изменена концепция организации динамического распределения памяти: отсутствуют способы программного освобождения динамически выделенной памяти с помощью деструктора, понятие которого исключено из Java. Вместо этого реализована система автоматического освобождения памяти «сборщик мусора», выделенной с помощью оператора **new**. Программист может только рекомендовать системе освободить выделенную динамическую память.

В отличие от C++, Java не поддерживает перегрузку операторов, беззнаковые целые, прямое индексирование памяти и, как следствие, указатели. В Java существуют конструкторы, но отсутствуют деструкторы, т.к. применяется автоматическая сборка мусора, запрещены оператор **goto** и слово **const**, хотя они являются зарезервированными словами языка.

Кроме ключевых слов в Java существуют три литерала: **null**, **true**, **false**, не относящиеся к ключевым и зарезервированным словам, а также зарезервированное слово **var**, значение которого зависит от его позиции в коде. Слово **var** используется вместо типа локальной переменной метода, для которой существует инициализатор, определяющий тип: **var str = "sun"**; Введены также новые ключевые слова **record** и **yield**.

Ключевые и зарезервированные(*) слова языка Java:

abstract	default	if	protected	throw
assert	do	implements	public	throws
boolean	double	import	record	transient
break	else	instanceof	return	try
byte	enum	int	short	var*
case	extends	interface	static	void
catch	final	long	strictfp	volatile
char	finally	native	super	while
class	float	new	switch	yield
const*	for	package	synchronized	
continue	goto*	private	this	

Простое приложение

Изучение любого языка программирования удобно начинать с программы передачи символического сообщения на консоль.

```
// # 1 # простое линейное приложение # HelloTutorial.java
```

```
public class HelloTutorial {
    public static void main(String[] args) {
        System.out.println("tutorial->https://docs.oracle.com/javase/tutorial/");
    }
}
```

Здесь функция **main()**, с которой начинается выполнение любой программы Java, является методом класса **HelloTutorial**. Такая простая структура класса не является хорошей. Пусть в процессе тестирования или внедрения системы окажется, что фразу необходимо заменить на другую, например, в конце поставить восклицательный знак. Для этого программисту придется обыскивать весь код, искать места, где встречается указанная фраза, и заменять ее новой. Во избежание подобных проблем сообщение лучше хранить в отдельном методе или константе (а еще лучше — в файле) и при необходимости вызывать его. Тогда изменение текста сообщения приведет к локальному изменению одной-единственной строки кода. В следующем примере этот код будет переписан с использованием двух классов, реализованных на основе простейшего применения объектно-ориентированного программирования:

```
/* # 2 # простое объектно-ориентированное приложение # FirstProgram.java */
```

```
package by.epam.learn.main;
public class FirstProgram {
    public static void main(String[] args) {
        // declaring and creating an object
        TutorialAction action = new TutorialAction();
        // calling a method that outputs a string
        action.printMessage("tutorial-> https://docs.oracle.com/javase/tutorial/");
    }
}
```

```
/* # 3 # простой класс # TutorialAction.java */
```

```
class TutorialAction {
    void printMessage(String msg) { // method definition
        // output string
        System.out.println(msg);
    }
}
```

Здесь класс **FirstProgram** используется для того, чтобы определить метод **main()**, который вызывается автоматически интерпретатором Java и может называться контроллером этого примитивного приложения. Метод **main()** получает в качестве параметра аргументы командной строки **String[]args**, представляющие массив строк, и является открытым (**public**) членом класса. Это означает, что метод **main()** может быть виден и доступен любому классу. Ключевое слово **static** объявляет методы и переменные класса, используемые при работе с классом в целом, а не только с объектом класса. Символы верхнего и нижнего регистров здесь различаются. Тело метода **main()** содержит объявление объекта:

```
TutorialAction action = new TutorialAction();
```

и вызов его метода:

```
action.printMessage("tutorial-> https://docs.oracle.com/javase/tutorial/");
```

Вывод строки **tutorial-> <https://docs.oracle.com/javase/tutorial/>** в примере осуществляет метод **println()** (ln — переход к новой строке после вывода) статического поля-объекта **out** класса **System**, который подключается к приложению автоматически вместе с пакетом **java.lang**. Приведенную программу необходимо поместить в файл **FirstProgram.java** (расширение **.java** обязательно), имя которого должно совпадать с именем **public**-класса.

Перед объявлением класса располагается строка

```
package by.epam.learn.main;
```

указывающая на принадлежность класса пакету с именем **by.epam.learn.main**, который является на самом деле каталогом на диске. Для приложения, состоящего из двух классов, наличие пакетов не является необходимостью. Однако даже при отсутствии слова **package** классы будут отнесены к пакету по умолчанию (*unnamed*), размещенному в корне проекта. Если же приложение состоит из нескольких сотен классов, то размещение классов по пакетам является жизненной необходимостью.

Классы из примеров 2 и 3 могут сохраняться как в одном файле, так и в двух файлах **FirstProgram.java** и **TutorialAction.java**. На практике следует хранить классы в отдельных файлах, что позволяет всем разработчикам проекта быстрее воспринимать концепцию приложения в целом.

```
/* # 4 # простое объектно-ориентированное приложение # FirstProgram.java */
```

```
package by.epam.learn.main;
import by.epam.learn.action.TutorialAction; // import a class from another package
public class FirstProgram {
    public static void main(String[] args) {
        TutorialAction action = new TutorialAction();
        action.printMessage("tutorial-> https://docs.oracle.com/javase/tutorial/");
    }
}
```

```
// # 5 # простой класс # TutorialAction.java
```

```
package by.epam.learn.action;
public class TutorialAction {
    public void printMessage(String msg) {
        System.out.println(msg);
    }
}
```

Установка JDK и IDE

Чтобы начать программировать, необходимо скачать установку Java с официального сайта производителя *oracle.com* по линку <https://www.oracle.com/java/technologies/javase-downloads.html>.

Затем установить на компьютер. При инсталляции рекомендуется указывать для размещения корневой каталог. Если JDK установлена в директории (для Windows) **c:\Java\jdk[version]**, то каталог, который компилятор Java будет рассматривать как корневой для иерархии пакетов, можно вручную задавать с помощью переменной среды окружения в виде: **CLASSPATH=.;c:\Java\jdk[version]**.

Переменной задано еще одно значение «.» для использования текущей директории, например, **c:\workspace** в качестве рабочей для хранения своих собственных приложений.

Чтобы можно было вызывать сам компилятор и другие исполняемые программы, переменную **PATH** нужно проинициализировать в виде **PATH=c:\Java\jdk[version]\bin**.

Этот путь указывает на месторасположение файлов **javac.exe** и **java.exe**. В различных версиях операционных систем путь к JDK может указываться различными способами.

JDK (*Java Development Kit*) — полный набор для разработки и запуска приложений, состоящий из компилятора, утилит, исполнительной системы JRE, библиотек, документации.

JRE (*Java Runtime Environment*) — минимальный набор для исполнения приложений, включающий JVM, но без средств разработки.

JVM (*Java Virtual Machine*) — базовая часть исполняющей системы Java, которая интерпретирует байт-код Java, скомпилированный из исходного текста Java-программы для конкретной операционной системы.

Однако при одновременном использовании нескольких различных версий компилятора и различных библиотек применение переменных среды окружения начинает мешать эффективной работе, так как при выполнении приложения поиск класса осуществляется независимо от версии. Когда виртуальная машина обнаруживает класс с подходящим именем, она его и подгружает.

Такая ситуация предрасполагает к ошибкам, порой трудноопределимым. Поэтому переменные окружения начинающим программистам лучше не определять вовсе.

Для компиляции и запуска приложений можно использовать два способа:

- 1) Командная строка;
- 2) IDE (*IntelliJ IDEA, Eclipse, NetBeans etc.*).

IDE позволяют создавать, компилировать и запускать приложения в значительно более удобной форме, чем с помощью командной строки.

IntelliJ IDEA — интегрированная среда разработки программного обеспечения для Java, разработанная компанией JetBrains.

Загрузить IntelliJ IDEA можно на официальном сайте <https://www.jetbrains.com/idea/>. Среда IDEA используется для разработки приложений, поэтому необходимо установить Java Development Kit (JDK). Студентам и магистрантам предоставляется возможность установить полную лицензионную версию после заполнения регистрационной формы <https://www.jetbrains.com/shop/eform/students>.

После установки IntelliJ IDEA программа предложит настроить среду разработки: выбрать тему и установить дополнительные плагины.

Eclipse — сообщество открытого кода, или *open source*, чьи проекты сфокусированы на создании открытой платформы для разработки, развертывания, управления приложениями с использованием различных фреймворков, инструментов и сред исполнения. Загрузить продукты Eclipse можно на официальном сайте <http://eclipse.org/>.

Для установки Eclipse Java IDE необходимо распаковать загруженный архив в ту директорию, в которой будет храниться IDE. Предварительно также необходимо установить JDK.

Компиляция и запуск приложения

Простейший способ компиляции — вызов строчного компилятора из корневого каталога, в котором находится каталог **by**, каталог **epam** и так далее:

```
javac by/epam/learn/action/TutorialAction.java
javac by/epam/learn/main/FirstProgram.java
```

При успешной компиляции создаются файлы **FirstProgram.class** и **TutorialAction.class**, имена которых совпадают с именами классов. Запустить этот байт-код можно с помощью интерпретатора Java:

```
java by.epam.learn.main.FirstProgram
```

Здесь к имени приложения **FirstProgram.class** добавляется путь к пакету от корня проекта **by.epam.learn.main**, в котором он расположен.

Обработка аргументов командной строки, передаваемых в метод **main()**, относится к необходимым программам в самом начале обучения языку. Аргументы

представляют последовательность строк, разделенных пробелами, значения которых присваиваются объектам массива **String[] args**. Элементу **args[0]** присваивается значение первой строки после имен компилятора и приложения. Количество аргументов находится в значении **args.length**. Поле **args.length** является константным полем класса, массива, значение которого не может быть изменено на протяжении всего жизненного цикла объекта массива.

```
/* # 6 # вывод аргументов командной строки в консоль # PrintArgumentMain.java */
package by.epam.learn.arg;
public class PrintArgumentMain {
    public static void main(String[] args){
        for (String str : args) {
            System.out.printf("Argument--> %s\n", str);
        }
    }
}
```

В данном примере используется цикл **for** для неиндексируемого перебора всех элементов и метод форматированного вывода **printf()**. Тот же результат был бы получен при использовании традиционного вида цикла **for**

```
for (int i = 0; i < args.length; i++) {
    System.out.println("Argument--> " + args[i]);
}
```

Запуск этого приложения осуществляется с помощью следующей командной строки вида:

```
java by.epam.learn.arg.PrintArgumentMain 2020 Oracle "Java SE"
```

что приведет к выводу на консоль следующей информации:

```
Argument--> 2020
Argument--> Oracle
Argument--> Java SE
```

Приложение, запускаемое с аргументами командной строки, может быть использовано как один из примитивных способов ввода в приложение внешних данных.

Основы классов и объектов

Классы в языке Java объединяют поля класса, методы, конструкторы, логические блоки и внутренние классы. Основные отличия от классов C++: все функции определяются внутри классов и называются *методами*; невозможно создать метод, не являющийся методом класса, или объявить метод вне класса; спецификаторы доступа **public**, **private**, **protected** воздействуют *только на те*

объявления полей, методов и классов, перед которыми они стоят, а не на участках от одного до другого спецификатора, как в C++; элементы по умолчанию не устанавливаются в **private**, а доступны для классов из данного пакета. Объявление класса в общем виде следующее:

```
[specifiers] class ClassName [extends SuperClass] [implements List_interfaces] {
    /* implementation */
}
```

Спецификатор доступа к классу может быть **public** (класс доступен в данном пакете и вне пакета). По умолчанию, если спецификатор класса **public** не задан, он устанавливается в дружественный (*friendly* или *private-package*). Такой класс доступен только в текущем пакете. Спецификатор *friendly* так же, как и *private-package*, при объявлении вообще не используется и не является ключевым словом языка. Это слово используется в сленге программистов, чтобы обозначить значение по умолчанию.

Кроме этого, спецификатор может быть **final** (класс не может иметь подклассов) и **abstract** (класс может содержать абстрактные нереализованные методы, объект такого класса создать нельзя).

Класс наследует все свойства и методы суперкласса (базового класса), указанного после ключевого слова **extends**, и может включать множество интерфейсов, перечисленных через запятую после ключевого слова **implements**. Интерфейсы относительно похожи на абстрактные классы, содержащие только статические константы и не имеющие конструкторов, но имеют целый ряд серьезных архитектурных различий.

Все классы любого приложения условно разделяются на две группы: классы — носители информации, и классы, работающие с этой информацией. Классы, обладающие информацией, содержат данные о предметной области приложения. Например, если приложение предназначено для управления воздушным движением, то предметной областью будут самолеты, билеты, багаж, пассажиры и пр. При проектировании классов информационных экспертов важна инкапсуляция, обеспечивающая значениям полей классов корректность информации.

В качестве примера с нарушением инкапсуляции можно рассмотреть класс **Coin** в приложении по обработке монет.

```
// # 7 # простой пример класса носителя информации # Coin.java
```

```
package by.epam.learn.bean;
public class Coin {
    public double diameter; // encapsulation violation
    private double weight; // correct encapsulation
    public double getDiameter() {
        return diameter;
    }
}
```

```
public void setDiameter(double value) {
    if (value > 0) {
        diameter = value;
    } else {
        diameter = 0.01; // default value
    }
}
public double takeWeight() { // incorrect method name
    return weight;
}
public void setWeight(double value) {
    weight = value;
}
}
```

Класс **Coin** содержит два поля **diameter** и **weight**, помеченные как **public** и **private**. Значение поля **weight** можно изменять только при помощи методов, например, **setWeight (double value)**. В некоторых ситуациях замена некорректного значения на значение по умолчанию может привести к более грубым ошибкам в дальнейшем, поэтому часто вместо замены производится генерация исключения. Поле **diameter** доступно непосредственно через объект класса **Coin**. Поле, объявленное таким способом, считается объявленным с нарушением «тугой» инкапсуляции, следствием чего может быть нарушение корректности информации, как это показано ниже:

```
// # 8 # демонстрация последствий нарушения инкапсуляции # CoinMain.java
```

```
package by.epam.learn.main;
import by.epam.learn.bean.Coin;
public class CoinMain {
    public static void main(String[] args) {
        Coin coin = new Coin();
        coin.diameter = -0.12; // incorrect: direct access--> ob.setWeight(100);
        // coin.weight = -150; // field is not available: compile error
    }
}
```

Чтобы поле **diameter** стало недоступно напрямую, а компиляция кода вида

```
coin.diameter = -0.12;
```

стала невозможной, следует поле **diameter** класса **Coin** объявить в виде

```
private double diameter;
```

тогда строка с попыткой прямого присваивания значения поля с помощью ссылки на объект приведет к ошибке компиляции.

```
// # 9 # «Туго» инкапсулированный класс (Java Bean) # Coin.java
```

```
package by.epam.learn.bean;
public class Coin {
    private double diameter;
    private double weight;
    public double getDiameter() {
        return diameter;
    }
    public void setDiameter(double value) {
        if(value > 0) {
            diameter = value;
        } else {
            System.out.println("Negative diameter!");
        }
    }
    public double getWeight() { // correct name
        return weight;
    }
    public void setWeight(double value) {
        weight = value;
    }
}
```

Проверка корректности входящей извне информации осуществляется в методе **setDiameter(double value)** и позволяет уведомить о нарушении инициализации объекта. Доступ к **public**-методам объекта класса осуществляется только после создания объекта данного класса.

```
/* # 10 # создание объекта, доступ к полям и методам объекта # CompareCoin.java
# CoinMain.java */
```

```
package by.epam.learn.action;
import by.epam.learn.bean.Coin;
public class CompareCoin {
    public void compareDiameter(Coin first, Coin second) {
        double delta = first.getDiameter() - second.getDiameter();
        if (delta > 0) {
            System.out.println("The first coin is more than the second for " + delta);
        } else if (delta == 0) {
            System.out.println("Coins have the same diameter");
        } else {
            System.out.println("The second coin is more than the first on " + -delta);
        }
    }
}

package by.epam.learn.main;
import by.epam.learn.bean.Coin;
import by.epam.learn.action.CompareCoin;
```

```
public class CoinMain {
    public static void main(String[] args) {
        Coin coin1 = new Coin();
        coin1.setDiameter(-0.11); // error message
        coin1.setDiameter(0.12); // correct
        coin1.setWeight(150);
        Coin coin2 = new Coin();
        coin2.setDiameter(0.21);
        coin2.setWeight(170);
        CompareCoin compare = new CompareCoin();
        compare.compareDiameter(coin1, coin2);
    }
}
```

Компиляция и выполнение данного кода приведут к выводу на консоль следующей информации:

Negative diameter!

The second coin is more than the first on 0.09

Объект класса создается за два шага. Сначала объявляется ссылка на объект класса. Затем с помощью оператора **new** создается экземпляр объекта, например:

```
Coin coin; // declaring an object reference
coin = new Coin(); // object instantiation
```

Однако эти два действия обычно объединяют в одно:

```
Coin coin = new Coin();
```

Оператор **new** вызывает конструктор, в данном примере используется конструктор по умолчанию без параметров, но в круглых скобках могут размещаться аргументы, передаваемые конструктору, если у класса объявлен конструктор с параметрами. Операция присваивания для объектов означает, что две ссылки будут указывать на один и тот же участок памяти.

Метод **compareDiameter(Coin first, Coin second)** выполняет два действия, которые следует разделять: выполняет сравнение и печатает отчет. Действия слишком различны по природе, чтобы быть совмещенными. Естественным решением будет изменить возвращаемое значение метода на **int** и оставить в нем только вычисления.

```
/* # 11 # метод сравнения экземпляров по одному полю */
```

```
public int compareDiameter(Coin first, Coin second) {
    int result = 0;
    double delta = first.getDiameter() - second.getDiameter();
    if (delta > 0) {
        result = 1;
    }
}
```

```

    } else if (delta < 0) {
        result = -1;
    }
    return result;
}

```

Формирование отчета следует поместить в другой метод другого класса.

Объектные ссылки

Java работает не с объектами, а со ссылками на объекты, размещаемыми в динамической памяти с помощью оператора **new**. Это объясняет то, что операции сравнения ссылок на объекты не имеют смысла, так как при этом сравниваются адреса. Для сравнения объектов на эквивалентность по значению необходимо использовать специальные методы, например, **equals(Object ob)**. Этот метод наследуется в каждый класс из суперкласса **Object**, который лежит в корне дерева иерархии всех классов и должен переопределяться в подклассе для определения эквивалентности содержимого двух объектов этого класса.

```
/* # 12 # сравнение ссылок и объектов # ComparisonString.java */
```

```

package by.epam.learn.main;
public class ComparisonString {
    public static void main(String[ ] args) {
        String str1, str2;
        str1 = "Java";
        str2 = str1; // variable refers to the same string
        System.out.println("comparison of references " + (str1 == str2)); // true
        str2 = new String("Java"); // is equivalent to str2 = new String(str1);
        System.out.println("comparison of references " + (str1 == str2)); // false
        System.out.println("comparison of values " + str1.equals(str2)); // true
    }
}

```

В результате выполнения действия **str2 = str1** получается, что обе ссылки ссылаются на один и тот же объект. Оператор «==» возвращает **true** при сравнении ссылок только в том случае, если они ссылаются на один и тот же объект.

Если же ссылку **str2** инициализировать конструктором **new String(str1)**, то создается новый объект в другом участке памяти, который инициализируется значением, взятым у объекта **str1**. В итоге существуют две ссылки, каждая из которых независимо ссылается на объект, который никак физически не связан с другим объектом. Поэтому оператор сравнения ссылок возвращает результат **false**, так как ссылки ссылаются на различные участки памяти. Объекты обладают одинаковыми значениями, что легко определяется вызовом метода **equals(Object ob)**.

Если в процессе разработки возникает необходимость в сравнении по значению объектов классов, созданных программистом, для этого следует переопределить в данном классе метод **equals(Object ob)** в соответствии с теми критериями сравнения, которые существуют для объектов данного типа или по стандартным правилам, заданным в документации.

Консольный ввод\вывод

Консоль предоставляет пользователю простой способ взаимодействия с программой посредством потоков ввода\вывода. В Java взаимодействие с консолью обеспечивает класс **System**, а его статические поля **in**, **out** и **err** обеспечивают ввод\вывод. Для вывода информации на консоль используют методы **System.out.println()** и другие.

Взаимодействие с консолью с помощью потока (объекта класса) **System.in** представляет собой один из простейших способов ввода. При создании первых приложений такого рода передача в них информации является единственно доступной для начинающего программиста. В следующем примере рассматривается ввод информации в виде символа из потока ввода, связанного с консолью, и последующего вывода на консоль символа и его числового кода.

```
// # 13 # чтение символа из потока System.in # ReadCharMain.java 24

package by.epam.learn.console;
import java.io.IOException;
public class ReadCharMain {
    public static void main(String[] args) {
        int x;
        try {
            x = System.in.read();
            char c = (char)x;
            System.out.println("Character Code: " + c + " = " + x);
        }
        catch (IOException e) {
            System.err.println("i/o error " + e);
        }
    }
}
```

Обработка исключительной ситуации **IOException**, которая может возникнуть в операциях ввода/вывода и в любых других взаимодействиях с внешними источниками данных, осуществляется в методе **main()** с помощью реализации блока **try-catch**. Если ошибок при выполнении не возникает, выполняется блок **try**, в противном случае генерируется исключительная ситуация, и выполнение программы перехватывает блок **catch**.

Ввод информации осуществляется посредством чтения строки из консоли с помощью возможностей объекта класса **Scanner**, имеющего возможность соединяться практически с любым потоком/источником информации: строкой, файлом, сокетом, адресом в интернете, с любым объектом, из которого можно получить ссылку на поток ввода.

```
// # 14 # чтение строки из консоли # ScannerMain.java
```

```
package by.epam.learn.console;
import java.util.Scanner;
public class ScannerMain {
    public static void main(String[] args) {
        System.out.println("Enter name and press <Enter> & number and press <Enter>:");
        Scanner scan = new Scanner(System.in);
        String name = scan.next();
        System.out.println("hello, " + name);
        int num = scan.nextInt();
        System.out.println("number= " + num);
        scan.close();
    }
}
```

В результате запуска приложения будет выведено, например, следующее:

Enter name and press <Enter> & number and press <Enter>:

ostap

hello, ostap

777

number= 777

Класс **Scanner** объявляет ряд методов для ввода: **next()**, **nextLine()**, **nextInt()**, **nextDouble()** и др.

Позже будут рассмотрены более удобные способы извлечения информации из потока ввода с помощью класса **Scanner**, в качестве которого может фигурировать не только консоль, но и дисковый файл, строка, сокетное соединение и пр.

Base code conventions

При выборе имени класса, поля, метода использовать цельные слова, полностью исключить сокращения, кроме общепринятых. По возможности опускать предлоги и очевидные связующие слова. Аббревиатуры использовать только в том случае, когда они очевидны. Если избежать сокращения не получается, надо помнить, что начало важнее конца, согласные важнее гласных.

Имя класса всегда пишется с большой буквы: **Coin**, **Developer**.

Если имя класса состоит из двух и более слов, то второе и следующие слова пишутся слитно с предыдущим и начинаются с большой буквы: **AncientCoin**, **FrontendDeveloper**.

Имя метода всегда пишется с маленькой буквы: **perform()**, **execute()**.

Если имя метода состоит из двух и более слов, то второе и следующие слова пишутся слитно с предыдущим и начинаются с большой буквы: **performTask()**, **executeBaseAction()**.

Имя поля класса, локальной переменной и параметра метода всегда пишется с маленькой буквы: **weight**, **price**.

Если имя поля класса, локальной переменной и параметра метода состоит из двух и более слов, то второе и следующие слова пишутся слитно с предыдущим и начинаются с большой буквы: **priceTicket**, **typeProject**.

Константы и перечисления пишутся в верхнем регистре: **DISCOUNT**, **MAX_RANGE**.

Все имена пакетов пишутся с маленькой буквы. Сокращения допустимы только в случае, если имя пакета слишком длинное: 10 или более символов. Использование цифр и других символов нежелательно.

Вопросы к главе 1

1. Что имеется в виду, когда говорится: Java-язык программирования и Java-платформа?
2. Расшифровать аббревиатуры JVM, JDK и JRE. Показать, где они физически расположены и что собой представляют.
3. JVM-JDK-JRE. Кто кого включает и как взаимодействуют.
4. Как связаны имя Java-файла и классы, которые в этом файле объявляются?
5. Как скомпилировать и запустить класс, используя командную строку?
6. Что такое *classpath*? Зачем в переменных среды окружения прописывать пути к установленному JDK?
7. Если в *classpath* есть две одинаковые библиотеки (или разные версии одной библиотеки), объект класса из какой библиотеки создастся?
8. Объяснить различия между терминами «объект» и «ссылка на объект».
9. Какие области памяти использует Java для размещения простых типов, объектов, ссылок, констант, методов, пул строк и т.д.
10. Почему метод **main()** объявлен как **public static void**?
11. Возможно ли в сигнатуре метода **main()** поменять местами слова **static** и **void**?
12. Будет ли вызван метод **main()** при запуске приложения, если слова **static** или **public** отсутствуют?
13. Классы какого пакета импортируются в приложение автоматически?

Задания к главе 1

Вариант А

1. Приветствовать любого пользователя при вводе его имени через командную строку.
2. Отобразить в окне консоли аргументы командной строки в обратном порядке.
3. Вывести заданное количество случайных чисел с переходом и без перехода на новую строку.
4. Ввести пароль из командной строки и сравнить его со строкой-образцом.
5. Ввести целые числа как аргументы командной строки, подсчитать их суммы и произведения. Вывести результат на консоль.
6. Вывести фамилию разработчика, дату и время получения задания, а также дату и время сдачи задания.

Вариант В

Ввести с консоли n целых чисел. На консоль вывести:

1. Четные и нечетные числа.
2. Наибольшее и наименьшее число.
3. Числа, которые делятся на 3 или на 9.
4. Числа, которые делятся на 5 и на 7.
5. Все трехзначные числа, в десятичной записи которых нет одинаковых цифр.
6. Простые числа.
7. Отсортированные числа в порядке возрастания и убывания.
8. Числа в порядке убывания частоты встречаемости чисел.
9. «Счастливые» числа.
10. Числа-палиндромы, значения которых в прямом и обратном порядке совпадают.
11. Элементы, которые равны полусумме соседних элементов.

Тестовые задания к главе 1

Вопрос 1.1.

Какие объявления параметров корректны для метода **public static void main?** (выбрать два)

- a) `String args []`
- b) `String [] args`
- c) `Strings args []`
- d) `String args`

Вопрос 1.2.

Какой из предложенных пакетов содержит **class System**? (выбрать один)

- a) java.io
- b) java.base
- c) java.util
- d) java.lang

Вопрос 1.3.

Какая команда выполнит компиляцию Java приложения **First.java**? (выбрать один)

- a) javac First
- b) java First.class
- c) javac First.java
- d) java First.java
- e) java First

Вопрос 1.4.

Какие слова являются ключевыми словами Java? (выбрать два)

- a) classpath
- b) for
- c) main
- d) out
- e) void

Вопрос 1.5.

Дан код:

```
public class Main {
    public static void main(String[] args) {
        for(int x = 0; x < 1; x++)
            System.out.print(x);
            System.out.print(x);
    }
}
```

Что будет в результате компиляции и запуска? (выбрать один)

- a) 01
- b) 00
- c) 11
- d) compilation fails
- e) runtime error

Вопрос 1.6.

Дан код:

Каков будет вывод в консоль, если код запускается из командной строки

java PRIVATE ? (выбрать один)

```
public class P {  
    public static void main(String[] s) {  
        System.out.print(s[1] + s[3] + s[2] + s[1]);  
    }  
}
```

- a) PIRP
- b) IEVI
- c) RVIR
- d) compilation fails
- e) runtime error

ТИПЫ ДАННЫХ И ОПЕРАТОРЫ

Программирование всегда осуждалось светскими, духовными и прочими властями.

Любая программа манипулирует данными и объектами с помощью операторов. Каждый оператор производит результат из значений своих операндов или изменяет непосредственно значение операнда.

Базовые типы данных и литералы

Java — язык объектно-ориентированного программирования, однако не все данные в языке есть объекты. Для повышения производительности в нем, кроме объектов, используются базовые типы данных, значения которых (литералы) размещаются в стековой памяти. Для каждого базового типа имеются также классы-оболочки, которые инкапсулируют данные базовых типов в объекты, располагаемые в динамической памяти (*heap*). Базовые типы обеспечивают более высокую производительность вычислений по сравнению с объектами классов-оболочек и другими объектами. Что является основной причиной применения в Java базовых типов, а не объектной модели полностью. Значения базовых типов данных, записанных по правилам языка Java, называют литералами.

Определено восемь базовых типов данных, размер каждого из которых остается неизменным независимо от платформы (рис. 2.1.).

Беззнаковых типов в Java не существует. Каждый тип данных определяет множество значений и их представление в памяти. Для каждого типа определен набор операций над его значениями.

В Java используются целочисленные литералы, например: **42** — целое (**int**) десятичное число, **042** — восьмеричное целое число, **0x42b** — шестнадцатеричное целое число, **0b101010** — двоичное целое число. Целочисленные литералы по умолчанию относятся к типу **int**. Если необходимо определить длинный литерал типа **long**, в конце указывается символ **L** (например: **0xffffL**). Если значение числа больше значения, помещающегося в **int** (**2147483647**), то Java автоматически полагает, что оно типа **long**.

В Java для удобства восприятия литералов стало возможно использовать знак «**_**» при объявлении больших чисел, то есть вместо **int value = 4200000**

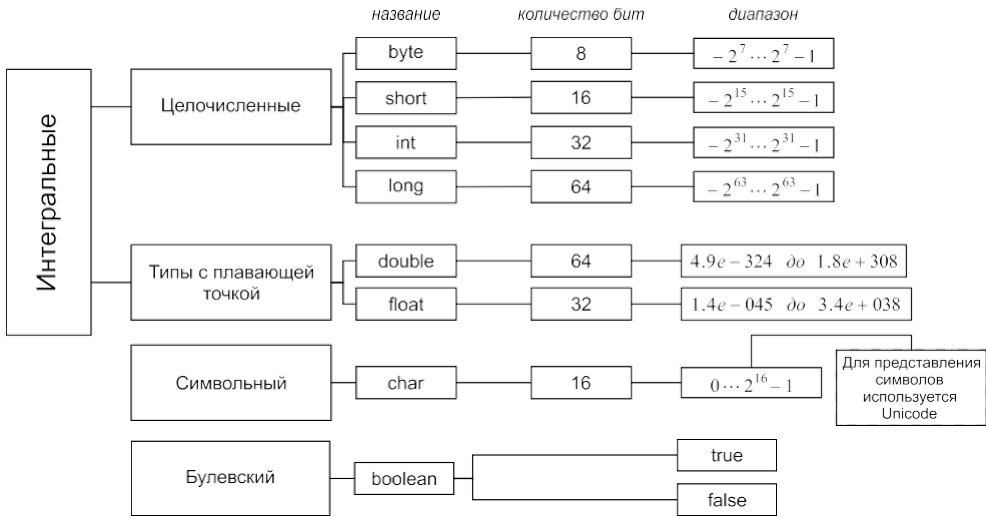


Рис. 2.1. Базовые типы данных и их свойства

можно записать **int value = 4_200_000**. Эта форма применима и для чисел с плавающей запятой. Однако некорректно: `_7` или `7_`.

Литералы с плавающей точкой записываются в виде **1.618** или в экспоненциальной форме **0.117E-5** и относятся к типу **double**. Таким образом, действительные числа относятся к типу **double**. При объявлении такого литерала можно использовать символы **d** и **D**, а именно **1.618d**. Если необходимо определить литерал типа **float**, то в конце литерала необходимо добавить символ **F** или **f**. По стандарту IEEE 754 введены понятие бесконечности **+Infinity** и **-Infinity**, число большее или меньшее любого другого числа при делении на ноль в типах с плавающей запятой, а также значение **NaN** (Not a Number), которое может быть получено, например, при извлечении квадратного корня из отрицательного числа.

К булевым литералам относятся значения **true** и **false**. Литерал **null** — значение по умолчанию для объектной ссылки.

Символьные литералы определяются в апострофах (**'a'**, **'\n'**, **'\042'**, **'\ub76f'**). Для размещения символов используется формат Unicode, в соответствии с которым для каждого символа отводится два байта. В формате Unicode первый байт содержит код управляющего символа или национального алфавита, а второй байт соответствует стандартному ASCII коду, как в C++. Любой символ можно представить в виде **'\ucode'**, где **code** представляет двухбайтовый шестнадцатеричный код символа. Java поддерживает управляющие символы, не имеющие графического изображения; **'\n'** — новая строка, **'\r'** — переход к началу, **'\f'** — новая страница, **'\t'** — табуляция, **'\b'** — возврат на один символ, **'\uxxxx'** — шестнадцатеричный символ Unicode, **'\dddd'** — восьмеричный символ и др.

В настоящее время Java обеспечивает поддержку стандарта Unicode 10.0.0 возможностями классов **Character**, **String**.

Переменные могут быть либо членами класса, либо локальными переменными метода, либо параметрами метода. По стандартным соглашениям имена переменных не могут начинаться с цифры, в именах не могут использоваться символы арифметических и логических операторов, а также символ '#'. Применение символов '\$' и '_' допустимо, в том числе и в первой позиции имени, но нежелательно. Каждая переменная должна быть объявлена с одним из указанных выше типов.

Можно конечно записать такой корректный с точки зрения компилятора, но абсолютно бессмысленный код:

```
int _ = 2;
int __ = 3;
int___ = _ * __;
```

но это будет яркий пример антипрограммирования. В приведенном фрагменте ни одна переменная не указывает на смысл хранимого им значения и результата вычислений.

Но если переменным дать имена,

```
int height = 2;
int width = 3;
int rectangleArea = height * width;
```

то станет ясно, что здесь приведен процесс вычисления площади прямоугольника.

Переменная базового типа, объявленная как член класса, хранит нулевое значение, соответствующее своему типу. Если переменная объявлена как локальная переменная в методе, то перед использованием она обязательно должна быть проинициализирована, так как локальная переменная не инициализируется по умолчанию нулем. Область действия и время жизни такой переменной ограничена блоком {}, в котором она объявлена, то есть код

```
int height = 2;
{ int width = 3;
}
int rectangleArea = height * width;
```

компилироваться не будет, так как переменная **width** недоступна (или не видна) вне блока {}.

Строки, заключенные в двойные апострофы, считаются литералами и размещаются в пуле литералов, но в то же время такие строки представляют собой объекты класса **String**. При инициализации строки создается объект класса **String**. При работе со строками, кроме методов класса **String**, можно использовать единственный в языке перегруженный оператор «+» конкатенации (слияния) строк. Конкатенация строки с объектом любого другого типа добавляет к исходному

объекту-строке строковое представление объекта другого типа. Строковый литерал заключается в двойные кавычки, это не ASCII-строка, а объект из набора (массива) символов. Строковые литералы в большинстве своем должны объявляться как константы **final static** поля классов для экономии памяти.

```
String name = "Sun";
name += '1';
name = name + 42;
```

В тоже время будет некорректно

```
str = "javac" - "c"; // error. subtraction of a line is impossible.
```

Оператор «+» для строк и объектов не перегружен. Самостоятельно запрограммировать перегрузку оператора также невозможно, так как перегрузка операторов в Java не реализована.

В арифметических выражениях автоматически выполняются расширяющие преобразования типа

byte → short → int → long → float → double

Это значит, что любая операция с участием различных типов даст результат, тип которого будет соответствовать большему из типов операндов. Например, результатом сложения значений типа **double** и **long** будет значение типа **double**.

Java автоматически расширяет тип каждого **byte** или **short** операнда до **int** в арифметических выражениях. Для сужающих диапазон значений преобразований необходимо производить явное преобразование вида (type)value. Например:

```
int number = 42;
byte value = (byte) number; // transformation int to byte
value = 42 + 1; // correct // line 1
value = 42 + 101; // compile error // line 2
```

строка, помеченная *line 1*, компилируется, несмотря на то, что операция сложения дает результатом **int**, но так как операнды константы, то компилятор вычисляет результирующее значение 43, которое не выходит за пределы допустимых значений типа **byte** (от -128 до 127), и преобразует результат к **byte**. В строке *line 2* компилятор вычисляет значение 143, и оно выходит за допустимые пределы, поэтому и не может быть преобразовано к **byte**.

```
byte b = 1;
final byte B = 2;
b = B + 1; // ok!
```

Аналогичная ситуация возникает и в случае сложения константы **B** с числовой константой, если результат не выходит за границы типа **byte**.

Далее фрагмент

```
byte b = 1;
b = b + 100;
```

не компилируется, так как в выражении присутствует переменная **b** и компилятор в общем случае предупреждает, что результат операции может выходить за допустимые пределы. Проблема компиляции решается явным преобразованием:

```
b = (byte)(b + 100);
```

При явном преобразовании (*type*)*value* возможно усечение значения. Если значение **b** изменить так, чтобы результат выходил за пределы границ для **byte**:

```
byte b = 42;
b = (byte)(b + 100);
```

то **b** получит значение **-114**, и программист должен понимать, какой смысл ему в этом результате.

Приведение типов не потребуется и в случае, если результатом будет тип **byte**, а в операции используется константа типа **int**:

```
byte b = 42;
final int VALUE = 1;
b = b + VALUE;
```

где компилятор определяет результат **43** как допустимый.

При инициализации полей класса и локальных переменных методов с использованием арифметических операторов автоматически выполняется неявное приведение литералов к объявленному типу без необходимости его явного указания, если только их значения находятся в допустимых пределах. В операциях присваивания нельзя присваивать переменной значение более длинного типа, в этом случае необходимо явное преобразование типа. Исключение составляют операторы инкремента «++», декремента «--» и сокращенные операторы (+=, /= и т.д.), которые сами выполняют приведение типов:

```
byte b = 42;
int i = 1;
b += i++; // ok!
b += 1000; // ok!
```

В Java 10 введено понятие *type inference* — исключение из объявления типа локальной переменной с помощью **var**. Теперь при объявлении локальных переменных метода вместо

```
int counter = 1;
String name = new String();
```

можно записать

```
var counter = 1;
var name = new String();
```

При просмотре данного выражения компилятор обрабатывает правую часть выражения и присваивает его тип переменной. **var** — зарезервированное имя типа, не является ключевым словом и может использоваться как имя переменной или метода.

То есть корректны объявления переменной и метода:

```
var var = 1;
void var(){}

```

Применение **var** экономит несколько символов при создании кода и избегает дублирования имен типов при объявлении переменной.

Рекомендуется избегать применения **var** в случае, когда значение типа данных неочевидно при прочтении кода:

```
var a = ob.method();

```

Документирование кода

В языке Java используются блочные и однострочные комментарии `/* */` и `//`. Введен также особый вид комментария `/** */`, который содержит описание класса/метода/поля с помощью дескрипторов вида:

- @author** — задает сведения об авторе;
- @version** — задает номер версии класса;
- @exception** — задает имя класса исключения;
- @param** — описывает параметры, передаваемые методу;
- @return** — описывает тип, возвращаемый методом;
- @deprecated** — указывает, что метод устаревший и у него есть более совершенный аналог;
- @since** — определяет версию, с которой метод (член класса, класс) присутствует;
- @throws** — описывает исключение, генерируемое методом;
- @see** — что следует посмотреть дополнительно.

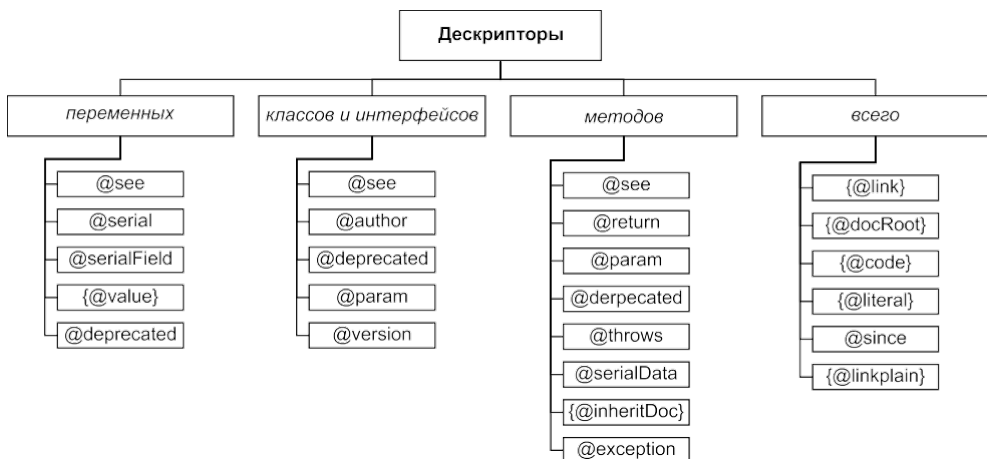


Рис. 2.2. Дескрипторы документирования кода

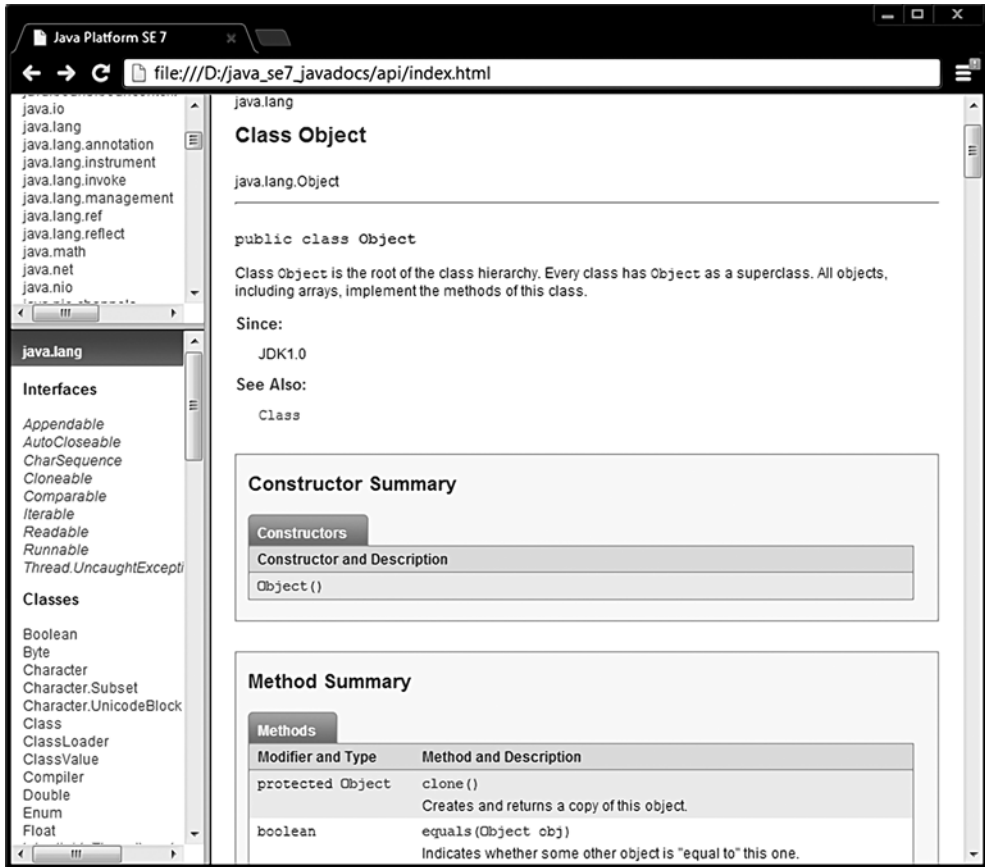


Рис. 2.3. Сгенерированная документация для класса *Object*

Из java-файла, содержащего такие комментарии, соответствующая утилита **javadoc.exe** может извлекать информацию для документирования классов и сохранения ее в виде html-документа. В качестве примера и образца для подражания следует рассматривать исходный код языка Java и документацию, сгенерированную на его основе (рис. 2.3.).

```
/* # 1 # фрагмент класса Object с дескрипторами документирования # Object.java */
```

```
package java.lang;
/**
 *Class {@code Object} is the root of the class hierarchy.
 *Every class has {@code Object} as a superclass. All objects,
 * including arrays, implement the methods of this class.
 *@author unascrbed
 *@see java.lang.Class
```

```

*@since JDK1.0
*/
public class Object {
    /**
     *Indicates whether some other object is "equal to" this one.
     * MORE COMMENTS HERE
     *@param obj the reference object with which to compare.
     *@return {@code true} if this object is the same as the obj
     *argument; {@code false} otherwise.
     *@see hashCode()
     *@see java.util.HashMap
     */
    public boolean equals(Object obj) {
        return (this == obj);
    }
    /**
     *Creates and returns a copy of this object.
     *MORE COMMENTS HERE
     *@return a clone of this instance.
     *@exception CloneNotSupportedException if the object's class does not
     *support the {@code Cloneable} interface. Subclasses
     *that override the {@code clone} method can also
     *throw this *exception to indicate that an instance cannot be cloned.
     *@see java.Lang.Cloneable
     */
    protected native Object clone() throws CloneNotSupportedException;
    // more code here
}

```

Всегда следует помнить, что точные названия классов, их полей и методов улучшают восприятие кода и уменьшают размер комментариев. Наличие комментария должно еще больше облегчить скорость восприятия разработанного кода. Код системы будет читаться чаще и больше по времени, чем требуется на его создание. Комментарии помогут программисту, сопровождающему код, быстрее разобраться в нем и грамотнее использовать или изменять его.

Документирование кода следует проводить в первую очередь для интерфейсов и суперклассов. Переопределенные и реализованные методы не документируются.

Операторы

Операторы Java практически совпадают с операторами в других современных языках программирования и имеют приоритет, как приведенный на рисунке 2.4. Операторы работают с базовыми типами, для которых они определены, и объектами классов-оболочек над базовыми типами. Кроме этого, операторы «+» и «+=» производят также действия по конкатенации операндов типа **String**. Логические операторы «==», «!=» и оператор присваивания «=» применимы

К операндам любого объектного и базового типов, а также литералам. Применение оператора присваивания к объектным типам часто приводит к ошибке несовместимости типов во время выполнения приложения с генерацией исключения **ClassCastException**, поэтому такие операции необходимо контролировать. Деление на ноль в целочисленных типах вызывает исключительную ситуацию, переполнение не контролируется.

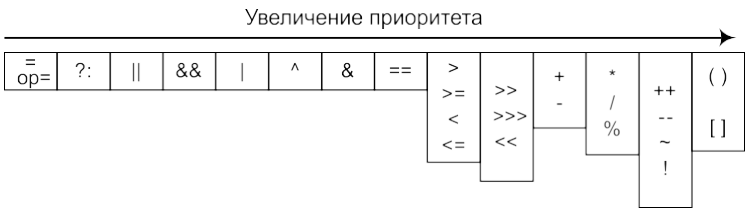


Рис. 2.4. Таблица приоритетов операций

Оператор «=» во всех операциях имеет самый низкий приоритет и выполняется самым последним.

```
int f;  
int g;  
int h;  
h = g = f = 42;
```

В выражении:

```
int result= 4 + 2 * 7 - 8;
```

первой будет выполнена операция умножения, затем в порядке очереди равноправные операции сложения и вычитания, последним выполняется присваивание как обладающее самым низким приоритетом.

Над числами с плавающей запятой выполняются арифметические операции и операции отношения, как и в других алгоритмических языках.

Арифметические операторы

+	Сложение	/	Деление (или выделение целой части для целочисленных операндов)
+=	Сложение (с присваиванием)	/=	Деление (с присваиванием)
-	Бинарное вычитание и унарное изменение знака	%	Остаток от деления (деление по модулю)
-=	Вычитание (с присваиванием)	%=	Остаток от деления (с присваиванием)
*	Умножение	++	Инкремент (увеличение значения на единицу)
*=	Умножение (с присваиванием)	--	Декремент (уменьшение значения на единицу)

Для целочисленных типов операция деления отбрасывает остаток от деления (деление нацело).

```
int x = 1;
int y = 3;
int z = x / y;
```

Результат: **z** будет равно **0**.

По этой же причине следующий фрагмент также даст результат 0.

```
z = x / y * y;
```

Если же операция деления производится для чисел с плавающей запятой, то результат будет, как и в обычной арифметике.

```
float f = 1;
int y = 3;
float r = f / y * y;
```

Результат: **r** будет равно **1.0**.

Оператор «%» позволяет получить остаток от деления для всех числовых типов данных:

```
int x = 1;
int y = 3;
int z = x % y;
```

Результат: **z** будет равно **1**.

Операторы инкремента и декремента в общем случае аналогичны действию **i = i + 1** и **i = i - 1**, но при присваивании нужно учитывать префиксную или постфиксную запись оператора. Префиксная запись первым действием увеличивает значение операнда на единицу и затем присваивает результат, постфиксная же работает в обратном порядке.

```
int i = 0;
int j = i++; // j=0 and i=1
int k = ++i; // i=2 and k=2
```

Результат: **j** получит значение **0**, **k** получит значение **2**, **i** получит значение **2**.

По-особому работает постфиксный оператор в случае использования с той же самой переменной. В этом случае значение постфиксного увеличения на единицу утрачивается, так как присваивание уже произошло.

```
int i = 0;
i = i++;
```

Результат: **i** получит значение **0**.

Битовые операторы над целочисленными типами

	Или	>>	Сдвиг вправо
=	Или (с присваиванием)	>>=	Сдвиг вправо (с присваиванием)
&	И	>>>	Сдвиг вправо с появлением нулей
&=	И (с присваиванием)	>>>=	Сдвиг вправо с появлением нулей и присваиванием
^	Исключающее или	<<	Сдвиг влево
^=	Исключающее или (с присваиванием)	<<=	Сдвиг влево с присваиванием
~	Унарное отрицание		

Примеры работы битовых операторов с числовыми типами приведены ниже.

```
int b1 = 0b1110; // 14
int b2 = 0b1001; // 9
int i = 0;
System.out.println(b1 + "|" + b2 + " = " + (b1|b2));
System.out.println(b1 + "&" + b2 + " = " + (b1&b2));
System.out.println(b1 + "^" + b2 + " = " + (b1^b2));
System.out.println(" ~" + b2 + " = " + ~b2);
System.out.println(b1 + ">>" + ++i + " = " + (b1>>i));
System.out.println(b1 + "<<" + i + " = " + (b1<<i++));
System.out.println(b1 + ">>>" + i + " = " + (b1>>>i));
```

Результатом выполнения данного кода будет:

```
14|9 = 15
14&9 = 8
14^9 = 7
~9 = -10
14>>1 = 7
14<<1 = 28
14>>>2 = 3
```

Операторы отношения

<	Меньше	>	Больше
<=	Меньше либо равно	>=	Больше либо равно
==	Равно	!=	Не равно

Эти операторы применяются для сравнения символов, целых и вещественных чисел, а также для сравнения ссылок при работе с объектами.

Логические операторы

	Или	&&	И	!	Унарное отрицание
--	-----	----	---	---	-------------------

Логические операции выполняются только над значениями типов **boolean** и **Boolean** (**true** или **false**).

Если один из операндов оператора «||» имеет значение **true**, то и результат всей операции равен **true**.

```
boolean a = true;
boolean b = false;
System.out.println(a || b);
```

Результат: **true**.

Если один из операндов оператора «&&» имеет значение **false**, то и результат всей операции равен **false**. Если оба операнда имеют значения **true**, то только в этом случае результатом оператора будет **true**.

```
System.out.println(a && b);
```

Результат: **false**.

При работе логических операторов в ситуации, если результат определяется одним операндом и нет необходимости проверять значение второго, то виртуальная машина к нему и не обращается. Проиллюстрировать такой функционал можно следующим примером:

```
int x = 0;
int y = 0;
System.out.println( x++ == y++ || x++ != y++ );
System.out.println("x=" + x + ", y=" + y);
```

Результат:

true

x=1, y=1.

Выражение справа от оператора «||» было вычислено, а выражение слева — нет.

В случае с битовым оператором «|» всегда вычисляются оба операнда, как в следующем фрагменте:

```
System.out.println( x++ == y++ | x++ != y++ );
System.out.println("x=" + x + ", y=" + y);
```

Результат:

true

x=2, y=2.

В операторе «&&» в случае истинного значения первого операнда будет вычисляться второй. Если же первый операнд имеет значение **false**, то второй выполнен не будет.

```
System.out.println( x++ == y++ && x++ != y++ );
System.out.println("x=" + x + ", y=" + y);
```

Результат:

false

x=2, y=2.

Если логические операторы расположены последовательно, то действия выполняются до тех пор, пока результат не будет очевиден для конкретного оператора. В следующем фрагменте отработает только первый операнд, так для оператора «||» результат выражения будет определяться его значением **true**.

```
System.out.println( x++ == y++ || x++ != y++ && x++ == y++);
System.out.println("x= " + x + ", y= " + y);
```

Результат:

true

x=1, y=1.

Если же операция состоит из битовых операторов, то вычисляются все без исключения операнды.

```
System.out.println( x++ == y++ | x++ != y++ & x++ == y++);
System.out.println("x= " + x + ", y= " + y);
```

Результат:

true

x=3, y=3.

К логическим операторам относится также оператор определения принадлежности типа **instanceof** и тернарный оператор «?:» (if-then-else).

Тернарный оператор «?:» используется в выражениях вида:

```
boolean_value ? expression_one : expression_two;
type value = boolean_value ? expression_one : expression_two;
```

Если *boolean_value* равно **true**, вычисляется значение выражения *expression_one*, и оно становится результатом всего оператора, иначе результатом является значение выражения *expression_two*, которое может быть присвоено значению *value*. Например:

```
int determineBonus(int numberProducts) {
    int bonus = numberProducts > 7 ? 10 : 0;
    return bonus;
}
```

Если число купленных предметов более семи, то клиент получает **bonus** в размере десятипроцентной скидки, в противном случае скидка не предоставляется.

Такое применение делает оператор простым для понимания так же, как и следующий вариант:

```
int result = experience > requirements ? acceptToProject() : learnMore();
```

где оба метода должны возвращать значение типа **int**.

Оператор **instanceof** возвращает значение **true**, если объект является экземпляром данного типа. Например, для иерархии наследования:

```
/* # 2 # учебные курсы в учебном заведении: иерархия */
```

```
class Course {
    void method(){}
}
class BaseCourse extends Course { /* */}
class FreeCourse extends BaseCourse { /* */}
class OptionalCourse extends Course { /* */}
```

Применение оператора **instanceof** может выглядеть следующим образом при вызове метода **doAction(Course course)**:

```
void doAction(Course course) {
    if (course instanceof BaseCourse) {
        // for BaseCourse and FreeCourse
        BaseCourse base = (BaseCourse)course;
        base.method();
    } else if (course instanceof OptionalCourse) {
        // for OptionalCourse
        OptionalCourse optional = (OptionalCourse)course;
        optional.method();
    } else {
        // for Course or null
    }
}
```

Результатом действия оператора **instanceof** будет истина, если объект является объектом данного класса или одного из его подклассов, но не наоборот. Проверка на принадлежность объекта к классу **Object** всегда даст истину, поскольку любой класс является наследником класса **Object**. Результат применения этого оператора по отношению к ссылке на значение **null** — всегда ложь, потому что **null** нельзя причислить к какому-либо классу. В то же время литерал **null** можно передавать в методы по ссылке на любой объектный тип и использовать в качестве возвращаемого значения. Базовому типу значение **null** присвоить нельзя так же, как использовать ссылку на базовый тип в операторе **instanceof**.

В Java 14 в операторе **instanceof** были объединены проверка на тип объекта и его преобразование к этому типу. Тогда метод **doAction(Course course)** можно переписать в виде:

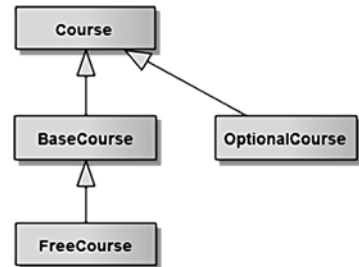


Рис. 2.5. Иерархия наследования

```
void doAction(Course course) {
    if (course instanceof BaseCourse base) {
        base.method();
    } else if (course instanceof OptionalCourse optional) {
        optional.method();
    } else {
        // code
    }
}
```

где **method()** — полиморфный метод класса **Course**, переопределенный в под-классах.

Классы-оболочки

Кроме базовых типов данных, в языке Java используются соответствующие классы-оболочки (wrapper-классы) из пакета **java.lang**: **Boolean**, **Character**, **Integer**, **Byte**, **Short**, **Long**, **Float**, **Double**. Объекты этих классов хранят те же значения, что и соответствующие им базовые типы.

Объект любого из этих классов представляет собой экземпляр класса в динамической памяти, в котором хранится его неизменяемое значение. Значения базовых типов хранятся в стеке и не являются объектами.

Классы, соответствующие числовым базовым типам, находятся в библиотеке **java.lang**, являются наследниками абстрактного класса **Number** и реализуют

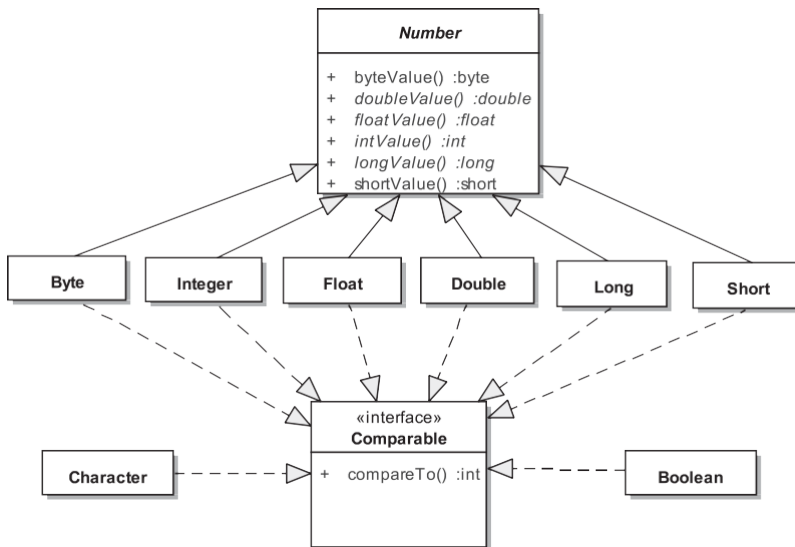


Рис. 2.6. Иерархия классов-оболочек

интерфейс **Comparable<T>**. Этот интерфейс определяет возможность сравнения объектов одного типа между собой с помощью метода **int compareTo(T ob)**. Объекты классов-оболочек по умолчанию получают значение **null**. Как было сказано выше, базовые типы данных по умолчанию инициализируются нулями.

Создаются экземпляры числовых, символьного и булевского, классов с помощью статических методов **valueOf(String s)**, **parseType(String s)** и **decode(String s)** с параметрами типа **String** или статического метода-фабрики **valueOf(type v)**, где **type** параметр базового типа.

```
Integer value = Integer.valueOf(42);
```

В следующем примере рассмотрено три стандартных способа создания объекта **Integer** на основе преобразования строки в число:

```
/* # 3 # преобразование строки в целое число # */
```

```
String arg = "42";
try {
    int value1 = Integer.parseInt(arg);
    Integer value2 = Integer.valueOf(arg);
    Integer value3 = Integer.decode(arg);
} catch (NumberFormatException e) {
    System.err.println("invalid number format: " + arg + " : " + e);
}
```

Использование конструктора для создания объектов не применяется.

Для устойчивой работы приложения все операции по преобразованию строки в типизированные значения желательно заключать в блок **try-catch** для перехвата и обработки возможного исключения.

Метод **decode(String s)** преобразует строки, заданные в виде восьмеричных или шеснадцатеричных литералов в соответствующие им десятичные числа.

```
Integer.decode("0x22");
Integer.decode("042");
```

У некоторых из приведенных методов также есть перегруженные версии при использовании разных систем счисления и представления чисел.

Методы **valueOf(String s, int radix)** и **parseInt(String s, int radix)** преобразуют число в строке и в указанной системе счисления в десятичное число, например:

```
Integer.valueOf("42", 8);
Integer.valueOf("100010", 2);
Integer.valueOf("22", 16);
```

Результатом преобразования во всех случаях будет число **34**.

Объект класса-оболочки может быть преобразован к базовому типу обычным присваиванием или методом **typeValue()**, где **type** один из базовых типов данных.

Объект класса **Boolean** также следует создавать с помощью статических методов:

```
boolean bool1 = Boolean.parseBoolean("true"); // true
Boolean bool2 = Boolean.valueOf("Hello"); // false
Boolean bool3 = Boolean.valueOf(true);
```

При этом если в качестве параметра выступает строка со словом **true**, записанном символами в любом регистре, то результатом будет всегда истинное значение. Строка с любым другим набором символов всегда при преобразовании даст значение **false**.

Булевский объект также задается константами:

```
Boolean bool4 = Boolean.TRUE;
Boolean bool5 = Boolean.FALSE;
```

В классе **Boolean** объявлены три статических метода **Boolean.logicalAnd(boolean value1, boolean value2)**, **Boolean.logicalOr(boolean value1, boolean value2)**, **Boolean.logicalXor(boolean value1, boolean value2)**, заменяющие базовые логические операции. Теперь вместо выражения *boolean_expr1 && boolean_expr2* возможна альтернатива **Boolean.logicalAnd(boolean_expr1, boolean_expr2)**.

Класс **Character** не является подклассом **Number**, и этому классу нет необходимости поддерживать интерфейс классов, предназначенных для хранения результатов арифметических операций. Вместо этого класс **Character** имеет большое число специфических методов для работы с символами и их представлением. Ниже приведены способы преобразования символа в число, если символ является числовым или представлен кодом числа.

```
char ch0 = '7';
Character ch1 = Character.valueOf(ch0);
int value0 = Character.digit(ch0, 10); // Character into int, value0 will be 7
int value1 = Character.digit(55, 10); // CodePoint into int, value1 will be 7
int value2 = Character.getNumericValue(ch1);
```

Обратное преобразование из типизированного значения (в частности **int**) в строку можно выполнить следующими способами:

```
int value = 42;
String good1 = Integer.toString(value); // good
String good2 = String.valueOf(value); // good
String bad = "" + value; // very bad
```

В Java определен процесс автоматической инкапсуляции данных базовых типов в соответствующие объекты оболочки и обратно (автоупаковка/автораспаковка). При этом нет необходимости в явном создании соответствующего объекта с использованием методов:

```
Integer iob = 420; // autoboxing
Integer i = 42; // integer cache from -128 to 127
```

При инициализации объекта класса-оболочки значением базового типа преобразование типов в некоторых ситуациях необходимо указывать явно, то есть код

```
Float f = 42; // it will be correct (float)42 or 42F instead of 42
```

вызывает ошибку компиляции.

С другой стороны, однако справедливо:

```
Float f1 = Float.valueOf(42);
Float f0 = Float.valueOf("42");// or valueOf("42f") or valueOf("42.0")
```

Автораспаковка — процесс извлечения из объекта-оболочки значения базового типа. Вызовы методов **intValue()**, **doubleValue()** и им подобных для преобразования объектов в значения базовых типов становятся излишними.

Допускается участие объектов оболочек в арифметических операциях, однако не следует этим злоупотреблять, поскольку упаковка/распаковка является ресурсоемким процессом:

```
// autoboxing & unboxing:
Integer i = 420; // autoboxing
++i; // unboxing + operation + autoboxing
int j = i; // unboxing
```

Однако следующий код генерирует исключительную ситуацию **NullPointerException** при попытке присвоить базовому типу значение **null** объекта класса **Integer**. Литерал **null** — не объект, поэтому и не может быть преобразован к литералу «0»:

```
Integer j = null; // the object is not instantiated ! This is not zero!
int i = j; // generating an exception at runtime
```

Сравнение объектов классов оболочек между собой происходит по ссылкам, как и для других объектных типов. Для сравнения значений объектов следует использовать метод **equals()**.

```
int i = 128;
Integer a = i; // autoboxing
Integer b = i;
System.out.println("a==i " + (a == i)); // true - unboxing and comparing values
System.out.println("b==i " + (b == i)); // true
System.out.println("a==b " + (a == b)); // false - references to different objects
System.out.println("equals ->" + a.equals(i) + b.equals(i) + a.equals(b)); // true true true
```

Метод **equals()** сравнивает не значения объектных ссылок, а значения объектов, на которые установлены эти ссылки. Поэтому вызов **a.equals(b)** возвращает значение **true**.

Если в приведенном выше примере значению **int i** будет присвоено значение в диапазоне от -128 до 127, то операции сравнения объектных и базовых ссылок будут давать результат **true** во всех случаях. Литералы указанного

диапазона кэшируются по умолчанию, для чего используется класс **IntegerCache**. Дело в том, что при создании кода очень часто применяются литералы небольших значений типа -1, 0, 1, 8, 16, и чтобы не увеличивать затраты памяти на создание новых объектов, применяется кэширование. Диапазон кэширования может быть расширен параметрами виртуальной машины.

Значение базового типа может быть передано в метод **equals()**. Однако ссылка на базовый тип не может вызывать методы:

```
i.equals(a); // compile error
```

При автоупаковке значения базового типа возможны ситуации с появлением некорректных значений и непроверяемых ошибок.

Существуют два класса для работы с высокоточной арифметикой — **java.math.BigInteger** и **java.math.BigDecimal**, которые поддерживают целые числа и числа с фиксированной точкой произвольной длины. При вычислениях для типов **double** и **float** возникает проблема точности, так как для этих типов существует понятие «машинный ноль», это означает, что при вычислениях после 15-го разряда для **double** и после 6-го для **float** появляются «лишние» значащие цифры. Для устранения таких побочных эффектов следует использовать **BigDecimal**:

```
float res = 0.4f - 0.3f;
BigDecimal big1 = new BigDecimal("0.4");
BigDecimal big2 = new BigDecimal("0.3");
BigDecimal bigRes = big1.subtract(big2, MathContext.DECIMAL32);
```

Результатом выполнения будет:

0.099999994

0.1

Константы класса **MathContext** определяют число знаков после запятой при округленных вычислениях.

Для операций сложения, умножения и деления применяются методы **add()**, **multiply()** и **divide()**, которые представлены в нескольких перегруженных версиях.

Проблема «машинного нуля» проявляется и при операциях сравнения, а именно результатом выполнения следующего оператора

```
boolean res1 = 1.00000001f == 1.00000002f;
```

так же, как и

```
boolean res2 = 1 == 1f / 3 * 3;
```

будет **true**.

Оператор условного перехода

Оператор условного перехода **if** имеет следующий синтаксис:

```
if (boolean_value) {
    // line 1: main success scenario
} else {
    // line 2: scenario variation
}
```

Если выражение *boolean_value* принимает значение **true**, то выполняется группа операторов *line 1*, иначе — группа операторов *line 2*.

```
if (boolean_value) {
    System.out.println("value is Valid");
} else {
    System.out.println("value is Broken");
}
```

В отличие от приведенного варианта

```
if (boolean_value) {
    System.out.println("value is Valid");
}
System.out.println("More Opportunities"); // this code will always be executed
```

оператор **else** может отсутствовать, в этом случае операторы, расположенные после окончания оператора **if**, выполняются вне зависимости от значения булевого выражения оператора **if**. Фигурные скобки обозначают составной оператор.

Если после оператора **if** следует только одна инструкция для выполнения, то фигурные скобки для выделения блока кода можно опустить, но делать это нежелательно.

```
if (boolean_value)
    ifCode(); // depends on the condition
restOfCode(); // this code will always be executed
```

При корректировке кода может понадобиться добавить строку кода, например, вызов метода **checkRole()** перед вызовом **ifCode()**. Поведение программы резко изменится и не будет соответствовать идее, что метод **ifCode()** вызывается только при истинности условия *boolean_value*.

```
if (boolean_value)
    checkRole(); // depends on the condition
    ifCode(); // this code will always be executed !!!
    restOfCode();
```

Теперь указанный метод будет вызываться независимо от выполнения условного оператора. Во избежание таких нелепых ошибок следует использовать

фигурные скобки даже при одной исполняемой строке кода. К тому же код будет выглядеть более понятным, что немаловажно.

```
if (boolean_value) {
    checkRole();
    ifCode();
}
restOfCode();
```

Этих же правил следует придерживаться и для оформления циклов.

Если в операторе **if-else** выполняются действия в одну строку кода, то более выгодно заменить его на тернарный оператор. Тогда вместо:

```
int value = 1;
if (value <= 0) {
    value++;
} else {
    value--;
}
```

можно записать

```
int value = 1;
value = value <= 0 ? value++ : value--;
```

данная запись более компактна и легче читается.

Допустимо также использование конструкции «лесенки» **if-else if-else**.

```
if (value >= 1 && value <= 4) {
    // more code 1
} else if (value == 9) {
    // more code 2
} else if (value == 0) {
    // more code 3
} else {
    // more code 4
}
```

Операторы условного перехода следует применять так, чтобы нормальный ход выполнения программы был очевиден. После **if** следует располагать код, удовлетворяющий нормальной работе алгоритма, после **else** — побочные и исключительные варианты. Используется и обратный порядок в случае **if без else**, например, при проверке значения ссылки на **null**. Более выгодный вариант использования:

```
if (obj == null) {
    throw new IllegalArgumentException();
}

/* base algorithm - begin
...
...
base algorithm - end */
```

Следующий вариант менее выгодный с точки зрения чтения кода, так как приходится искать где-то внизу блок **else** с альтернативой появления некорректного для разработчика значения ссылки:

```
if (obj != null) {
    /* base algorithm - begin
       base algorithm - end */
} else {
    throw new IllegalArgumentException();
}
```

Оператор выбора

«Лесенку» **if-else-if** рекомендуется заменить на более удобный в данной ситуации оператор множественного выбора **switch**. Оператор **switch** в общем случае выглядит:

```
switch(value) {
    case const1:
        // code 1
        break; // not required
        ...
    case constN:
        // code N
        break; // not required
    default:
        // code
}
```

А в случае замены «лесенки»:

```
switch (value) {
    case 1:
    case 2:
    case 3:
    case 4:
        // more code 1
        break;
    case 9:
        // more code 2
        break;
    case 0:
        // more code 3
        break;
    default:
        // more code 4
}
```

При совпадении значения **value** со значением **val1** выполняется следующий за ним вариант. Затем, если отсутствует оператор **break**, выполняются подряд

все блоки операторов до тех пор, пока не встретится оператор **break**. Если значение **value** не совпадает ни с одним из значений в **case**, то выполняется блок **default**. Создание качественного кода подразумевает обязательное использование блока **default**, что позволяет снизить число потенциальных ошибок приложения при альтернативном течении сценария. Блок **default** можно размещать и между блоками **case**, но это плохой способ его применения. Данный блок всегда должен стоять после всех **case**, и должен быть не пустым. Значения **const1,..., constN** должны быть константами и могут иметь значения типа **int**, **byte**, **short**, **char**, **enum** или **String**:

```
/* # 4 # тип String в операторе switch */
```

```
public int defineLevel(String role) {
    int level;
    switch (role) { // or role.toLowerCase()
        case "guest":
            level = 0;
            break;
        case "client":
            level = 1;
            break;
        case "moderator":
            level = 2;
            break;
        case "admin":
            level = 3;
            break;
        default:
            throw new IllegalArgumentException("non-authentic role = " + role);
    }
    return level;
}
```

Параметр типа **String** разумно применять в случае одноразового использования набора литералов из списка **case**. При многократном использовании этого набора литералов следует задуматься об организации класса-перечисления, что позволит избежать ошибок «по невнимательности» при частом включении в код обработки набора информации, содержащего строковые литералы.

Аналогично для оператора **switch** нормальное исполнение алгоритма следует располагать в инструкциях **case**, а именно, наиболее вероятные варианты размещаются раньше остальных, альтернативные или для значений по умолчанию — в инструкции **default**. Цепочки вызовов **if-else-if** и **switch-case** иногда при грамотном проектировании можно заменить вызовом полиморфного метода.

В Java 14 добавлены новые формы оператора **switch**:

- не требующие оператора **break**,
- позволяющие объединять повторяющиеся метки через запятую,
- применяющие лямбда-выражения,
- допускающие применение в форме метода.

Оператор **switch** может возвращать значение. Естественно, все **case** должны возвращать значение одного типа, либо приводимые к нему. Вместо символа «:» и **return** проще использовать лямбда-выражения, о которых речь пойдет в главе «Интерфейсы и аннотации»:

```
/* # 5 # лямбда в операторе switch */
```

```
public int defineLevel(String role) {
    return switch (role) {
        case "guest" -> 0;
        case "client" -> 1;
        case "moderator" -> 2;
        case "admin" -> 3;
        default -> {
            System.out.println("non-authentic role = " + role);
            yield -1;
        }
    };
}
```

При использовании в блоке кода внутри **switch** новое ключевое слово **yield** возвращает значение из блока кода.

Возвращаемое значение может присваиваться результату прямым присваиванием:

```
/* # 6 # прямое присваивание оператора switch */
```

```
public int defineLevel(String role) {
    var result = switch (role) {
        case "guest" -> 0;
        case "client" -> 1;
        case "moderator" -> 2;
        case "admin" -> 3;
        default -> {
            System.out.println("non-authentic role = " + role);
            yield -1;
        }
    };
    return result;
}
```

Вместо нескольких инструкций **case** появилась возможность перечислять их через запятую:

```
/* # 7 # список значений в операторе switch */
```

```
int value = 1;
switch (value) {
    case 1, 2, 3, 4 -> System.out.println("1, 2, 3 or 4");
    case 777 -> System.out.println("Range: " + value);
    case 0 -> System.out.println("0");
    default -> System.out.println("Default");
}
```

Возвращаемое значение оператора **switch** можно также сразу передавать как параметр метода:

```
/* # 8 # оператор switch как метод */
```

```
System.out.println(
    switch (value) {
        case 2, 3, 4 -> "2,3 or 4";
        case 777 -> "Range: " + value;
        case 0 -> "0";
        default -> "Default";
    });
```

Циклы

Цикл представляет собой последовательность операций, которые могут выполняться более одного раза и завершаться при выполнении некоторого граничного условия.

В Java существует четыре вида циклов. Основные три:

Цикл с предусловием — **while** (*boolean_value*) { */* code */* }

Цикл с постусловием — **do** { */* code */* } **while** (*boolean_value*);

Цикл с параметром — **for** (*expression_1; boolean_value; expression_3*) { */* code */* }

Циклы выполняются до тех пор, пока булевское выражение *boolean_value* равно **true**.

Тело цикла **do\while** будет выполнено минимум один раз:

```
int value = 42;
do {
    value++;
} while (value < 40);
```

Условие перехода на следующую итерацию проверяется в конце цикла.

Если заменить его циклом **while**:

```
int value = 42;
while(value < 40) {
    value++;
}
```

то тело такого цикла не будет выполнено ни разу.

При разработке этих видов циклов следует аккуратно проверять код, чтобы цикл не «сорвался» в бесконечный цикл.

В цикле с параметром, по традиции, *expression_1* — начальное выражение, *boolean_value* — условие выполнения цикла, *expression_3* — выражение, выполняемое в конце итерации цикла (как правило, это изменение начального значения).

```
for (int i = 0; i <= 10; i++) {
    // code
}
```

Цикл может использовать, если необходимо «движение» сразу по нескольким переменным цикла:

```
for (int i = 0, j = 20; i <= j + 2; i++, j--) {
    // code
}
```

Любой блок или даже все блоки цикла **for** могут быть пустыми,

```
for (;;) {
    // code
}
```

но в этом случае цикл станет бесконечным, как и цикл

```
while(true) {}
```

В таких случаях необходимо предусмотреть выход из цикла, например, применением оператора **break**.

В Java 5 введен еще один цикл полного перебора, упрощающий доступ к массивам и коллекциям:

```
for(Type nameObject : nameIterateObject) { /* code */ }
```

При работе с массивами и коллекциями с помощью данного цикла можно получить гарантированный доступ ко всем их элементам без использования индексов.

```
int[] arr = {1, 3, 5, 7, 9};
for (int elem : arr) {
    System.out.printf("%d ", elem);
}
```

Однако цикл с параметром позволяет заменять значение элемента итерируемого массива на новое:

```
String[] strings = {"a", "b", "c"};
for (int i = 0; i < strings.length; i++) {
    strings[i] += "java";
}
System.out.println(Arrays.toString(strings));
```

В тоже время для цикла полного перебора:

```
for (String string : strings) {
    string += "se";
}
System.out.println(Arrays.toString(strings));
```

Изменять значения элементов массива или любого другого итерируемого объекта с помощью оператора «=» в таком цикле невозможно. Но можно вызывать методы, изменяющие внутреннее состояние объекта. Данный цикл способен обрабатывать коллекции и единичные объекты, если его классы реализуют интерфейсы **Iterable** или **Iterator**.

Оператор **break** при использовании в теле цикла прекращает работу цикла. Управление передается оператору, следующему в коде за оператором цикла. Оператор **break** рекомендуется использовать *только* для циклов **while** или **do\while**.

Оператор **continue** прекращает выполнение текущей итерации цикла и переходит к проверке условий и при их выполнении к следующей итерации цикла.

Некоторые рекомендации при проектировании циклов:

- цикл **for** следует использовать при необходимости выполнения алгоритма строго определенное количество раз. Циклы **while** и **do\while** используются в случаях, когда неизвестно точное число итераций для достижения результата, например, поиск необходимого значения в массиве или коллекции. Цикл **while(true){}** эффективно применяется в многопоточных приложениях для организации бесконечных циклов;
- для цикла **for** не рекомендуется в теле цикла изменять индекс цикла;
- в циклах **for** не следует использовать оператор **break**;
- для индексов следует применять осмысленные имена;
- циклы не должны быть слишком длинными. Такой цикл претендует на выделение в отдельный метод;
- вложенность циклов не должна превышать трех.

В языке Java расширились возможности оператора прерывания цикла **break** и оператора прерывания итерации цикла **continue**, которые можно использовать с меткой, например:

```
int j = -3;
OUT: while(true) {
    for(;;) {
        while (j < 10) {
            if (j == 0) {
                break OUT;
            } else {
                j++;
                System.out.printf("%d ", j);
            }
        }
    }
}
```

```

    }
    }
}
System.out.print("end");

```

Здесь оператор **break** разрывает цикл, помеченный меткой **OUT**. Тем самым решается вопрос об отсутствии необходимости в операторе **goto** для выхода из самого внутреннего из вложенных циклов. Такое использование является плохим примером проектирования циклов и на практике никогда не встречается.

Массивы

Массив в Java представляет собой класс с типом `[]`, при этом имя объекта класса массива является объектной ссылкой на динамическую память, в которой хранятся элементы массива. Элементами массива, в свою очередь, могут быть значения базового типа или объекты. Элементы массива проиндексированы, индексирование элементов начинается с нуля. Для объявления ссылки на массив можно записать пустые квадратные скобки после имени типа, например: `int a[]`. Аналогичный результат получится при записи `int []a`.

Все массивы динамические. Существуют два способа создания массива: с помощью оператора **new**

```

int[] arInt = new int[11]; // 11 zeros
String str[] = new String[7]; // 7 null

```

или с помощью прямой инициализации присваиванием значений элементам массива в фигурных скобках

```

int[] arInt1 = {5, 7, 9, -5, 6}; // short declaration of the array
int[] arInt2 = new int[] {5, 7, 9, -5, 6}; // is identical to the previous one

```

Значения элементов неинициализированного массива, для которого выделена память, устанавливаются в значения по умолчанию для массива базового типа или **null** для массива объектных ссылок.

Ссылка на самый верхний в иерархии объект класса **Object** может быть проинициализирована массивом любого типа и любой размерности.

```

Object arObj1 = new float[5]; // the array is an object
Object arObj2 = new int[105]; // the array is an object

```

Обратное действие требует обязательного приведения типов и корректно только в случае, если тип значений массива и тип ссылки совпадают.

```

float[] arRefFloat = (float[]) arObj1;

```

Если же ссылка на массив объявлена с указанием типа, то она может содержать данные только указанного типа и присваиваться другой ссылке такого же типа. Приведение типов в этом случае вызывает ошибку компиляции.

Присваивание **arInt1=arInt** приведет к утрате элементов массива **arInt1**, в итоге две ссылки будут установлены на один массив **arInt**, то есть будут ссылаться на один и тот же участок памяти.

Массив представляет собой безопасный объект, поскольку все ссылки на элементы инициализируются при создании и доступ к элементам невозможен из-за пределов границ массива. Размерность массива хранится в его **final**-поле **length** типа **int**.

В памяти объект массива занимает значительно больше места, чем сумма байт его элементов, а именно:

- каждому массиву выделяется дополнительная память (overhead) размером 8 байт;
- массиву предоставляется еще один overhead в 4 байта, в которых хранится размер **length** типа **int**;
- для ссылки выделяется 4 байта;
- размер всех объектов выравнивается до 8 байт, если их тип данных занимает меньший объем памяти.

В итоге массив массивов занимает в памяти в 2 и более раз больше места, чем сумма байт его типов данных. Многомерных массивов в Java не существует, но можно объявлять массив ссылок на массивы. Для экономии памяти при объявлении таких массивов следует размеры объявлять по возрастанию. Например:

```
byte array[2][42];
```

вместо

```
byte array[42][2];
```

Для задания начальных значений массива массивов существует специальная форма инициализатора, например:

```
int[][] matrix = { {1}, {2, 3}, {4, 5, 6}, {7, 42, 9, 8} };
```

Первый индекс указывает на порядковый номер массива, например, **matrix[3]** [1] указывает на второй элемент четвертого массива, а именно, на значение **42**.

Массивы объектов внешне не отличаются от массивов базовых типов. В действительности они представляют собой массивы ссылок, проинициализированных по умолчанию значением **null**. Выделение памяти для хранения объектов массива должно производиться для каждой объектной ссылки в отдельности.

В пакете **java.util** находится утилитный класс **Arrays**, в котором объявлен набор статических методов обработки массива, а именно для поиска, заполнения, сравнения, сортировки, преобразования. Некоторые из них:

String toString(parameters) — преобразование массива в строку;

int binarySearch(parameters) — бинарный поиск значения в массивах примитивных и объектных типов. Возвращает позицию первого совпадения;

boolean equals(parameters) — сравнение двух массивов на эквивалентность;
int compare(parameters) — сравнение двух массивов на больше, меньше или равно. Возвращает 1 или -1 на первом же индексе, где найдены отличные друг от друга значения. Знак зависит от того, элемент первого массива больше соответствующего элемента, второго массива или наоборот;

void sort(parameters) — сортировка массива или его части с использованием интерфейса **Comparator** и без него.

Практический пример работы этих методов:

```
/* # 9 # методы класса Arrays # LearnMainArrays */
package by.bsu.learn.array;
import java.util.Arrays;
public class LearnMainArrays {
    public static void main(String[] args) {
        int[] arr1 = {1, 9, 3, 8, 7};
        int[] arr2 = {1, 9, 3, 11, 5};
        System.out.println(Arrays.equals(arr1, arr2)); // false
        System.out.println(Arrays.compare(arr1, arr2)); // arr1[3]<arr2[3]. Print: -1
        System.out.println(Arrays.binarySearch(arr1, 3)); // arr1[2] = 3
        Arrays.sort(arr1);
        System.out.println(Arrays.toString(arr1)); // after sorting [1, 3, 7, 8, 9]
    }
}
```

Вопросы к главе 2

1. Какие примитивные типы Java существуют, как создать переменные примитивных типов?
2. Объяснить процедуру, по которой переменные примитивных типов передаются в методы как параметры.
3. Каков размер примитивных типов? Как размер примитивных типов зависит от разрядности платформы?
4. Что такое преобразование (приведение) типов и зачем оно необходимо? Какие примитивные типы не приводятся ни к какому другому типу.
5. Объяснить, что такое явное и неявное приведение типов, привести примеры, когда такое преобразование имеет место.
6. Что такое литералы в Java-программе? Дать описание классификации литералов.
7. Как записываются литералы различных видов и типов в Java-программе?
8. Как осуществляется работа с типами при вычислении арифметических выражений в Java?
9. Что такое классы-оболочки, для чего они предназначены? Что значит: объект класса оболочки — константный объект.

10. Объяснить разницу между примитивными и ссылочными типами данных. Пояснить существующие различия, при передаче параметров примитивных и ссылочных типов в методы. Объяснить, как константные объекты ведут себя при передаче в метод.
11. Перечислить известные арифметические, логические и битовые операторы, определить случаи их употребления. Что такое приоритет оператора, как определить, в какой последовательности будут выполняться операции в выражении, если несколько из них имеют одинаковый приоритет.
12. Какие правила выполнения операций с плавающей точкой в Java? Как определить, что результатом вычисления стала бесконечность или «нечисло»?
13. Что такое autoboxing и unboxing? Принцип действия на примерах.
14. Что такое **var**? Можно ли переменной или методу дать имя **var**? Достоинства и недостатки.
15. Объяснить работу операторов **if**, **switch**, **while**, **do-while**, **for**, **for-each**. Написать корректные примеры работы этих операторов.
16. Объяснить работу оператора **instanceof**. Что будет результатом работы оператора, если слева от него будет стоять ссылка, равная **null**?
17. Дать определение массиву. Как осуществляется индексация элементов массива. Как необходимо обращаться к *i*-му элементу массива?
18. Привести способы объявления и инициализации одномерных и двумерных массивов примитивных и ссылочных типов. Чем отличаются массивы примитивных и ссылочных типов?
19. Объяснить, что представляет собой двумерный массив в Java, что такое «массив массивов». Как узнать количество строк и количество элементов в каждой строке для «массива массивов»?
20. Объяснить ситуации, когда в коде Java могут возникнуть следующие исключительные ситуации **java.lang.ArrayIndexOutOfBoundsException** и **java.lang.ArrayStoreException**.

Задания к главе 2

Вариант А

В приведенных ниже заданиях необходимо вывести внизу фамилию разработчика, дату и время получения задания, а также дату и время сдачи задания. Добавить комментарии в программы в виде */** комментарий */*. В заданиях на числа объект можно создавать в виде массива символов.

Ввести *n* чисел с консоли.

1. Найти самое короткое и самое длинное число. Вывести найденные числа и их длину.
2. Упорядочить и вывести числа в порядке возрастания (убывания) значений их длины.

3. Вывести на консоль те числа, длина которых меньше (больше) средней, а также длину.
4. Найти число, в котором число различных цифр минимально. Если таких чисел несколько, найти первое из них.
5. Найти количество чисел, содержащих только четные цифры, а среди них — количество чисел с равным числом четных и нечетных цифр.
6. Найти число, цифры в котором идут в строгом порядке возрастания. Если таких чисел несколько, найти первое из них.
7. Найти число, состоящее только из различных цифр. Если таких чисел несколько, найти первое из них.
8. Среди чисел найти число-палиндром. Если таких чисел больше одного, найти второе.
9. Найти корни квадратного уравнения. Параметры уравнения передавать с командной строкой.

Вариант В

1. Вывести на экран таблицу умножения.
2. Вывести элементы массива в обратном порядке.
3. Определить принадлежность некоторого значения k интервалам (n, m) , $[n, m)$, $(n, m]$, $[n, m]$.
4. Вывести на экран все числа от 1 до 100, которые делятся на 3 без остатка.
5. Сколько значащих нулей в двоичной записи числа 129?
6. В системе счисления с некоторым основанием десятичное число 81 записывается в виде 100. Найти это основание.
7. Написать код программы, которая бы переводила числа из десятичной системы счисления в любую другую.
8. Написать код программы, которая бы переводила числа одной любой системы счисления в любую другую.
9. Ввести число от 1 до 12. Вывести на консоль название месяца, соответствующего данному числу. Осуществить проверку корректности ввода чисел.

Вариант С

Ввести с консоли n -размерность матрицы $a[n][n]$. Задать значения элементов матрицы в интервале значений от $-n$ до n с помощью генератора случайных чисел.

1. Упорядочить строки (столбцы) матрицы в порядке возрастания значений элементов k -го столбца (строки).
2. Выполнить циклический сдвиг заданной матрицы на k позиций вправо (влево, вверх, вниз).
3. Найти и вывести наибольшее число возрастающих\убывающих элементов матрицы, идущих подряд.

4. Найти сумму элементов матрицы, расположенных между первым и вторым положительными элементами каждой строки.
5. Вывести числа от 1 до k в виде матрицы $N \times N$ слева направо и сверху вниз.
6. Округлить все элементы матрицы до целого числа.
7. Повернуть матрицу на 90, 180 или 270 градусов против часовой стрелки.
8. Вычислить определитель матрицы.
9. Построить матрицу, вычитая из элементов каждой строки матрицы ее среднее арифметическое.
10. Найти максимальный элемент(ы) в матрице и удалить из матрицы все строки и столбцы, его содержащие.
11. Уплотнить матрицу, удаляя из нее строки и столбцы, заполненные нулями.
12. В матрице найти минимальный элемент и переместить его на место заданного элемента путем перестановки строк и столбцов.
13. Преобразовать строки матрицы таким образом, чтобы элементы, равные нулю, располагались после всех остальных.
14. Найти количество всех седловых точек матрицы (матрица A имеет седловую точку $A(i, j)$, если $A(i, j)$ является минимальным элементом в i -й строке и максимальным в j -м столбце).
15. Перестроить матрицу, переставляя в ней строки так, чтобы сумма элементов в строках полученной матрицы возрастала.
16. Найти число локальных минимумов. Соседями элемента матрицы назовем элементы, имеющие с ним общую сторону или угол. Элемент матрицы называется локальным минимумом, если он строго меньше всех своих соседей.
17. Найти наименьший среди локальных максимумов. Элемент матрицы называется локальным максимумом, если он строго больше всех своих соседей.
18. Перестроить заданную матрицу, переставляя в ней столбцы так, чтобы значения их характеристик убывали. Характеристикой столбца прямоугольной матрицы называется сумма модулей его элементов.
19. Путем перестановки элементов квадратной вещественной матрицы добиться того, чтобы ее максимальный элемент находился в левом верхнем углу, следующий по величине — в позиции (2, 2), следующий по величине — в позиции (3, 3) и т.д., заполнив таким образом всю главную диагональ.

Тестовые задания к главе 2

Вопрос 2.1.

Какое значение примет переменная *value* в результате выполнения фрагмента кода? (выбрать один)

```
boolean flag = true;
int value;
if (flag) {
```

```
value = flag ? 1 : 2;
} else {
    value = flag ? 3 : 4;
}
System.out.print(value);
```

- a) 1
- b) 2
- c) 3
- d) 4

Вопрос 2.2.

Какое значение будет выведено в консоль в результате выполнения цикла? (выберите один)

```
for (int i = 0, j = 0; i < 3 && j <= 0; i++, j--) {
    if (i != j) {
        continue;
    }
    System.out.print (i + " " + j);
}
```

- a) 0 0
- b) 1 -1
- c) 2 -2
- d) Ничего не будет выведено

Вопрос 2.3.

Какое из следующих объявлений массива корректно? (выбрать один)

- a) String array [] = new String {"j" "s" "e"};
- b) String array [] = { "j" " " v" "m"};
- c) String array = {"j", "d", "k"};
- d) String array [] = {"j", "r", "e"};

Вопрос 2.4.

Что будет результатом компиляции и запуска следующего кода? (выбрать один)

```
public class Quest {
    public static void main(String args[]) {
        char[] array = {'a', 'b', 'c'};
        for (char element : array) {
            element += element;
        }
        System.out.println(Arrays.toString(array));
    }
}
```

- a) [a, b, c]
- b) [b, c, d]
- c) [aa, bb, cc]
- d) [bb, cc, dd]
- e) compilation fails

Вопрос 2.5.

Даны объявления строк:

```
String s1 = new String("Java12");
String s2 = new String("12");
String s3 = new String();
```

Какая из следующих операций корректна? (выбрать один)

- a) `s1 + s2`
- b) `s1 - s2`
- c) `s2 || s3`
- d) `s2 | s3`
- e) `s1 ++ s3`

Вопрос 2.6.

Какие из ключевых слов могут быть использованы при объявлении метода класса? (выбрать два)

- a) `final`
- b) `volatile`
- c) `abstract`
- d) `enum`
- e) `default`

Вопрос 2.7.

Какие из следующих имен переменных корректны? (выбрать три)

- a) `int j1;`
- b) `int _j2;`
- c) `int j_3;`
- d) `int #j4;`
- e) `int @j5;`

Вопрос 2.8.

Какое из предложенных значений представляет значение по умолчанию для типа **boolean**? (выбрать один)

- a) `true`
- b) `false`
- c) `Boolean.TRUE`
- d) `Boolean.FALSE`

КЛАССЫ И МЕТОДЫ

Любая действующая программа устарела.

Первый Закон Программирования

Класс соответствует типу данных как описание совокупности объектов предметной области приложения с общими атрибутами, методами, отношениями и семантикой.

Объект — обладающий именем набор данных (полей и свойств объекта) и методов, имеющих доступ к ним. Имя объекта используется для работы с его полями и методами.

Любой объект относится к определенному классу. В классе дается обобщенное описание некоторого набора родственных объектов.

Объект — конкретный экземпляр класса.

Объект — логическая модель реальной сущности.

Объектно-ориентированное программирование — это методология программирования, основанная на использовании классов и объектов.

В ООП используются принципы:

- инкапсуляции;
- наследования;
- полиморфизма, в частности, «позднего связывания».

Инкапсуляция (*encapsulation*) — принцип, объединяющий данные и код, защищающий данные от прямого внешнего доступа и неправильного использования. Другими словами, доступ к данным класса должен осуществляться, в соответствии с принципами инкапсуляции, только посредством методов этого же класса.

Наследование (*inheritance*) — процесс, посредством которого один класс может наследовать поля и методы другого класса, изменять их и добавлять к ним поля и методы, характерные только для его состояния и поведения. Наследование бывает двух видов:

- одиночное наследование — подкласс (производный класс) имеет один и только один суперкласс (предок);
- множественное наследование — класс может иметь любое количество подклассов. В Java для классов запрещено, а для интерфейсов разрешено.

Полиморфизм (*polymorphism*) — механизм, использующий одну и ту же сигнатуру метода для решения похожих, но несколько отличающихся задач

В различных классах при наследовании от суперкласса. Целью полиморфизма является использование одного имени при выполнении общих для суперкласса и подклассов действий. Это необходимо для создания общего интерфейса для близких по смыслу действий.

Механизм «позднего связывания» или «динамического полиморфизма» в процессе выполнения программы определяет принадлежность объекта конкретному классу и производит вызов метода, относящегося к классу, объект которого был использован. Механизм «позднего связывания» позволяет определять версию полиморфного (виртуального) метода во время выполнения программы, то есть невозможно на этапе компиляции определить, какая версия переопределенного метода будет вызвана на этапе выполнения программы.

Краеугольным камнем наследования и полиморфизма предстает следующая парадигма: *«объект подкласса может использоваться всюду, где используется объект суперкласса»*. То есть при добавлении к иерархии классов нового подкласса существующий код с экземпляром нового подкласса будет работать точно так же, как и со всеми другими экземплярами классов в иерархии.

При вызове метода сначала он ищется в самом классе. Если метод существует, то он вызывается. Если же метод в текущем классе отсутствует, то обращение происходит к родительскому классу, и вызываемый метод ищется в этом классе. Если поиск неудачен, то он продолжается вверх по иерархическому дереву вплоть до корня (суперкласса всех классов, класса **Object**) иерархии.

Подробнее полиморфизм будет рассмотрен в следующей главе.

Классы — основной элемент абстракции, отвечающий за реализацию назначенного ему контракта и обеспечивающий сокрытие реализации. Классы объединяются в пакеты, которые связаны друг с другом только через ограниченное количество методов и классов. Имя класса в пакете должно быть уникальным. Физически пакет представляет собой каталог, в который помещаются файлы с расширением **.java**, содержащие реализацию классов.

Классы позволяют провести декомпозицию поведения сложной системы до множества элементарных взаимодействий связанных объектов. Класс определяет структуру и/или поведение некоторого элемента предметной области. Под элементом следует понимать представление как физической сущности (например: *Заказ, Товар*), так и логической (например: *УправлениеСчетом, СоставлениеОтчета*).

Определение класса в общем виде без параметризации, без наследования и реализации интерфейсов (наследование по умолчанию только от **Object**) имеет вид:

```
[public] [final] [abstract] class TypeName {
}
```

Спецификатор доступа **public** определяет внешний (*enclosing*) класс, как доступный из других пакетов.

Переменные класса, экземпляра класса и константы

Класс создается в случае необходимости группировки данных и/или действий (методов) под общим именем, то есть создания нового типа данных, объединяющего, инкапсулирующего свойства и действия, связанные с одной предметной областью.

В классах используются следующие данные:

- поля — переменные, объявленные в классе;
- локальные переменные — в методе или блоке кода;
- параметры метода — переменные в объявлении метода.

Поля класса имеют спецификатор и тип. Тип полей класса может быть базовый или объектный.

Со спецификатором **static** объявляется для всего класса статическая переменная класса, которая имеет общее значение для всех экземпляров класса. Без спецификатора **static** объявляются переменные экземпляра класса, имеющие уникальные и независимые значения для каждого объекта класса. Поля класса, как и методы, объявляются также со спецификаторами доступа **public**, **private**, **protected** или по умолчанию без спецификатора. Кроме полей-членов класса, в методах класса используются локальные переменные и параметры методов. В отличие от переменных класса, инкапсулируемых нулевыми элементами, переменные методов не инициализируются по умолчанию.

Поля класса со спецификатором **final** не могут быть переприсвоены (т.е. использоваться справа от оператора присваивания «=») после инициализации. Спецификатор **final** можно использовать не только для поля класса, но и для локальной переменной, объявленной в методе, а также для параметра метода. Это единственный спецификатор, применяемый с параметром метода или локальной переменной.

В следующем примере приводятся объявление и инициализация значений полей класса и локальных переменных метода, а также использование параметров метода:

```
/* # 1 # типы атрибутов и переменных # Order.java */

package by.epam.learn.entity;
// class is available from other packages
public class Order {
    // class instance variable
    private long orderId;
    // class/static variable
    static int bonus;
    // class instance final variable
    public final int CURRENT_RANGE = (int)(Math.random() * 42);
```

```
// class instance final variable
private final MutableType mutable = new MutableType();
// class/static final variable
public final static int PURCHASE_TAX = 5;
// constructors
// methods
public double calculatePrice(double price, int counter) {
    // local variable of the method does not get the default value
    double amount;
    // amount++; compile error, variable not initialized
    amount = (price - bonus) * counter; // initialization of a local variable
    double tax = amount * PURCHASE_TAX/100;
    return amount + tax;
}
}
```

Если значение переменной изменить нельзя, что справедливо для неизменяемых типов данных, к которым относятся все базовые типы и класс **String**, то объявление имени такой переменной со спецификатором **final** должно быть в верхнем регистре **CURRENT_RANGE**, **PURCHASE_TAX**. Если же внутренне состояние **final**-переменной можно изменить (с помощью вызова метода класса), то имя такой переменной должно быть записано в таком же стиле, как и для изменяемых полей класса и локальных переменных. Переменные базовых типов хранятся в сегменте памяти *stack*, переменные объектных типов — в сегменте *heap*, коды приложения, описания классов относят к памяти *non-heap*. К этой же памяти относятся и статические переменные.

Ограничение доступа

Язык Java предоставляет несколько уровней защиты, обеспечивающих возможность настройки области видимости данных и методов. Из-за наличия пакетов Java работает с четырьмя категориями видимости между классами и элементами классов:

- **private** — члены класса доступны только членам данного класса;
- по умолчанию (*package-private*) — члены класса доступны классам, находящимся в том же пакете;
- **protected** — члены класса доступны классам, находящимся в том же пакете, и подклассам — в других пакетах;
- **public** — члены класса доступны для всех классов в этом и других пакетах.

Член класса (поле, конструктор или метод), объявленный **public**, доступен из любого места вне класса. Спецификатор **public** для методов и конструкторов **public**-классов обеспечивает внешний доступ к функциональности пакета, в котором он объявлен. Если данный спецификатор отсутствует, то класс и его методы могут использоваться только в текущем пакете.

Все, что объявлено **private**, доступно только конструкторам и методам внутри класса и нигде больше. Они выполняют служебную или вспомогательную роль в пределах класса, и их функциональность (методов и конструкторов) не предназначена для внешнего использования. Закрытие **private**-полей обеспечивает инкапсуляцию.

Если у члена класса вообще не указан спецификатор уровня доступа, то такой член класса будет виден и доступен из подклассов и классов того же пакета. Именно такой уровень доступа используется по умолчанию. Такой член класса обеспечивает исполнение **public**-методами **public**-классов своей функциональности.

Если же необходимо, чтобы элемент был доступен из другого пакета, но только подклассам того класса, которому он принадлежит, нужно объявить такой элемент со спецификатором **protected**. Спецификатор применим, если поле часто используется в подклассе или если метод предназначен для переопределения и использования его функциональности в другом пакете. Для конструктора наличие **protected** обеспечивает саму возможность наследования от этого класса в другом пакете.

Действие спецификатора доступа распространяется только на тот элемент класса, перед которым стоит такой спецификатор.

Конструкторы

Конструктор — особого вида метод, который по имени автоматически вызывается при создании экземпляра класса с помощью оператора **new**. При корректном проектировании класса конструктор не должен выполнять никаких других обязанностей, кроме инициализации полей класса и проверки непротиворечивости конструирования объекта.

Свойства конструктора:

- Конструктор имеет то же имя, что и класс; вызывается не просто по имени, а только вместе с ключевым словом **new** при создании экземпляра класса.
- Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.
- Конструкторов в классе может быть несколько, но не менее одного.
- Если конструктор в классе явно не определен, то компилятор предоставляет конструктор по умолчанию без параметров.
- Если же конструктор с параметрами определен, то конструктор по умолчанию становится недоступным, и для его вызова необходимо явное объявление такого конструктора.
- Конструктор подкласса при его создании всегда наделяется возможностью вызова конструктора суперкласса. Этот вызов может быть явным или неявным и располагается только в первой строке кода конструктора подкласса.
- Если конструктору суперкласса нужно передать параметры, то необходим явный вызов из конструктора подкласса **super(parameters)**.

- Конструктор может объявляться только со спецификаторами видимости: **public**, **private**, **protected** или по умолчанию.
- Конструктор не может быть объявлен как **static**, **final**, **abstract**, **synchronized**, **native**.
- Если к конструктору добавить возвращаемое значение, то он перестанет быть конструктором, а превратится в метод данного класса. Компилятор при этом выдаст предупреждение о том, что в классе присутствуют методы с таким же именем, как и класс, что является грубым нарушением соглашения о написании кода.

Деструктор в языке Java не используется, объект уничтожается «сборщиком мусора» после определения невозможности его дальнейшего использования (потери ссылки). Некоторым аналогом деструктора являлся метод **finalize()**, в тело которого помещается код по освобождению занятых объектом ресурсов, но в настоящее время данный метод помечен как *deprecated* и не рекомендован к использованию. Тем не менее виртуальная машина станет вызывать его каждый раз, когда «сборщик мусора» будет уничтожать объект класса, которому не соответствует ни одна ссылка.

В следующем примере объявлен класс **Account** с полями, конструкторами и методами для инициализации и извлечения значений атрибутов.

```
/* # 2 # конструкторы # Account.java */
```

```
package by.epam.learn.transfer.bean;
public class Account {
    private long accountId;
    private double asset;
    // constructor without parameters
    public Account() {
        super();
        /* if class is declared without constructors, then compiler will provide it in
           this form */
    }
    // constructor with parameters
    public Account(long accountId) {
        super(); /* access to superclass constructor is explicitly optional, compiler
                  will add it automatically */
        this.accountId = accountId;
    }
    // constructor with parameters
    public Account(long accountId, double asset) {
        this.accountId = accountId;
        this.asset = asset;
    }
    public double getAsset() {
        return asset;
    }
}
```

```

public void setAsset(double asset) {
    this.asset = asset;
}
public long getAccountId () {
    return accountId;
}
public void setId(long accountId) {
    this.accountId = accountId;
}
}

```

Кроме данных и методов каждый экземпляр класса (объект) имеет неявную ссылку **this** на себя, которая передается также неявно и нестатическим методам класса. После этого каждый метод «знает», какой объект его вызвал. Вместо обращения к атрибуту **accountId** в методах можно писать **this.accountId**, хотя и не обязательно, так как записи **accountId**, **this.accountId** и **Account.this.accountId** равносильны. Но если в методе объявлена локальная переменная или параметр метода с таким же именем, как и поле класса, то для обращения к полю класса использование **this** обязательно. Без использования указателя обращение всегда будет производиться к локальной переменной, так как просто не существует другого способа ее идентификации.

Объект класса **Account** может быть создан тремя способами, вызывающими один из трех конструкторов:

```

Account account1 = new Account();
Account account2 = new Account(42L);
Account account3 = new Account(42L, 0.7);

```

Оператор **new** вызывает конструктор, поэтому в круглых скобках могут стоять аргументы, передаваемые конструктору.

Методы

Метод — основной элемент структурирования кода. Методы должны выполнять небольшое конкретное одиночное действие и быть короткими и понятными. Метод должен иметь имя, точно отражающее его поведение. Все методы в Java объявляются только внутри классов и используются для работы с данными класса, локальными и передаваемыми параметрами. Простейшее определение метода имеет вид:

```

ReturnType nameMethod(parameters) {
    // body of method
    return value; // if need
}

```

Если метод не возвращает значение (тип возвращаемого значения метода **void**), ключевое слово **return** отсутствует или записывается без аргумента в виде:

```
return;
```

Хорошим тоном считается наличие только одного оператора **return**, расположенного в последней строке метода.

Вместо пустого списка параметров метода тип **void** не указывается, а только пустые скобки. Вызов методов осуществляется через объект или класс (для статических методов):

```
nameObject.nameMethod(value_parameters);
```

Для того чтобы создать метод, нужно внутри объявления класса написать объявление метода и затем реализовать его тело. Объявление метода как минимум должно содержать тип возвращаемого значения (включая **void**) и имя метода. В приведенном ниже объявлении метода элементы, заключенные в квадратные скобки, являются не обязательными.

```
[public/private/protected] [static] [abstract] [final] [synchronized] [native] [<T>] Return Type  
nameMethod(parameters) [throws exceptions]
```

Как и для полей класса, спецификатор доступа к методам может быть **public**, **private**, **protected** и по умолчанию. При этом методы суперкласса можно перегружать или переопределять в порожденном подклассе.

Еще раз следует напомнить, что объявленные в методе переменные являются локальными переменными метода, а не членами классов, и не инициализируются значениями по умолчанию при создании объекта класса или вызове метода.

Проектирование методов

При распределении обязанностей между классами выделяются так называемые классы бизнес-логики, в которые помещаются методы, обрабатывающие информацию из передаваемых им объектов. Такие классы могут и не иметь атрибутов. Если атрибуты присутствуют, то они обычно играют чисто служебную роль для облегчения методам класса выполнения их функциональности.

Использование параметров метода

Сигнатура разрабатываемого программистом метода должна ясно указывать пользователю на его цель, на необходимые для ее достижения данные и на ожидаемый результат.

За примерами хорошего проектирования методов следует обратиться к методам из библиотек Java. Результат работы метода и способ передачи ему данных должны быть интуитивно понятны, как:

static double max(double a, double b) метод класса **Math**:

- находит максимальное значение из двух переданных методу аргументов;
- все необходимые данные для работы метода передаются в качестве параметров, а в возвращаемое значение помещается результат.

Было бы несколько труднее разобраться в работе метода, если бы он объявлялся в классе, например, так:

```
/* # 3 # плохое проектирование метода max() # MathUgly*/

public class MathUgly {
    private double a;
    private double b;
    public void setA(double a) {
        this.a = a;
    }
    public void setB(double b) {
        this.b = b;
    }
    public double max() {
        return a > b ? a : b;
    }
}
```

Для его корректного использования сначала пришлось бы инициализировать поля класса, а затем только вызывать его метод **max()**.

```
MathUgly math = new MathUgly();
math.setA(1);
math.setB(2);
double maxValue = math.max();
```

А само применение метода неочевидно, поэтому пришлось бы заглянуть в документацию, чтобы понять, как передать значения для метода поиска максимального.

Еще один хороший пример проектирования метода из класса **String**:

Метод **boolean isEmpty()** возвращает значение **true**, если длина строки равна нулю, т.е. строка не содержит ни одного символа. Метод вызывается на объекте типа **String**, который передается в метод по ссылке **this** и обрабатывается в методе, в возвращаемое значение которого и передается результат.

В обоих случаях прочтение сигнатуры метода позволяет предположить, какие исходные данные нужны для его работы и каков будет результат.

Пример некорректного проектирования метода с необоснованным объявлением полей, которые на самом деле должны быть просто параметрами метода:

```
// # 4 # некачественный способ создания метода перевода денег со счета на счет
# TransferAction.java

package by.epam.learn.transfer.action;
import by.epam.learn.transfer.bean.Account;
public class TransferAction {
    private Account from;
    private Account to;
    private double transactionAsset;
```

```
private boolean isDone;
public TransferAction(double asset, Account from, Account to) {
    if (asset > 0 && from != null && to != null ) {
        this.transactionAsset = asset;
        this.from = from;
        this.to = to;
    } else {
        throw new IllegalArgumentException();
    }
}
public boolean isDone() {
    return isDone;
}
public void transferIntoAccount() {
    double demand = from.getAsset() - transactionAsset;
    if (demand >= 0) {
        from.setAsset(demand);
        to.addAsset(transactionAsset);
        isDone = true;
    }
}
}
```

Для выполнения действия придется выполнить дополнительное конфигурирование объекта. Необходима инициализация полей **from**, **to**, **transactionAsset**. Программисту, использующему метод **transferIntoAccount()**, придется разбираться не только с методом, но и с целым классом, конструктором и полями, чтобы выполнить корректное действие. Кроме того, сам метод становится плохо читаемым, так как у него отсутствуют параметры и возвращаемое значение. А если у класса появятся и другие методы и конструкторы, то разобраться в логике создания объектов для выполнения конкретного действия будет еще сложнее.

Для устарания проблемы достаточно всю ответственность за хранение и проверку входных данных передать методу **transferIntoAccount()**, либо валидацию параметров вынести в класс-валидатор.

После рефакторинга в более корректном виде класс может быть таким:

```
/* # 5 # корректный метод # TransferAction.java */
```

```
package by.epam.learn.transfer.action;
import by.epam.learn.transfer.bean.Account;
public class TransferAction {
    public boolean transferIntoAccount(double asset, Account from, Account to) {
        if (asset <= 0 || from == null || to == null ) {
            throw new IllegalArgumentException();
        }
        boolean isDone = false;
        double demand = from.getAsset() - asset;
        if (demand >= 0) {
```

```

        from.setAsset(demand);
        to.addAsset(asset);
        isDone = true;
    }
    return isDone;
}
}

```

Класс стал проще: исчезли все поля, конструктор и метод **isDone()**. Прочитав сигнатуру метода **transferIntoAccount()** теперь легко понять, какие объекты нужны для выполнения действия и каков будет результат.

Если данные невозможно обработать, то генерируется исключение. Если действие невозможно выполнить из-за недостатка средств на исходном счете, то возвращается **false**.

Объект класса может быть создан в любом пакете приложения, если конструктор класса объявлен со спецификатором **public**. Спецификатор **private** не позволит создавать объекты вне класса, так как ограничивает видимость пределами класса, а спецификатор «по умолчанию» — вне пакета. Спецификатор **protected** позволяет создавать объекты в текущем пакете и делает конструктор доступным для обращения к подклассам данного класса в других пакетах.

Использование параметра метода для получения результата

Может возникнуть ситуация, когда необходимо по итогам метода вернуть более одного значения. Какие могут быть быстрые варианты решения? Примеры плохого проектирования метода.

- Создать класс, поля которого будут соответствовать возвращаемым значениям. Такой новый класс не представляет собой описание сущности системы, поэтому возникают проблемы и с его именованием. Область применения его ограничена только в качестве возвращаемого значения одного метода одного класса.
- Одно из значений возвращать как возвращаемое значение метода, а остальные сохранить как поля класса, в котором объявлен метод. Тогда уже сам класс, содержащий такой метод, начнет утрачивать свой смысл, объявляя поля, связанные с выполнением одного метода и никак не связанные с другими методами.

Можно воспользоваться хорошим решением, предлагаемым в самой Java. В интерфейсе **java.io.InputStream** метод **int read(byte[] b)**.

Метод **int read(byte[] b)** читает массив байт из потока ввода и возвращает число прочитанных байт. То есть необходимы два возвращаемых значения. При вызове данного метода требуется объявить перед вызовом метода массив байт в виде:

```
byte[] byteResult = new byte[16];
```

И передать ссылку **byteResult** в метод **read()**. По окончании работы метод возвратит число прочитанных байт как возвращаемое значение, сами прочитанные байты можно просто взять из заполненного методом массива **byteResult**.

Пусть метод должен вычислить и вернуть число разрядов, передаваемого целого числа, и вернуть массив или список, состоящий из цифр этого числа.

Метод, решающий обе проблемы, может быть записан в виде:

```
// # 6 # метод с параметром возвращаемым значением # MethodAction.java
```

```
package by.epam.learn.type;
public class MethodAction {
    public int numberParser(int number, int[] numbers){
        String str = String.valueOf(number);
        int length = str.length();
        for (int i=0; i < length; ++i) {
            char code = str.charAt(i);
            int n = Character.digit(code, 10);
            numbers[i] = n;
        }
        return length;
    }
}
```

а способ вызова достаточно простой:

```
// # 7 # вызов метода и извлечение результатов из параметра
# MethodReturnMain.java
```

```
package by.epam.learn.type;
public class MethodReturnMain {
    public static void main(String[] args) {
        MethodAction method = new MethodAction();
        int num = 739_015_428;
        int[] result = new int[10];
        int length = method.numberParser(num, result);
        System.out.println(length);
        for (int j = 0; j < length; j++)
            System.out.printf("%d, ", result[j]);
    }
}
```

В результате будет выведено:

9

7, 3, 9, 0, 1, 5, 4, 2, 8,

Использование параметра метода как возвращаемого значения позволяет не создавать новые классы и не хранить полученные результаты вне метода. Однако сложностью является необходимое предварительное объявление объекта ре-

зультатов. В данном случае в качестве параметра лучше использовать коллекцию, в частности **List<Integer>**.

Использование возвращаемого значения

Стандартной задачей для создаваемого метода представляется поиск и/или создание какого-либо объекта на основе передаваемых в метод данных. Необходимо найти некоторую сущность по заданному признаку в некотором хранилище/файле/репозитории/БД и проч., например, по значению **orderId**. При проектировании метода это означает, что возвращаемым значением метода как раз следует сделать объект класса **Order**, и он будет в единственном числе из-за наличия уникального идентификатора.

```
public Order findOrderById(long orderId) {
    Order entity = null;
    try {
        // order search
        order = // order found
    } catch(SQLException e) { // or IOException etc.
        // log
    }
    return order;
}
```

Если объект успешно найден, то реализуется основной положительный сценарий и вроде бы проблема решена — метод готов к использованию. Однако исходя из логики возможна ситуация, когда объект не может быть получен по нескольким причинам. Наиболее очевидны две причины: объект с заданным идентификатором **orderId** не найден; при выполнении поиска возникла исключительная ситуация.

В случае генерации исключения будет записан лог, а метод возвратит значение **null**. Такое использование метода **findOrderById()** приведет к дезинформации пользователя. Пользователь метода, если не прочтает логи, что очень вероятно, будет считать что сущность с таким **id** не найдена и дальнейшую логику работы приложения будет строить на этом ложном основании, что может привести в итоге к некорректному результату работы всей системы. Такие ошибки достаточно сложно выявить, так как метод отрабатывает корректно, но возвращает результат, который можно трактовать двояко.

Устранить двойственность в данном примере следует заменой записи лога в блоке **catch** на генерацию проверяемого исключения. Подробнее про генерацию исключений будет рассказано в главе «Исключения и ошибки».

```
public Order findOrderById(long orderId) throws CustomException {
    Order order= null;
    try {
        // order search
        order = // order found
    }
```

```

    } catch(SQLException e) {
        throw new CustomException(e);
    }
    return order;
}

```

Теперь работа метода становится однозначной. В случае успешного поиска возвращается объект, в случае отсутствия подходящего объекта возвращается **null**, в ситуации возникновения ошибки при поиске генерируется исключение.

Такие результаты выполнения метода достаточно типичны для большинства методов и при проектировании метода следует исходить из указанных выше соображений.

Но возможны нюансы, например, для методов, отвечающих за валидацию данных. Фактически метод **isEmpty()** класса **String** является типичным методом-валидатором. Такие методы должны возвращать только конкретный результат типа **boolean**, вне зависимости от того, что произошло в теле метода.

Возвращение методом значения **null** также далеко не безобидно по причине возможности генерации на возвращаемом объекте исключения **NullPointerException**, что требует от разработчика внимания и проверки результата работы метода на этот самый **null**. Проблема **null** является проблемой на «миллион долларов» и пока не решена. Однако существует механизм класса **Optional<T>**, который четко демонстрирует пользователю, что метод может возвращать **null**.

Если существует возможность не возвращать значение null, то не следует возвращать его!

От возвращения значения **null** избавиться можно, заменив возвращаемое значение метода на коллекцию:

```

public List<Order> findOrderByName(String name) throws CustomException {
    List<Order> orders = new ArrayList<>();
    try {
        // searching
        orders = // orders found
    } catch (SQLException e) {
        throw new CustomException(e);
    }
    return orders;
}

```

Что позволяет в ситуации, когда требуемый объект\объекты не были найдены, вернуть пустой список. Эту же конфигурацию метода (с оговорками) можно применять даже когда заранее известно, что найден будет только один объект.

Оболочка Optional

Класс **java.util.Optional<T>** представляет собой оболочку для любого объектного типа. Смысл класса в обозначении проблемы возвращения значения **null**,

методом осуществляющим поиск/обработку/генерацию какого-либо объекта в ситуации, если это действие не дало приемлемого результата.

Когда программист использует метод, возвращающий некоторую ссылку на значение, то он планирует использовать полученный объект. Если метод возвращает **null**, то в этом случае его использование гарантированно приведет к генерации **NullPointerException**. Программисту придется вводить в код дополнительную проверку на **null**, чтобы защитить свой код от исключения.

Класс **Optional<T>** не решает проблему возвращения **null**, но его использование в качестве возвращаемого значения ясно обозначает ее.

Сначала будет представлен метод в классической реализации:

```
/* # 8 # метод, который может вернуть null # OrderAction.java */

package by.epam.learn.optional;
import by.epam.learn.entity.Order;
import java.util.List;
import java.util.stream.Collectors;
public class OrderAction {
    public Order findById(List<Order> orders, long id) {
        Order order = null;
        List<Order> result = orders.stream()
                                   .filter(o -> id == o.getOrderid())
                                   .collect(Collectors.toList());

        if(result.size() != 0) {
            order = result.get(0);
        }
        return order;
    }
}
```

который возвратит **null**, если требуемый объект не найден.

Замена возвращаемого значения на **Optional<T>** делает код метода проще:

```
/* # 9 # метод, который возвращает объект в оболочке */

public Optional<Order> findById(List<Order> orders, long id) {
    List<Order> result = orders.stream()
                               .filter(o -> id == o.getOrderid())
                               .collect(Collectors.toList());
    Optional<Order> optionalOrder =
        result.size() != 0 ? Optional.of(result.get(0)) : Optional.empty();
    return optionalOrder;
}
```

Возможности **Stream<T>** будут подробно рассмотрены в другой главе.

Статические методы класса создают новый объект:

<T> Optional<T> empty() — создает «пустой» объект;

<T> Optional<T> of(T value) — создает объект с ненулевым содержимым;

<T> Optional<T> ofNullable(T value) — создает объект, содержимое которого может быть нулевым.

Код этого метода можно сделать еще короче:

```
public Optional<Order> findById(List<Order> orders, long id) {
    Optional<Order> optionalOrder = orders.stream()
        .filter(o -> id == o.getOrderId())
        .findAny();

    return optionalOrder;
}
```

Каким же образом тогда работать с возвращаемым значением такого метода? Возможностей довольно много.

```
/* # 10 # обработка Optional # OptionalMain.java */
```

```
package by.epam.learn.optional;
import by.epam.learn.entity.Order;
import java.util.*;
public class OptionalMain {
    public static void main(String[] args) {
        List<Order> list = new ArrayList<>();
        list.add(new Order(71L, 100D));
        list.add(new Order(18L, 132D));
        list.add(new Order(24L, 210D));
        list.add(new Order(35L, 693D));
        OrderAction action = new OrderAction();
        Optional<Order> optionalOrder = action.findById(list, 24); // replaced by 23
        if(optionalOrder.isPresent()) {
            System.out.println(optionalOrder.get());
        }
        Set<Order> set = new HashSet<>();
        optionalOrder.ifPresent(set::add); // or o->set.add(o)
        System.out.println(set);
    }
}
```

Некоторые методы класса:

T get() — извлекает объект из оболочки, но если оболочка пуста, то генерирует **NullPointerException**, поэтому лучше сначала удостовериться в наличии объекта в оболочке;

boolean isPresent(), boolean isEmpty() — проверяют пуста оболочка или нет;

void ifPresent(Consumer<? super T> action) — если объект присутствует в оболочке, то выполняет над ним действие;

void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction) — если объект присутствует в оболочке, то выполняет над ним действие **action**, если отсутствует, то действие **emptyAction**.

Представлена также целая серия методов, извлекающих объект из оболочки, но в случае его **отсутствия** выполняющих другое действие:

T orElse(T other) — возвращает объект **other**;

T orElseThrow(), <T extends Throwable> T orElseThrow(Supplier<? extends X> exception) — генерирует исключение **NoSuchElementException** или заданное исключение.

Современные технологии программирования очень активно используют возможности класса **Optional<T>**, поэтому изучить и применять его возможности необходимо.

В пакете **java.util** объявлены контейнеры для работы с числовыми значениями: **OptionalInt**, **OptionalLong** и **OptionalDouble**. Функциональность их практически повторяет **Optional<T>**, но предназначены они только для хранения конкретных типов данных: **int**, **long** и **double**.

Статические методы и поля

Методы и поля со спецификатором **static** логически привязаны к классу и представляют собой приближенный аналог глобальных переменных и методов.

Роль статических методов в приложении очень проста: ***вызов статического метода не может быть никаким образом заменен другой реализацией. Выбор статического метода раз и навсегда определяется на этапе компиляции.***

Поле данных, объявленное в классе как **static**, общее для всех объектов класса и называется переменной класса. Переменная класса может быть использована без создания экземпляра класса. Если один объект изменит значение такого поля, то это изменение увидят все объекты. Для работы со статическими полями используются статические методы, объявленные со спецификатором **static**. Нестатические методы могут обращаться к статическим полям и методам напрямую без всяких дополнительных условий. Свойства статических методов:

- Статические методы являются методами класса, не привязаны ни к какому объекту и не содержат указателя **this** на конкретный экземпляр, вызвавший метод.
- Статические методы реализуют парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции.
- По причине недоступности указателя **this** статические поля и методы не могут обращаться к нестатическим полям и методам напрямую, так как они не «знают», к какому объекту относятся, да и сам экземпляр класса может быть не создан.
- Для обращения к статическим полям и методам достаточно имени класса, в котором они определены.
- Переопределение статических методов невозможно, так как обращение к статическому методу осуществляется посредством имени класса, которому они принадлежат.

```
// # 11 # Статические метод и поле # TransferAction.java
```

```
package by.epam.learn.action;
public class TransferAction {
    public static int taxRate = 12; // static field
    private double amountTaxes; // non static field
    public double getAmountTaxes() { // non static method
        return amountTaxes;
    }
    public static void increaseTaxRate() { // static method
        // this cannot be used - the object does not exist
        // amountTaxes cannot be used - the object does not exist
        /* methods getAmountTaxes() and transferIntoAccount() cannot be called -
           the object does not exist */
        taxRate++;
    }
    public boolean transferIntoAccount(double asset, Account from, Account to) {
        if (asset <= 0 || from == null || to == null ) {
            throw new IllegalArgumentException(); // or custom exception
        }
        boolean isDone = false;
        double currentAmountTaxes = asset/taxRate;
        amountTaxes += currentAmountTaxes;
        double demand = from.getAsset() - asset * ( 1 + currentAmountTaxes);
        if (demand >= 0) {
            from.setAsset(demand);
            to.addAsset(asset);
            isDone = true;
        }
        return isDone;
    }
}
```

Вызов метода **increaseTaxRate()** осуществляется без создания объекта:

```
TransferAction.increaseTaxRate();
```

но и переопределить статический метод уже нельзя.

Для двух объектов

```
TransferAction action1 = new TransferAction();
TransferAction action2 = new TransferAction();
```

значение **action1.taxRate** и **action2.taxRate** равно **12**, поскольку располагается в одной и той же области памяти вне объекта. Такая запись не корректна. Поэтому обращаться и изменять значение статического поля следует непосредственно через имя класса:

```
TransferAction.taxRate = 20;
```

или через статический метод, который делает изменения по определенному программистом алгоритму.

Вызов статического метода всегда следует осуществлять с помощью указания на имя класса, а не объекта. Статический метод можно вызывать также с использованием имени объекта, но такой вызов снижает качество кода, приводит к появлению соответствующего предупреждения и не будет логически корректным, хотя и не закончится ошибкой компиляции.

Статические методы используются при необходимости придать функциональности метода признак «окончателности», «неизменности» реализации алгоритма для данного класса.

Модификатор **final** и неизменяемость

Модификатор **final** используется для определения констант в качестве члена класса, локальной переменной или параметра метода. Методы, объявленные как **final**, нельзя замещать в подклассах. Для классов, объявленных со спецификатором **final**, нельзя создавать подклассы.

```
/* # 12 # final-поля и переменные # Card.java */
```

```
package by.epam.learn.finalvar;
public class Card {
    // uninitialized constant
    public final long BANK_ID; // initialization is not done by default!
    public Card() {
        // initialization in the constructor in this case is required!
        BANK_ID = 100_000; // only once
    }
    public Card(long id) {
        // initialization in the constructor in this case is required!!
        BANK_ID = id; // only once
    }
    public final boolean checkRights(final int NUMBER) {
        int value = 1;
        final int CODE = 42 + value;
        // NUMBER = 1; // compile error
        // CODE = 1; // compile error
        return CODE == NUMBER;
    }
}
```

или

```
/* # 13 # final-поля и переменные # Card.java */
```

```
package by.epam.learn.finalvalue;
public class Card {
    public final long BANK_ID;
    { BANK_ID = 111_111_111_111L; } // only once !!
    public Card() {
```

```
// initialization in the constructor is now impossible
}
public Card(long id) {
    // BANK_ID = id; // compile error
}
}
```

Константа может быть объявлена как поле класса, но не проинициализирована. В этом случае она должна быть проинициализирована в логическом блоке класса, заключенном в {}, или конструкторе, но только в одном из указанных мест. Значение по умолчанию константа получить не может в отличие от переменных класса. Константы могут быть объявлены в методах как локальные или как параметры метода. В обоих случаях значения таких констант изменять нельзя.

Абстрактные методы

Абстрактные методы размещаются в абстрактных классах или интерфейсах, тела у таких методов отсутствуют и должны быть реализованы в подклассах.

```
/* # 14 # абстрактный класс и метод # AbstractCardAction.java */
import java.math.BigDecimal;
public abstract class AbstractCardAction {
    private BigDecimal amount;
    public AbstractCardAction() { }
    public abstract void doPayment(BigDecimal amountPayment); /* no implementation */
    public void setAmount(BigDecimal amount) {
        this.amount = amount;
    }
}
```

При этом становится невозможным создание экземпляра:

```
AbstractCardAction action = new AbstractCardAction(); // compile error
```

Спецификатор **abstract** присутствует здесь как в объявлении метода, так и в объявлении класса.

В отличие от интерфейсов абстрактный класс может содержать и абстрактные, и неабстрактные методы, а может и не содержать ни одного абстрактного метода.

Подробнее абстрактные классы и интерфейсы изучаются в главе «Наследование и полиморфизм».

Модификатор native

Приложение на языке Java может вызывать методы, написанные на языке C++. Такие методы объявляются с ключевым словом **native**, которое сообщает компилятору, что метод реализован в другом месте. Например:

```
public native int loadCrypto(int value);
```

Методы, помеченные **native**, можно перегружать и переопределять обычными методами Java в подклассах.

Модификатор **synchronized**

При взаимодействии нескольких потоков в одном приложении необходимо синхронизировать методы, обращающиеся к общим данным. Когда интерпретатор обнаруживает метод **synchronized**, он включает код, блокирующий доступ к объекту при вызове этого метода и снимающий блок при его завершении.

Вызов методов уведомления о возвращении блокировки объекта **notifyAll()**, **notify()** и метода останова потока и освобождения блокировки **wait()** класса **Object** (суперкласса для всех классов языка Java) предполагает использование модификатора **synchronized**, так как эти методы предназначены для вызова на заблокированных объектах.

Логические блоки

При описании класса могут быть использованы логические блоки. Логическим блоком называется код, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса, например:

```
{ /* code */ }
static { /* code */ }
```

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов и обращения к полям текущего класса. При создании объекта класса они вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса. Операции с полями класса внутри логического блока до явного объявления этого поля возможны только при использовании ссылки **this**, представляющей собой ссылку на текущий объект.

Логический блок может быть объявлен со спецификатором **static**. В этом случае он вызывается только один раз в жизненном цикле приложения при создании объекта или при обращении к статическому методу (полю) данного класса.

На практике статические логические блоки могут применяться для проверки и инициализации базовых параметров, необходимых для функционирования приложения или класса. Нестатические логические блоки могут применяться для проверки и инициализации параметров конкретного объекта и для сокращения количества кода, если одинаковый код присутствует в каждом конструкторе.

```
/* # 15 # логические блоки при объявлении класса # Department.java
# DemoLogicMain.java */
```

```
package by.epam.learn.logic;
public class Department {
```

```

{
    System.out.println("logic (1) id=" + this.id);
}
static {
    System.out.println("static logic");
}
private int id = 42;
public Department(int id) {
    this.id = id;
    System.out.println("constructor id=" + id);
}
public int getId() {
    return id;
}
/* not very good location of the logical block */
System.out.println("logic (2) id=" + id);
}
}
package by.epam.learn.logic;
public class DemoLogicMain {
    public static void main(String[] args) {
        new Department(71);
        new Department(17);
    }
}

```

В результате выполнения программы будет выведено:

```

static logic
logic (1) id=0
logic (2) id=42
constructor id=71
logic (1) id=0
logic (2) id=42
constructor id=17

```

Во второй строке вывода поле **id** получит значение по умолчанию, так как память для него выделена при создании объекта, а значение еще не проинициализировано. В четвертой строке выводится значение поля **id**, равное **71**, так как после инициализации атрибута класса был вызван логический блок, получивший его значение.

Логические блоки не наследуются.

Перегрузка методов

Метод называется перегруженным, если существует несколько его версий с одним и тем же именем, но с разным списком параметров. Перегрузка реализует «раннее связывание», то есть версия вызываемого метода определяется на

этапе компиляции. Перегрузка может ограничиваться одним классом. Методы с одинаковыми именами, но с различными списком параметров и возвращаемыми значениями, могут находиться в разных классах одной цепочки наследования и также будут перегруженными. Если списки параметров идентичны, то имеет место механизм динамического полиморфизма — переопределение метода.

Статические методы могут перегружаться нестатическими, и наоборот, без ограничений.

При вызове перегруженных методов возможна неопределенность, когда компилятор будет не в состоянии выбрать тот или иной метод. Компилятор реагирует на эту ситуацию ошибкой компиляции.

```
/* # 16 # вызов перегруженных методов # NumberInfo.java */
```

```
package by.epam.learn.overload;
public class NumberInfo {
    public void viewNum(Integer i) { // 1
        System.out.printf("Integer=%d\n", i);
    }
    public void viewNum(int i) { // 2
        System.out.printf("int=%d\n", i);
    }
    public void viewNum(Float f) { // 3
        System.out.printf("Float=%.4f\n", f);
    }
    public void viewNum(Number n) { // 4
        System.out.println("Number=" + n);
    }
}

package by.epam.learn.overload;
public class RunnerNumberInfo {
    public static void main(String[] args) {
        NumberInfo info = new NumberInfo();
        Number[] num = {new Integer(7), 71, 3.14f, 7.2 };
        for (Number n : num) {
            info.viewNum(n);
        }
        info.viewNum(new Integer(8));
        info.viewNum(81);
        info.viewNum(4.14f);
        info.viewNum(8.2);
    }
}
```

Может показаться, что в результате компиляции и выполнения данного кода в цикле **for** будут последовательно вызваны все четыре метода, однако в консоль будет выведено:

```
Number=7
Number=71
```

Number=3.14

Number=7.2

Integer=8

int=81

Float=4.1400

Number=8.2

То есть во всех случаях при передаче в метод элементов массива был вызван четвертый метод. Это произошло вследствие того, что выбор варианта перегруженного метода происходит на этапе компиляции и зависит от типа массива **num**. То, что на этапе выполнения в метод передается другой тип (для первых трех элементов массива), не имеет никакого значения, так как выбор уже был осуществлен заранее.

При непосредственной передаче объекта в метод выбор производится в зависимости от типа ссылки на этапе компиляции.

С одной стороны, этот механизм снижает гибкость, с другой — все возможные ошибки при обращении к перегруженным методам отслеживаются на этапе компиляции, в отличие от переопределенных методов, когда их некорректный вызов приводит к возникновению исключений на этапе выполнения. При перегрузке всегда надо придерживаться следующих правил:

- не использовать сложных вариантов перегрузки;
- не использовать перегрузку с одинаковым числом параметров;
- заменять при возможности перегруженные методы на несколько разных методов.

Параметризованные классы

Параметризация (*generic*) классов и методов, позволяет использовать гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Применение *generic*-классов для создания типизированных коллекций будет рассмотрено в главе «Коллекции и Stream API». Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

Использование параметризации позволяет сократить число создаваемых классов и размер кода в методах.

Пусть необходимо разработать класс для описания сущности «Сообщение» в мессенджере. Сообщение может быть простым текстом, изображением, медиафайлом или их комбинацией. Каждое сообщение к тому же должно иметь идентификатор для хранения в БД. Чтобы описать все типы сообщений придется строить иерархию наследования из обычных классов. Применение *generic* позволит сократить описание группы сущностей до одного класса.

```
/* # 17 # объявление класса с двумя параметрами # Post.java */
```

Ниже приведен пример подобного *generic*-класса с двумя параметрами:

```
package by.epam.learn.messenger.entity;
public class Post <K extends Number, V> {
    private K postId;
    private V message;
    // fields & constructors & methods
}
```

Здесь **K**, **V** — фиктивные объектные типы, которые используются при объявлении членов класса и обрабатываемых данных. В качестве типа **K** допустимо использовать только подклассы класса **Number**. В качестве параметров классов запрещено применять базовые типы.

Объект класса **Post** можно создать, например, следующим образом:

```
Post<Short, String> post1 = new Post<Short, String>();
Post<Long, Picture> post2 = new Post<Long, Picture>();
Post<Integer, Media> post3 = new Post<Integer, Media>();
```

или

```
Post<Integer, Media> post3 = new Post<>(); // <> - operator diamond (Java 7)
```

Параметризованные типы обеспечивают типобезопасность (typesafe).

Присваивание **post1=post2** приводит к ошибке компиляции.

При создании объекта компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект, при этом все внешние признаки параметризации исчезнут, то есть проверка на принадлежность типу осуществима на этапе выполнения кода динамическим оператором **instanceof** в виде:

```
post instanceof Post
```

Ниже приведен пример параметризованного класса **Message** с конструкторами и методами, также инициализация и исследование поведения объектов при задании различных параметров.

```
/* # 18 # создание и использование объектов параметризованного класса
# Message.java # MessageMain.java */
```

```
package by.epam.learn.messenger.entity;
public class Message <T> {
    private T value;
    public Message() {}
    public Message(T value) {
        this.value = value;
    }
    public T getValue() {
```

```

    return value;
  }
  public void setValue(T value) {
    this.value = value;
  }
  public String toString() {
    if (value == null) {
      return null;
    }
    return value.getClass().getName() + " : " + value;
  }
}

package by.epam.learn.messenger.main;
import by.epam.learn.messenger.entity.Message;
public class MessageMain {
  public static void main(String[ ] args) {
    Message<Integer> ob1 = new Message<>();
    ob1.setValue(1); // only Integer or int
    int v1 = ob1.getValue();
    System.out.println(v1);
    Message<String> ob2 = new Message<>("Java");
    String v2 = ob2.getValue();
    System.out.println(v2);
    // ob1 = ob2; // compile error - parameterization is not covariant
    // default parameterization - Object
    Message ob3 = new Message(); // warning - raw type
    ob3 = ob1; // no compilation error - no parameterization
    System.out.println(ob3.getValue());
    ob3.setValue(new Byte((byte)1));
    ob3.setValue("Java SE 12");
    System.out.println(ob3);/* the type of the object is displayed, not the type of
                           parameterization */

    ob3.setValue(71);
    System.out.println(ob3);
    ob3.setValue(null);
  }
}

```

В результате выполнения этой программы будет выведено:

1

Java

null

java.lang.String: Java SE 12

java.lang.Integer: 71

В рассмотренном примере были созданы объекты типа **Message**: **ob1** на основе типа **Integer** и **ob2** на основе типа **String** при помощи различных конструкторов. При компиляции вся информация о *generic*-типах стирается и заменяется для членов класса и методов заданными типами или типом **Object**,

если параметр не задан, как для объекта **ob3**. Такая реализация необходима для обеспечения совместимости с кодом, созданным в предыдущих версиях языка.

Объявление *generic*-типа в виде `<T>`, несмотря на возможность использовать любой тип в качестве параметра, ограничивает область применения разрабатываемого класса. Переменные такого типа могут вызывать только методы класса **Object**. Доступ к другим методам ограничивает компилятор, предупреждая возможные варианты возникновения ошибок.

Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```
public class ValueExtension <T extends Type> {
    private T value;
}
```

Такая запись говорит о том, что в качестве типа **T** разрешено применять только классы, являющиеся наследниками (подклассами) реального класса **Type**, и, соответственно, появляется возможность вызова методов ограничивающих (*bound*) типов.

Часто возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом. В этом случае при определении метода следует применить метасимвол `«?»` или анонимный тип, или *wildcard*. Метасимвол также может использоваться с ограничением **extends** или **super** для передаваемого типа.

```
/* # 19 # использование метасимвола в параметризованном классе # Task.java
# TaskMain.java */
```

```
package by.epam.learn.entity;
public class Task<T extends Number> {
    private String name;
    private T mark;
    public Task(T mark, String name) {
        this.name = name;
        this.mark = mark;
    }
    public T getMark() {
        return mark;
    }
    public boolean equalsToMark(Task<T> task) {
        return roundMark() == task.roundMark();
    }
    private int roundMark() {
        return Math.round(mark.floatValue());
    }
}
package by.epam.learn.main;
```

```
import by.epam.learn.entity;
public class TaskMain {
    public static void main(String[ ] args) {
        Task<Double> task1 = new Task<Double>(71.41D, "JSE");// 71.5d
        Task<Double> task2 = new Task<Double>(71.45D, "JEE");// 71.5d
        System.out.println(task1.equalsToMark(task2));
        Task<Integer> task = new Task<Integer>(71, "Scala");
        // task1.equalsToMark(task); // compile error: incompatible types
    }
}
```

В результате будет выведено:

true.

Метод с параметром **Task<T>** может принимать исключительно объекты с инициализацией того же типа, что и вызывающий метод объект. Чтобы метод **equalsToMark()** мог распространить свои возможности на экземпляры класса **Task**, инициализированные любым допустимым типом, его следует переписать с использованием метасимвола «?» в виде:

```
public boolean equalsToMark(Task<?> task) {
    return roundMark() == task.roundMark();
}
```

Тогда при вызове **task1.equalsToMark(task)** ошибки компиляции не возникнет, и метод выполнит свою расширенную функциональность по сравнению объектов класса **Task**, инициализированных объектами различных допустимых типов. В противном случае было бы необходимо создавать новые перегруженные методы.

Для *generic*-типов существует целый ряд ограничений. Например, невозможно выполнить явный вызов конструктора *generic*-типа:

```
class FailedClass<T> {
    private T t = new T();
}
```

так как компилятор не знает, какой конструктор может быть вызван и какой объем памяти должен быть выделен при создании объекта.

По аналогичным причинам *generic*-поля не могут быть статическими, статические методы не могут иметь *generic*-параметры, например:

```
/* # 20 # неправильное объявление и использование полей параметризованного
класса # FailedTwo.java */
```

```
class FailedTwo <T> {
    static T t;
}
```

```
class FailedThree <K> {
    K k;
```

```
static void takeKey(K k) {
    // ...
}
}
```

Параметризованные методы

Параметризованный (*generic*) метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра, и может быть записан, например, в виде:

```
<T extends Type> ReturnType method(T arg) { }
<T> static T[] method(int count, T arg) { }
```

Описание типа должно находиться перед возвращаемым типом. Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после **extends**. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

Generic-методы могут находиться как в параметризованных классах, так и в обычных. Параметр метода может не иметь никакого отношения к параметру своего *generic*-класса. Причем такому методу разрешено быть статическим, так как параметризацию обеспечивает сам метод, а не класс, в котором он объявлен. Метасимволы применимы и к *generic*-методам.

```
/* # 21 # параметризованные конструкторы и методы # SimpleAction.java */
```

```
package by.epam.learn.method;
public class SimpleAction {
    public <T extends Course> SimpleAction (T course) {
    }
    public <T> SimpleAction() {
    }
    public <T extends Course> float calculateMark(T course) {
    }
    public <T> boolean printReport(T course) {
    }
    public <T> void check() {
    }
}
```

Создание экземпляра с параметром и вызов параметризованного метода с параметром выглядят следующим образом:

```
SimpleAction action = new SimpleAction(new Course());
action.printReport(new Course(42));
```

Создание экземпляра с использованием параметризованного конструктора без параметров требует указания типа параметра перед именем конструктора:

```
action = new <String>SimpleAction();
```

Аналогично для метода без параметров:

```
action.<Integer>check();
```

Методы с переменным числом параметров

Возникают ситуации, когда заранее неизвестно количество передаваемых экземпляров класса в метод. В обычной ситуации пришлось бы создавать несколько перегруженных методов с разным числом параметров одного типа. Другим решением будет один метод с параметром в виде массива или коллекции, что потребует предварительной организации соответствующего объекта массива или коллекции.

Применяются такие методы в случае, когда необходимо присвоить\изменить\использовать какое-либо свойство набора объектов, но заранее неизвестно, сколько таких объектов будет представлено. В некоторых ситуациях параметризованные методы являются альтернативой циклу.

Существует возможность передачи в метод нефиксированного числа параметров, что позволяет отказаться от предварительного создания сложного объекта для его последующей передачи в метод. Набор объектов, переданный в такой метод, преобразуется в массив с типом и именем, которые указаны в качестве параметров метода. Метод **printf()** с переменным числом аргументов уже применялся неоднократно в примерах предыдущих глав.

Список параметров метода выглядит в общем случае:

```
(Type... args)
```

а в случае необходимости передачи параметров других типов:

```
(Type1 t1, Type2 t2, TypeN tn, Type... args)
```

```
/* # 22 # определение количества параметров метода # DemoVarargMain.java */
```

```
package by.epam.learn.varargs;
public class DemoVarargMain {
    public static int defineArgCount(Integer... args) {
        if (args.length != 0) {
            for (int element : args) {
                System.out.printf("arg:%d ", element);
            }
        } else {
            System.out.print("No arg ");
        }
    }
}
```

```

        return args.length;
    }
    public static void main(String ... args) {
        System.out.println("N=" + defineArgCount(7, 42, 555));
        Integer[] arr = { 1, 2, 3, 4, 5, 42, 7 };
        System.out.println("N=" + defineArgCount(arr));
        System.out.println(defineArgCount());
        // defineArgCount(arr, arr); // compile error
        // defineArgCount(71, arr); // compile error
    }
}

```

В результате выполнения этой программы будет выведено:

```

arg:7 arg:42 arg:555 N=3
arg:1 arg:2 arg:3 arg:4 arg:5 arg:42 arg:7 N=7
No arg 0

```

В примере приведен простейший метод с переменным числом параметров. Метод **defineArgCount()** выводит все переданные ему аргументы и возвращает их количество. При передаче параметров в метод из них автоматически создается массив. Второй вызов метода в примере позволяет передать в метод массив. Метод может быть вызван и без аргументов.

Чтобы передать несколько массивов в метод по ссылке, следует использовать следующее объявление:

```
method(Type[]... args) { }
```

Методы с переменным числом аргументов могут быть перегружены:

```

void method(Integer...args) { }
void method(int x1, int x2) { }

```

Недопустимый способ:

```

void method(Type... args) { }
void method(Type[] args) { }

```

В следующем примере приведены три перегруженных метода и несколько вариантов их вызова. Отличительной чертой является возможность метода с аргументом **Object... args** принимать не только объекты, но и массивы:

```
/* # 23 # передача массивов # OverloadMain.java */
```

```

package by.epam.learn.overload;
public class OverloadMain {
    public static void printArgCount(Object... args) { // 1
        System.out.println("Object args: " + args.length);
    }
    public static void printArgCount(Integer[]...args) { // 2
        System.out.println("Integer[ ] args: " + args.length);
    }
}

```

```
public static void printArgCount(int... args) { // 3
    System.out.print("int args: " + args.length);
}
public static void main(String... args) {
    Integer[] i = { 1, 2, 3, 42, 5 };
    printArgCount(7, "No", true, null);
    printArgCount(i, i, i);
    printArgCount(i, 4, 42);
    printArgCount(i); // call method 1
    printArgCount(42, 7);
    // printArgCount(); //compile error: overload uncertainty!
}
}
```

В результате будет выведено:

```
Object args: 4
Integer[] args: 3
Object args: 3
Object args: 5
int args: 2
```

При передаче в метод **printArgCount()** единичного массива **i** компилятор отдает предпочтение методу с параметром **Object... args**. Так как имя массива является объектной ссылкой, указанный параметр будет ближайшим. Метод с параметром **Integer[]...args** не вызывается, потому что ближайшей объектной ссылкой для него будет **Object[]...args**. Метод с параметром **Integer[]...args** будет вызван для единичного массива только в случае отсутствия метода с параметром **Object...args**.

При вызове метода без параметров возникает неопределенность из-за возможности однозначного выбора.

Не существует также ограничений и на переопределение подобных методов.

Единственным ограничением является то, что параметр вида **Type...args** должен быть единственным и последним в объявлении списка параметров метода, то есть записи вида: **(Type1... args, Type2 obj)** и **(Type1... args, Type2... args)** приведут к ошибке компиляции.

Перечисления

При разработке приложений достаточно часто встречаются сущности, число значений которых ограничено естественным образом: страны мира, месяцы года, дни недели, марки транспортных средств, типы пользователей и многие другие. В этих случаях до Java 5 прибегали к следующему решению:

```
/* # 24 # объявление набора констант # RoleOldStyle.java */
```

```
package by.epam.learn.type;
public class RoleOldStyle {
```

```
public final static int GUEST = 0;
public final static int CLIENT = 1;
public final static int MODERATOR = 2;
public final static int ADMIN = 3;
private RoleOldStyle(){}
```

Обрабатывались такие значения следующим образом:

```
/* # 25 # использование набора констант # RoleOldStyleMain.java */

package by.epam.learn.type;
public class RoleOldStyleMain {
    public static void main(String[] args) {
        int role = 1;
        switch (role){
            case 0:
                System.out.println("guest can only watch");
            case 1:
                System.out.println("client can place orders");
            case 2:
                System.out.println("moderator can manage his section");
            case 3:
                System.out.println("admin controls everything");
        }
    }
}
```

Не очень очевидное решение. Но Java 5 был введен класс-перечисление, и тогда в данной задаче код становится более читаемым:

```
/* # 26 # объявление набора констант-элементов перечисления # Role.java
# RoleMain.java */

package by.epam.learn.type;
public enum Role {
    GUEST, CLIENT, MODERATOR, ADMIN
}

package by.epam.learn.type;
public class RoleMain {
    public static void main(String[] args) {
        Role role = Role.valueOf("Admin".toUpperCase());
        switch (role){
            case GUEST:
                System.out.println(role + " can only watch");
            case CLIENT:
                System.out.println(role + " can place orders");
            case MODERATOR:
                System.out.println(role + " can manage his section");
        }
    }
}
```

```

        case ADMIN:
            System.out.println(role + " controls everything");
        }
    }
}

```

Типобезопасные перечисления (*typesafe enums*) в Java представляют собой классы и являются подклассами абстрактного класса **java.lang.Enum**. Вместо слова **class** при описании перечисления используется слово **enum**. При этом объекты перечисления инициализируются прямым объявлением без помощи оператора **new**. При инициализации хотя бы одного перечисления происходит инициализация всех без исключения оставшихся элементов данного перечисления.

В операторах **case** используются константы без уточнения типа перечисления, так как его тип определен в **switch**.

Перечисление как подкласс класса **Enum** может содержать поля, конструкторы и методы, реализовывать интерфейсы. Каждый элемент **enum** может использовать методы:

static enumType[] values() — возвращает массив, содержащий все элементы перечисления в порядке их объявления;

static <T extends Enum<T>> T valueOf(Class<T> enumType, String arg) — создает элемент перечисления, соответствующий заданному типу и значению передаваемой строки;

static enumType valueOf(String arg) — создает элемент перечисления, соответствующий значению передаваемой строки;

int ordinal() — возвращает позицию элемента перечисления, начиная с нуля, следствием чего является возможность сравнения элементов перечисления между собой на больше\меньше соответствующими операторами;

String name() — возвращает имя элемента, так же как и **toString()**;

int compareTo(T t) — сравнивает элементы на больше, меньше либо равно.

Пример создания объекта:

```

Role role = Role.valueOf("client".toUpperCase());

```

Класс перечисления может объявлять собственные методы, и, следовательно, экземпляры перечисления могут к этим методам обращаться. Перечисления представляют собой классы, а классам положено явно или неявно объявлять конструкторы.

```

/* # 27 # перечисление с полями и методами # Role.java */

```

```

package by.epam.learn.type;
public enum Role {
    GUEST("guest"), CLIENT("client"), MODERATOR("moderator"), ADMIN("administrator");
    private String typeName;
    Role(String typeName) {
        this.typeName = typeName;
    }
}

```

```
/* # 28 # объявление перечисления с методом # Shape.java */
```

```
package by.epam.learn.type;
public enum Shape {
    RECTANGLE, TRIANGLE, CIRCLE;
    public double defineSquare(double ... x) {
        // here may be checking the parameters for correctness
        double area;
        switch (this) {
            case RECTANGLE:
                area = x[0] * x[1];
                break;
            case TRIANGLE:
                area = x[0] * x[1] / 2;
                break;
            case CIRCLE:
                area = Math.pow(x[0], 2) * Math.PI;
                break;
            default:
                throw new EnumConstantNotPresentException(getDeclaringClass(), name());
        }
        return area;
    }
}
```

```
/* # 29 # применение перечисления # ShapeMain.java */
```

```
package by.epam.learn.type;
public class ShapeMain {
    public static void main(String args[ ]) {
        double x = 2, y = 3;
        Shape sh;
        sh = Shape.RECTANGLE;
        System.out.printf("%9s = %5.2f\n", sh, sh.defineSquare (x, y));
        sh = Shape.valueOf(Shape.class, "TRIANGLE");
        System.out.printf("%9s = %5.2f\n", sh, sh.defineSquare (x, y));
        sh = Shape.CIRCLE;
        System.out.printf("%9s = %5.2f\n", sh, sh.defineSquare (x));
    }
}
```

В результате будет выведено:

```
RECTANGLE = 6,00
TRIANGLE = 3,00
CIRCLE = 12,57
```

Каждый из элементов перечисления в данном случае содержит арифметическую операцию, ассоциированную с методом **defineSquare()**. Без **throw** данный код не будет компилироваться, так как компилятор не исключает появления

неизвестного элемента. Данная инструкция позволяет указать на возможную ошибку при появлении необъявленной фигуры. Поэтому и при введении нового элемента желательно добавлять соответствующий ему **case**.

Перечисление является классом, поэтому в его теле можно объявлять, кроме методов, также поля и конструкторы. Все конструкторы вызываются автоматически при инициализации любого из элементов. Конструктор не может быть объявлен со спецификаторами **public** и **protected**, так как не вызывается явно и перечисление не может быть суперклассом. Поля перечисления используются для сохранения дополнительной информации, связанной с его элементом.

Метод **toString()** реализован в классе **Enum** для вывода элемента в виде строки. Если переопределить метод **toString()** в конкретной реализации перечисления, то можно выводить не только значение элемента, но и значения его полей, то есть предоставить полную информацию об объекте, как и определяется контрактом метода.

Метод **ordinal()** определяет порядковый номер элемента перечисления, обратная задача не совсем тривиальна, но решается. Определение элемента перечисления по позиции с применением метода **values()** преобразования элементов перечисления в массив:

```
public static Optional<Role> valueOf(int roleId) {
    return Arrays.stream(values())
        .filter(roleType -> roleType.ordinal() == roleId)
        .findFirst();
}
```

При разработке класса перечисления возможны различные ошибки. В частности:

```
enum WrongPlanType {
    AIRBUS_A320, AIRBUS_A380, AIRBUS_A330, BOEING_737_800, BOEING_777
}
class WrongPlane {
    private WrongPlanType type;
}
```

Для описания типов и марок самолетов создано перечисление, но такой класс в итоге может оказаться очень большим из-за количества моделей конкретного производителя. Решить проблему легко разделением ответственности за хранение типа производителя в перечислении, а марки модели в обычной строке:

```
enum PlanType {
    AIRBUS, BOEING
}
class Plane {
    private PlanType type;
    private String model;
}
```

На перечисления накладывается целый ряд ограничений. Им запрещено:

- быть суперклассами;
- быть подклассами;
- быть абстрактными;
- быть параметризованными;
- создавать экземпляры, используя ключевое слово **new**.

Immutable и record

Если в системе необходим объект, внутреннее состояние которого нельзя изменить, то процедура реализации такой задачи представляется в виде:

```
/* # 30 # класс для создания неизменяемых объектов # ImmutableType.java */

package by.bsu.learn.immutable;
public final class ImmutableType {
    private String name;
    private int id;
    public ImmutableType(String name, int id) {
        this.name = name;
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public int getId() {
        return id;
    }
}
// equals, hashCode, toString
}
```

Такой объект от создания и до уничтожения не может быть изменен, что уменьшает затраты на безопасность при использовании в конкурирующих операциях. Классов с неизменяемым внутренним состоянием в стандартной реализации Java достаточно много. Следует вспомнить класс **String**.

Класс с поведением **Immutable**, тем не менее, может содержать методы для создания объекта того же типа с внутренним состоянием, отличающимся от исходного, что оправданно с точки зрения ресурсов, только если такие изменения происходят не слишком часто.

Если полем такого *immutable* класса необходимо объявить изменяемый тип, то следует предусмотреть, чтобы соответствующий ему *get*-тер возвращал копию или неизменяемый объект, а не ссылку на объект. Например, если поле такого класса объявлено как

```
List<String> strings;
```

то метод может выглядеть так:

```
public List<String> getStrings() {
    String[] arr = {};
    return List.of(strings.toArray(arr));
}
```

Ключевое слово **record** представляет еще один способ создать класс с неизменяемыми экземплярами. Этот класс является подклассом класса **java.lang.Record** и, естественно, не может наследовать другой класс, а также сам не может выступать в роли суперкласса. Реализация интерфейсов разрешается. Также класс **record** может объявлять собственные методы.

```
/* # 31 # класс-record для создания неизменяемых объектов # ImmutableRec.java */
```

```
package by.epam.learn.immutable;
public record ImmutableRec(String name, int id) {
    void method() {}
}
```

Такая запись гарантирует неизменяемость значений полей записи и избавляет от необходимости создавать конструктор, методы **equals(Object o)**, **hashCode()** и **toString()**, которые для **record** генерируются автоматически. Вместо геттеров генерируются методы с именем поля. В данном случае это **name()** и **id()**.

```
/* # 32 # методы класса-record # RecordMain.java */
```

```
package by.epam.learn.immutable;
public class RecordMain {
    public static void main(String[] args) {
        ImmutableRec object = new ImmutableRec("Jan", 777);
        System.out.println(object.id());
        System.out.println(object.name());
        System.out.println(object.toString());
        ImmutableRec object2 = new ImmutableRec("Jan", 777);
        System.out.println(object.equals(object2));
    }
}
```

В результате будет выведено:

```
777
Jan
ImmutableRec[name=Jan, id=777]
true
```

Декомпозиция

Корпоративные информационные системы предоставляют пользователю огромное количество сервисов и манипулируют очень большим количеством

самой разнообразной информации. В разработке таких сложных и многообразных систем участвуют зачастую сотни программистов-разработчиков. Такие системы состоят из большого количества подсистем и программных модулей, которые взаимодействуют между собой через ограниченное число интерфейсов (методов). Главное же заключается в том, что система, состоящая из сотен тысяч (миллионов) строк кода, не может создаваться, существовать, развиваться и поддерживаться, если при ее разработке не использовались принципы объектно-ориентированного программирования, в частности, декомпозиция.

Под *декомпозицией* следует понимать определение физических и логических сущностей, а также принципов их взаимодействия на основе анализа предметной области создаваемой системы.

Все системы состоят из классов. Все классы системы взаимодействуют с теми или иными классами этой же системы.

Объяснение принципов декомпозиции можно рассмотреть на простейшем примере. Пусть требуется решить следующую задачу: *создать систему, позволяющую умножать целочисленные матрицы друг на друга.*

Начинающий программист, знающий о том, что существуют классы, конструкторы и методы, может предложить решение поставленной проблемы в следующем виде:

```
/* # 33 # произведение двух матриц # Matrix.java */
```

```
public class Matrix {
    private int[][] a;
    private int n;
    private int m;
    public Matrix(int nn, int mm) {
        n = nn;
        m = mm;
        // creation and filling with random values
        a = new int[n][m];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                a[i][j] = (int)(Math.random() * 10);
            }
        }
        show();
    }
    public Matrix(int nn, int mm, int k) {
        n = nn;
        m = mm;
        a = new int[n][m];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                a[i][j] = k;
            }
        }
    }
}
```

```

        if(k != 0) {
            show();
        }
    }
}

public void show() {
    System.out.println("matrix : " + a.length + " by " + a[0].length);
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < a[0].length; j++) {
            System.out.print(a[i][j] + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    int n = 2, m = 3, l = 4;
    Matrix p = new Matrix(n, m);
    Matrix q = new Matrix(m, l);
    Matrix r = new Matrix(n, l, 0);
    for (int i = 0; i < p.a.length; i++) {
        for (int j = 0; j < q.a[0].length; j++) {
            for (int k = 0; k < p.a[0].length; k++) {
                r.a[i][j] += p.a[i][k] * q.a[k][j];
            }
        }
    }
    System.out.println("matrix multiplication result: ");
    r.show();
}
}

```

Программа полностью работоспособна, но следует взглянуть на нее внимательнее:

- создан только один класс, маловато, но и задача невелика;
- класс обладает лишними полями, значения которых зависят от значений других полей;
- в классе объявлены два конструктора, оба выделяют память под матрицу и заполняют ее элементами, переданными или сгенерированными. Оба конструктора решают похожие задачи и не проверяют на корректность входные значения, т.к. решают слишком обширные задачи;
- определен метод **show()** для вывода матрицы на консоль, что ограничивает способы общения класса с внешними для него классами;
- задача умножения решается в методе **main()**, и класс является одноразовым, т.е. для умножения двух других матриц придется код умножения копировать в другое место;
- реализован только основной положительный сценарий, например, не выполняется проверка размерности при умножении, и, как следствие, отсутствует реакция приложения на некорректные данные.

Ниже приведена попытка переработки (рефакторинга) созданного приложения таким образом, чтобы существовала возможность поддержки и расширения возможности системы при возникновении сопутствующих задач.

```
/* # 34 # класс хранения матрицы # Matrix.java */
```

```
package by.epam.learn.entity;
import by.epam.learn.exception.MatrixException;
public class Matrix {
    private int[][] a;
    public Matrix(int[][] a) {
        this.a = a;
    }
    public Matrix(int n, int m) throws MatrixException {
        if (n < 1 || m < 1) { // check input
            throw new MatrixException();
        }
        a = new int[n][m];
    }
    public int getVerticalSize() {
        return a.length;
    }
    public int getHorizontalSize() {
        return a[0].length;
    }
    public int getElement(int i, int j) throws MatrixException {
        if (checkRange(i, j)) { // check i & j
            return a[i][j];
        } else {
            throw new MatrixException();
        }
    }
    public void setElement(int i, int j, int value) throws MatrixException {
        if (checkRange(i, j)) {
            a[i][j] = value;
        } else {
            throw new MatrixException();
        }
    }
    @Override
    public String toString() {
        final String BLANK = " ";
        StringBuilder s = new StringBuilder("\nMatrix : " + a.length + "x"
            + a[0].length + "\n");
        for (int [ ] row : a) {
            for (int value : row) {
                s.append(value).append(BLANK);
            }
            s.append("\n");
        }
    }
}
```

```

        return s.toString();
    }
    private boolean checkRange(int i, int j) { // check matrix range
        if (i < 0 || i > a.length - 1 || j < 0 || j > a[0].length - 1) {
            return false;
        } else {
            return true;
        }
    }
}

```

В классе **Matrix** объявлен **private**-метод **checkRange(int i, int j)** для проверки параметров на предельные допустимые значения во избежание невынужденных ошибок. Однако условный оператор **if** в этом методе выглядит запутанным, во-первых, нарушая правило положительного сценария, то есть разумно ожидать, что параметры метода с индексами элемента матрицы будут корректными, и, исходя из этого, строить условие, а в представленном варианте все наоборот; во-вторых, при значении **true** в условии оператора метод возвращает значение **false**, что выглядит противоречивым.

Метод следует переписать в виде:

```

private boolean checkRange(int i, int j) {
    if (i >= 0 && i < a.length && j >= 0 && j < a[0].length) {
        return true;
    } else {
        return false;
    }
}

```

или, что еще лучше:

```
return (i >= 0 && i < a.length && j >= 0 && j < a[0].length);
```

заменяв условие в операторе на противоположное, возвращая непротиворечащее результату значение.

```
/* # 35 # класс-создатель матрицы # MatrixCreator.java */
```

```

package by.epam.learn.creator;
import by.epam.learn.entity.Matrix;
import by.epam.learn.exception.MatrixException;
public class MatrixCreator {
    public void fillRandomized(Matrix matrix, int minValue, int maxValue) {
        int v = matrix.getVerticalSize();
        int h = matrix.getHorizontalSize();
        for(int i = 0; i < v; i++) {
            for(int j = 0; j < h; j++) {
                try {
                    int value = (int) ((Math.random() * (maxValue - minValue)) + minValue);
                    matrix.setElement(i, j, value);
                }
            }
        }
    }
}

```

```

        } catch (MatrixException e) {
            // Log: exception impossible
        }
    }
}
}
// public int[][] createArray(int n, int m, int minValue, int maxValue) { /*code*/
// public void createFromFile(Matrix matrix, File f) { /* code */ }
// public void createFromStream(Matrix matrix, Stream stream) { /* code */ }
}

```

Инициализация элементов матрицы различными способами вынесена в отдельный класс, методы которого могут в зависимости от условий извлекать значения для инициализации элементов из различных источников.

```
/* # 36 # класс действия над матрицей # Multiplicator.java */
```

```

package by.epam.learn.action;
import by.epam.learn.entity.Matrix;
import by.epam.learn.exception.MatrixException;
public class Multiplicator {
    public Matrix multiply(Matrix p, Matrix q) throws MatrixException {
        int v = p.getVerticalSize();
        int h = q.getHorizontalSize();
        int controlSize = p.getHorizontalSize();
        if (controlSize != q.getVerticalSize()) {
            throw new MatrixException("incompatible matrices");
        }
        Matrix result = new Matrix(v, h);
        try {
            for (int i = 0; i < v; i++) {
                for (int j = 0; j < h; j++) {
                    int value = 0;
                    for (int k = 0; k < controlSize; k++) {
                        value += p.getElement(i, k) * q.getElement(k, j);
                    }
                    result.setElement(i, j, value);
                }
            }
        } catch (MatrixException e) {
            // Log: exception impossible
        }
        return result;
    }
}

```

Все манипуляции взаимодействия объектов-матриц между собой и с другими объектами должны быть сгруппированы и вынесены в отдельные логические классы, что дает возможность другому разработчику быстро разобраться в смысле класса и модернизировать его.

```
/* # 37 # исключительная ситуация при индексировании объекта-матрицы
# MatrixException.java */
```

```
package by.epam.learn.exception;
public class MatrixException extends Exception {
    public MatrixException() {
    }
    public MatrixException(String message) {
        super(message);
    }
    public MatrixException(String message, Throwable cause) {
        super(message, cause);
    }
    public MatrixException(Throwable cause) {
        super(cause);
    }
}
```

Создание собственных исключений позволяет разработчику при их возникновении быстро локализовать и исправить ошибку.

```
/* # 38 # класс, запускающий приложение # MatrixMain.java */
```

```
package by.epam.learn.main;
import by.epam.learn.action.Multiplier;
import by.epam.learn.creator.MatrixCreator;
import by.epam.learn.entity.Matrix;
import by.epam.learn.exception.MatrixException;
public class MatrixMain {
    public static void main(String[] args) {
        try {
            MatrixCreator creator = new MatrixCreator();
            Matrix p = new Matrix(2, 3);
            creator.fillRandomized(p, 2, 8);
            System.out.println("Matrix first is: " + p);
            Matrix q = new Matrix(3, 4);
            creator.fillRandomized(q, 2, 7);
            System.out.println("Matrix second is: " + q);
            Multiplier multiplier = new Multiplier();
            Matrix result = multiplier.multiply(p, q);
            System.out.println("Matrices product is " + result);
        } catch (MatrixException e) {
            System.err.println("Error of creating matrix " + e);
        }
    }
}
```

Одним из вариантов выполнения кода может быть следующий:

Matrix first is:

Matrix: 2x3

3 4 7

4 7 2

Matrix second is:

Matrix: 3x4

3 3 6 6

5 6 5 5

4 3 3 4

Matrices product is

Matrix: 2x4

57 54 59 66

55 60 65 67

Выше был приведен пример простейшей декомпозиции. При разработке приложений любой сложности всегда следует производить анализ предметной области, определять абстракции и разделять задачу на логические взаимодействующие части. Тем не менее границы, отделяющие хороший дизайн приложения от посредственного, достаточно размыты и зависят от общего уровня компетентности команды разработчиков и правил, принятых в проекте.

Рекомендации при проектировании классов

При создании класса следует придерживаться некоторых правил. Принцип единственной ответственности. Каждый класс должен иметь простое назначение. Решать в идеале единственную задачу. Классу следует давать такое имя, чтобы его пользователю была понятна роль класса в пакете или приложении. Если класс отвечает за хранение информации, то функциональность работы с этой информацией должна быть базовой. Манипулированием информацией через объект должны заниматься другие классы, которых может оказаться достаточно много.

Класс должен быть разработан так, чтобы внесение в него изменений было относительно простой задачей.

Код конструктора должен заниматься только инициализацией объекта. Следует избегать вызовов из конструктора других методов, за исключением **final**, **static**, **private**. Иначе такой метод может быть переопределен в подкласс и исказить процесс инициализации объекта.

Использовать инкапсуляцию нестатических и неконстантных полей.

Классы-сущности должны применять для доступа к полям классов, хранящих информацию, корректные методы типа **get**, **set**, **is**, а также желательно реализовать методы **equals()**, **hashCode()**, **clone()**, **toString()** и имплементировать интерфейсы **Comparable** и **Serializable**.

Если разрабатываемый класс кажется сложным, следует разбить его на несколько простых.

По возможности избегать слишком длинных методов. От 25–30-и строк — длинный метод. Следует, если это возможно, разбить метод на несколько, или даже создать для этой цели новый класс.

Если метод используется только другими методами этого класса, следует объявлять его как **private**.

Определить и распределить по разным классам функциональности, которые могут изменяться в процессе разработки, от тех, которые будут постоянными.

Хороший дизайн кода отличается высоким уровнем декомпозиции.

Если в разных участках класса или нескольких классов востребован один и тот же фрагмент кода, следует выделить его в отдельный метод.

Избегать длинного списка аргументов. Приближаясь к числу семь, список аргументов становится не воспринимаемым при чтении. Возможно, следует объединить группы аргументов в новый тип данных.

Не использовать «волшебные числа», «волшебные строки». Логичнее вынести их в **final static** атрибуты или за пределы кода, например, в файл.

Вопросы к главе 3

1. Дать определение таким понятиям как «класс» и «объект». Привести примеры объявления класса и создания объекта класса. Какие спецификаторы можно использовать при объявлении класса?
2. Как определить, какие поля и методы необходимо определить в классе? Привести пример. Какие спецификаторы можно использовать с полями, а какие с методами (и что они значат)?
3. Что такое конструктор? Как отличить конструктор от любого другого метода? Сколько конструкторов может быть в классе?
4. Что такое конструктор по умолчанию? Может ли в классе совсем не быть конструкторов? Объяснить, какую роль выполняет оператор **this()** в конструкторе?
5. Какова процедура инициализации полей класса и полей экземпляра класса? Когда инициализируются поля класса, а когда — поля экземпляров класса? Какие значения присваиваются полям по умолчанию? Где еще в классе полям могут быть присвоены начальные значения?
6. JavaBeans: основные требования к классам Bean-компонентов, соглашения об именах.
7. В каких областях памяти хранятся значения и объекты, массивы?
8. Дать определение перегрузке методов. Чем удобна перегрузка методов? Указать, какие методы могут перегружаться, и какими методами они могут быть перегружены?
9. Можно ли перегрузить методы в базовом и производном классах? Можно ли **private** метод базового класса перегрузить **public** методом производного?
10. Можно ли перегрузить конструкторы, и можно ли при перегрузке конструкторов менять атрибуты доступа у конструкторов?

11. Свойства конструктора. Способы его вызова.
12. Объяснить, что такое раннее и позднее связывание? Перегрузка — это раннее или позднее связывание?
13. Объяснить правила, которым следует компилятор при разрешении перегрузки; в том числе, если аргументы методов перегружаются примитивными типами, между которыми возможно неявное приведение, или ссылочными типами, состоящими в иерархической связи.
14. Перегрузка. Можно ли менять модификатор доступа метода, если да, то каким образом?
15. Перегрузка. Можно ли менять возвращаемый тип метода, если да, то как? Можно ли менять тип передаваемых параметров?
16. Объяснить, что такое неявная ссылка **this**? В каких методах эта ссылка присутствует, а в каких — нет, и почему?
17. Что такое финальные поля, какие поля можно объявить со спецификатором **final**? Где можно инициализировать финальные поля?
18. Что такое статические поля, статические финальные поля и статические методы. К чему имеют доступ статические методы? Можно ли перегрузить и переопределить статические методы? Наследуются ли статические методы?
19. Что такое логические и статические блоки инициализации? Сколько их может быть в классе, в каком порядке они могут быть размещены и в каком порядке вызываются?
20. Что представляют собой методы с переменным числом параметров, как передаются параметры в такие методы, и что представляет собой такой параметр в методе? Как осуществляется выбор подходящего метода, при использовании перегрузки для методов с переменным числом параметров?
21. Каким образом передаются переменные в методы, по значению или по ссылке?
22. **Mutable** и **Immutable** классы. Привести примеры. Как создать класс, который будет **immutable**?
23. Какие свойства у класса, объявленного как **record**?
24. Что такое **static**? Что будет, если значение атрибута изменить через объект класса? Всегда ли **static** поле содержит одинаковые значения для всех его объектов?
25. **Generics**. Что это такое и для чего применяются. Во что превращается во время компиляции и выполнения? Использование **wildcards**.
26. Что такое **enum**? Какими свойствами обладает? Область применения.
27. Класс **Optional**. Как помогает бороться с проблемой возвращения методом значения **null**?

Задания к главе 3

Вариант А

Создать классы, спецификации которых приведены ниже. Определить конструкторы и методы **setTun()**, **getTun()**, **toString()**. Определить дополнительно методы в классе, создающем массив объектов. Задать критерий выбора данных и вывести эти данные на консоль. В каждом классе, обладающем информацией, должно быть объявлено несколько конструкторов.

1. **Student:** id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Факультет, Курс, Группа.

Создать массив объектов. Вывести:

- а) список студентов заданного факультета;
- б) списки студентов для каждого факультета и курса;
- с) список студентов, родившихся после заданного года;
- д) список учебной группы.

2. **Customer:** id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета.

Создать массив объектов. Вывести:

- а) список покупателей в алфавитном порядке;
- б) список покупателей, у которых номер кредитной карточки находится в заданном интервале.

3. **Patient:** id, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, Диагноз.

Создать массив объектов. Вывести:

- а) список пациентов, имеющих данный диагноз;
- б) список пациентов, номер медицинской карты которых находится в заданном интервале.

4. **Abiturient:** id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.

Создать массив объектов. Вывести:

- а) список абитуриентов, имеющих неудовлетворительные оценки;
- б) список абитуриентов, у которых сумма баллов выше заданной;
- с) выбрать заданное число n абитуриентов, имеющих самую высокую сумму баллов (вывести также полный список абитуриентов, имеющих полу проходную сумму).

5. **Book:** id, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Тип переплета.

Создать массив объектов. Вывести:

- а) список книг заданного автора;
- б) список книг, выпущенных заданным издательством;
- с) список книг, выпущенных после заданного года.

6. **House:** id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации.

Создать массив объектов. Вывести:

- а) список квартир, имеющих заданное число комнат;
- б) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;
- с) список квартир, имеющих площадь, превосходящую заданную.

7. **Phone:** id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров.

Создать массив объектов. Вывести:

- а) сведения об абонентах, у которых время внутригородских разговоров превышает заданное;
- б) сведения об абонентах, которые пользовались междугородной связью;
- с) сведения об абонентах в алфавитном порядке.

8. **Car:** id, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер.

Создать массив объектов. Вывести:

- а) список автомобилей заданной марки;
- б) список автомобилей заданной модели, которые эксплуатируются больше n лет;
- с) список автомобилей заданного года выпуска, цена которых больше указанной.

9. **Product:** id, Наименование, UPC, Производитель, Цена, Срок хранения, Количество.

Создать массив объектов. Вывести:

- а) список товаров для заданного наименования;
- б) список товаров для заданного наименования, цена которых не превосходит заданную;
- с) список товаров, срок хранения которых больше заданного.

10. **Train:** Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс).

Создать массив объектов. Вывести:

- а) список поездов, следующих до заданного пункта назначения;
- б) список поездов, следующих до заданного пункта назначения и отправляющихся после заданного часа;
- с) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.

11. **Bus:** Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег.

Создать массив объектов. Вывести:

- а) список автобусов для заданного номера маршрута;
- б) список автобусов, которые эксплуатируются больше заданного срока;
- с) список автобусов, пробег у которых больше заданного расстояния.

12. **Airline:** Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели.

Создать массив объектов. Вывести:

- а) список рейсов для заданного пункта назначения;
- б) список рейсов для заданного дня недели;
- с) список рейсов для заданного дня недели, время вылета для которых больше заданного.

Вариант В

Реализовать методы сложения, вычитания, умножения и деления объектов (для тех классов, объекты которых могут поддерживать арифметические действия).

1. Определить класс **Дробь (Рациональная Дробь)** в виде пары чисел m и n . Объявить и инициализировать массив из k дробей, ввести/вывести значения для массива дробей. Создать массив/список/множество объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента.
2. Определить класс **Комплекс**. Создать массив/список/множество размерности n из комплексных координат. Передать его в метод, который выполнит сложение/умножение его элементов.
3. Определить класс **Квадратное уравнение**. Реализовать методы для поиска корней, экстремумов, а также интервалов убывания/возрастания. Создать массив/список/множество объектов и определить наибольшие и наименьшие по значению корни.
4. Определить класс **Полином** степени n . Объявить массив/список/множество из m полиномов и определить сумму полиномов массива.
5. Определить класс **Интервал** с учетом включения/невключения концов. Создать методы по определению пересечения и объединения интервалов, причем интервалы, не имеющие общих точек, пересекаются/объединяются не могут. Объявить массив/список/множество n интервалов и определить расстояние между самыми удаленными концами.
6. Определить класс **Точка** на плоскости (в пространстве) и во времени. Задать движение точки в определенном направлении. Создать методы по определению скорости и ускорения точки. Проверить для двух точек возможность пересечения траекторий. Определить расстояние между двумя точками в заданный момент времени.
7. Определить класс **Треугольник** на плоскости. Определить площадь и периметр треугольника. Создать массив/список/множество объектов и подсчитать количество треугольников разного типа (равносторонний, равнобедренный, прямоугольный, произвольный). Определить для каждой группы наибольший и наименьший по площади (периметру) объект.
8. Определить класс **Четырехугольник** на плоскости. Определить площадь и периметр четырехугольника. Создать массив/список/множество объектов и подсчитать количество четырехугольников разного типа (квадрат,

- прямоугольник, ромб, произвольный). Определить для каждой группы наибольший и наименьший по площади (периметру) объект.
9. Определить класс **Окружность** на плоскости. Определить площадь и периметр. Создать массив/список/множество объектов и определить группы окружностей, центры которых лежат на одной прямой. Определить наибольший и наименьший по площади (периметру) объект.
 10. Определить класс **Прямая** на плоскости (пространстве). Определить точки пересечения прямой с осями координат. Определить координаты пересечения двух прямых. Создать массив/список/множество объектов и определить группы параллельных прямых.

Вариант С

1. Определить класс **Полином** с коэффициентами типа **Рациональная Дробь**. Объявить массив/список/множество из n полиномов и определить сумму полиномов массива.
2. Определить класс **Прямая** на плоскости (в пространстве), параметры которой задаются с помощью **Рациональной Дробь**. Определить точки пересечения прямой с осями координат. Определить координаты пересечения двух прямых. Создать массив/список/множество объектов и определить группы параллельных прямых.
3. Определить класс **Полином** с коэффициентами типа **Комплексное число**. Объявить массив/список/множество из m полиномов и определить сумму полиномов массива.
4. Определить класс **Дробь** в виде пары (m, n) с коэффициентами типа **Комплексное число**. Объявить и инициализировать массив из k дробей, ввести/вывести значения для массива дробей. Создать массив/список/множество объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента.
5. Определить класс **Комплекс**, действительная и мнимая часть которой представлены в виде **Рациональной Дробь**. Создать массив/список/множество размерности n из комплексных координат. Передать его в метод, который выполнит сложение/умножение его элементов.
6. Определить класс **Окружность** на плоскости, координаты центра которой задаются с помощью **Рациональной Дробь**. Определить площадь и периметр. Создать массив/список/множество объектов и определить группы окружностей, центры которых лежат на одной прямой. Определить наибольший и наименьший по площади (периметру) объект.
7. Определить класс **Точка** в пространстве, координаты которой задаются с помощью **Рациональной Дробь**. Создать методы по определению расстояния между точками и расстояния до начала координат. Проверить для трех точек возможность нахождения на одной прямой.

8. Определить класс **Точка** в пространстве, координаты которой задаются с помощью **Комплексного числа**. Создать методы по определению расстояния между точками и расстояния до начала координат.
9. Определить класс **Треугольник** на плоскости, вершины которого имеют тип **Точка**. Определить площадь и периметр треугольника. Создать массив/список/множество объектов и подсчитать количество треугольников разного типа (равносторонний, равнобедренный, прямоугольный, произвольный). Определить для каждой группы наибольший и наименьший по площади (периметру) объект.
10. Определить класс **Четырехугольник** на плоскости, вершины которого имеют тип **Точка**. Определить площадь и периметр четырехугольника. Создать массив/список/множество объектов и подсчитать количество четырехугольников разного типа (квадрат, прямоугольник, ромб, произвольный). Определить для каждой группы наибольший и наименьший по площади (периметру) объект.
11. Определить класс **Вектор**. Реализовать методы инкремента, декремента, индексирования. Определить массив из m объектов. Каждую из пар векторов передать в методы, возвращающие их скалярное произведение и длины. Вычислить и вывести углы между векторами.
12. Определить класс **Вектор**. Реализовать методы для вычисления модуля вектора, скалярного произведения, сложения, вычитания, умножения на константу. Объявить массив объектов. Написать метод, который для заданной пары векторов будет определять, являются ли они коллинеарными или ортогональными.
13. Определить класс **Вектор** в $\mathbb{R}^3$. Реализовать методы для проверки векторов на ортогональность, проверки пересечения неортогональных векторов, сравнения векторов. Создать массив из m объектов. Определить компланарные векторы.
14. Определить класс **Булева матрица (BoolMatrix)**. Реализовать методы для логического сложения (дизъюнкции), умножения и инверсии матриц. Реализовать методы для подсчета числа единиц в матрице и упорядочения строк в лексикографическом порядке.
15. Построить класс **Булев вектор (BoolVector)**. Реализовать методы для выполнения поразрядных конъюнкции, дизъюнкции и отрицания векторов, а также подсчета числа единиц и нулей в векторе.
16. Определить класс **Множество символов**. Реализовать методы для определения принадлежности заданного элемента множеству; пересечения, объединения, разности двух множеств. Создать методы сложения, вычитания, умножения (пересечения), индексирования, присваивания. Создать массив объектов и передавать пары объектов в метод другого класса, который строит множество, состоящее из элементов, входящих только в одно из заданных множеств.

17. Определить класс **Определенный интеграл** с аналитической подынтегральной функцией. Создать методы для вычисления значения по формуле левых прямоугольников, по формуле правых прямоугольников, по формуле средних прямоугольников, по формуле трапеций, по формуле Симпсона (параболических трапеций).
18. Определить класс **Массив**. Создать методы сортировки: обменная сортировка (метод пузырька); обменная сортировка «Шейкер-сортировка», сортировка посредством выбора (метод простого выбора), сортировка вставками: метод хеширования (сортировка с вычислением адреса), сортировка вставками (метод простых вставок), сортировка бинарного слияния, сортировка Шелла (сортировка с убывающим шагом).

Тестовые задания к главе 3

Вопрос 3.1.

Дано объявление класса: `class A{}`. Выбрать корректное объявление конструктора (выбрать один).

- a) `A() {this.super();}`
- b) `A() {Object.super();}`
- c) `A() {A.super();}`
- d) `A() {}`

Вопрос 3.2.

Дан код:

```
class Base{
    void method(int i) {
        System.out.print(" int ");
    }
    void method(long i) {
        System.out.print(" long ");
    }
}
public class Main {
    public static void main(String[] args) {
        Base base = new Base();
        base.method(1L);
        base.method(1_000_000);
    }
}
```

Что будет выведено в результате компиляции и выполнения кода? (выбрать один)

- a) `int int`
- b) `long long`

- c) int long
- d) long int
- e) compilation fails

Вопрос 3.3.

Какие из следующих объявлений представляют корректное объявление метода? (выбрать три)

- a) protected abstract void method();
- b) final static void method(){}
- c) public protected void method(){}
- d) default void method();
- e) private final void method(){}
- f) public static method(){}

Вопрос 3.4.

Какие из следующих объявлений представляют корректное объявление класса, объявленного в пакете? (выбрать два)

- a) final abstract class Type {}
- b) public static class Type {}
- c) final public class Type {}
- d) protected abstract class Type {}
- e) class Type {}
- f) abstract default class Type {}

Вопрос 3.5.

Дан код:

```
class Hope {
    static void action(){
        System.out.print(1);
    }
}
class Quest {
    static Hope hope;
    public static void main(String[] args) {
        hope.action();
    }
}
```

Каким будет результат компиляции и запуска приложения? (выбрать один)

- a) compilation fails
- b) NullPointerException при запуске
- c) 1
- d) null

Вопрос 3.6.

Дан код:

```
public class A {
    int a;
    A(int a) {
        this.a = a;
    }
}

public class Quest {
    public static void main(String[] args) {
        A a1 = new A(0);
        A a2 = new A(0);
        System.out.print(a1 == a2);
        System.out.print(", " + a1.equals(a2));
        System.out.print(", " + (a1.hashCode()==a2.hashCode()));
    }
}
```

Каким будет результат компиляции и запуска приложения? (выбрать один)

- a) false, true, false
- b) false, false, false
- c) false, true, true
- d) false, false, true
- e) true, true, false

Вопрос 3.7.

Дан код:

```
class GenericNum<T extends Number> {
    T number;
    GenericNum(T t) {
        number = t;
    }
    T get() {
        return number;
    }
}

class GenericsMain {
    public static void main(String[] args) {
        GenericNum<Integer> i1 = new GenericNum<>(500);
        GenericNum<Integer> i2 = new GenericNum<>(500);
        System.out.print(i1.get() == i2.get());
        System.out.print(i1.get().intValue() == i2.get().intValue());
    }
}
```

Каким будет результат компиляции и запуска программы? (выбрать один)

- a) falsefalse
- b) falsetrue
- c) truefalse
- d) truetrue

Вопрос 3.8.

Дан код:

```
public class Quest9 {
    enum Numbers {ONE, TWO, THREE, FOUR, FIVE}
    public static void main(String[] args) {
        Numbers n = Numbers.ONE;
    }
}
```

Сколько раз будет вызван конструктор перечисления Numbers?

- a) 0
- b) 1
- c) 5

Вопрос 3.9.

Какие из перечисленных методов класса Object являются final-методами?

(выбрать четыре)

- a) getClass()
- b) finalize()
- c) notify()
- d) wait()
- e) notifyAll()
- f) clone()

НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Если делегированию полномочий уделять внимание, ответственность накопится внизу, подобно осадку.

Закон делегирования Раска

Наследование

Отношение между классами, при котором характеристики одного класса (суперкласса) передаются другому классу (подклассу) без необходимости их повторного определения, называется *наследованием*.

Подкласс наследует поля и методы суперкласса, используя ключевое слово **extends**. Класс может также реализовать любое число интерфейсов, используя ключевое слово **implements**. Подкласс имеет прямой доступ ко всем открытым переменным и методам родительского класса, как будто они находятся в подклассе. Исключения составляют члены класса, помеченные **private** (во всех случаях) и «по умолчанию» для подкласса в другом пакете. В любом случае (даже если ключевое слово **extends** отсутствует) класс автоматически наследует свойства суперкласса всех классов — класса **Object**.

Множественное наследование классов запрещено, аналог предоставляет реализация интерфейсов, которые не являются классами и содержат описание набора методов, задающих поведение объекта класса, реализующего эти интерфейсы. Наличие общих методов, которые должны быть реализованы в разных классах, обеспечивают им сходную функциональность.

Подкласс дополняет члены суперкласса своими полями и/или методами и/или переопределяет методы суперкласса. Если имена методов совпадают, а параметры различаются, то такое явление называется перегрузкой методов (статическим полиморфизмом).

Если же совпадают имена и параметры методов, то этот механизм называется динамическим полиморфизмом. То есть в подклассе можно объявить (переопределить) метод с тем же именем, списком параметров и возвращаемым значением, что и у метода суперкласса.

Способность ссылки динамически определять версию переопределенного метода в зависимости от переданного по ссылке типа объекта называется *полиморфизмом*.

Полиморфизм является основой для реализации механизма динамического или «позднего связывания».

В следующем классе переопределяемый метод **doPayment()** находится в суперклассе **CardAction** и его подклассе **CreditCardAction**. В соответствии с принципом полиморфизма по ссылке вызывается метод наиболее близкий к текущему объекту.

```
/* # 1 # наследование класса и переопределение метода # CardAction.java
# CreditCardAction.java # CardRunner.java */
```

```
package by.epam.learn.inheritance;
public class CardAction {
    public void doPayment(double amountPayment) {
        System.out.println("complete from debt card");
    }
}

package by.epam.learn.inheritance;
public class CreditCardAction extends CardAction {
    @Override // is used when override a method in sub class
    public void doPayment(double amountPayment) { // override method
        System.out.println("complete from credit card");
    }
    public boolean checkCreditLimit() { // own method
        return true;
    }
}

package by.epam.learn.inheritance;
public class CardRunner {
    public static void main(String[] args) {
        CardAction action1 = new CardAction();
        CardAction action2 = new CreditCardAction();
        CreditCardAction cc = new CreditCardAction();
        // CreditCardAction cca = new CardAction(); // compile error: class cast
        action1.doPayment(15.5); // method of CardAction
        action2.doPayment(21.2); // polymorphic method: CreditCardAction
        // dc2.checkCreditLimit(); // compile error: non-polymorphic method
        ((CreditCardAction) action2).checkCreditLimit(); // ok
        cc.doPayment(7.0); // polymorphic method: CreditCardAction
        cc.checkCreditLimit(); // non-polymorphic method CreditCardAction
        ((CreditCardAction) action1).checkCreditLimit(); // runtime error: class cast
    }
}
```

Объект по ссылке **action1** создается при помощи вызова конструктора класса **CardAction** и, соответственно, при вызове метода **doPayment()** вызывается версия метода из класса **CardAction**. При создании объекта **action2** ссылка типа **CardAction** инициализируется объектом типа **CreditCardAction**. При

таким способом инициализации ссылка на суперкласс получает доступ к методам, переопределенным в подклассе.

При объявлении совпадающих по сигнатуре (имя, тип, область видимости) полей в суперклассе и подклассах их значения не переопределяются и никак не пересекаются, т.е. существуют в одном объекте независимо друг от друга. Такое решение является плохим примером кода, который не используется в практическом программировании. Не следует использовать вызов методов, которые можно переопределить, в конструкторе. Это действие может привести к некорректной работе конструктора при инициализации полей объекта и в целом некачественному созданию объекта. Для доступа к полям текущего объекта можно использовать указатель **this**, для доступа к полям суперкласса — указатель **super**.

К чему может привести вызов полиморфных методов в конструкторе и объявление идентичных полей иерархии наследования рассмотрено ниже:

```
/* # 2 # Вызов полиморфного метода из конструктора # Dumb.java # Dumber.java */

package by.epam.learn.inheritance;
class Dumb {
    {
        this.id = 6;
    }
    int id;
    Dumb() {
        System.out.println("constructor Dumb ");
        id = getId(); // ~ this.getId(); // ~ Dumb.this.getId();
        System.out.println(" id=" + id);
    }
    int getId() { // 1
        System.out.println("getId() of Dumb ");
        return id;
    }
}
class Dumber extends Dumb {
    int id = 9;
    Dumber() {
        System.out.println("constructor Dumber");
        id = this.getId();
        System.out.println(" id=" + id);
    }
    @Override
    int getId() { // 2
        System.out.println("getId() of Dumber");
        return id;
    }
}
```

В результате создания экземпляра **Dumb dumb = new Dumber()** последовательно будет выведено:

constructor Dumb

getId() of Dumber

id=0

constructor Dumber

getId() of Dumber

id=9

Метод **getId()** объявлен в классе **Dumb** и переопределен в его подклассе **Dumber**. Перед вызовом конструктора **Dumber()** вызывается конструктор класса **Dumb**. Но так как создается объект класса **Dumber**, то вызывается ближайший к создаваемому объекту метод **getId()**, объявленный в классе **Dumber**, который, в свою очередь, оперирует полем **id**, еще не проинициализированным для класса **Dumber**. В результате **id** получит значение по умолчанию, т.е. ноль.

Разработчик класса **Dumb** предполагал, что объект будет создаваться по его правилам, и метод будет работать так всегда. Но если метод переопределен в подклассе, то, соответственно, и в конструкторе суперкласса будет вызвана переопределенная версия, которая может выполнять совершенно другие действия, и создание объекта пойдет уже по другому пути.

Поля с одинаковыми именами в подклассе не замещаются. Объект подкласса будет иметь в итоге два поля с одинаковыми именами. У разработчика появится проблема их различить. Воспользовавшись преобразованием типов вида **((Dumber) dumb).id**, можно получить доступ к полю **id** из подкласса, но это в случае открытого доступа. Если поле приватное, то эта простая задача становится проблемой.

Практического смысла в одинаковых именах полей в иерархии наследования не просматривается. Это просто ошибка проектирования, которой следует избегать.

Методы, объявленные как **private**, не переопределяются.

```
/* # 3 # попытка наследования приватного метода # Dumb.java # Dumber.java */
```

```
class Dumb {
    private void action() {
        System.out.println("Dumb");
    }
}
class Dumber extends Dumb {
    @Override // compile error
    void action() {
        System.out.println("Dumber");
    }
}
```

Аннотация **@Override** будет выдавать ошибку компиляции из-за того, что она просто не видит переопределяемый метод.

Без этой аннотации код будет компилироваться, только цепочка переопределения будет начинаться с версии метода в подклассе и никак не будет связана с версией метода суперкласса.

Классы и методы **final**

Запрещено переопределять метод в порожденном классе, если в суперклассе он объявлен со спецификатором **final**:

```
/* # 4 # попытка переопределения final-метода # MasterCardAction.java
# VisaCardAction.java */

package by.epam.learn.inheritance;
public class MasterCardAction extends CreditCardAction{
    @Override// doPayment() cannot be polymorphic
    public final void doPayment(double amountPayment) {
        System.out.println("complete from MasterCard");
    }
}

package by.epam.learn.inheritance;
public class VisaCardAction extends MasterCardAction {
    //@@Override
    //public void doPayment(double amountPayment) {// impossible method
    //}
}
```

Если разработчик объявляет метод как **final**, следовательно, он считает, что его версия в этой ветви наследования метода окончательна и переопределение\совершенствованию не подлежит.

Что же общего у **final**-метода со статическим? Версия статического метода жестко определяется на этапе компиляции. И если **final**-метод вызван на объекте класса **MasterCardAction**, в котором он определен, или на объекте любого его подкласса, например, **VisaCardAction**, то также в точке вызова будет зафиксирован вызов именно этой версии метода.

```
/* # 5 # вызов полиморфного метода # CardService.java */

public class CardService {
    public void doService(CardAction action, double amount){
        action.doPayment(amount);
    }
}
```

При передаче в метод **doService()** объектов классов **CardAction** или **CreditCardAction** будут вызваны их собственные версии методов **doPayment()**, определенные в каждом из классов, что представляет собой еще одну иллюстрацию полиморфизма.

При передаче же в метод **doService()** объектов классов **MasterCardAction** или **VisaCardAction** будет вызвана версия метода **doPayment()**, определенная в классе **MasterCardAction**, так как в классе **VisaCardAction** метод не определен, то будет вызвана версия метода ближайшая по восходящей цепочке наследования.

Применение **final**-методов также показательно при разработке конструктора класса. Процесс инициализации экземпляра должен быть строго определен и не подвергаться изменениям. Исключить подмену реализации метода, вызываемого в конструкторе, следует объявлением метода как **final**, т.е. при этом метод не может быть переопределен в подклассе. Подобное объявление гарантирует обращение именно к этой реализации. Корректное определение вызова метода класса из конструктора представлено ниже:

```
/* # 6 # вызов нестатического final-метода из конструктора # AuthenticationService.java */

package by.epam.learn.service;
public class AuthenticationService {
    public AuthenticationService() {
        authenticate();
    }
    public final void authenticate() {
        //appeal to the database
    }
}
```

Если убрать **final** в объявлении метода суперкласса, то становится возможным переопределить метод в подклассе.

```
/* # 7 # переопределение не final-метода # NewAuthenticationService.java */

package by.epam.learn.service;
public class NewAuthenticationService extends AuthenticationService {
    @Override
    public void authenticate() {
        //empty method
    }
}
```

Тогда при создании объекта подкласса конструктор суперкласса вызовет версию метода из подкласса как самую близкую по типу создаваемого объекта, и проверка данных в БД не будет выполнена.

Рекомендуется при разработке классов из конструкторов вызывать методы, на которые не распространяются принципы полиморфизма. Метод может быть еще объявлен как **private** или **static** с таким же результатом.

Нельзя создать подкласс для класса, объявленного со спецификатором **final**:

```
public final class String { /* code */ }
class MegaString extends String { /* code */ } //impossible: compile error
```

Если необходимо создать собственную реализацию с возможностями **final**-класса, то создается класс-оболочка, где в качестве поля представлен **final**-класс. В свою очередь, необходимые или даже все методы делегируются из **final**-класса, и им придается необходимая разработчику функциональность.

Такой подход гарантирует невозможность прямого использования класса-оболочки вместо оборачиваемого класса и наоборот.

```
// # 8 # класс-оболочка для класса String # WrapperString.java
```

```
package by.epam.learn.string;
public class WrapperString {
    private String str;
    public WrapperString() {
        str = new String();
    }
    public WrapperString(String str) {
        this.str = str;
    }
    public int length() { // delegate method
        return str.length();
    }
    public boolean isEmpty() { // delegate method
        return str.isEmpty();
    }
    public int indexOf(int arg) { // delegate method
        int value = // new realization
        return value;
    }
    // other methods
}
```

Класс **WrapperString** не является наследником класса **String**, и его объект не может быть использован для передачи по ссылке на класс **String**. Класс **WrapperString**, в свою очередь, уже может быть суперклассом, поэтому его поведение можно изменять.

Использование **super** и **this**

Ключевое слово **super** применяется для обращения к конструктору супер-класса и для доступа к полю или методу суперкласса. Например:

```
super(parameters); // call to superclass constructor
super.id = 42; // superclass attribute reference
super.getId(); // superclass method call
```

Первая форма **super** применяется только в конструкторах для обращения к конструктору суперкласса только в качестве первой строки кода конструктора и только один раз.

Вторая форма **super** используется для доступа из подкласса к переменной **id** суперкласса. Третья форма специфична для Java и обеспечивает вызов из подкласса метода суперкласса, что позволяет избежать рекурсивного вызова в случае, если вызываемый с помощью **super** метод переопределен в данном подклассе. Причем, если в суперклассе этот метод не определен, то будет осуществляться поиск по цепочке наследования до тех пор, пока он не будет найден.

Во всех случаях с использованием **super** можно обратиться только к ближайшему суперклассу, т.е. «перескочить» через суперкласс, чтобы обратиться к его суперклассу, невозможно.

Следующий код показывает, как, используя **this**, можно строить одни конструкторы на основе использования возможностей других.

```
// # 9 # super и this в конструкторе # Point1D.java # Point2D.java # Point3D.java
```

```
package by.epam.learn.point;
public class Point1D {
    private int x;
    public Point1D(int x) {
        this.x = x;
    }
}

package by.epam.learn.point;
public class Point2D extends Point1D {
    private int y;
    public Point2D(int x, int y) {
        super(x);
        this.y = y;
    }
}

package by.epam.learn.point;
public class Point3D extends Point2D {
    private int z;
    public Point3D(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }
    public Point3D() {
        this(-1, -1, -1); // call Point3D constructor with parameters
    }
}
```

В классе **Point3D** второй конструктор для завершения инициализации объекта обращается к первому конструктору. Такая конструкция применяется в случае, когда в класс требуется добавить конструктор по умолчанию с обязательным использованием уже существующего конструктора.

Ссылка **this** используется, если в методе объявлены локальные переменные с тем же именем, что и переменные экземпляра класса. Локальная переменная имеет

преимущество перед полем класса и закрывает к нему доступ. Чтобы получить доступ к полю класса, требуется воспользоваться явной ссылкой **this** перед именем поля, так как поле класса является частью объекта, а локальная переменная нет.

Инструкция **this()** должна быть единственной в вызывающем конструкторе и быть первой по счету выполняемой операцией, иначе возникает возможность вызова нескольких конструкторов суперкласса или ветвления при обращении к конструктору суперкласса. Компилятор выполнять подобные действия запрещает.

Возможна ситуация с заикливанием обращений конструкторов друг к другу, что также запрещено:

```
// # 10 # ошибка с заикливанием вызова конструктора # Point1D.java
```

```
public class Point1D {
    private int x;

    public Point1D(int x) { // recursive constructor invocation
        this();
        this.x = x;
    }
    public Point1D() { // recursive constructor invocation
        this(42);
    }
}
```

Переопределение методов и полиморфизм

Способность Java делать выбор метода, исходя из типа объекта во время выполнения, называется «**поздним связыванием**». При вызове метода его поиск происходит сначала в данном классе, затем в суперклассе, пока метод не будет найден или не достигнут **Object** — суперкласс для всех классов.

Если два метода с одинаковыми именами и возвращаемыми значениями находятся в одном классе, то списки их параметров должны отличаться. То же относится к методам, наследуемым из суперкласса. Такие методы являются перегружаемыми (overloading). При обращении вызывается доступный метод, список параметров которого совпадает со списком параметров вызова.

Если объявление метода подкласса полностью, включая параметры, совпадает с объявлением метода суперкласса (порождающего класса), то метод подкласса переопределяет (overriding) метод суперкласса. Переопределение методов является основой концепции динамического связывания, реализующей полиморфизм. Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. Таким образом, тип объекта определяет версию метода на этапе выполнения. В следующем примере рассматривается реализация полиморфизма на основе динамического связывания. Так как

суперкласс содержит методы, переопределенные подклассами, то объект суперкласса будет вызывать методы различных подклассов в зависимости от того, на объект какого подкласса у него имеется ссылка.

```
/* # 11 # динамическое связывание методов # Point1D.java # Point2D.java
# Point3D.java # PointReport.java # PointMain.java */
```

```
package by.epam.learn.point;
public class Point1D {
    private int x;
    public Point1D(int x) {
        this.x = x;
    }
    public double length() {
        return Math.abs(x);
    }
    @Override
    public String toString() {
        return "x=" + x;
    }
}

package by.epam.learn.point;
public class Point2D extends Point1D {
    private int y;
    public Point2D(int x, int y) {
        super(x);
        this.y = y;
    }
    @Override
    public double length() {
        return Math.hypot(super.length(), y);
        /* just length() is impossible, because the method will call itself, which will
        lead to infinite recursion and an error at runtime */
    }
    @Override
    public String toString() {
        return super.toString() + " y=" + y;
    }
}

package by.epam.learn.point;
public class Point3D extends Point2D {
    private int z;
    public Point3D(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }
    public Point3D() {
        this(-1, -1, -1);
    }
}
```

```

@Override
public double length() {
    return Math.hypot(super.length(), z);
}
@Override
public String toString() {
    return super.toString() + " z=" + z;
}
}

package by.epam.learn.point;
public class PointReport {
    public void printReport(Point1D point) {
        System.out.printf("length=%.2f %s\n", point.length(), point);
        // out.print(point.toString()) is identical to out.print(point)
    }
}

package by.epam.learn.point;
public class PointMain {
    public static void main(String[] args) {
        PointReport report = new PointReport();
        Point1D point1 = new Point1D(-7);
        report.printReport(point1);
        Point2D point2 = new Point2D(3, 4);
        report.printReport(point2);
        Point3D point3 = new Point3D(3, 4, 5);
        report.printReport(point3);
    }
}

```

Результат:

length=7.00 x=-7

length=5.00 x=3 y=4

length=7.07 x=3 y=4 z=5

Аннотация **@Override** позволяет выделить в коде переопределенный метод и сгенерирует ошибку компиляции в случае, если программист изменит имя метода, типы его параметров или их количество в описании сигнатуры полиморфного метода.

Следует помнить, что при вызове метода обращение **super** всегда производится к ближайшему суперклассу. Переадресовать вызов, минуя суперкласс, невозможно! Аналогично при вызове **super()** в конструкторе обращение происходит к соответствующему конструктору непосредственного суперкласса.

Основной вывод: выбор версии переопределенного метода производит-ся на этапе выполнения кода.

Все нестатические методы Java являются виртуальными (ключевое слово **virtual**, как в C++, не используется).

Статические методы можно перегружать и «переопределять» в подклассах, но их доступность всегда зависит от типа ссылки и атрибута доступа, и никогда — от типа самого объекта.

Методы подставки

После выхода пятой версии языка появилась возможность при переопределении методов указывать другой тип возвращаемого значения, в качестве которого можно использовать только типы, находящиеся ниже в иерархии наследования, чем тип возвращаемого значения метода суперкласса.

```
/* # 12 # методы-подставки # Point1DCreator.java # Point2DCreator.java # PointMain.java */

package by.epam.learn.inheritance.service;
import by.epam.learn.point.Point1D;
import java.util.Random;
public class Point1DCreator {
    public Point1D create() {
        System.out.println("log in Point1DCreator");
        return new Point1D(new Random().nextInt(100));
    }
}

package by.epam.learn.inheritance.service;
import by.epam.learn.point.Point2D;
import java.util.Random;
public class Point2DCreator extends Point1DCreator {
    @Override
    public Point2D create() {
        System.out.println("log in Point2DCreator");
        Random random = new Random();
        return new Point2D(random.nextInt(10), random.nextInt(10));
    }
}

package by.epam.learn.inheritance;
import by.epam.learn.inheritance.service.Point1DCreator;
import by.epam.learn.inheritance.service.Point2DCreator;
import by.epam.learn.point.Point1D;
import by.epam.learn.point.Point2D;
public class PointMain {
    public static void main(String[] args) {
        Point1DCreator creator1 = new Point2DCreator();
        // Point2D point = creator1.create(); // compile error
        Point1D pointA = creator1.create(); /* when compiling - overLoad,
                                           when running - overriding */

        System.out.println(pointA);
        Point2DCreator creator2 = new Point2DCreator();
        Point2D pointB = creator2.create();
        System.out.println(pointB);
    }
}
```

В обоих случаях создания объекта будут созданы объекты класса **Point2D**:

log in Point2DCreator

x=5 y=4

log in Point2DCreator

x=2 y=7

В данной ситуации при компиляции в подклассе **Point2DCreator** создаются два метода **create()**. Один имеет возвращаемое значение **Point2D**, другой (явно невидимый) — **Point1D**. При обращении к методу **create()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении срабатывает полиморфизм и вызывается метод с возвращаемым значением **Point2D**.

Получается, что на этапе компиляции метод-подставка ведет себя как обычный метод, видимый по типу ссылки, а на этапе выполнения включается механизм переопределения и срабатывает метод из подкласса.

На практике методы подставки могут использоваться для расширения возможностей класса по прямому извлечению (без преобразования) объектов подклассов, инициализированных в ссылке на суперкласс.

«Переопределение» статических методов

Для статических методов принципы «позднего связывания» не работают. Динамический полиморфизм к статическим методам класса неприменим, так как обращение к статическому атрибуту или методу осуществляется по типу ссылки, а не по типу объекта, через который производится обращение. Версия вызываемого статического метода всегда определяется на этапе компиляции. При использовании ссылки для доступа к статическому члену компилятор при выборе метода учитывает тип ссылки, а не тип объекта, ей присвоенного.

```
/* # 13 # поведение статического метода при «переопределении» # StaticMain.java */
```

```
class StaticDumb {
    public static void go() {
        System.out.println("go() from StaticDumb ");
    }
}
class StaticDumber extends StaticDumb {
    // @Override - compile error
    public static void go() { // similar to dynamic polymorphism
        System.out.println("go() from StaticDumber ");
    }
}
class StaticMain {
    public static void main(String[] args) {
        StaticDumb dumb = new StaticDumber();
    }
}
```

```
dumb.go(); // warning: static member accessed via instance reference
StaticDumber dumber = null;
dumber.go(); // will not NullPointerException !
}
}
```

В результате выполнения данного кода будет выведено:

go() from StaticDumb
go() from StaticDumber

При таком способе инициализации объекта **dumb** метод **go()** будет вызван из класса **StaticDumb**. Если же спецификатор **static** убрать из объявления методов, то вызов методов будет осуществляться в соответствии с принципами полиморфизма.

Статические методы всегда следует вызывать через имя класса, в котором они объявлены, а именно:

```
StaticDumb.go();
StaticDumber.go();
```

Вызов статических методов через объект считается нетипичным и нарушающим смысл статического определения метода.

Абстракция

Множество моделей предметов реального мира обладают некоторым набором общих характеристик и правил поведения. Абстрактное понятие «Геометрическая фигура» может содержать описание геометрических параметров и расположения центра тяжести в системе координат, а также возможности определения площади и периметра фигуры. Однако в общем случае дать конкретную реализацию приведенных характеристик и функциональности невозможно ввиду слишком общего их определения. Для конкретного понятия, например, «Квадрат», дать описание линейных размеров и определения площади и периметра не составляет труда. Абстрагирование понятия должно предоставлять абстрактные характеристики предмета реального мира, а не его ожидаемую реализацию. Грамотное выделение абстракций позволяет структурировать код программной системы в целом и повторно использовать абстрактные понятия для конкретных реализаций при определении новых возможностей абстрактной сущности.

Абстрактные классы объявляются с ключевым словом **abstract** и содержат объявления абстрактных методов, которые не реализованы в этих классах, а будут реализованы в подклассах. Абстрактный класс может и не содержать вовсе абстрактных методов. Предназначение такого класса — быть вершиной иерархии его различных реализаций.

Объекты таких классов нельзя создать с помощью оператора **new**, но можно создать объекты подклассов, которые реализуют все эти методы. При этом

допустимо объявлять ссылку на абстрактный класс, но инициализировать ее можно только объектом производного от него класса. Абстрактные классы могут содержать и полностью реализованные методы, а также конструкторы и поля данных.

С помощью абстрактного класса объявляется контракт (требования к функциональности) для его подклассов. Примером может служить уже рассмотренный выше абстрактный класс **Number** и его подклассы **Byte**, **Float** и другие. Класс **Number** объявляет контракт на реализацию ряда методов по преобразованию данных к значению конкретного базового типа, например, **floatValue()**. Можно предположить, что реализация метода будет различной для каждого из классов-оболочек. Объект класса **Number** нельзя создать явно при помощи его собственного конструктора.

```
/* # 14 # абстрактный класс и метод # AbstractCardAction.java */
```

```
package by.epam.learn.inheritance;
public abstract class AbstractCardAction {
    private long actionId;
    public AbstractCardAction() {
    }
    public void check() {
    }
    public abstract void doPayment(double amountPayment);
}
```

```
/* # 15 # подкласс абстрактного класса # CreditCardAction.java # */
```

```
package by.epam.learn.inheritance;
public class CreditCardAction extends AbstractCardAction {
    @Override
    public void doPayment(double amountPayment) {
        // code
    }
}
```

Ссылка **action** на абстрактный суперкласс инициализируется объектом подкласса, в котором реализованы все абстрактные методы суперкласса:

```
AbstractCardAction action;
// action = new AbstractCardAction(); //compile error: cannot create object!
action = new CreditCardAction();
action.doPayment(7);
```

С помощью этой ссылки могут вызываться также и неабстрактные методы абстрактного класса, если они не переопределены в подклассе:

```
action.check();
```

Полиморфизм и расширение функциональности

В объектно-ориентированном программировании применение наследования предоставляет возможность расширения и дополнения программного обеспечения, имеющего сложную структуру с большим количеством классов и методов. В задачи суперкласса в этом случае входит определение интерфейса (набора методов), как способа взаимодействия для всех подклассов.

В следующем примере приведение к базовому типу происходит в выражении:

```
AbstractQuest quest1 = new DragnDropQuest();
AbstractQuest quest2 = new SingleChoiceQuest();
```

Суперкласс **AbstractQuest** предоставляет общий интерфейс (методы) для своих подклассов. Подклассы **DragnDropQuest** и **SingleChoiceQuest** переопределяют эти определения для обеспечения уникального поведения.

Небольшое приложение демонстрирует возможности полиморфизма.

```
/* # 16 # общий пример на полиморфизм # AbstractQuest.java # DragnDropQuest.
java # SingleChoiceQuest.java # Answer.java # QuestFactory.java # QuizMain.java */
```

```
package by.epam.learn.inheritance.quiz;
public abstract class AbstractQuest {
    private long questId;
    private String content;
    // constructors, methods
    public abstract boolean check(Answer answer);
}

package by.epam.learn.inheritance.quiz;
import java.util.Random;
public class DragnDropQuest extends AbstractQuest {
    // constructors, methods
    @Override
    public boolean check(Answer answer) {
        return new Random().nextBoolean(); // demo
    }
}

package by.epam.learn.inheritance.quiz;
import java.util.Random;
public class SingleChoiceQuest extends AbstractQuest {
    // constructors, methods
    @Override
    public boolean check(Answer answer) {
        return new Random().nextBoolean();
    }
}

package by.epam.learn.inheritance.quiz;
```

```

public class Answer {
    // fields, constructors, methods
}

package by.epam.learn.inheritance.quiz;
public class QuestFactory { // pattern Factory Method (simplest)
    public static AbstractQuest getQuestFromFactory(int mode) {
        switch (mode) {
            case 0:
                return new DragnDropQuest();
            case 1:
                return new SingleChoiceQuest();
            default:
                throw new IllegalArgumentException("illegal mode");
                // assert false; // bad
                // return null; // ugly
        }
    }
}

package by.epam.learn.inheritance.quiz;
import java.util.Random;
public class TestGenerator {
    public AbstractQuest[] generateTest(final int NUMBER_QUESTS, int maxMode) {
        AbstractQuest[] test = new AbstractQuest[NUMBER_QUESTS];
        for (int i = 0; i < test.length; i++) {
            int mode = new Random().nextInt(maxMode); // stub
            test[i] = QuestFactory.getQuestFromFactory(mode);
        }
        return test;
    }
}

package by.epam.learn.inheritance.quiz;
public class TestAction {
    public int checkTest(AbstractQuest[] test) {
        int counter = 0;
        for (AbstractQuest s : test) {
            // вызов полиморфного метода
            counter = s.check(new Answer()) ? ++counter : counter;
        }
        return counter;
    }
}

package by.epam.learn.inheritance.quiz;
public class QuizMain {
    public static void main(String[] args) {
        TestGenerator generator = new TestGenerator();
        AbstractQuest[] test = generator.generateTest(60, 2); // 60 questions of 2 types
        // here should be the code of the test process ...
        TestAction action = new TestAction();
        int result = action.checkTest(test);
        System.out.println(result + " correct answers, " + (60 - result) + " incorrect");
    }
}

```

В процессе выполнения приложения будет случайным образом сформирован массив-тест из вопросов разного типа, будет выполнена проверка теста, а общая информация об ответах на них будет выведена на консоль:

27 correct answers, 33 incorrect

Класс **QuestFactory** содержит метод **getQuestFromFactory(int numMode)**, который возвращает ссылку на случайно выбранный объект подкласса класса **AbstractQuest** каждый раз, когда он вызывается. Приведение к базовому типу производится оператором **return**, который возвращает ссылку на **DragnDropQuest** или **SingleChoiceQuest**. Метод **main()** содержит массив из ссылок **AbstractQuest**, заполненный с помощью вызова **getQuestFromFactory()**. На этом этапе известно, что имеется некоторое множество ссылок на объекты базового типа и ничего больше (не больше, чем знает компилятор). Когда происходит перемещение по этому массиву, метод **check()** вызывается для каждого случайным образом выбранного объекта.

Если понадобится в дальнейшем добавить в систему, например, класс **MultiplyChoiceQuest**, то это потребует только переопределения метода **check()** и добавления одной строки в код метода **getQuestFromFactory()**, что делает систему легко расширяемой.

Невозможно приравнять ссылки на классы, находящиеся в разных ветвях наследования, так как не существует никакого способа привести один такой тип к другому.

Класс Object

На вершине иерархии классов находится класс **Object**, суперкласс для всех классов. Изучение класса **Object** и его методов необходимо, т.к. его свойствами обладают все классы Java. Ссылочная переменная типа **Object** может указывать на объект любого другого класса, на любой массив, так как массив реализован как класс-наследник **Object**.

В классе **Object** определен набор методов, который наследуется всеми классами:

- **protected Object clone()** — создает и возвращает копию вызывающего объекта;
- **public boolean equals(Object ob)** — предназначен для использования и переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов одного и того же типа;
- **public Class<? extends Object> getClass()** — возвращает экземпляр типа **Class**;
- **protected void finalize()** — (deprecated) автоматически вызывается сборщиком мусора (garbage collection) перед уничтожением объекта;
- **public int hashCode()** — вычисляет и возвращает хэш-код объекта (число, в общем случае вычисляемое на основе значений полей объекта);
- **public String toString()** — возвращает представление объекта в виде строки.

Методы **notify()**, **notifyAll()** и **wait()**, **wait(int millis)** будут рассмотрены в главе «Потоки выполнения».

Если при создании класса предполагается проверка логической эквивалентности объектов, которая не выполнена в суперклассе, следует переопределить два метода: **boolean equals(Object ob)** и **int hashCode()**. Кроме того, переопределение этих методов необходимо, если логика приложения предусматривает использование элементов в коллекциях. Метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь — в противном случае. Реализация метода в классе **Object** возвращает истину только в том случае, если обе ссылки указывают на один и тот же объект, а конкретно:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

При переопределении метода **equals()** должны выполняться соглашения, предусмотренные спецификацией языка Java, а именно:

- рефлексивность — объект равен самому себе;
- симметричность — если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- транзитивность — если метод **equals()** возвращает значение **true** при сравнении объектов **x** и **y**, а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- непротиворечивость — при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- ненулевая ссылка при сравнении с литералом **null** всегда возвращает значение **false**.

При создании информационных классов также рекомендуется переопределить методы **hashCode()** и **toString()**, чтобы адаптировать их действия для создаваемого типа.

Метод **int hashCode()** переопределен, как правило, в каждом классе и возвращает число, являющееся уникальным идентификатором объекта, зависящим в большинстве случаев только от значения объекта. Его следует переопределять всегда, когда переопределен метод **equals()**. Метод **hashCode()** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

- все одинаковые по содержанию объекты *одного типа* **должны** иметь одинаковые хэш-коды;
- различные по содержанию объекты *одного типа* **могут** иметь различные хэш-коды;
- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен.

Один из способов создания правильного метода **hashCode()**, гарантирующий выполнение соглашений, приведен ниже для класса **Student**.

Метод **toString()** следует переопределять таким образом, чтобы, кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (т.е. всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**. Метод **toString()** класса **Object** возвращает строку с описанием объекта в виде:

getClass().getName() + '@' + Integer.toHexString(hashCode())

Метод вызывается автоматически, когда объект передается в поток вывода методами **println()**, **print()** и некоторыми другими.

```
/* # 17 # переопределение методов equals(), hashCode(), toString() # Student.java */
```

```
package by.epam.learn.entity;

public class Student {
    private int id;
    private String name;
    private int yearOfStudy;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getYearOfStudy() {
        return yearOfStudy;
    }
    public void setYearOfStudy(int yearOfStudy) {
        this.yearOfStudy = yearOfStudy;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        Student student = (Student) o;
        if (id != student.id) {
```

```

        return false;
    }
    if (yearOfStudy != student.yearOfStudy) {
        return false;
    }
    return name != null ? name.equals(student.name) : student.name == null;
}
@Override
public int hashCode() {
    return id + 31 * yearOfStudy + (name != null ? name.hashCode() : 0);
}

@Override
public String toString() {
    final StringBuilder sb = new StringBuilder("Student{");
    sb.append("id=").append(id);
    sb.append(", name=").append(name).append('\n');
    sb.append(", yearOfStudy=").append(yearOfStudy);
    sb.append('}');
    return sb.toString();
}
}

```

Выражение **id + 31 * yearOfStudy** гарантирует различные результаты вычислений при перемене местами значений полей, а именно: если **id=1** и **yearOfStudy=2**, то в результате будет получено **33**, если значения поменять местами, то **63**. Такой подход применяется при наличии у классов полей базовых типов.

Формально всем трем правилам определения метода **hashCode()** удовлетворяет и такая реализация:

```

public int hashCode() {
    return 42;
}

```

но все же следует признавать корректной ту реализацию метода, при которой для различных по содержанию объектов значения хэш-кодов были различными.

Метод **equals()** переопределяется для класса **Student** таким образом, чтобы убедиться в том, что полученный объект является объектом типа **Student**, а также сравнить содержимое полей **id**, **name** и **yearOfStudy** соответственно у вызывающего метод объекта и объекта, передаваемого в качестве параметра. Для подкласса всегда придется создавать собственную реализацию метода.

Клонирование объектов

Объекты в методы передаются по ссылке, в результате чего метод получает ссылку на объект, находящийся вне метода. Если в методе изменить значение поля объекта, это изменение коснется исходного объекта. Во избежание такой ситуации для защиты внешнего объекта следует создать клон (копию) объекта в методе.

Класс **Object** содержит **protected**-метод **clone()**, осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод **clone()** как **public** для обеспечения возможности вызова из другого пакета. В переопределенном методе следует вызвать базовую версию метода **super.clone()**, которая и выполняет собственно клонирование. Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс **Cloneable**. Интерфейс **Cloneable** не содержит методов, относится к помеченным (tagged) интерфейсам, а его реализация гарантирует, что метод **clone()** класса **Object** возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение **CloneNotSupportedException**.

При использовании этого механизма объект создается без вызова конструктора, на уровне виртуальной машины выполняется побитовое копирование объекта в другую часть памяти. В языке C++ аналогичный механизм реализован с помощью конструктора копирования.

Чтобы продемонстрировать реализацию клонирования классу **Student** следует добавить имплементацию интерфейса **Cloneable**:

```
public class Student implements Cloneable
```

а в сам класс реализацию метода **clone()**:

```
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
```

Тогда безопасное клонирование можно представить в виде:

```
/* # 18 # безопасная передача по ссылке # CloneMain.java */
```

```
public class CloneMain {
    private static void preparation(Student student) {
        try {
            student = (Student) student.clone(); // cloning
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        student.setId(1000);
        System.out.println("->id = " + student.getId());
    }
    public static void main(String[] args) {
        Student ob = new Student();
        ob.setId(71);
        System.out.println("id = " + ob.getId());
        preparation(ob);
        System.out.println("id = " + ob.getId());
    }
}
```

В результате будет выведено:

```
id = 71
->id = 1000
id = 71
```

Если закомментировать вызов метода **clone()**, то выведено будет следующее:

```
id = 71
->id = 1000
id = 1000
```

То есть отключение клонирования привело к тому, что изменение состояния объекта в методе отразилось и на исходном объекте.

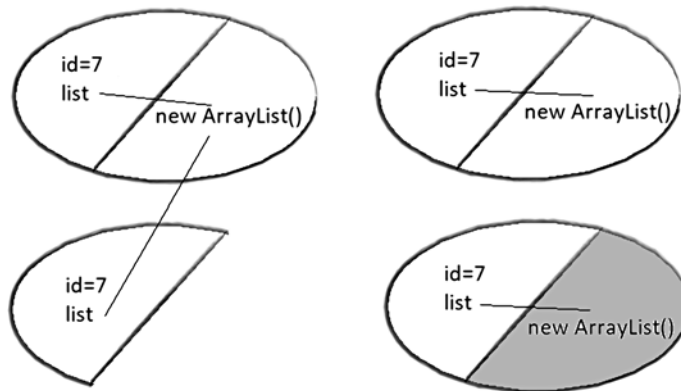


Рис. 4.1. «Неглубокое» и «глубокое» клонирование

Решение эффективно только в случае, когда поля клонируемого объекта представляют собой значения базовых типов и их оболочек или неизменяемых (immutable) объектных типов. Если же поле клонируемого типа является изменяемым объектным типом, то для корректного клонирования требуется другой подход. Причина заключается в том, что при создании копии поля оригинал и копия представляют собой ссылку на один и тот же объект.

В этой ситуации следует также клонировать и объект поля класса, если он сам поддерживает клонирование.

```
/* # 19 # глубокое клонирование # Abiturient.java */
```

```
package by.epam.learn.entity;
import java.util.ArrayList;
public class Abiturient implements Cloneable {
    private int id = 7;
    private ArrayList<Byte> list = new ArrayList<>();
```

```

public ArrayList<Byte> getList() {
    return list;
}
public void setList(ArrayList<Byte> list) {
    this.list = list;
}
@Override
public Abiturient clone() {
    Abiturient copy = null;
    try {
        copy = (Abiturient)super.clone();
        copy.list = (ArrayList<Byte>) list.clone();
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return copy;
}
@Override
public String toString() {
    final StringBuilder sb = new StringBuilder("Abiturient{");
    sb.append("id=").append(id);
    sb.append(", list=").append(list);
    sb.append('}');
    return sb.toString();
}
}

```

Клонирование возможно лишь, если тип атрибута класса также реализует интерфейс **Cloneable** и переопределяет метод **clone()**. В противном случае вызов метода невозможен, так как он просто недоступен.

```
/* # 20 # демонстрация глубокого клонирования # CloneMain.java */
```

```

package by.epam.learn.main;
import by.epam.learn.entity.Abiturient;
import java.util.ArrayList;
public class CloneMain {
    public static void main(String[] args) {
        Abiturient abiturient = new Abiturient();
        ArrayList<Byte> list = new ArrayList<>();
        list.add((byte)1);
        list.add((byte)2);
        abiturient.setList(list);
        Abiturient abiturient1 = abiturient.clone();
        list = abiturient1.getList();
        list.remove(0);
        list.add((byte)9);
        System.out.println(abiturient);
        System.out.println(abiturient1);
    }
}

```

Клонированный объект не имеет никаких зависимостей от оригинала:

```
Abiturient{id=7, list=[1, 2]}
Abiturient{id=7, list=[2, 9]}
```

Следовательно, если класс имеет суперкласс, то для реализации механизма клонирования текущего класса необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений **final** для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

Если заменить объявление **ArrayList<Byte>** на **CustomCollection<Task>**, где **Task** — изменяемый тип, то клонирование должно затрагивать и внутреннее состояние коллекции.

«Сборка мусора» и освобождение ресурсов

Так как объекты создаются динамически с помощью операции **new**, а уничтожаются автоматически, то желательно знать механизм ликвидации объектов и способ освобождения памяти. Автоматическое освобождение памяти, занимаемой объектом, выполняется с помощью механизма «сборки мусора». Когда никаких ссылок на объект не существует, т.е. все ссылки на него вышли из области видимости программы, предполагается, что объект больше не нужен, и память, занятая объектом, может быть освобождена. «Сборка мусора» происходит нерегулярно во время выполнения программы. Форсировать «сборку мусора» невозможно, можно лишь «рекомендовать» выполнить ее вызовом метода **System.gc()** или **Runtime.getRuntime().gc()**, но виртуальная машина выполнит очистку памяти тогда, когда сама посчитает это удобным. Вызов метода **System.runFinalization()** приведет к запуску метода **finalize()** для объектов, утративших ссылки. Начиная с версии Java 9 метод **finalize()** помечен как **deprecated**, тем самым не рекомендован к использованию, как и не был рекомендован еще в первые годы существования языка Java, но механизм его работы важен для понимания принципов работы «сборщика мусора».

Иногда объекту нужно выполнять некоторые действия перед освобождением памяти. Например, освободить внешние ресурсы. Для обработки таких ситуаций могут применяться два способа: конструкция **try-finally** и механизм **autocloseable**. Указанные способы являются предпочтительными, абсолютно надежными и будут рассмотрены в девятой главе.

Запуск стандартного механизма **finalization** определяется алгоритмом «сборки мусора», и до его непосредственного исполнения может пройти сколько угодно много времени. Из-за всего этого поведение метода **finalize()** может повлиять на корректную работу программы, особенно при смене JVM. Если существует возможность освободить ресурсы или выполнить другие подобные действия без привлечения этого механизма, то лучше без него обойтись.

Виртуальная машина вызывает этот метод всегда, когда она собирается уничтожить объект данного класса. Внутри метода **protected void finalize()**, вызываемого непосредственно перед освобождением памяти, следует определить действия, которые должны быть выполнены до уничтожения объекта.

Ключевое слово **protected** запрещает доступ к **finalize()** коду, определенному вне этого класса. Метод **finalize()** вызывается только перед самой «сборкой мусора», а не тогда, когда объект выходит из области видимости, т.е. заранее невозможно определить, когда **finalize()** будет выполнен, и недоступный объект может занимать память довольно долго. В принципе, этот метод может быть вообще не выполнен!

Недопустимо в приложении доверять такому методу критические по времени действия по освобождению ресурсов.

Если не вызвать явно метод **finalize()** суперкласса, то он не будет вызван автоматически. Еще одна опасность: если при выполнении данного метода возникнет исключительная ситуация, она будет проигнорирована и приложение продолжит выполняться, что также представляет опасность для его корректной работы.

Пакеты

Любой класс Java относится к определенному пакету, который может быть неименованным (unnamed или default package), если оператор **package** отсутствует. Оператор **package**, помещаемый в начале исходного программного файла, определяет именованный пакет, т.е. область в пространстве имен классов, в которой определяются имена классов, содержащихся в этом файле. Действие оператора **package** указывает на месторасположение файла относительно корневого каталога проекта. Например:

```
package by.bsu.eun.entity;
```

При этом программный файл будет помещен в подкаталог с названием **by.bsu.eun.entity**. Имя пакета при обращении к классу из другого пакета присоединяется к имени класса **by.bsu.eun.entity.Student**.

В проектах пакеты именуются следующим образом:

- обратный Интернет-адрес производителя или заказчика программного обеспечения, а именно для **www.bsu.by** получится **by.bsu**;
- далее следует имя проекта (обычно сокращенное), например, **eun**;
- затем располагаются пакеты, определяющие собственно приложение. Общая форма файла, содержащего исходный код Java, может быть следующая:
- одиночный оператор **package** (необязателен, но крайне желателен);
- любое количество операторов **import** (необязательны);
- одиночный открытый (**public**) класс (необязателен);
- любое количество классов пакета (необязательны и нежелательны).

При использовании классов перед именем класса через точку надо добавлять полное имя пакета, к которому относится данный класс. На рисунке приведен далеко не полный список пакетов реального приложения. Из названий пакетов можно определить, какие примерно классы в нем расположены, не заглядывая внутрь. При создании пакета всегда следует руководствоваться простым правилом: называть его именем простым, но отражающим смысл, логику поведения и функциональность объединенных в нем классов.

```
by.bsu.eun
by.bsu.eun.administration.type
by.bsu.eun.administration.dbhelper
by.bsu.eun.common.type
by.bsu.eun.common.dbhelper.annboard
by.bsu.eun.common.dbhelper.course
by.bsu.eun.common.dbhelper.guestbook
by.bsu.eun.common.dbhelper.learnresult
by.bsu.eun.common.dbhelper.message
by.bsu.eun.common.dbhelper.news
by.bsu.eun.common.dbhelper.prepinfo
by.bsu.eun.common.dbhelper.statistic
by.bsu.eun.common.dbhelper.subjectmark
by.bsu.eun.common.dbhelper.subject
by.bsu.eun.common.dbhelper.test
by.bsu.eun.common.dbhelper.user
by.bsu.eun.common.menu
by.bsu.eun.common.object
by.bsu.eun.common.servlet
by.bsu.eun.common.tool
by.bsu.eun.consultation.type
by.bsu.eun.consultation.dbhelper
by.bsu.eun.consultation.object
by.bsu.eun.core.type
by.bsu.eun.core.dbhelper
by.bsu.eun.core.exception
by.bsu.eun.core.filter
by.bsu.eun.core.manager
by.bsu.eun.core.taglib
```

Рис. 4.2. Организация пакетов приложения

Каждый класс добавляется в указанный пакет при компиляции. Например:

```
/* # 21 # применение пакета # CommonObject.java */

package by.bsu.eun.object;
public class CommonObject {
    // more code
}
```

Класс начинается с указания того, что он принадлежит пакету **by.bsu.eun.object**. Другими словами, это означает, что файл **CommonObject.java** находится в каталоге **object**, который, в свою очередь, находится в каталоге **bsu**, и так далее. Нельзя переименовывать пакет, не переименовав каталог, в котором хранятся его классы. Чтобы получить доступ к классу из другого пакета, перед именем такого класса указывается имя пакета: **by.bsu.eun.object.CommonObject**. Чтобы избежать таких длинных имен при создании объектов классов, используется ключевое слово **import**. Например:

```
import by.bsu.eun.object.CommonObject;
```

или

```
import by.bsu.eun.object.*;
```

Во втором варианте импортируется весь пакет, что означает возможность доступа к любому классу пакета, но только не к подпакету и его классам. В практическом программировании следует использовать индивидуальный **import** класса, чтобы при анализе кода была возможность быстро определить месторасположение используемого класса.

Доступ к классу из другого пакета можно осуществить еще одним способом (не очень рекомендуемым):

```
/* # 22 # применение полного имени пакета при наследовании # UserStatistic.java */
```

```
package by.bsu.eun.usermng;
public class UserStatistic extends by.bsu.eun.object.CommonObject {
    // more code
}
```

Такая запись используется, если в классе нужен доступ к классам, имеющим одинаковые имена.

При импорте класса из другого пакета рекомендуется всегда указывать полный путь с указанием имени импортируемого класса. Это позволяет в большом проекте легко найти определение класса, если возникает необходимость посмотреть исходный код класса.

```
/* # 23 # применение полного пути к классу при импорте # CreatorStatistic.java */
```

```
package by.bsu.eun.action;
import by.bsu.eun.object.CommonObject;
import by.bsu.eun.usermng.UserStatistic;
public class CreatorStatistic extends CommonObject {
    public UserStatistic statistic;
}
```

Если пакет не существует, то его необходимо создать до первой компиляции, если пакет не указан, класс добавляется в пакет без имени (unnamed). При этом unnamed-каталог не создается. Однако в реальных проектах классы вне пакетов не создаются, и не существует причин отступать от этого правила.

Статический импорт

При вызове статических методов и обращении к статическим константам приходится использовать в качестве префикса имя класса, что утяжеляет код и снижает скорость его восприятия.

```
// # 24 # обращение к статическому методу и константе # ImportMain.java
```

```
package by.epam.learn.demo;
public class ImportMain {
    public static void main(String[ ] args) {
        System.out.println(2 * Math.PI * 3);
        System.out.println(Math.floor(Math.cos(Math.PI / 3)));
    }
}
```

Статические константы и статические методы класса можно использовать без указания принадлежности к классу, если применить статический импорт,

```
import static java.lang.Math.*;
```

как это показано ниже.

```
// # 25 # статический импорт методов и констант # ImportLuxMain.java
```

```
package by.epam.learn.demo;
import static java.lang.Math.*;
public class ImportLuxMain {
    public static void main(String[ ] args) {
        System.out.println(2 * PI * 3);
        System.out.println(floor(cos(PI / 3)));
    }
}
```

Если необходимо получить доступ только к одной статической константе или методу, то импорт производится в следующем виде:

```
import static java.lang.Math.E; // for one constant
import static java.lang.Math.cos; // for one method
```

Рекомендации при проектировании иерархии

При построении иерархии необходимо помнить, что отношение между классами можно выразить как «is-a», или «является». *Студент* «является» *Человеком*. Поля класса находятся с классом в отношении «has-a», или «содержит». *Студент* «содержит» *номер зачетной книжки*.

Наследование и переопределение методов используются для реализации отличий поведения. Если наследование можно заменить агрегацией, то следует

так и поступить. Нет смысла создавать подкласс *Студент-заочник*, если можно в подкласс *Студент* добавить поле *Форма обучения*.

При наследовании в новые классы добавляются новые возможности в виде полей и методов или переопределения методов. Если новых возможностей не обнаруживается, то использование наследования, как правило, не имеет для этого оснований.

Базовая функциональность должна определяться в вершине иерархии проектируемых классов. Если в подклассе добавляются новые методы, характеризующие поведение иерархии в целом, следует заняться перепроектированием. Но нельзя учесть все возможные изменения иерархии в процессе разработки. Избыточный функционал придется поддерживать. Лучше на поздних стадиях добавить методы в суперкласс, чем пытаться понять, зачем нужен метод, который никто еще не использовал.

Не использовать значения переменных для характерного изменения поведения. Для этих целей следует создать подкласс и переопределить метод.

Для различных семантических сущностей не создавать общую иерархию, даже если действия для них идентичны по форме. Блокировка *Теста* и блокировка *Студента* — действия суть похожие, но отличные по своему функционалу и последствиям.

Вопросы к главе 4

1. Принципы ООП.
2. Правила переопределения метода `boolean equals(Object o)`.
3. Зачем переопределять методы `hashCode()` и `equals()` одновременно?
4. Написать метод `equals()` для класса, содержащего одно поле типа **String**.
5. Правила переопределения метода `int hashCode()`. Можно ли в качестве результата возвращать константу?
6. Правила переопределения метода `clone()`.
7. Чем отличаются **finally** и **finalize**? Для чего используется ключевое слово **final**?
8. JavaBeans: основные требования к классам Bean-компонентов, соглашения об именах.
9. Как работает *Garbage Collector*. Какие самые распространенные алгоритмы? Можно ли самому указать сборщику мусора, какой объект удалить из памяти.
10. В каких областях памяти хранятся значения и объекты, массивы?
11. Чем является класс **Object**? Перечислить известные методы класса **Object**, указать их назначение.
12. Что такое хэш-значение? Объяснить, почему два разных объекта могут сгенерировать одинаковые хэш-коды?
13. Для чего используется наследование классов в java-программе? Привести пример наследования. Поля и методы, помеченные модификатором доступа **private**, наследуются?

14. Как вызываются конструкторы при создании объекта производного класса? Что в конструкторе класса делает оператор **super()**?
15. Возможно ли в одном конструкторе использовать операторы **super()** и **this()**?
16. Объяснить утверждения: «ссылка базового класса может ссылаться на объекты своих производных типов» и «объект производного класса может быть использован везде, где ожидается объект его базового типа». Верно ли обратное и почему?
17. Что такое переопределение методов? Зачем оно нужно? Можно ли менять возвращаемый тип при переопределении методов? Можно ли менять атрибуты доступа при переопределении методов? Можно ли переопределить методы в рамках одного класса?
18. Определить правило вызова переопределенных методов. Можно ли статические методы переопределить нестатическими и наоборот?
19. Какие свойства имеют финальные методы и финальные классы? Зачем их использовать?
20. Какие применяются правила приведения типов при наследовании. Записать примеры явного и неявного преобразования ссылочных типов. Объяснить, какие ошибки могут возникать при явном преобразовании ссылочных типов.
21. Что такое объект класса **Class**? Чем использование метода **getClass()** и последующего сравнения возвращенного значения с **Type.class** отличается от использования оператора **instanceof**?
22. Что такое абстрактные классы и методы? Зачем они нужны? Бывают ли случаи, когда абстрактные методы содержат тело? Можно ли в абстрактных классах определять конструкторы? Могут ли абстрактные классы содержать неабстрактные методы? Можно ли от абстрактных классов создавать объекты и почему?
23. Для чего служит интерфейс **Cloneable**? Как правильно переопределить метод **clone()** класса **Object**, для того чтобы объект мог создавать свои адекватные копии?
24. Что такое перечисления в Java. Как объявить перечисление? Чем являются элементы перечислений? Кто и когда создает экземпляры перечислений?
25. Могут ли перечисления реализовывать интерфейсы или содержать абстрактные методы? Могут ли перечисления содержать статические методы?
26. Можно ли самостоятельно создать экземпляр перечисления? А ссылку типа перечисления? Как сравнить, что в двух переменных содержится один и тот же элемент перечисления и почему именно так?
27. Что такое параметризованные классы? Для чего они необходимы? Привести пример параметризованного класса и пример создания объекта параметризованного класса.
28. Ссылки какого типа могут ссылаться на объекты параметризованных классов? Можно ли создать объект, параметризовав его примитивным типом данных?

29. Какие ограничения на вызов методов существуют у параметризованных полей? Как эти ограничения снимает использование при параметризации ключевого слова **extends**?
30. Как параметризуются статические методы, как определяется конкретный тип параметризованного метода? Можно ли методы экземпляра класса параметризовать отдельно от параметра класса, и если «да», то как тогда определять тип параметра?
31. Что такое **wildcard**? Привести пример его использования?
32. Для чего используется параметризация `<? extends Type>`, `<? super Type>`?

Задания к главе 4

Вариант А

Создать приложение, удовлетворяющее требованиям, приведенным в задании. Наследование применять только в тех заданиях, в которых это логически обосновано. Аргументировать принадлежность классу каждого создаваемого метода и корректно переопределить для каждого класса методы **equals()**, **hashCode()**, **toString()**.

1. Создать объект класса **Текст**, используя классы **Предложение**, **Слово**. Методы: дополнить текст, вывести на консоль текст, заголовок текста.
2. Создать объект класса **Автомобиль**, используя классы **Колесо**, **Двигатель**. Методы: ехать, заправляться, менять колесо, вывести на консоль марку автомобиля.
3. Создать объект класса **Самолет**, используя классы **Крыло**, **Шасси**, **Двигатель**. Методы: летать, задавать маршрут, вывести на консоль маршрут.
4. Создать объект класса **Государство**, используя классы **Область**, **Район**, **Город**. Методы: вывести на консоль столицу, количество областей, площадь, областные центры.
5. Создать объект класса **Планета**, используя классы **Материк**, **Океан**, **Остров**. Методы: вывести на консоль название материка, планеты, количество материков.
6. Создать объект класса **Звездная система**, используя классы **Планета**, **Звезда**, **Луна**. Методы: вывести на консоль количество планет в звездной системе, название звезды, добавление планеты в систему.
7. Создать объект класса **Компьютер**, используя классы **Винчестер**, **Дисковод**, **Оперативная память**, **Процессор**. Методы: включить, выключить, проверить на вирусы, вывести на консоль размер винчестера.
8. Создать объект класса **Квадрат**, используя классы **Точка**, **Отрезок**. Методы: задание размеров, растяжение, сжатие, поворот, изменение цвета.
9. Создать объект класса **Круг**, используя классы **Точка**, **Окружность**. Методы: задание размеров, изменение радиуса, определение принадлежности точки данному кругу.

10. Создать объект класса **Щенок**, используя классы **Животное**, **Собака**. Методы: вывести на консоль имя, подать голос, прыгать, бегать, кусать.
11. Создать объект класса **Наседка**, используя классы **Птица**, **Воробей**. Методы: летать, петь, нести яйца, высиживать птенцов.
12. Создать объект класса **Текстовый файл**, используя классы **Файл**, **Директория**. Методы: создать, переименовать, вывести на консоль содержимое, дополнить, удалить.
13. Создать объект класса **Одномерный массив**, используя классы **Массив**, **Элемент**. Методы: создать, вывести на консоль, выполнить операции (сложить, вычесть, перемножить).
14. Создать объект класса **Простая дробь**, используя класс **Число**. Методы: вывод на экран, сложение, вычитание, умножение, деление.
15. Создать объект класса **Дом**, используя классы **Окно**, **Дверь**. Методы: закрыть на ключ, вывести на консоль количество окон, дверей.
16. Создать объект класса **Цветок**, используя классы **Лепесток**, **Бутон**. Методы: расцвести, завянуть, вывести на консоль цвет бутона.
17. Создать объект класса **Дерево**, используя классы **Лист**, **Ветка**. Методы: зацвести, опать листьям, покрыться инеем, пожелтеть листьям.
18. Создать объект класса **Пианино**, используя классы **Клавиша**, **Педаль**. Методы: настроить, играть на пианино, нажимать клавишу.
19. Создать объект класса **Фотоальбом**, используя классы **Фотография**, **Страница**. Методы: задать название фотографии, дополнить фотоальбом фотографией, вывести на консоль количество фотографий.
20. Создать объект класса **Год**, используя классы **Месяц**, **День**. Методы: задать дату, вывести на консоль день недели по заданной дате, рассчитать количество дней, месяцев в заданном временном промежутке.
21. Создать объект класса **Сутки**, используя классы **Час**, **Минута**. Методы: вывести на консоль текущее время, рассчитать время суток (утро, день, вечер, ночь).
22. Создать объект класса **Птица**, используя классы **Крылья**, **Клюв**. Методы: летать, садиться, питаться, атаковать.
23. Создать объект класса **Хищник**, используя классы **Когти**, **Зубы**. Методы: рычать, бежать, спать, добывать пищу.
24. Создать объект класса **Гитара**, используя класс **Струна**, **Скворечник**. Методы: играть, настраивать, заменять струну.

Вариант В

Создать консольное приложение, удовлетворяющее следующим требованиям:

- Использовать возможности ООП: классы, наследование, полиморфизм, инкапсуляция.
- Каждый класс должен иметь отражающее смысл название и информативный состав.

- Наследование должно применяться только тогда, когда это имеет смысл.
 - При кодировании должны быть использованы соглашения об оформлении кода `java code convention`.
 - Классы должны быть грамотно разложены по пакетам.
 - Консольное меню должно быть минимальным.
 - Для хранения параметров инициализации можно использовать файлы.
1. **Цветочница.** Определить иерархию цветов. Создать несколько объектов-цветов. Собрать букет (используя аксессуары) с определением его стоимости. Провести сортировку цветов в букете на основе уровня свежести. Найти цветок в букете, соответствующий заданному диапазону длин стеблей.
 2. **Новогодний подарок.** Определить иерархию конфет и прочих сладостей. Создать несколько объектов-конфет. Собрать детский подарок с определением его веса. Провести сортировку конфет в подарке на основе одного из параметров. Найти конфету в подарке, соответствующую заданному диапазону содержания сахара.
 3. **Домашние электроприборы.** Определить иерархию электроприборов. Включить некоторые в розетку. Подсчитать потребляемую мощность. Провести сортировку приборов в квартире на основе мощности. Найти прибор в квартире, соответствующий заданному диапазону параметров.
 4. **Шеф-повар.** Определить иерархию овощей. Сделать салат. Подсчитать калорийность. Провести сортировку овощей для салата на основе одного из параметров. Найти овощи в салате, соответствующие заданному диапазону калорийности.
 5. **Звукозапись.** Определить иерархию музыкальных композиций. Записать на диск сборку. Подсчитать продолжительность. Провести перестановку композиций диска на основе принадлежности к стилю. Найти композицию, соответствующую заданному диапазону длины треков.
 6. **Камни.** Определить иерархию драгоценных и полудрагоценных камней. Отобрать камни для ожерелья. Подсчитать общий вес (в каратах) и стоимость. Провести сортировку камней ожерелья на основе ценности. Найти камни в ожерелье, соответствующие заданному диапазону параметров прозрачности.
 7. **Мотоциклист.** Определить иерархию амуниции. Экипировать мотоциклиста. Подсчитать стоимость. Провести сортировку амуниции на основе веса. Найти элементы амуниции, соответствующие заданному диапазону параметров цены.
 8. **Транспорт.** Определить иерархию подвижного состава железнодорожного транспорта. Создать пассажирский поезд. Подсчитать общую численность пассажиров и багажа. Провести сортировку вагонов поезда на основе уровня комфортности. Найти в поезде вагоны, соответствующие заданному диапазону параметров числа пассажиров.
 9. **Авиакомпания.** Определить иерархию самолетов. Создать авиакомпанию. Посчитать общую вместимость и грузоподъемность. Провести сортировку

- самолетов компании по дальности полета. Найти самолет в компании, соответствующий заданному диапазону параметров потребления горючего.
10. **Таксопарк.** Определить иерархию легковых автомобилей. Создать таксопарк. Подсчитать стоимость автопарка. Провести сортировку автомобилей парка по расходу топлива. Найти автомобиль в компании, соответствующий заданному диапазону параметров скорости.
 11. **Страхование.** Определить иерархию страховых обязательств. Собрать из обязательств дериватив. Подсчитать стоимость. Провести сортировку обязательств в деривативе на основе уменьшения степени риска. Найти обязательство в деривативе, соответствующее заданному диапазону параметров.
 12. **Мобильная связь.** Определить иерархию тарифов мобильной компании. Создать список тарифов компании. Подсчитать общую численность клиентов. Провести сортировку тарифов на основе размера абонентской платы. Найти тариф в компании, соответствующий заданному диапазону параметров.
 13. **Фургон кофе.** Загрузить фургон определенного объема грузом на определенную сумму из различных сортов кофе, находящихся к тому же в разных физических состояниях (зерно, молотый, растворимый в банках и пакетиках). Учитывать объем кофе вместе с упаковкой. Провести сортировку товаров на основе соотношения цены и веса. Найти в фургоне товар, соответствующий заданному диапазону параметров качества.
 14. **Игровая комната.** Подготовить игровую комнату для детей разных возрастных групп. Игрушек должно быть фиксированное количество в пределах выделенной суммы денег. Должны встречаться игрушки родственных групп: маленькие, средние и большие машины, куклы, мячи, кубики. Провести сортировку игрушек в комнате по одному из параметров. Найти игрушки в комнате, соответствующие заданному диапазону параметров.
 15. **Налоги.** Определить множество и сумму налоговых выплат физического лица за год с учетом доходов с основного и дополнительного мест работы, авторских вознаграждений, продажи имущества, получения в подарок денежных сумм и имущества, переводов из-за границы, льгот на детей и материальной помощи. Провести сортировку налогов по сумме.
 16. **Счета.** Клиент может иметь несколько счетов в банке. Учитывать возможность блокировки/разблокировки счета. Реализовать поиск и сортировку счетов. Вычисление общей суммы по счетам. Вычисление суммы по всем счетам, имеющим положительный и отрицательный балансы отдельно.
 17. **Туристические путевки.** Сформировать набор предложений клиенту по выбору туристической путевки различного типа (отдых, экскурсии, лечение, шопинг, круиз и т.д.) для оптимального выбора. Учитывать возможность выбора транспорта, питания и числа дней. Реализовать выбор и сортировку путевок.
 18. **Кредиты.** Сформировать набор предложений клиенту по целевым кредитам различных банков для оптимального выбора. Учитывать возможность

досрочного погашения кредита и/или увеличения кредитной линии.
Реализовать выбор и поиск кредита.

Тестовые задания к главе 4

Вопрос 4.1.

Объявлены классы X и Y. Заполнить пустое место в методе **action()** так, чтобы был корректно вызван метод из класса X (выбрать один).

```
class X {
    void action(){}
}
class Y extends X {
    static void action(int i){
        ____ .action();
    }
}
```

- a) super
- b) this
- c) X
- d) new X()

Вопрос 4.2.

Объявлены два класса в иерархии наследования:

```
class Base {
    public void method() {
        System.out.println("base");
    }
}
class Derived extends Base {
    public static void main(String[] args) {
        Base base = (Base)new Derived();
        base.method();
    }
    public void method() {
        System.out.print("derived");
    }
}
```

Что будет выведено в результате компиляции и выполнения кода? (выбрать один)

- a) compilation fails
- b) runtime exception
- c) derived
- d) base

Вопрос 4.3.

Дан код:

```
class A {
    A() {
        System.out.print(0);
    }
    A(int i) {
        System.out.print(i);
    }
}
class B extends A {
    B(){super(1);}
    B(int i){}
    public static void main(String args[]) {
        new B();
        new B(2);
    }
}
```

Что будет выведено в результате компиляции и запуска кода? (выбрать один)

- a) compilation fails
- b) runtime exception
- c) 10
- d) 12
- e) 01
- f) 21

Вопрос 4.4.

Дан код:

```
class SubClass extends Base {
    public static final int NUM = 0;
}
class Base {
    public static final int NUM = 1;
    public static void main(String[] args) {
        Base base = new Base();
        SubClass subClass = new SubClass();
        System.out.print(Base.NUM);
        System.out.print(SubClass.NUM);
        System.out.print(base.NUM);
        System.out.print(((Base) subClass).NUM);
    }
}
```

Что выведется на консоль в результате запуска этой программы? (выбрать один)

- a) 1000
- b) 1010
- c) 1011
- d) 1111
- e) 1110

Вопрос 4.5.

Дан класс и его подкласс:

```
class Base {
    void action() {}
}
class Current extends Base {
    // line 1
}
```

Какие из представленных методов могут быть корректно подставлены вместо комментария *line 1*? (выбрать три)

- a) `int action() {}`
- b) `void action(){}`
- c) `public void action(){}`
- d) `private void action(){}`
- e) `protected void action(){}`

Вопрос 4.6.

Дан класс и его подкласс:

```
class Base {
    private long idBase;
    public Base(long idBase) {
        this.idBase = idBase;
    }
}
class Current extends Base {
}
```

Какие из утверждений истинны? (выбрать один)

- a) код компилируется без изменений
- b) код компилируется, если `Current() { Base (); }` добавить к классу `Current`
- c) код компилируется, если `Base() { Current (); }` добавить к классу `Base`
- d) код компилируется, если `Base() { this(1); }` добавить к классу `Base`
- e) код компилируется, если `Base() { Base (1); }` добавить к классу `Base`

ВНУТРЕННИЕ КЛАССЫ

Внутри каждой большой задачи сидит маленькая, пытающаяся пробиться наружу.

Закон больших задач Хоара

Классы могут взаимодействовать друг с другом не только посредством наследования и использования ссылок, но и посредством организации логической структуры с определением одного класса в теле другого.

В Java можно определить (вложить) один класс внутри определения другого класса, что позволяет группировать классы, *логически связанные друг с другом*, и динамично управлять доступом к ним. С одной стороны, обоснованное использование в коде внутренних классов делает его более эффективным и понятным. С другой, применение внутренних классов есть один из способов *сокрытия кода*, так как внутренний класс может быть недоступен и не виден вне класса-владельца. Внутренние классы также могут использоваться в качестве *блоков прослушивания событий*. Решение о включении одного класса внутрь другого может быть принято при тесном и частом взаимодействии двух классов.

Одной из причин использования внутренних классов является возможность быть подклассом любого класса независимо от того, подклассом какого класса является внешний класс. Фактически при этом реализуется ограниченное множественное наследование со своими преимуществами и проблемами.

При необходимости доступа к защищенным полям и методам некоторого класса может появиться множество подклассов в разных пакетах системы. В качестве альтернативы можно сделать этот подкласс внутренним, и методами класса-владельца предоставить доступ к интересующим защищенным полям и методам.

В качестве примеров можно рассмотреть взаимосвязи классов *Студент*, *Адрес*. Объект класса *Адрес* как единое целое характеризует объект класса *Студент*, расположен внутри (невидим извне) объекта *Студент*. Его состояние есть часть состояния *Студента*. Оба этих объекта связаны. Перед инициализацией объекта внутреннего класса *Адрес* должен быть создан объект внешнего класса *Студент*. Классы связаны описанием общего состояния.

Если необходимо определить и связать класс с некоторой функциональностью, очень близкой этому классу, то применяется статическое вложение класса, делающее независимым объект с вложенной функциональностью от

класса-владельца, но логически через имя внешнего класса связывает с ним. Объект такого класса можно создать, не создавая объект класса-владельца.

Такие статические вложенные классы объявляются с модификатором **static**. Статические классы могут обращаться к нестатическим членам включающего класса не напрямую, а только через его объект.

Нестатические внутренние классы имеют доступ ко всем переменным и методам своего внешнего класса-владельца и требуют последовательного создания объектов внешнего и внутреннего классов.

Применение анонимных классов и их подмножества: лямбда-выражений, позволяет сократить количество кода. Анонимные классы сокращают число подклассов, лямбда-выражения записываются еще короче и с более высоким акцентом на функционал объекта.

Внутренние (inner) классы

Нестатические вложенные классы принято называть внутренними, или *inner* классами.

Связь между внешним и внутренним классами при этом определяется необходимостью привязывания логической сущности внутреннего класса к сущности внешнего класса.

Доступ к элементам внутреннего класса возможен из внешнего только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса. Объект внутреннего класса всегда ассоциируется (скрыто хранит ссылку) с создавшим его объектом внешнего класса — так называемым внешним, или *enclosing* объектом.

Пусть изначально описание класса **Student** представлено в виде:

```
package by.epam.learn.study;
public class Student {
    private int studentId;
    private String name;
    private int group;
    private String faculty;
    private String city;
    private String street;
    private int houseId;
    private int flatId;
    private String email;
    private String skype;
    private long phoneNumber;
    // constructors, methods
}
```

В описании класса можно целый набор данных объединить под общим именем **Address** и выделить его как внутренний класс. Класс **Student** станет более коротким и понятным.

Внешний и внутренний классы могут выглядеть в итоге так:

```
/* # 1 # объявление внутреннего класса и использование его в качестве поля
# Student.java */
```

```
package by.epam.learn.study;
public class Student {
    private int studentId;
    private String name;
    private int group;
    private String faculty;
    private Address address;
    // private, protected - may be
    public class Address { // inner class: begin
        private String city;
        private String street;
        private int houseId;
        private int flatId;
        private String email;
        private String skype;
        private long phoneNumber;
        // more code
    } // inner class: end
}
```

Внутренний класс может быть использован любым членом своего внешнего класса, а может и не использоваться вовсе, хотя в этом случае утрачивается его смысл.

Использование объекта внутреннего класса вне своего внешнего класса возможно только при наличии доступа (видимости) и при объявлении ссылки в виде:

```
Student.Address address = new Student().new Address();
```

Здесь сначала создается объект внешнего класса, а затем объект внутреннего класса. Другим способом объект внутреннего класса создать не получится. Объекту внутреннего класса совершенно не обязательно быть полем класса-владельца. Основное отличие от внешнего класса состоит в больших возможностях ограничения видимости и сокрытия реализации внутреннего класса по сравнению с обычным внешним классом. Внутренний класс может быть объявлен как **private**, что обеспечивает его полную невидимость вне класса-владельца и надежное сокрытие реализации. В этом случае ссылку **address**, приведенную выше, объявить было бы нельзя. Создать объект такого класса можно только в методах и логических блоках внешнего класса. Использование **protected** позволяет получить доступ к внутреннему классу для класса в другом пакете, являющегося суперклассом внешнего класса.

При компиляции внутренний класс получает собственный модуль для интерпретации, соответствующий внутреннему классу, который получит имя

Student\$Address.class. Внешний же класс будет скомпилирован в обычный файл **Student.class**.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, как будто они его собственные, в то же время внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутреннего класса. Доступ будет разрешен по имени в том числе и к полям, объявленным как **private**. Внутренние классы не могут содержать статические поля и методы, кроме **final static**. Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования. Допустимо наследование следующего вида:

```
/* # 2 # наследование от внешнего и внутреннего классов # SubStudent.java */
```

```
package by.epam.learn.study;
public class SubStudent extends Student {
    public class SubAddress extends Address {
    }
}
```

Если внутренний класс наследуется обычным образом другим классом (поле **extends** указывается *ИмяВнешнегоКласса.ИмяВнутреннегоКласса*), то он теряет доступ к полям своего внешнего класса, в котором был объявлен.

```
/* # 3 # наследование от внутреннего класса # FreeAddress.java */
```

```
package by.epam.learn.study;
public class FreeAddress extends Student.Address {
    public FreeAddress() {
        new Student().super();
    }
    public FreeAddress(Student student) {
        student.super();
    }
}
```

В данном случае конструктор класса **FreeAddress** должен объявлять объект класса **Student** или получать его в виде параметра, что позволит получить доступ к ссылке на внутренний класс **Address**, наследуемый классом **FreeAddress**.

Внутренние классы позволяют решить проблему множественного наследования сущности, когда требуется наследовать свойства нескольких классов.

При объявлении внутреннего класса могут использоваться модификаторы **final**, **abstract**, **private**, **protected**, **public**.

```
class Outer {
    private class Inner {}
}
```

Поля внешнего класса видны внутреннему классу так, будто они его собственные, модификаторы видимости игнорируются. Применить ссылку **this** также не получится, так как **this** внутри класса **Inner** указывает на его собственный объект, и ни в коем случае не на его владельца. Поэтому **this.id** будет давать ошибку компиляции.

```
/* # 4 # доступ к полям внешнего класса # Owner.java */
```

```
public class Owner {
    private int id;
    public class Inner {
        public void buildId() {
            id += 1000;
        }
    }
}
```

Внешний класс напрямую не видит никаких компонентов своего внутреннего класса.

Вопрос доступа к полям внешнего класса можно рассмотреть путем объявления полей с одинаковыми именами во внутреннем и внешнем классах, чего на практике делать не рекомендуется.

```
/* # 5 # некорректное объявление полей классов # DumberOwner.java */
```

```
public class DumberOwner {
    private int id;
    public class DumberInner {
        private int id;
        public void buildId(int id) {
            this.id = id + 100 * DumberOwner.this.id;
        }
    }
}
```

Поле **id** объявлено как поле класса **DumberInner** и при попытке доступа к полю **id** класса **DumberOwner** обращения типа **id** или **this.id** приведут к обращению к полю внутреннего класса. Для доступа к **id** внешнего класса нужно указать на имя класса, которому принадлежит объект, а именно **DumberOwner.this.id**.

Внутренний класс может быть объявлен также внутри метода или логического блока внешнего (owner) класса. Видимость такого класса регулируется областью видимости блока, в котором он объявлен. Но внутренний класс сохраняет доступ ко всем полям и методам внешнего класса, а также ко всем константам, объявленным в текущем блоке кода. Класс, объявленный внутри метода, не может быть объявлен как **static**, а также не может содержать статические поля и методы.

```
/* # 6 # внутренний класс, объявленный внутри метода # AbstractTeacher.java  
# TeacherCreator.java # Teacher.java # TeacherLogic.java # StudyMain.java */
```

```
package by.epam.learn.study;
public abstract class AbstractTeacher {
    private int id;
    public AbstractTeacher(int id) {
        this.id = id;
    }
    public abstract boolean remandStudent(Student student);
}
package by.epam.learn.study;
public class Teacher extends AbstractTeacher {
    public Teacher(int id) {
        super(id);
    }
    @Override
    public boolean remandStudent(Student student) {
        return false;
    }
}
package by.epam.learn.study;
public class TeacherCreator {
    public static AbstractTeacher createTeacher(int id) {
        int value = 0;
        // class declaration inside a method
        class Rector extends AbstractTeacher {
            Rector(int id) {
                super(id);
            }
            @Override
            public boolean remandStudent(Student student) {
                // value++; compile error
                boolean result = false;
                if (student != null) {
                    // student status change code in the database
                    result = true;
                }
                return result;
            }
        } // inner class: end
        if (isRectorId(id)) {
            return new Rector(id);
        } else {
            return new Teacher(id);
        }
    }
    private static boolean isRectorId(int id) {
        // checking id in the database
    }
}
```

```

        return (id == 6); // stub
    }
}
package by.epam.learn.study;
public class TeacherLogic {
    public void expelledProcess(int rectorId, Student student) {
        AbstractTeacher teacher = TeacherCreator.createTeacher(rectorId);
        boolean result = teacher.remandStudent(student);
        System.out.println("Student expelled: " + result);
    }
}
package by.epam.learn.study;
public class StudyMain {
    public static void main(String[] args) {
        TeacherLogic logic = new TeacherLogic();
        Student student = new Student();
        logic.expelledProcess(42, student);
        logic.expelledProcess(6, student);
    }
}

```

В результате будет выведено:

Student expelled: false

Student expelled: true

Класс **Rector** объявлен в методе **createTeacher(int id)** и, соответственно, объекты этого класса можно создавать только внутри метода, из любого другого метода или конструктора внешнего класса внутренний класс недоступен. Однако существует единственная возможность получить ссылку на класс, объявленный внутри метода, и использовать его специфические свойства, как в данном случае, при наследовании внутренним классом функциональности обычного класса, в частности, **AbstractTeacher**. При компиляции данного кода с внутренним классом ассоциируется объектный модуль со сложным именем **TeacherCreator\$1Rector.class**, тем не менее однозначно определяющим связь между внешним и внутренним классами. Цифра **1** в имени говорит о том, что в других методах класса также можно объявить внутренний класс с таким же именем.

Свойства внутренних классов:

- Доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего класса;
- Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса;
- Объект внутреннего класса имеет ссылку на объект своего внешнего класса (**enclosing**);
- Внутренние классы не могут содержать **static**-полей и методов, кроме **final static**;
- Внутренние классы могут быть производными от других классов;

- Внутренние классы могут быть суперклассами;
- Внутренние классы могут реализовывать интерфейсы;
- Внутренние классы могут быть объявлены с параметрами **final**, **abstract**, **private**, **protected**, **public**;
- Если необходимо создать объект внутреннего класса где-нибудь, кроме внешнего нестатического метода класса, то нужно определить тип объекта как *OwnerType.InnerType*;
- Внутренний класс может быть объявлен внутри метода или логического блока внешнего класса, видимость класса регулируется видимостью того блока, в котором он объявлен. Однако внутренний класс сохраняет доступ ко всем полям и методам внешнего класса, а также **final**-переменным, объявленным в текущем блоке кода;
- Локальному внутреннему классу, объявленному внутри метода или логического блока, модификатор доступа не требуется, так как он все равно не доступен напрямую вне метода.

Вложенные (nested) классы

Если не существует жесткой необходимости в одновременном обязательном существовании объекта внутреннего класса и объекта внешнего класса, то есть смысл сделать такой внутренний класс статическим, который будет тогда называться вложенным, или *nested* классом.

Связь между внешним и внутренним классами определяется необходимостью привязывания логической функциональности внутреннего класса.

Вложенный класс логически связан с классом-владельцем, но его объект может быть использован независимо от объекта внешнего класса. Такой класс обычно определяет дополнительный функционал для класса-владельца.

При объявлении такого внутреннего класса присутствует служебное слово **static**, и такой класс называется вложенным (*nested*). Если класс объявлен внутри интерфейса, то он получает спецификаторы **public static** по умолчанию. Такой класс способен наследовать другие классы, реализовывать интерфейсы и являться объектом наследования для любого класса, обладающего необходимыми правами доступа. В то же время статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса, а напрямую иметь доступ только к статическим полям и методам внешнего класса. Для создания объекта вложенного класса объект внешнего класса создавать нет необходимости. Подкласс вложенного класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс. Если предполагается использовать внутренний класс в качестве подкласса, следует исключить использование в его теле любых прямых обращений к членам класса-владельца.

```
/* # 7 # вложенный класс-компаратор # Student.java # ComparingMain.java */
```

```
package by.epam.learn.nested;
import java.util.Comparator;
public class Student {
    private int studentId;
    private String name;
    private int group;
    private float averageMark;
    public Student(int studentId, String name, int group, float averageMark) {
        this.studentId = studentId;
        this.name = name;
        this.group = group;
        this.averageMark = averageMark;
    }
    public String getName() {
        return name;
    }
    public int getGroup() {
        return group;
    }
    public float getAverageMark() {
        return averageMark;
    }
    // nested classes
    public static class GroupComparator implements Comparator<Student> {
        @Override
        public int compare(Student o1, Student o2) {
            return o1.group - o2.group;
        }
    }
    public static class NameComparator implements Comparator<Student> {
        @Override
        public int compare(Student o1, Student o2) {
            return o1.name.compareTo(o2.name);
        }
    }
    public static class MarkComparator implements Comparator<Student> {
        @Override
        public int compare(Student o1, Student o2) {
            return Float.compare(o2.averageMark, o1.averageMark);
        }
    }
}
package by.epam.learn.nested;
import java.util.Comparator;
public class ComparingMain {
    public static void main(String[] args) {
        Student st1 = new Student(2341757, "Mazaliyk", 3, 5.42f);
```

```

Student st2 = new Student(2341742, "Polovinkin", 1, 5.42f);
// creating a static class object
Student.NameComparator nameComparator = new Student.NameComparator();
int result1 = nameComparator.compare(st1, st2);
System.out.println(st1.getName() + " [" + result1 + "] " + st2.getName());
Student.MarkComparator markComparator = new Student.MarkComparator();
int result2 = markComparator.compare(st1, st2);
System.out.println(st1.getAverageMark() + " [" + result2+ "] " + st2.getAverageMark());
Student.GroupComparator groupComparator = new Student.GroupComparator();
int result3 = groupComparator.compare(st1, st2);
System.out.println(st1.getGroup() + " [" + result3+ "] " + st2.getGroup());
}
}

```

Результатом будет:

Mazaliyk [-3] Polovinkin

5.42 [0] 5.42

3 [2] 1

Объект **nameComparator** вложенного класса создается с использованием имени внешнего класса без вызова его конструктора. Если во вложенном классе объявлен статический метод, то он просто вызывается при указании полного относительного пути к нему.

```
/* # 8 # видимость полей # Owner.java */
```

```

public class Owner {
    private int value = 1;
    static int statValue = 2;
    public static class Nested {
        {
            statValue++;
            // value++; invisible
        }
    }
}

```

Статическому классу доступно только статическое содержимое класса-владельца.

Класс, вложенный в интерфейс, по умолчанию статический. На него не накладывается никаких особых ограничений, и он может содержать поля и методы как статические, так и нестатические.

```
/* # 9 # класс, вложенный в интерфейс # Logic.java # NestedLogic.java */
```

```

package by.epam.learn.nested;
public interface Logic {
    void doLogic();
}

```

```

class NestedLogic { // public static: default parameters
    public long value;
    public NestedLogic() { /* code */ }
    public static void assign() { /* code */ }
    public void accept() { /* code */ }
}

```

Такой внутренний класс использует пространство имен интерфейса. Вызов статического метода выглядит так:

```
Logic.NestedLogic.assign();
```

Свойства вложенных классов:

- Вложенный класс может быть базовым, производным, реализующим интерфейс;
- Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса;
- Вложенный класс имеет доступ к статическим полям и методам внешнего класса;
- Подкласс вложенного класса не наследует возможность доступа к членам внешнего класса, которыми наделен его суперкласс;
- Класс, вложенный в интерфейс, статический по умолчанию;
- Статический метод вложенного класса вызывается при указании полного относительного пути к нему.

Анонимные (anonymous) классы

Анонимные (безымянные) внутренние классы применяются для придания уникальной функциональности отдельно взятому экземпляру, для обработки событий, реализации блоков прослушивания, реализации интерфейсов, запуска потоков и т.д. Можно объявить анонимный класс, который будет расширять другой класс или реализовывать интерфейс при объявлении одного-единственного объекта, когда остальным объектам этого класса будет соответствовать реализация метода, определенная в самом классе. Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора **new**.

С появлением функциональных интерфейсов появилось понятие анонимного объекта-функции, то есть объекта, основное назначение которого передать реализацию конкретной функциональности в анонимном виде:

```
FunctionalInterface lambdaFunction = () -> doAction();
```

Функциональные интерфейсы будут рассматриваться позже в главе 7.

Анонимные классы эффективно используются, как правило, для реализации (переопределения) одного или нескольких методов. Этот прием эффективен

В случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области или одномоментного применения объекта.

Анонимные классы, как и остальные внутренние, допускают вложенность друг в друга, что может сильно запутать код и сделать эти конструкции непонятными, поэтому в практике программирования данная техника не используется.

Конструктор анонимного класса определить невозможно. Нельзя создать анонимный класс для **final**-класса.

```
/* # 10 # анонимные классы # StudentAction.java # StudentActionMain.java */

package by.epam.learn.inner.anonymous;
public class StudentAction {
    private static final int BASE_COEFFICIENT = 6;
    public double defineScholarship(float averageMark) {
        double value = 100;
        if (averageMark > BASE_COEFFICIENT) {
            value *= 1 + (BASE_COEFFICIENT / 10.0);
        }
        return value;
    }
}

package by.epam.learn.inner.anonymous;
public class StudentActionMain {
    public static void main(String[] args) {
        StudentAction action = new StudentAction();// usually object
        StudentAction actionAnon = new StudentAction() { // anonymous class object
            int base = 9; // invisible
            @Override
            public double defineScholarship(float averageMark) {
                double value = 100;
                if (averageMark > base) {
                    value *= 1 + (base / 10.0);
                }
                return value;
            }
        };
        System.out.println(action.defineScholarship(9.05f));
        System.out.println(actionAnon.defineScholarship(9.05f));
    }
}
```

В результате будет выведено:

160.0

190.0

Анонимный класс может использовать только неизменяемые параметры и локальные переменные метода, в котором он создан. При запуске приложения происходит объявление объекта **actionAnon** с применением анонимного класса,

в котором переопределяется метод **defineScholarship()**. Вызов данного метода на объекте **actionAnon** приводит к вызову версии метода из анонимного класса, который компилируется в объектный модуль с именем **StudentActionMain\$1.class**. Процесс создания второго объекта с анонимным типом применяется в программировании значительно чаще, особенно при реализации классов-адаптеров и реализации интерфейсов в блоках прослушивания. В анонимном классе разрешено объявлять собственные поля и методы, которые не доступны объекту вне этого класса.

Для перечисления объявление анонимного внутреннего класса выглядит несколько иначе, так как инициализация всех элементов происходит при первом обращении к типу. Поэтому и анонимный класс реализуется только внутри объявления типа **enum**, как это сделано в следующем примере.

```
/* # 11 # анонимный класс в перечислении # Shape.java # EnumMain.java */
```

```
package by.epam.learn.inner.anonymous;
import java.util.StringJoiner;
public enum Shape {
    RECTANGLE (2, 3) {
        public double computeSquare() {
            return this.getA() * this.getB();
        }
    }, TRIANGLE (2, 3) {
        public double computeSquare() {
            return this.getA() * this.getB() / 2;
        }
    };
    private double a;
    private double b;

    Shape(double a, double b) {
        this.a = a;
        this.b = b;
    }
    public double getA() {
        return a;
    }
    public void setA(double a) {
        this.a = a;
    }
    public double getB() {
        return b;
    }
    public void setB(double b) {
        this.b = b;
    }
    public abstract double computeSquare();
    @Override
```

```

    public String toString() {
        return new StringJoiner(", ", Shape.class.getSimpleName() + "[", "]")
            .add("a=" + a).add("b=" + b).toString();
    }
}
package by.epam.learn.inner.anonymous;
import java.util.Arrays;
public class EnumMain {
    public static void main(String[] args) {
        Arrays.stream(Shape.values()).forEach(s -> System.out.println(s.computeSquare()));
    }
}

```

В результате будет выведено:

6.0

3.0

Свойства анонимных классов:

- расширяет другой класс или реализует интерфейс при объявлении одного единственного объекта, остальным объектам будет соответствовать реализация, определенная в самом классе;
- объявление анонимного объекта выполняется одновременно с созданием его объекта с помощью оператора **new**;
- конструкторы анонимных классов ни определить, ни переопределить нельзя;
- анонимные классы допускают вложенность друг в друга (нежелательно использовать);
- объявление анонимного класса в перечислении отличается от простого анонимного класса, поскольку инициализация всех элементов происходит при первом обращении к типу.

Ситуации, в которых следует использовать внутренние классы:

- выделение самостоятельной логической части сложного класса;
- сокрытие реализации;
- одномоментное использование переопределенных методов;
- реализация обработчиков событий;
- запуск потоков выполнения;
- отслеживание внутреннего состояния, например, с помощью **enum**.

Вопросы к главе 5

1. Что такое внутренние, вложенные и анонимные классы? Как определить классы такого вида? Как создать объекты классов такого вида.
2. Перечислить возможности доступа к членам внешнего класса, которым наделены вложенные классы?

3. Перечислить возможности доступа к членам внешнего класса, которым наделены внутренние классы?
4. Перечислить возможности доступа к членам внешнего класса, которым наделены анонимные классы?
5. Могут ли классы внутри классов быть базовыми, производными или реализующими интерфейсы?
6. Как решить проблему множественного наследования с применением внутренних классов?
7. Можно ли анонимный класс создать от абстрактного класса?
8. Можно ли анонимный класс создать от **final**-класса?
9. Во что компилируется анонимный внутренний класс в классе? В методе?
10. Можно ли создать анонимный статический внутренний класс?
11. Как получить доступ к внутреннему классу, объявленному внутри метода извне метода?

Задания к главе 5

Вариант А

1. Создать класс **Notepad** с внутренним классом или классами, с помощью объектов которого могут храниться несколько записей на одну дату.
2. Создать класс **Payment** с внутренним классом, с помощью объектов которого можно сформировать покупку из нескольких товаров.
3. Создать класс **Account** с внутренним классом, с помощью объектов которого можно хранить информацию обо всех операциях со счетом (снятие, платежи, поступления).
4. Создать класс **Зачетная Книжка** с внутренним классом, с помощью объектов которого можно хранить информацию о сессиях, зачетах, экзаменах.
5. Создать класс **Department** с внутренним классом, с помощью объектов которого можно хранить информацию обо всех должностях отдела и обо всех сотрудниках, когда-либо занимавших конкретную должность.
6. Создать класс **Catalog** с внутренним классом, с помощью объектов которого можно хранить информацию об истории выдач книги читателям.
7. Создать класс **Европа** с внутренним классом, с помощью объектов которого можно хранить информацию об истории изменения территориального деления на государства.
8. Создать класс **City** с внутренним классом, с помощью объектов которого можно хранить информацию о проспектах, улицах, площадях.
9. Создать класс **BlueRayDisc** с внутренним классом, с помощью объектов которого можно хранить информацию о каталогах, подкаталогах и записях.
10. Создать класс **Mobile** с внутренним классом, с помощью объектов которого можно хранить информацию о моделях телефонов и их свойствах.

11. Создать класс **Художественная Выставка** с внутренним классом, с помощью объектов которого можно хранить информацию о картинах, авторах и времени проведения выставок.
12. Создать класс **Календарь** с внутренним классом, с помощью объектов которого можно хранить информацию о выходных и праздничных днях.
13. Создать класс **Shop** с внутренним классом, с помощью объектов которого можно хранить информацию об отделах, товарах и услугах.
14. Создать класс **Справочная Служба Общественного Транспорта** с внутренним классом, с помощью объектов которого можно хранить информацию о времени, линиях маршрутов и стоимости проезда.
15. Создать класс **Computer** с внутренним классом, с помощью объектов которого можно хранить информацию об операционной системе, процессоре и оперативной памяти.
16. Создать класс **Park** с внутренним классом, с помощью объектов которого можно хранить информацию об аттракционах, времени их работы и стоимости.
17. Создать класс **Cinema** с внутренним классом, с помощью объектов которого можно хранить информацию об адресах кинотеатров, фильмах и времени начала сеансов.
18. Создать класс **Программа Передач** с внутренним классом, с помощью объектов которого можно хранить информацию о названии телеканалов и программ.
19. Создать класс **Фильм** с внутренним классом, с помощью объектов которого можно хранить информацию о продолжительности, жанре и режиссерах фильма.

Тестовые задания к главе 5

Вопрос 5.1.

Дано:

```
class Garden {
    public static class Plant {}
}
```

Выбрать корректное объявление объекта внутреннего класса (выбрать один).

- a) `Plant plant = new Plant();`
- b) `Garden.Plant plant = new Garden.Plant();`
- c) `Plant plant = new Garden.Plant();`
- d) `Garden.Plant plant = new Garden().new Plant();`

Вопрос 5.2.

Дан код:

```
class Outer {
    public int size = 0;
    static class Inner {
        public void incrementSize() {
            /* line 1 */ += 1;
        }
    }
}
```

Что необходимо записать вместо комментария *line 1*, чтобы код класса Inner компилировался без ошибок? (выбрать один)

- a) size
- b) Outer.size
- c) new Outer().size
- d) Outer.Inner.size
- e) нет верного варианта

Вопрос 5.3.

Дано:

```
class Outer{
    class Inner{}
    public void outerMethod() {
        // место создания объекта класса Inner
    }
}
```

Какой вариант вызовет ошибку компиляции при создании объекта класса Inner? (выбрать один)

- a) Inner a = new Inner();
- b) Inner b = Outer.new Inner();
- c) Inner c = new Outer().new Inner();
- d) Outer.Inner d = new Outer().new Inner();
- e) Outer.Inner e = new Inner();

Вопрос 5.4.

Дан код:

```
classClazz{
    {System.out.print("clazz ");}
    public void method() {
        System.out.print("method1 ");
    }
}
```

```
class Owner{
    {System.out.print("owner ");}
    private int N=10;
    class Inner extends Clazz {
        {System.out.print("inner ");}
        public void method(){
            System.out.print("method2 ");
        }
    }
}
class Quest {
    public static void main(String[] args) {
        new Owner().new Inner().method();
    }
}
```

Каким будет результат компиляции и запуска приложения? (выбрать один)

- a) compilation fails
- b) clazz owner inner method2
- c) owner clazz inner method2
- d) owner clazz inner method1

Вопрос 5.5.

Дан код:

```
class Clazz {
    static int i = 1;
}
class Outer {
    class Inner extends Clazz {
    }
}
class Quest {
    public static void main(String[] args) {
        System.out.println(_____);
    }
}
```

Какое обращение к переменной *i* корректно из метода *main*? (выбрать один)

- a) Outer.Inner.Clazz.i
- b) new Outer.Inner().i
- c) Outer.Inner.super.i
- d) Outer.Inner.i

ИНТЕРФЕЙСЫ И АННОТАЦИИ

*Решение сложной задачи поручайте ленивому
сотруднику — он найдет более легкий путь.*

Закон Хлейда

Интерфейсы

Назначение интерфейса — описание или спецификация функциональности, которую должен реализовывать каждый класс, его имплементирующий. Класс, реализующий интерфейс, предоставляет к использованию объявленный интерфейс в виде набора **public**-методов в полном объеме.

Интерфейсы подобны абстрактным классам, хотя и не являются классами. В языке Java существует три вида интерфейсов: интерфейсы, определяющие функциональность для классов посредством описания методов, но не их реализации; функциональные интерфейсы, специфицирующие в одном абстрактном методе свое применение; интерфейсы, реализация которых автоматически придает классу определенные свойства. К последним относятся, например, интерфейсы **Cloneable**, **Serializable**, **Externalizable**, отвечающие за клонирование и сохранение объекта (сериализацию) в информационном потоке соответственно.

Общее определение интерфейса имеет вид:

```
[public] interface Name [extends NameOtherInterface,..., NameN] {  
    // constants, methods  
}
```

Все объявленные в интерфейсе абстрактные методы автоматически трактуются как **public abstract**, а все поля — как **public static final**, даже если они так не объявлены. Интерфейсы могут объявлять статические методы. В интерфейсах могут объявляться методы с реализацией с ключевым словом **default**. Эти методы могут быть **public** или **private**.

Класс может реализовывать любое число интерфейсов, указываемых через запятую после ключевого слова **implements**, дополняющего определение класса. После этого класс обязан реализовать все абстрактные методы, полученные им от интерфейсов, или объявить себя абстрактным классом.

Если необходимо определить набор функциональности для какого-либо рода деятельности, например, для управления счетом в банке, то следует использовать интерфейс вида:

```
/* # 1 # объявление интерфейса управления банковским счетом # AccountAction.java */
```

```
package by.epam.learn.advanced;
public interface AccountAction{
    boolean openAccount();
    boolean closeAccount();
    void blocking();
    default void unBlocking() {}
    double depositInCash(int accountId, int amount);
    boolean withdraw(int accountId, int amount);
    boolean convert(double amount);
    boolean transfer(int accountId, double amount);
}
```

В интерфейсе обозначены, но не реализованы, действия, которые может производить клиент со своим счетом. Реализация не представлена из-за возможности различного способа выполнения действия в конкретной ситуации. А именно: счет может блокироваться автоматически, по требованию клиента или администратором банковской системы. В каждом из трех указанных случаев реализация метода **blocking()** будет уникальной и никакого базового решения предложить невозможно. С другой стороны, наличие в интерфейсе метода заставляет класс, его имплементирующий, предоставить реализацию методу. Программист получает повод задуматься о способе реализации функциональности, так как наличие метода в интерфейсе говорит о необходимости той или иной функциональности всем классам, реализующим данный интерфейс.

Метод **unblocking()** предоставляет дефолтную реализацию метода, которая может быть при необходимости переопределена в подклассе.

Интерфейс следует проектировать ориентированным на выполнение близких по смыслу задач, например, разделить действия по созданию, закрытию и блокировке счета от действий по снятию и пополнению средств. Такое разделение разумно в ситуации, когда клиенту посредством Интернет-системы не предоставляется возможность открытия, закрытия и блокировки счета. Тогда вместо одного общего интерфейса можно записать два специализированных: один для администратора, второй — для клиента.

```
/* # 2 # общее управление банковским счетом # AccountBaseAction.java */
```

```
package by.epam.learn.advanced;
public interface AccountBaseAction {
    boolean openAccount();
    boolean closeAccount();
    void blocking();
    default void unBlocking() {}
}
```

```
/* # 3 # операционное управление банковским счетом # AccountManager.java */
```

```
package by.epam.learn.advanced;
public interface AccountManager {
    double depositInCash(int accountId, int amount);
    boolean withdraw(int accountId, int amount);
    boolean convert(double amount);
    boolean transfer(int accountId, double amount);
}
```

Реализация интерфейса при реализации всех методов выглядит следующим образом:

```
/* # 4 # реализация общего управления банковским счетом
# AccountBaseActionImpl.java */
```

```
package by.epam.learn.advanced.impl;
import by.epam.learn.advanced.AccountBaseAction;
public class AccountBaseActionImpl implements AccountBaseAction {
    public boolean openAccount() {
        // more code
    }
    public boolean closeAccount() {
        // more code
    }
    public void blocking() {
        // more code
    }
    public void unBlocking() {
        // more code
    }
}
```

Если по каким-либо причинам для данного класса не предусмотрена реализация метода или его реализация нежелательна, рекомендуется генерация не-проверяемого исключения в теле метода, а именно:

```
public boolean blocking() {
    throw new UnsupportedOperationException();
}
```

Менее хорошим примером будет реализация в виде:

```
public boolean blocking() {
    return false;
}
```

так как пользователь метода может считать реализацию корректной.

В интерфейсе не могут быть объявлены поля без инициализации, интерфейс в качестве полей содержит только **public static final** константы.

Множественное наследование между интерфейсами допустимо. Классы, в свою очередь, интерфейсы только реализуют. Класс может наследовать один суперкласс и реализовывать произвольное число интерфейсов. В Java интерфейсы обеспечивают большую часть той функциональности, которая в C++ представляется с помощью механизма множественного наследования.

При именовании интерфейса в конец его названия может добавляться **able** в случае, если в названии присутствует действие (глагол). Не рекомендуется в имени интерфейса использовать слово **Interface** или любое его сокращение. В конец имени класса, реализующего интерфейс, для указания на источник действий можно добавлять слово **Impl**. Реализации интерфейсов помещать следует в подпакет с именем **impl**.

Интерфейсы можно применить, например, в задаче, где на поверхности линии соединяются в фигуры. Описание функционала методов по обработке информации о линиях и фигурах:

```
/* # 5 # объявление интерфейсов # LineGroupAction.java */
```

```
package by.epam.learn.advanced;
import by.epam.learn.entity.AbstractShape;
public interface LineGroupAction {
    double computePerimeter(AbstractShape shape);
}
```

```
/* # 6 # наследование интерфейсов # ShapeAction.java */
```

```
package by.epam.learn.advanced;
import by.epam.learn.entity.AbstractShape;
public interface ShapeAction extends LineGroupAction {
    double computeSquare(AbstractShape shape);
}
```

Класс, который будет реализовывать интерфейс **ShapeAction**, должен будет определить все методы из цепочки наследования интерфейсов. В данном случае это методы **computePerimeter()** и **computeSquare()**.

```
/* # 7 # абстрактная фигура # AbstractShape.java */
```

```
package by.epam.learn.entity;
public class AbstractShape {
    private long shapeId;
    public long getShapeId() {
        return shapeId;
    }
    public void setShapeId(long shapeId) {
        this.shapeId = shapeId;
    }
}
```

Интерфейсы обычно объявляются как **public**, потому что описание функциональности, предоставляемое ими, может быть использовано в других пакетах проекта. Интерфейсы с областью видимости в рамках пакета, атрибут доступа по умолчанию, могут использоваться только в этом пакете и нигде более.

Реализация интерфейсов классом может иметь вид:

```
[public] class NameClass implements Name1, Name2,..., NameN { /* */ }
```

Здесь *Name1, Name2,..., NameN* — перечень используемых интерфейсов. Класс, который реализует интерфейс, должен предоставить полную реализацию всех абстрактных методов, объявленных в интерфейсе. Кроме того, данный класс может объявлять свои собственные методы.

```
/* # 8 # реализация интерфейса # RectangleAction.java */
```

```
package by.epam.learn.advanced.impl;
import by.epam.learn.entity.AbstractShape;
import by.epam.learn.entity.Rectangle;
import by.epam.learn.advanced.ShapeAction;
public class RectangleAction implements ShapeAction {
    @Override
    public double computeSquare(AbstractShape shape) {
        double square;
        if (shape instanceof Rectangle rectangle) {
            square = rectangle.getHeight() * rectangle.getWidth();
        } else {
            throw new IllegalArgumentException("Incompatible shape " + shape.getClass());
        }
        return square;
    }
    @Override
    public double computePerimeter(AbstractShape shape) {
        double perimeter;
        if (shape instanceof Rectangle rectangle) {
            perimeter = 2 * (rectangle.getWidth() + rectangle.getHeight());
        } else {
            throw new IllegalArgumentException("Incompatible shape " + shape.getClass());
        }
        return perimeter;
    }
}
```

```
/* # 9 # реализация интерфейса # TriangleAction.java */
```

```
package by.epam.learn.advanced.impl;
import by.epam.learn.entity.AbstractShape;
import by.epam.learn.entity.RightTriangle;
import by.epam.learn.advanced.ShapeAction;
public class TriangleAction implements ShapeAction {
```

```

@Override
public double computeSquare(AbstractShape shape) {
    double square;
    if (shape instanceof RightTriangle triangle) {
        square = 1./2 * triangle.getSideA() * triangle.getSideB();
    } else {
        throw new IllegalArgumentException("Incompatible shape " + shape.getClass());
    }
    return square;
}

@Override
public double computePerimeter(AbstractShape shape) {
    double perimeter = 0;
    if (shape instanceof RightTriangle triangle) {
        double a = triangle.getSideA();
        double b = triangle.getSideB();
        perimeter = a + b + Math.hypot(a, b);
    } else {
        throw new IllegalArgumentException("Incompatible shape " + shape.getClass());
    }
    return perimeter;
}
}

```

При неполной реализации интерфейса класс должен быть объявлен как абстрактный, что говорит о необходимости реализации абстрактных методов уже в его подклассе.

```

/* # 10 # неполная реализация интерфейса # PentagonAction.java */

package by.epam.learn.advanced.impl;
import by.epam.learn.entity.AbstractShape;
import by.epam.learn.advanced.ShapeAction;
public abstract class PentagonAction implements ShapeAction {
    @Override
    public double computePerimeter(AbstractShape shape) {
        // code
    }
}

```

Классы для определения фигур, используемые в интерфейсах:

```

/* # 11 # прямоугольник, треугольник # Rectangle.java # RightTriangle.java */

package by.epam.learn.entity;
public class Rectangle extends AbstractShape {
    private double height;
    private double width;
    public Rectangle(double height, double width) {
        this.height = height;
    }
}

```

```

        this.width = width;
    }
    public double getHeight() {
        return height;
    }
    public double getWidth() {
        return width;
    }
}
package by.epam.learn.entity;
import java.util.StringJoiner;
public class RightTriangle extends AbstractShape {
    private double sideA;
    private double sideB;
    public RightTriangle(double sideA, double sideB) {
        this.sideA = sideA;
        this.sideB = sideB;
    }
    public double getSideA() {
        return sideA;
    }
    public void setSideA(double sideA) {
        this.sideA = sideA;
    }
    public double getSideB() {
        return sideB;
    }
    public void setSideB(double sideB) {
        this.sideB = sideB;
    }
    @Override
    public String toString() {
        return new StringJoiner(", ", RightTriangle.class.getSimpleName() + "[", "]")
            .add("sideA=" + sideA).add("sideB=" + sideB).toString();
    }
}

```

```
/* # 12 # СВОЙСТВА ССЫЛОК НА ИНТЕРФЕЙС # ActionMain.java */
```

```

package by.epam.learn.advanced;
import by.epam.learn.advanced.impl.RectangleAction;
import by.epam.learn.advanced.impl.TriangleAction;
import by.epam.learn.entity.Rectangle;
import by.epam.learn.entity.RightTriangle;
public class ActionMain {
    public static void main(String[] args) {
        ShapeAction action;
        try {
            Rectangle rectShape = new Rectangle(2, 5);
            action = new RectangleAction();

```

```

        System.out.println("Square rectangle: " + action.computeSquare(rectShape));
        System.out.println("Perimeter rectangle: " + action.computePerimeter(rectShape));
        RightTriangle trShape = new RightTriangle(3, 4);
        action = new TriangleAction();
        System.out.println("Square triangle: " + action.computeSquare(trShape));
        System.out.println("Perimeter triangle: " + action.computePerimeter(trShape));
        action.computePerimeter(rectShape); // runtime exception
    } catch (IllegalArgumentException e) {
        System.err.println(e.getMessage());
    }
}
}

```

В результате будет выведено:

Square rectangle: 10.0

Perimeter rectangle: 14.0

Square triangle: 6.0

Perimeter triangle: 12.0

Incompatible shape class by.epam.learn.entity.Rectangle

Допустимо объявление ссылки на интерфейсный тип или использование ее в качестве параметра метода. Такая ссылка может указывать на экземпляр любого класса, который реализует объявленный интерфейс. При вызове метода через такую ссылку будет вызываться его реализованная версия, основанная на текущем экземпляре класса. Выполняемый метод разыскивается динамически во время выполнения, что позволяет создавать классы позже кода, который вызывает их методы.

Параметризация интерфейсов

Реализация для интерфейсов, параметры методов которых являются ссылками на иерархически организованные классы, в данном случае представлена достаточно тяжеловесно. Необходимость проверки принадлежности обрабатываемого объекта к допустимому типу снижает гибкость программы и увеличивает количество кода, в том числе и на генерацию, и обработку исключений.

Сделать реализацию интерфейса удобной, менее подверженной ошибкам и практически исключаящей проверки на принадлежность типу можно достаточно легко, если при описании интерфейса добавить параметризацию в виде:

```
/* # 13 # параметризация интерфейса # ShapeGeneric.java */
```

```

package by.epam.learn.advanced;
import by.epam.learn.entity.AbstractShape;
public interface ShapeGeneric<T extends AbstractShape> {

```

```
double computeSquare(T shape);
double computePerimeter(T shape);
}
```

Параметризованный тип **T extends AbstractShape** указывает, что в качестве параметра методов может использоваться только подкласс **AbstractShape**, что мало чем отличается от случая, когда тип параметра метода указывался явно. Но когда дело доходит до реализации интерфейса, то указывается конкретный тип объектов, являющихся подклассами **AbstractShape**, которые будут обрабатываться методами данного класса, а в качестве параметра метода также прописывается тот же самый конкретный тип:

```
/* # 14 # реализация интерфейса с указанием типа параметра
# RectangleGeneric.java # TriangleGeneric.java */
```

```
package by.epam.learn.advanced.impl;
import by.epam.learn.advanced.ShapeGeneric;
import by.epam.learn.entity.Rectangle;
public class RectangleGeneric implements ShapeGeneric<Rectangle> {
    @Override
    public double computeSquare(Rectangle shape) {
        return shape.getHeight() * shape.getWidth();
    }
    @Override
    public double computePerimeter(Rectangle shape) {
        return 2 * (shape.getWidth() + shape.getHeight());
    }
}

package by.epam.learn.advanced.impl;
import by.epam.learn.advanced.ShapeGeneric;
import by.epam.learn.entity.RightTriangle;
public class TriangleGeneric implements ShapeGeneric<RightTriangle> {
    @Override
    public double computeSquare(RightTriangle shape) {
        return 0.5 * shape.getSideA() * shape.getSideB();
    }
    @Override
    public double computePerimeter(RightTriangle shape) {
        double a = shape.getSideA();
        double b = shape.getSideB();
        double perimeter = a + b + Math.hypot(a, b);
        return perimeter;
    }
}
```

На этапе компиляции исключается возможность передачи в метод объекта, который не может быть обработан, т.е. код **action.computePerimeter(rectShape)** спровоцирует ошибку компиляции, если **action** инициализирован объектом класса **TriangleAction**.

Применение параметризации при объявлении интерфейсов в данном случае позволяет избавиться от лишних проверок и преобразований типов при реализации непосредственно самого интерфейса и использовании созданных на их основе классов.

```
/* # 15 # использование параметризованных интерфейсов # GenericMain.java */
```

```
package by.epam.learn.advanced;
import by.epam.learn.advanced.impl.RectangleGeneric;
import by.epam.learn.advanced.impl.TriangleGeneric;
import by.epam.learn.entity.Rectangle;
import by.epam.learn.entity.RightTriangle;
public class GenericMain {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(2, 5);
        ShapeGeneric<Rectangle> rectangleGeneric = new RectangleGeneric();
        RightTriangle triangle = new RightTriangle(3, 4);
        ShapeGeneric<RightTriangle> triangleGeneric = new TriangleGeneric();
        System.out.println("Square rectangle: " + rectangleGeneric.computeSquare(rectangle));
        System.out.println("Perimeter rectangle: " + rectangleGeneric.computePerimeter(rectangle));
        System.out.println("Square triangle: " + triangleGeneric.computeSquare(triangle));
        System.out.println("Perimeter triangle: " + triangleGeneric.computePerimeter(triangle));
        // triangleGeneric.computePerimeter(rectangle); // compile error
    }
}
```

Аннотации

Аннотации — это метатеги, которые добавляются к коду и применяются к объявлению пакетов, классов, конструкторов, методов, полей, параметров и локальных переменных. Аннотации всегда обладают некоторой информацией и связывают эти дополнительные данные и все перечисленные конструкции языка. Фактически аннотации представляют собой их дополнительные модификаторы, применение которых не влечет за собой изменений ранее созданного кода. Аннотации позволяют избежать создания шаблонного кода во многих ситуациях, активируя утилиты для его генерации из аннотаций в исходном коде.

В языке Java определено несколько встроенных аннотаций для разработки новых аннотаций — **@Retention**, **@Documented**, **@Target** и **@Inherited** — из пакета **java.lang.annotation**. Из других аннотаций выделяются — **@Override**, **@Deprecated** и **@SuppressWarnings** — из пакета **java.lang**. Широкое использование аннотаций в различных технологиях и фреймворках обуславливается возможностью сокращения кода и снижения его связанности.

В следующем коде приведено объявление аннотации.

```
/* # 16 # многочленная аннотация класса # BaseAction.java */

@Target(ElementType.TYPE)
public @interface BaseAction {
    int level();
    String sqlRequest();
}
```

Ключевому слову **interface** предшествует символ **@**. Такая запись сообщает компилятору об объявлении аннотации. В объявлении также есть два метода-члена: **int level()**, **String sqlRequest()**.

После объявления аннотации ее можно использовать для аннотирования объявлений класса за счет объявления со значением **ElementType.TYPE** целевой аннотации **@Target**. Объявление любого типа может быть аннотировано. Даже к аннотации можно добавить аннотацию. Во всех случаях аннотация предшествует объявлению.

Применяя аннотацию, нужно задавать значения для ее методов-членов, если при объявлении аннотации не было задано значение по умолчанию. Далее приведен фрагмент, в котором аннотация **BaseAction** сопровождает объявление класса:

```
/* # 17 # примитивное использование аннотации класса # Base.java */

@BaseAction(level = 2, sqlRequest = "SELECT name, phone FROM phonebook")
public class Base {
    public void doAction() {
        Class<Base> clazz = Base.class;
        BaseAction action = (BaseAction) clazz.getAnnotation(BaseAction.class);
        System.out.println(action.level());
        System.out.println(action.sqlRequest());
    }
}
```

Данная аннотация помечает класс **Base**. За именем аннотации, начинающимся с символа **@**, следует заключенный в круглые скобки список инициализирующих значений для методов-членов. Для того чтобы передать значение методу-члену, имени этого метода присваивается значение. Таким образом, в приведенном фрагменте строка «**SELECT name, phone FROM phonebook**» присваивается методу **sqlRequest()**, члену аннотации типа **BaseAction**. При этом в операции присваивания после имени **sqlRequest** нет круглых скобок. Когда методу-члену передается инициализирующее значение, используется только имя метода. Следовательно, в данном контексте методы-члены выглядят как поля.

Все аннотации содержат только объявления методов, добавлять тела этим методам не нужно, так как их реализует сам язык. Кроме того, эти методы не могут содержать параметров секции **throws** и действуют скорее как поля.

Допустимые типы возвращаемого значения: базовые типы, **String**, **Enum**, **Class** и массив любого из вышеперечисленных типов.

Все типы аннотаций автоматически расширяют интерфейс **Annotation** из пакета **java.lang.annotation**. В этом интерфейсе приведен метод **annotationType()**, который возвращает объект типа **Class**, представляющий вызывающую аннотацию.

Если необходим доступ к аннотации в процессе выполнения приложения, то перед объявлением аннотации задается правило сохранения **RUNTIME** в виде кода:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME)
```

предоставляющее максимальную продолжительность существования аннотации. С правилом **SOURCE** аннотация существует только в исходном тексте программы и отбрасывается во время компиляции. Аннотация с правилом сохранения **CLASS** помещается в процессе компиляции в файл **Name.class**, но недоступна в JVM во время выполнения.

Основные типы аннотаций: аннотация-маркер, одночленная и многочленная. Аннотация-маркер не содержит методов-членов. Цель — пометить объявление. В этом случае достаточно присутствия аннотации. Поскольку у интерфейса аннотации-маркера нет методов-членов, достаточно определить наличие аннотации:

```
public @interface MarkerAnnotation {}
```

Для проверки наличия аннотации используется метод **isAnnotationPresent()**. Одночленная аннотация содержит единственный метод-член. Для этого типа аннотации допускается краткая условная форма задания значения для метода-члена. Если есть только один метод-член, то просто указывается его значение при создании аннотации. Имя метода-члена указывать не нужно. Но для того, чтобы воспользоваться краткой формой, следует для метода-члена использовать имя **value()**.

Многочленные аннотации содержат несколько методов-членов. Поэтому используется полный синтаксис (*parameter_name = value*) для каждого параметра.

Реализация обработки аннотации, приведенная в методе **doAction()** класса **Base**, крайне примитивна. Разработчику каждый раз при использовании аннотаций придется писать код по извлечению значений полей-членов и реакцию метода, класса и поля на значение аннотации. Необходимо привести реализацию аннотации таким образом, чтобы программисту достаточно было только аннотировать класс, метод или поле и передать нужное значение. Реакция системы на аннотацию должна быть автоматической.

Ниже приведен пример работы с аннотацией, реализованной на основе **Reflection API**. Примеры корректного использования аннотаций короткими не бывают.

Аннотация **BankingAnnotation** предназначена для задания уровня проверки безопасности при вызове метода.

```
/* # 18 # одночленная аннотация метода # BankingAnnotation.java */

package by.epam.learn.annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface BankingAnnotation {
    SecurityLevelType securityLevel() default SecurityLevelType.MEDIUM;
}
```

При отсутствии явного значения **securityLevel** будет использовано дефолтное.

```
/* # 19 # возможные значения уровней безопасности # SecurityLevelType.java */

package by.epam.learn.annotation;
public enum SecurityLevelType {
    LOW, MEDIUM, HIGH
}
```

Методы класса логики системы выполняют действия с банковским счетом. В зависимости от различных причин конкретным реализациям интерфейса **AccountManager** могут понадобиться дополнительные действия помимо основных.

```
/* # 20 # аннотирование методов # AccountManagerImpl.java */

package by.epam.learn.annotation;
import by.epam.learn.advanced.AccountManager;
public class AccountManagerImpl implements AccountManager {
    @BankingAnnotation(securityLevel = SecurityLevelType.HIGH)
    public double depositInCash(int accountId, int amount) {
        System.out.println("deposit in total: " + amount);
        return 0;
    }
    @BankingAnnotation(securityLevel = SecurityLevelType.HIGH)
    public boolean withdraw(int accountId, int amount) {
        System.out.println("amount withdrawn: " + amount);
        return true;
    }
    // run again without comment
    // @BankingAnnotation(securityLevel = SecurityLevelType.LOW)
    public boolean convert(double amount) {
        System.out.println("amount converted: " + amount);
        return true;
    }
}
```

```
@BankingAnnotation // default value MEDIUM
public boolean transfer(int accountId, double amount) {
    System.out.println("amount transferred: " + amount);
    return true;
}
}
```

```
/* # 21 # конфигурирование и запуск # BankingMain.java */
```

```
package by.epam.learn.annotation;
import by.epam.learn.advanced.AccountManager;
public class BankingMain {
    public static void main(String[] args) {
        AccountManager manager = SecurityFactory.registerSecurityObject(new AccountManagerImpl());
        manager.depositInCash(10128336, 6);
        manager.withdraw(64092376, 2);
        manager.convert(200);
        manager.transfer(64092376, 300);
    }
}
```

Подход с вызовом некоторого промежуточного метода для включения в обработку аннотаций достаточно распространен и, в частности, используется в технологиях JPA, Hibernate, Spring и других.

Вызов метода **createSecurityObject()**, регистрирующего методы класса для обработки аннотаций, можно разместить в конструкторе некоторого промежуточного абстрактного класса, реализующего интерфейс **AccountManager**, или в статическом логическом блоке самого интерфейса. Тогда реагировать на аннотации будут все методы всех реализаций интерфейса **AccountManager**.

Класс, определяющий действия приложения в зависимости от значения **SecurityLevelType**, передаваемого в аннотированный метод.

```
/* # 22 # логика обработки значения аннотации # SecurityService.java */
```

```
package by.epam.learn.annotation;
import java.lang.reflect.Method;
import java.util.Arrays;
public class SecurityService {
    public void onInvoke(SecurityLevelType level, Method method, Object[] args){
        System.out.printf("%s() was invoked with parameters: %s ",
                                method.getName(), Arrays.toString(args));

        switch (level) {
            case LOW -> System.out.println("security: " + level);
            case MEDIUM -> System.out.println("security: " + level);
            case HIGH -> System.out.println("security: " + level);
            default -> throw new EnumConstantNotPresentException(
                                    SecurityLevelType.class, level.toString());
        }
    }
}
```

Статический метод **registerSecurityObject(AccountManager target)** класса-фабрики создает для экземпляра, созданного на основе реализации интерфейса **AccountManager** экземпляр-заместитель, обладающий, кроме всех свойств оригинала, возможностью неявного обращения к бизнес-логике, выбор которой определяется выбором значения для поля аннотации.

```
/* # 23 # создание прокси-экземпляра, включающего функциональность
SecurityService # SecurityFactory.java */
```

```
package by.epam.learn.annotation;
import by.epam.learn.advanced.AccountManager;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
public class SecurityFactory {
    public static AccountManager registerSecurityObject(AccountManager target) {
        return (AccountManager) Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new SecurityInvocationHandler(target));
    }
    private static class SecurityInvocationHandler implements InvocationHandler {
        private Object targetObject;
        SecurityInvocationHandler(Object target) {
            this.targetObject = target;
        }
        public Object invoke(Object proxy, Method method, Object[] args)
            throws InvocationTargetException, NoSuchMethodException, IllegalAccessException {
            SecurityService service = new SecurityService();
            Method invocationMethod = targetObject.getClass().getMethod(method.getName(),
                (Class[]) method.getGenericParameterTypes());
            BankingAnnotation annotation = invocationMethod.getAnnotation(BankingAnnotation.class);
            if (annotation != null) {
                // method annotated
                service.onInvoke(annotation.securityLevel(), invocationMethod, args);
            } else {
                /* if annotation of method is required, it should
                throw an exception to the fact of its absence */
                /* throw new InvocationTargetException(null, "method " + invocationMethod
                    + " should be annotated "); */
            }
            return method.invoke(targetObject, args);
        }
    }
}
```

Будет выведено:

depositInCash() was invoked with parameters: [10128336, 6] security: HIGH

--method execution: deposit in total: 6

withdraw() was invoked with parameters: [64092376, 2] security: HIGH

--method execution: amount withdrawn: 2

--method execution: amount converted: 200.0

transfer() was invoked with parameters: [64092376, 300.0] security: MEDIUM

--method execution: amount transferred: 300.0

Использование аннотации может быть обязательным или опциональным. Если метод **invoke()** при отсутствии аннотации у метода генерирует исключение, то это указывает на обязательность явного использования аннотации методами класса, имплементированными при реализации интерфейса. Отсутствие генерации исключения в блоке после **else** приведет к тому, что аннотирование всех методов класса **AccountManager** необязательно. Сам метод будет вызван, несмотря на отсутствие аннотации.

Вопросы к главе 6

1. Что такое интерфейс? Как определить и реализовать интерфейс в java-программе?
2. Можно ли описывать в интерфейсе конструкторы и создавать объекты?
3. Можно ли создавать интерфейсные ссылки и если да, то на какие объекты они могут ссылаться?
4. Какие идентификаторы по умолчанию имеют поля интерфейса?
5. Какие идентификаторы по умолчанию имеют методы интерфейса?
6. Чем отличается абстрактный класс от интерфейса?
7. Когда применять интерфейс логичнее, а когда абстрактный класс?
8. Бывают ли интерфейсы без методов? Для чего?
9. Могут ли классы внутри классов реализовывать интерфейсы?
10. Возможно ли анонимный класс создать на основе реализации интерфейса?
11. Привести два способа объявления статического метода в интерфейсе?

Задания к главе 6

Вариант А

Разработать проект управления процессами на основе создания и реализации интерфейсов для следующих предметных областей:

1. **Полиграфические издания.** Возможности: оформить договор; открыть\редактировать\верстать издание; отправить на печать; отказаться от издания; оплатить издание; возобновить\закрыть издание. Добавить специализированные методы для Книги, Журнала, Учебного пособия.
2. **Абонент мобильного оператора.** Возможности: оформить договор; открыть счет и номер; редактировать учетную запись абонента; получить всю

- актуальную информацию по номеру абонента; проверить состояние баланса и остаток трафика; просмотреть детализацию и информацию о платежах; сменить тарифный план, оператора; пополнить счет; закрыть счет и номер. Добавить специализированные методы для Учетной записи интернет и корпоративного абонента.
3. **Конфеты.** Возможности: выпустить партию конфет; получить информацию по продукции определенного производителя; добавить и редактировать информацию о продукции; добавить новые ингредиенты; создать новый тип конфет; снять с производства определенный тип конфет. Добавить специализированные методы для Конфеты, Шоколад, Леденец, Ирис.
 4. **Растение.** Возможности: задать и редактировать информацию о растении; получить информацию об имеющихся растениях в оранжерее/растениях определенного вида/месте происхождения; закупить новый вид растений; произвести полив; задать температуру; установить освещение; выкопать определенный вид растений (убрать из оранжереи). Добавить специализированные методы для Кустовое растение, Цветковое растение, Комнатное растение.
 5. **Авиарейс.** Возможности: получить или изменить различную информацию о рейсе: номер рейса, пункт назначения, марка судна, статус судна (загрузка, разгрузка, заправка, в пути, в ремонте, готов к вылету, требуется ремонт); узнать среднее время рейса; узнать снаряженную массу; узнать количество топлива; отправить в пункт назначения; отремонтировать; заправить; загрузить; разгрузить; узнать хрупкость/ценность груза. Добавить дополнительные возможности для грузового и пассажирского рейса.
 6. **Вагон.** Возможности: получить или изменить различную информацию о вагоне: регистрационный номер вагона, пункт назначения, владелец, статус (загрузка, разгрузка, в пути, в ремонте, готов к отправке, требуется ремонт); узнать снаряженную массу; отправить в пункт назначения; обслужить; отремонтировать; загрузить; разгрузить; узнать хрупкость/ценность груза. Добавить дополнительные возможности для грузового и пассажирского вагонов.
 7. **Средство передвижения.** Возможности: получить или изменить различную информацию о средстве передвижения: регистрационный номер, марка, модель, VIN-номер, владелец, тип движущей силы; заправить\отремонтировать\обслужить; пройти техосмотр. Добавить дополнительные возможности для автомобиля, велосипеда, самоката, мотоцикла, квадроцикла.
 8. **Лекарственный препарат.** Возможности: добавить действующее вещество; рассчитать дозировку; провести исследование вещества; изменить статус вещества (запрещенное, по рецепту, разрешенное); получить и изменить информации о действующем веществе. Добавить дополнительные возможности для Лекарства, Мази, Таблетки и Раствора.
 9. **Самолеты.** Возможности: пройти техосмотр; отремонтировать; осуществить рейс; заправить; получить\изменить информацию о транспортном

- средстве. Добавить дополнительные возможности для самолета (военного и гражданского самолета).
10. **Дом.** Возможности: построить дом; рассчитать цену за квадратный метр; узнать сколько комнат; увеличить площадь; сдавать дом в аренду; сделать ремонт (в какой-либо комнате). Добавить специализированные методы для Дома, Офисного здания, Торгового центра.
 11. **Банковский вклад.** Возможности: изменить продолжительность вклада (бессрочный/долгосрочный/краткосрочный); узнать, какой вклад (отзывной или безотзывной); закрыть один вклад и открыть новый с такими же условиями, но в другой валюте; рассчитать начисление процентов по вкладу.
 12. **Компьютер.** Возможности: создать новую модель компьютера; установить цену; добавить объем оперативной памяти; изменить разрешение экрана; изменить процессор компьютера; добавить новые комплектующие в базовую комплектацию. Добавить специализированные методы для Computer, SmartPhone, Pad.
 13. **Город.** Возможности: вывести основную информацию о городе (страна, год основания, популяция, площадь, расположение, бюджет, действующий мэр, язык общения); выбрать нового мэра города; увеличить/уменьшить популяцию населения; редактировать бюджет; изменить статус города (закрытие на карантин при определенном проценте зараженных); ввести/изменить закон. Добавить специализированные методы для Города, Деревни, Хутора.
 14. **Товар (Product).** Возможности: добавить товар; получить данные о товаре: id, наименование, UPC, производитель, цена, срок хранения, количество; изменить данные о товаре; переместить товар на склад, переместить товар в торговый зал; оплатить товар; списать товар. Добавить специализированные методы для масла, молока, творога.
 15. **Пациент (Patient).** Возможности: регистрировать пациента; оформить договор обслуживания; получить персональные данные: id, фамилия, имя, отчество, адрес, телефон, номер медицинской карты; изменить персональные данные; записаться на прием к врачу; записать диагноз/назначенное лечение; сдать лабораторные исследования; получить историю болезни; получить диагноз/назначение лечение; оплатить услуги; отказаться от обслуживания; возобновить обслуживание. Добавить специализированные методы для Пациента инфекционного отделения, Пациента стационара, Пациента с COVID-19.
 16. **Покупатель (Customer).** Возможности: оформить договор; получить/изменить персональные данные; добавить покупки; получить оплату; добавить скидки; расторгнуть договор; возобновить договор. Добавить специализированные методы для Оптового покупателя, VIP-покупателя.
 17. **Сувениры.** Возможности: просмотреть сувенир поставщика; создать макет; редактировать макет; оформить/расторгнуть договор с поставщиком;

- оплатить сувенир; отказаться от сувенира; произвести сувенир; убрать сувенир. Добавить специализированные методы для Блокнота, Майки, Магнита.
18. **Столовые приборы.** Возможности: просмотреть приборы определенного бренда; изменить материал изготовления приборов; произвести партию приборов; остановить производство приборов; оформить договор на производство приборов; оценить стоимость приборов; реставрировать приборы. Добавить специализированные методы для Вилка, Нож, Палочка для еды.
 19. **Планета.** Возможности: добавить планету; сообщить об открытии; добавить/изменить характеристики; рассчитать период вращения; построить модель планеты; изменить модель планеты; заказать дополнительные исследования; купить индивидуальное название; добавить снимки; напечатать отчет о планете; выбрать планеты по параметру. Добавить специализированные методы для Газовых гигантов, Ледяных гигантов, Карликовых планет.
 20. **Погода.** Возможности: ввести/получить данные мониторинга метеорологических условий; запросить данные; анализировать карты погоды; анализировать аэрологическую информацию; анализировать климатические данные; рассчитать прогнозные показатели; уточнить прогнозные показатели; обновить информацию; ввести фактические показатели; отправить на анализ; вывести прогноз погоды; оценить прогноз. Добавить специализированные методы для Срединный регион, Нижний регион, Верхний регион.
 21. **Электроинструменты.** Возможности: создать модель инструмента; ввести характеристики инструмента; добавить/изменить комплектующие; рассчитать стоимость производства; добавить чертежи; отправить на доработку; отказать в производстве; запустить в производство; отправить на тестирование; получить результаты тестов. Добавить специализированные методы для Пиление, Резка, Обработка дерева, Сверление отверстий.
 22. **Видеотека.** Возможности: создать/удалить видеотеку; создать/изменить/удалить актера/режиссера/фильм; отобразить информацию о самых новых/популярных фильмах; отобразить список фильмов; открыть фильм; улучшить/ухудшить качество видео; включить/выключить звук; закрыть фильм. Добавить специализированные методы для Библиотека, Сервер.
 23. **Расписание занятий.** Возможности: создать/удалить расписание; создать/изменить/удалить предмет/преподавателя; отобразить расписание для конкретного преподавателя/конкретной группы; отобразить список всех предметов/преподавателей; изменить аудиторию; отменить занятие; добавить дополнительное занятие; добавить выходной; посчитать количество занятий у преподавателя/группы. Добавить специализированные методы для Расписания занятий в школе, Расписания занятий в автошколе.
 24. **Письма.** Возможности: создать/удалить письмо, добавить/удалить отправителя; добавить текст письма; добавить/удалить приложения к письму; поставить дату отправки; отправить/принять письмо; проверить адресат на существование. Добавить специализированные методы для Заказного письма.

25. **Абитуриент.** Возможности: получить аттестат; заполнить заявление; зарегистрироваться\пройти вступительные испытания; подать документы в приемную комиссию. Добавить дополнительные возможности для Абитуриента БГУ, Абитуриента БГУИР.
26. **Литературное произведение.** Возможности: написать произведение; отредактировать произведение; опубликовать произведение; посвятить произведение кому-либо/чему-либо; перевести на другой язык; продать права на использование произведения; получить критику на произведение. Добавить дополнительные возможности для Стихотворение, Белый стих.

Тестовые задания к главе 6

Вопрос 6.1.

Дан код:

```
interface Readable {
    void read();
}
abstract class Editor {
    void edit(){}
}
interface Lexicon /* 1 */ /* 2 */ {
    /* 3 */
}
```

Какой набор строк, вставленный соответственно вместо комментариев 1, 2 и 3, позволит компилироваться интерфейсу **Lexicon** без ошибок? (выбрать один)

- a) extends; Editor; public void edit(){}
- b) implements; Readable; void read(){}
- c) extends; Editor; protected void edit(){}
- d) implements; Readable; public void read(){}

Вопрос 6.2.

Даны объявления интерфейсов и класс, их реализующий:

```
interface First {
    int i = 1;
}
interface Second {
    int i = 2;
}
public classClazz implements Second, First {
    public static void main(String[] args) {
        System.out.println(i);
    }
}
```

Каким будет результат компиляции и выполнения? (выбрать один)

- a) 1
- b) 2
- c) compilation fails
- d) runtime error

Вопрос 6.3.

Даны объявления класса и интерфейса:

```
interface Readable{
    void read();
}
class Read {
    public void read(){}
}
```

Какие объявления классов и интерфейсов корректны? (выбрать два)

- a) **interface** Editable **extends** Readable{ }
- b) **interface** Editable **implements** Readable{ }
- c) **class** Edit **implements** Readable **extends** Read { }
- d) **class** Edit **extends** Read **implements** Readable { }
- e) **class** Edit **implements** Readable, Read { }
- f) **class** Edit **extends** Readable, Read { }

Вопрос 6.4.

Даны объявления классов и интерфейсов:

```
interface First {}
interface Second extends First {}
class Klass implements Second{}
classClazz implements First {}
```

Какие объявления ссылок корректны? (выбрать три)

- a) First a = (First)**new**Clazz();
- b) Second b = (First)new Klass();
- c) First c = (First)**new** Klass();
- d) Second d = (Second) **new**Clazz();
- e) Klass e = (Second)newClazz();

ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ

Функциональное программирование нужно изучать под водой. Там слез не видно.

Функциональное программирование стало причиной появления в Java функциональных интерфейсов и лямбда-выражений, или попросту замыканий. Функциональное программирование снижает количество кода, но усложняет его понимание, что в общем противоречит парадигме о простоте языка. Но тенденции развития программирования требуют расширения возможностей.

В ФП практически отсутствует механизм создания абстракций, что характерно для ООП. ООП и ФП — это набор идей, а не прямых указаний к действию. Если разграничить их зоны ответственности, то обе парадигмы могут сосуществовать без конфликтов.

Все классы в Java-разработке условно можно разделить на две группы: классы-сущности, обладающие информацией, и классы-действия, манипулирующие этой информацией. Принципы ООП хорошо работают с абстрагированием данных, ФП — с классами, выполняющими действия, да и то не со всеми. И грамотное взаимодействие разных парадигм программирования позволит разрабатывать качественный код.

Функциональное программирование определило появление в Java функциональных интерфейсов и лямбда-выражений, или, по-другому, замыканий.

Методы `default` и `static` в интерфейсах

В Java 8 разрешено объявлять неабстрактные и статические методы в интерфейсах. Интерфейсы по-прежнему могут содержать абстрактные методы.

```
/* # 1 # интерфейс с дефолтным и статическим методом # Service.java */
```

```
package by.epam.learn.function;  
public interface Service {  
    default void anOperation() { // public  
        System.out.println("Service anOperation");  
        this.method();  
    }  
}
```

```
private void method() { // default not required
    System.out.println("private method");
}
static void action() { // public
    System.out.println("Service static action");
}
int define(int x1, int y1); // public abstract
void load(); // public abstract
}
```

При реализации классом такого интерфейса реализуются только абстрактные методы, **default**-методы могут переопределяться при необходимости.

Статический метод вызывается классическим способом, без реализации содержащего его интерфейса:

```
Service.action();
```

Для вызова **default**-метода нужно предоставить реализацию интерфейса.

```
/* # 2 # реализация интерфейса # ServiceImpl.java */
```

```
package by.epam.learn.function.impl;
import by.epam.learn.function.Service;
public class ServiceImpl implements Service {
    @Override
    public int define(int x, int y) {
        return x + y;
    }
    @Override
    public void load() {
        System.out.println("load()");
    }
}
```

Тогда вызов методов интерфейса **Service** можно представить в виде:

```
/* # 3 # использование реализации интерфейса # ServiceMain.java */
```

```
package by.epam.learn.function;
import by.epam.learn.function.impl.ServiceImpl;
public class ServiceMain {
    public static void main(String[] args) {
        Service.action(); // static method
        ServiceImpl service = new ServiceImpl();
        service.define(1, 2);
        service.load();
        service.anOperation(); // default method
    }
}
```

Появление методов по умолчанию в интерфейсах разрешило множественное наследование поведения, что не так уж редко встречается в практическом программировании.

Однако если класс реализует два интерфейса с **default**-методами, сигнатуры которых совпадают, то компилятор выдаст сообщение об ошибке, так как невозможно будет определить принадлежность метода при его вызове на объекте класса. При реализации классом методы интерфейса равноправны.

```
/* # 4 # интерфейс с методом, идентичным методу интерфейса Service
# ServiceApp.java */
```

```
package by.epam.learn.function;
public interface ServiceApp {
    default void anOperation() { // public
        System.out.println("ServiceApp anOperation");
    }
}
```

Единственным решением методов с одинаковыми сигнатурами при реализации интерфейсов **Service** и **ServiceApp** будет обязательное переопределение метода.

```
/* # 5 # реализация двух интерфейсов # ServiceTwoImpl.java */
```

```
package by.epam.learn.function.impl;
import by.epam.learn.function.Service;
import by.epam.learn.function.ServiceApp;
public class ServiceTwoImpl implements Service, ServiceApp {
    @Override
    public void anOperation() {
        System.out.println("necessary method implementation");
    }
    @Override
    public int define(int x, int y) {
        return x - y;
    }
    @Override
    public void load() {
    }
}
```

Такая же ситуация возникнет и при наследовании этих двух интерфейсов третьим.

```
/* # 6 # наследование двух интерфейсов # ServiceCommon.java */
```

```
package by.epam.learn.function;
public interface ServiceCommon extends Service, ServiceApp {
    @Override
```

```
default void anOperation() {
    System.out.println("necessary method implementation");
}
}
```

Со статическими методами подобных проблем не возникает по причине вызова статического метода через имя интерфейса его определяющего.

Функциональные интерфейсы

Функциональный интерфейс *должен* иметь *один* единственный абстрактный метод и любое число статических и **default**-методов. Для объявления такого интерфейса используется аннотация **@FunctionalInterface**. Интерфейс, помеченный этой аннотацией, предполагает его использование в виде лямбда-выражения, которое предлагает более лаконичный синтаксис при создании функций-объектов.

В Java объявлено достаточно интерфейсов с одним абстрактным методом, но тем не менее не помеченных аннотацией **@FunctionalInterface**. В общем случае отсутствие аннотации не предполагает использование такого интерфейса в лямбда-выражении, но и не запрещает его. К таким интерфейсам относятся **Comparable**, **Runnable**, **Callable** и другие. Интерфейсы без аннотации **@FunctionalInterface** желательно использовать как обычную абстракцию с ключевым словом **implements**.

Основные функциональные интерфейсы размещены в пакете **java.util.function**. К функциональным интерфейсам теперь относится и **Comparator**.

Прежде чем приступить к изучению стандартных функциональных интерфейсов, есть смысл создать собственный интерфейс и рассмотреть его свойства и применение.

```
/* # 7 # собственный функциональный интерфейс # ShapeService.java */
```

```
package by.epam.learn.function;
@FunctionalInterface
public interface ShapeService {
    double perimeter(double a, double b);
}
```

Если попробовать добавить еще один абстрактный метод или удалить существующий, то возникнет ошибка компиляции, так как интерфейс прекращает быть функциональным.

Реализация интерфейса в парадигме ООП в виде анонимного класса может выглядеть так:

```
/* # 8 # наследование двух интерфейсов # RectangleService.java */
```

```
package by.epam.learn.function.impl;
import by.epam.learn.function.ShapeService;
```

```
public class RectangleService implements ShapeService {
    @Override
    public double perimeter(double a, double b) {
        return 2 * (a + b);
    }
}
```

В виде лямбда реализация выглядит так:

```
ShapeService rectangleService = (a, b) -> 2 * (a + b);
```

На самом деле лямбда-выражение представляет сокращенную запись анонимного класса.

```
ShapeService rectangleService = new ShapeService() {
    @Override
    public double perimeter(double a, double b) {
        return 2 * (a + b);
    }
};
```

Эволюция в лямбда начинается с того, что опускается конструктор анонимного класса и имя метода интерфейса. Так как метод единственный в интерфейсе, то и его имя можно не упоминать. Параметры метода отделяются от тела метода оператором «->»:

```
ShapeService rectangleService = (double a, double b) -> {
    return 2 * (a + b);
};
```

Если тело метода состоит из одного оператора, то и фигурные скобки можно опустить.

```
ShapeService rectangleService = (double a, double b) -> 2 * (a + b);
```

Типы параметров метода также можно опустить, так как предполагается, что они известны из сигнатуры единственного абстрактного метода.

```
ShapeService rectangleService = (a, b) -> 2 * (a + b);
```

Применение лямбда-выражений становится возможным только для функциональных интерфейсов из-за единственного абстрактного метода, так как исчезает необходимость явно указывать имя метода.

```
/* # 9 # функциональный интерфейс как параметр метода # ActionType.java */
```

```
package by.epam.learn.function;
public class ActionType {
    private double x;
    private double y;
    public ActionType(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

```
public double action(ShapeService service) {
    return 10 * service.perimeter(x, y);
}
}
```

Тогда обращение к функциональному интерфейсу будет следующим:

```
double result = new ActionType(3, 5).action((a, b) -> (a + b) * 4);
```

Функция как объект создается во время выполнения. Внешне создается впечатление, что в метод передается функция. На самом деле в функцию передается объект, в который «завернута» функция. У этого объекта интересен только функционал, поэтому при практическом использовании программист может и не знать названия метода, который реально будет вызван.

Основная часть функциональных интерфейсов, более сорока, размещена в пакете **java.util.function**. Сами по себе все эти интерфейсы бесполезны и не имеют никакого смысла, но в виде лямбда-выражений обеспечивают практически всю функциональность методов **Stream** и **Collectors** при обработке потоков данных. Они существуют, прежде всего, для представления структуры, а не семантики.

Все представленные функциональные интерфейсы можно разделить на четыре группы: **Function<T,R>**, **Predicate<T>**, **Consumer<T>** и **Supplier<T>**.

Интерфейс Predicate

Интерфейс **Predicate<T>** представляет метод **boolean test(T t)**, возвращающий булево значение в зависимости от выполнения условия на объекте типа **T**. Основная область применения: выбор\поиск\фильтрация элементов из *stream* или коллекции по условию.

Реализация предиката может выглядеть следующим образом:

```
Predicate<String> predicateStr1 = s -> s.length() < 2;

Predicate<String> predicateStr2 = String::isBlank; // eq. s -> s.isBlank();

Predicate<Integer> predicateInt = i -> i >= 9;
```

Вызов этих функций довольно примитивен и опять же на практике в явном виде никогда не используется:

```
System.out.println(predicateStr1.test("java")); // false
System.out.println(predicateStr2.test(" ")); // true
System.out.println(predicateInt.test(16)); // true
```

Реальное применение предикатов в задаче, где дан массив строк, который преобразуется в *stream*, затем вызывается промежуточный метод **filter(Predicate<? super T> predicate)**, удаляющий из потока объекты, не удовлетворяющие условию предиката.

```
String[] arrayStr = {"as", "a", "the", " ", "d", "on", "and", ""};
System.out.println(Arrays.stream(arrayStr)
    .filter(s -> s.length() > 2)
    .collect(Collectors.toList()));
```

В результате будет получен список вида:

[the, and]

Терминальный метод **anyMatch(Predicate<? super T> predicate)** ищет в потоке хотя бы один объект, удовлетворяющий предикату, и только в этом случае возвратит истину. Другой метод **allMatch(Predicate<? super T> predicate)** возвратит истину, если все объекты потока удовлетворяют условию.

```
System.out.println(Arrays.stream(arrayStr).anyMatch(String::isBlank)); // true
int[] arrayInt = {1, 3, 5, 9, 17, 33, 65};
System.out.println(Arrays.stream(arrayInt).allMatch(i -> i >= 1)); // true
```

Композиции предикатов можно строить, используя методы: **default Predicate<T> and(Predicate<? super T> other)** — логическое «и».

```
Predicate<String> predicate1 = s -> s.contains("a");
System.out.println(Arrays.stream(arrayStr)
    .filter(predicate1.and(s -> s.contains("n")))
    .collect(Collectors.toList()));
```

Двум условиям одновременно удовлетворяет только одна строка:

[and]

Или, что тоже самое, но без использования ссылки на функцию **predicate1**:

```
System.out.println(Arrays.stream(arrayStr)
    .filter(((Predicate<String>) s -> s.contains("a")).and(s -> s.contains("n")))
    .collect(Collectors.toList()));
```

default Predicate<T> or(Predicate<? super T> other) — логическое «или».

Синтаксическая реализация подразумевает применение соответствующего типа данных в условии предиката. Метод **or()** тогда можно использовать для выбора из *stream* чисел меньше чем 4 или больше чем 32:

```
System.out.println(Arrays.stream(arrayInt)
    .filter(((IntPredicate) i -> i > 32).or(i -> i < 4))
    .boxed()
    .collect(Collectors.toList()));
```

Результатом будет такой набор значений:

Выведет в консоль: **[1, 3, 33, 65]**

Для упрощения работы с потоками чисел разработаны предикаты **IntPredicate**, **DoublePredicate**, **LongPredicate** с практически идентичным набором методов построения логических выражений.

default Predicate<T> negate() — логическое отрицание предиката.

Из *stream* строк будут выбраны все строки, которые не совпадают со строкой **and**.

```
System.out.println(Arrays.stream(arrayStr)
    .filter(((Predicate<String>)s -> s.contains("and")).negate())
    .collect(Collectors.toList()));
```

Выведет в консоль: [as, a, the, , d, on,]

static Predicate<T> not(Predicate<? super T> target) — более короткий вариант логического отрицания.

```
System.out.println(Arrays.stream(arrayStr)
    .filter(Predicate.not(s -> s.contains("and")))
    .collect(Collectors.toList()));
```

Результатом выполнения будет такой же набор данных, как и в предыдущем примере.

static Predicate<T> isEqual(Object targetRef) — возвращает предикат эквивалент метода **equals()** класса **Object**. Применяется для поиска точного совпадения объектов:

```
System.out.println(Arrays.stream(arrayStr)
    .filter(Predicate.isEqual("and"))
    .collect(Collectors.toList()));
```

Результат: **and**.

Обычный предикат, эквивалентный приведенному, записывается в виде:

```
s -> s.equals("and")
```

В пакет **java.util.function** объявлен еще один интерфейс-предикат **BiPredicate<T, U>** с абстрактным методом **boolean test(T t, U u)**.

```
BiPredicate<String, Integer> biPredicate = (s, max) -> s.length() <= max;
System.out.println(biPredicate.test("java", 7));
```

Результат: **true**.

Некоторые способы использования предикатов методами интерфейса **Stream**:

```
filter(Predicate<? super T> predicate);
remove(Predicate<? super E> filter);
allMatch(Predicate<? super T> predicate);
noneMatch(Predicate<? super T> predicate);
anyMatch(Predicate<? super T> predicate);
takeWhile(Predicate<? super T> predicate);
iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next).
```

Интерфейс Function

Основной абстрактный метод **R apply(T t)** принимает объект типа **T** и возвращает объект типа **R**. Его задача: выполнить действие над объектом одного типа и вернуть объект другого типа.

Создание объекта **Function** может выглядеть следующим образом:

```
Function<String, Integer> fun1 = s -> s.length();// String to Integer
Function<Integer, String> fun2 = i -> Integer.toBinaryString(i);//int to String
```

Первая функция получает строку и преобразует ее в число — длину строки.

Вторая — число преобразует в двоичное представление в виде строки.

Вызов этих функций в явном виде имеет мало смысла:

```
System.out.println(fun1.apply("internacionalization"));
```

Выведет в консоль результат: **20**

```
System.out.println(fun2.apply(20));
```

Выведет в консоль результат: **10100**

Реальное использование функциональных интерфейсов приводится в главе «Коллекции и Stream API», но понятный пример следует привести здесь.

Пусть дан массив строк, который преобразуется в *stream*, затем вызывается метод **map(Function<? super T, ? extends R> mapper)**, преобразующий поток объектов **String** в поток чисел-длин строк.

```
String[] arrayStr = {"as", "a", "the", "d", "on", "and"};
System.out.println(Arrays.stream(arrayStr)
    .map(fun1)
    .collect(Collectors.toList()));
```

Или, что является корректным и общепринятым использованием функционального интерфейса:

```
System.out.println(Arrays.stream(arrayStr)
    .map(s -> s.length())
    .collect(Collectors.toList()));
```

В обоих случаях будет выведено:

[2, 1, 3, 1, 2, 3]

Следующие два метода позволяют построить композицию из двух и более функций, которые будут в итоге вызываться как одна.

default <V> Function<V, R> compose(Function<? super V, ? extends T> before) — возвращает составную функцию, которая сначала применяет функцию **before** к своему входу, а затем применяет эту функцию к результату:

```
Function<Integer, Integer> fun3 = fun1.compose(fun2);
```

или

```
Function<Integer, Integer> fun3 = fun1.compose(i -> Integer.toBinaryString(i));
```

Первой будет вызвана функция **fun2**, она преобразует число в его двоичное представление **10100** в виде строки, затем на результат будет вызвана функция **fun2**, которая вычислит длину строки. То есть на входе в композицию функций будет передано число, и на выходе получится число. Если изменить функцию **fun2** так, чтобы она возвращала какой-либо другой тип данных, то на входе в композицию функций будет число, а на выходе другой тип данных.

```
System.out.println(fun1.compose(fun2).apply(17));
```

Даст результат: 5

Если применить эту композицию к массиву чисел, то выглядеть это будет более естественно:

```
int[] arrayInt = { 1, 3, 5, 9, 17, 33, 65};
System.out.println(Arrays.stream(arrayInt)
    .boxed()
    .map(fun1.compose(i -> Integer.toBinaryString(i)))
    .collect(Collectors.toList()));
```

Или, что тоже самое, но без использования ссылки **fun1**:

```
System.out.println(Arrays.stream(arrayInt)
    .boxed()
    .map(((Function<String, Integer>)s -> s.length())
        .compose(i -> Integer.toBinaryString(i)))
    .collect(Collectors.toList()));
```

Выведет в консоль: [1, 2, 3, 4, 5, 6, 7]

default <V> Function<T,V> andThen(Function<? super R, ? extends V> after) — возвращает составную функцию, которая сначала применяет эту функцию к своему входу, а затем применяет функцию **after** к результату.

Метод **andThen()** вызовет функции в порядке, обратном методу **compose()**.

```
Function<String, String> fun4 = fun1.andThen(fun2);
```

или

```
Function<String, String> fun4 = fun1.andThen(i -> Integer.toBinaryString(i));
```

Первой будет вызвана функция **fun1**, вычисляющая длину строки, а после — функция **fun2**, преобразующая число в двоичное представление в виде строки.

```
System.out.println(fun1.andThen(fun2).apply("java"));
```

Выведет в консоль: 100

Если применить эту композицию к массиву строк, то выглядеть это будет более естественно:

```
System.out.println(Arrays.stream(arrayStr)
    .map(fun1.andThen(i -> Integer.toBinaryString(i)))
    .collect(Collectors.toList()));
```

Выведет в консоль: **[10, 1, 11, 1, 10, 11]**

Или, что тоже самое, но без использования ссылки на функцию **fun1**:

```
System.out.println(Arrays.stream(arrayStr)
    .map(((Function<String, Integer>)s -> s.length()).andThen(i -> Integer.toBinaryString(i)))
    .collect(Collectors.toList()));
```

Еще один метод **static <T> Function<T, T> identity()** — возвращает функцию, которая всегда возвращает свой входной аргумент.

Все интерфейсы пакета **java.util.function**, имеющие в своем названии слова **Function** и **Operator**, на самом деле являются вариациями интерфейса **Function<T, R>**, не наследуя его при этом.

Интерфейс **UnaryOperator<T>** объявляет метод **T apply(T t)**. Отличием этого интерфейса от **Function** является только то, что и принимаемое, и возвращаемое значения метода **T apply(T t)** должны быть одного и того же типа, что в простом примере может соответствовать оператору инкремента:

```
UnaryOperator<Integer> operator = i -> ++i;
System.out.println(operator.apply(1)); // 2
```

Результат: **2**

Интерфейс **BiFunction<T, U, R>** объявляет метод **R apply(T t, U u)** с двумя параметрами, что также представляется более широкой версией **Function**.

```
BiFunction<Double, String, Integer> bi = (d, s) -> (Double.toString(d) + s).length();
int length = bi.apply(1.23, "java");
System.out.println(length);
```

Выведет в консоль результат: **8**

К еще более специализированным интерфейсам можно отнести:

ToIntFunction<T> — метод которого **int applyAsInt(T t)** принимает любой тип данных, но должен возвращать значение типа **int**.

IntFunction<R> — наоборот, его метод **R apply(int value)** принимает значение типа **int**, но может возвращать значение любого типа.

Интерфейс **BinaryOperator<T, T>** объявляет метод **T apply(T t1, T t2)**, что соответствует обычному бинарному оператору:

```
BinaryOperator<String> binaryOperator = (s1, s2) -> s1 + s2.toUpperCase();
System.out.println(binaryOperator.apply("oracle", "epam"));
```

Выведет в консоль: **oracleEPAM**

К интерфейсам группы **Function** также можно отнести интерфейс **java.util.Comparator<T>** с его методом **int compare(T o1, T o2)**. То есть это бинарная функция, принимающая два объекта одного типа и возвращающая значение

типа **int**. У компаратора есть своя функциональная роль по сравнению объектов на больше\меньше либо равно, используется коллекциями и *stream* для сортировок, будет рассмотрена в главе «Коллекции и Stream API».

Некоторые способы использования функциональных интерфейсов, аналогичных **Function**, методами интерфейса **Stream**:

```
reduce(BinaryOperator<T> accumulator);
sorted(Comparator<? super T> order);
max(Comparator<? super T> comparator);
min(Comparator<? super T> comparator);
map(Function<? super T,? extends R> mapper);
flatMap(Function<? super T,? extends Stream<? extends R>> mapper);
iterate(T seed, UnaryOperator<T> t);
mapToInt(ToIntFunction<? super T> mapper);
toArray(IntFunction<A[]> generator).
```

Интерфейс Consumer

Интерфейс **Consumer<T>** представляет абстрактный метод **void accept(T t)**, функция, принимающая объект типа **T** и выполняющая над ним некоторое действие. Результат действия можно сохранить во внешнем объекте, например, коллекции или вывести в поток вывода, например, в файл или на консоль.

В следующем примере **consumer** преобразует строку на основе разделителя в массив строк:

```
String str = "as a- the-d -on and";
String regex = "-";
Consumer<String> consumer = s -> System.out.println(Arrays.toString(s.split(regex)));
consumer.accept(str);
```

Выведет в консоль: **[as a, the, d , on and]**

Если изменить разделитель на

```
String regex = "\\s";
```

То будет выведено уже:

[as, a-, the-d, -on, and]

Еще один пример на **Consumer** с изменением поля объекта из массива объектов класса **RightTriangle** из примера #11 главы «Интерфейсы и аннотации».

```
/* # 10 # применение Consumer к массиву объектов # ConsumerMain.java */
```

```
package by.epam.learn.function;
import by.epam.learn.entity.RightTriangle;
import java.util.Arrays;
```

```
public class ConsumerMain {
    public static void main(String[] args) {
        RightTriangle[] triangles = {new RightTriangle(1, 2), new RightTriangle(3, 4)};
        Arrays.stream(triangles)
            .forEach(t -> t.setSideA(t.getSideA() * 10));
        System.out.println(Arrays.toString(triangles));
    }
}
```

Значение каждого поля **sideA** будет умножено на число **10**. В результате будет выведено:

[RightTriangle[sideA=10.0, sideB=2.0], RightTriangle[sideA=30.0, sideB=4.0]]

Вспомогательный метод **default Consumer<T> andThen(Consumer<? super T> after)** позволяет построить композицию из двух и более действий как одного целого:

```
String str = "as a- the-d -on and";
String regex1 = "\\s";
Consumer<String> consumer1 = s -> Arrays.toString(s.split(regex1));
String regex2 ="a";
Consumer<String> consumer2 = consumer1
    .andThen(s -> System.out.println(Arrays.toString(s.split(regex2))));
consumer2.accept(str);
```

Первый **consumer1** удалит из строки все знаки **regex1**, а второй — все знаки **regex2**. В итоге будет получено:

[, s , - the-d -on , nd]

Интерфейс **BiConsumer<T, U>** объявляет метод **void accept(T t, U u)** с двумя параметрами, что представляется как расширение возможностей **Consumer**. Применение этого метода позволит передавать множитель стороны треугольника в метод **accept()**.

```
RightTriangle triangle = new RightTriangle(1, 2);
BiConsumer<RightTriangle, Integer> consumer = (t, n) -> t.setSideA(t.getSideA()*n);
consumer.accept(triangle, 50);
System.out.println(triangle);
```

Результат:

RightTriangle[sideA=50.0, sideB=2.0]

Более специализированный родственный интерфейс **IntConsumer** объявляет метод **void accept(int value)** для действий над целым числом. Аналогичные методы содержат интерфейсы **DoubleConsumer**, **LongConsumer**.

Родственный интерфейсу **BiConsumer** интерфейс **ObjIntConsumer<T>** объявляет метод **void accept(T t, int value)**. С его помощью предыдущий пример переписывается несколько проще:

```
ObjIntConsumer<RightTriangle> consumer = (t, n) -> t.setSideA(t.getSideA() * n);
```

Аналогичные методы содержат интерфейсы **ObjDoubleConsumer**, **ObjLongConsumer**.

Некоторые способы использования **Consumer** методами интерфейса **Stream**:

```
forEach(Consumer<? super T> action);
forEachOrdered(Consumer<? super T> action);
peek(Consumer<? super T> action).
```

Интерфейс Supplier

Интерфейс **Supplier<T>** возвращает новый объект типа **T** методом **T get()**. Предназначен для создания новых объектов. Метод **get()** единственный в интерфейсе. Аналогичные интерфейсы предназначены для создания значений базовых типов:

BooleanSupplier и его метод **boolean getAsBoolean()**;

DoubleSupplier и его метод **double getAsDouble()**;

IntSupplier и его метод **int getAsInt()**;

LongSupplier и его метод **long getAsLong()**.

Прямое применение интерфейса для создания объектов выглядит следующим образом:

```
Supplier<StringBuilder> supplier = () -> new StringBuilder("java");
StringBuilder builder = supplier.get();
Supplier<int[]> supplier2 = () -> new int[10];
int[] arr = supplier2.get();
```

Можно завернуть **Supplier** в фабрику создания корректных массивов:

```
/* # 11 # фабрика массивов на основе Supplier # ArrayFactory.java */
```

```
package by.epam.learn.function.supplier;
import java.util.function.Supplier;
public class ArrayFactory {
    public static Supplier<int[]> buildArray(int size) {
        final int length = size > 0 ? size : 1;
        return () -> new int[length];
    }
}
```

Использование такой фабрики будет иметь вид:

```
int[] array = ArrayFactory.buildArray(10).get();
```

Хорошим использованием **Supplier** может быть генератор случайных чисел или уникальных идентификаторов на его основе.

Ниже приведен пример с имитацией простейшей игры в рулетку.

```
/* # 12 # генератор случайных чисел на основе Supplier # RouletteMain.java */

package by.epam.learn.function.supplier;
import java.util.Random;
import java.util.function.Supplier;
public class RouletteMain {
    public static void main(String[] args) {
        int yourNumber = 7;
        if (yourNumber < 0 || yourNumber > 36) {
            System.out.println("number should be 0<= and <=36");
            return;
        }
        //random value between 0 and 36
        Supplier<Integer> numberSupplier = () -> new Random().nextInt(37);
        int rouletteResult = numberSupplier.get();
        System.out.println("Roulette : " + rouletteResult + ", your bet: " + yourNumber);
        String result = rouletteResult == yourNumber ? "You win!" : "You lose";
        System.out.println(result);
    }
}
```

Строки с генерацией числа лучше завернуть в метод, который и будет возвращать объект **Supplier**.

Еще один пример показывает как **Supplier** позволяет немного сократить код операции и процесс преобразования его результата к строке:

```
/* # 13 # создание составного объекта на основе Supplier # SupplierMain.java */

package by.epam.learn.function.supplier;
import java.math.BigDecimal;
import java.math.MathContext;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.function.Supplier;
public class SupplierMain {
    public static void main(String[] args) {
        Supplier<String> supplierNumber = plus(1.123450989f, 2.000001f);
        System.out.println("res= " + supplierNumber.get());
        Supplier<String> supplierTime = buildTime("yyyy-MM-dd HH:mm:ss");
        System.out.println("time= " + supplierTime.get());
    }
    static Supplier<String> buildTime(String timePattern){
        DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern(timePattern);
        return () -> timeFormatter.format(LocalDateTime.now());
    }
    static Supplier<String> plus(float a, float b) {
        BigDecimal decimal = new BigDecimal(a);
        BigDecimal decimal2 = new BigDecimal(b);
    }
}
```

```
BigDecimal res = decimal.add(decimal2, MathContext.DECIMAL32);
return () -> res.toEngineeringString();
}
}
```

В результате будет выведено:

res= 3.123452

time= 2020-06-11 15:39:20

Использование **Supplier** методами интерфейса **Stream**:

```
generate(Supplier<? extends T> s);
collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator,
        BiConsumer<R, R> combiner).
```

Интерфейс Comparator

Интерфейс **java.util.Comparator<T>**, начиная с версии Java 8, приобрел свойства функционального интерфейса. Реализация интерфейса **Comparator<T>** представляет возможность его использования для сортировки наборов объектов конкретного типа по правилам, определенным для этого типа. Контракт интерфейса подразумевает реализацию метода **int compare(T ob1, T ob2)**, принимающего в качестве параметров два объекта, для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки.

Следует отметить, что реализация метода **equals()** класса **Object** предоставляет возможность проверить, эквивалентен один экземпляр другому или нет. На практике очень часто возникает необходимость сравнения объектов на больше/меньше либо равно. Метод **boolean equals(Object obj)**, также объявленный в интерфейсе **Comparator<T>**, рекомендуется переопределять, если экземпляр класса будет использоваться для хранения информации. Это необходимо для исключения противоречивой ситуации, когда для двух объектов метод **compare()** возвращает **0**, т.е. сообщает об их эквивалентности, в то же время метод **equals()** для этих же объектов возвращает **false**, так как данный метод не был никем определен и была использована его версия из класса **Object**. Кроме того, наличие метода **equals()** обеспечивает корректную работу метода семантического поиска и проверки на идентичность **contains(Object o)**, определенного в интерфейсе **java.util.Collection**, а следовательно, реализованного в любой коллекции.

С помощью лямбда-выражения можно привести базовую реализацию компаратора для строк по убыванию их длин.

```
Comparator<String> comparator = (s1, s2) -> s2.length() - s1.length();
```

Тогда сортировку массива строк по убыванию их длины можно провести способом:

```
String str = "and java course epam the rose lion wolf hero green white red white";
Arrays.stream(str.split("\\s"))
    .sorted(comparator)
    .forEach(s -> System.out.printf("%s ", s));
```

course green white white java epam rose lion wolf hero and the red

Метод **sorted(Comparator<? super T> c)** интерфейса **Stream<T>** автоматически вызывает метод **compare(T o1, T o2)** при сортировке элементов *stream*. Также при сортировке списка методами: **static <T> void sort(List<T> list, Comparator<? super T> c)** класса **Collections** и **void sort(Comparator<? super E> c)** интерфейса **List<T>** необходим компаратор.

Как вариант реализации компаратора можно использовать статический метод **comparing(Function<? super T, ? extends U> keyExtractor)** интерфейса **Comparator<T>**.

Кроме уже перечисленных методов, интерфейс **Comparator<T>** объявляет еще некоторые необходимые методы:

default Comparator<T> reversed() — примененный к уже созданному компаратору, делает сортировку в обратном направлении;

nullFirst(Comparator<? super T> comparator) и **nullLast(Comparator<? super T> comparator)** — применяются к компаратору для размещения всех **null** в начале или конце списка при сортировке;

thenComparing(Comparator<? super E> other), thenComparing(Function<? super T, ? extends U> keyExtractor) — **default**-методы, позволяющие произвести сортировку в сортировке. Например, все строки отсортировать по возрастанию длины и далее все строки с одинаковыми длинами отсортировать по алфавиту.

```
String str = "and java course epam the rose lion wolf hero green white red white";
Arrays.stream(str.split("\\s"))
    .sorted(Comparator.comparing(String::length).thenComparing(String::compareTo))
    .forEach(s -> System.out.printf("%s ", s));
```

Результат сортировки:

and red the epam hero java lion rose wolf green white white course

Если в процессе использования необходимы сортировки по различным полям класса, то реализацию компаратора можно вынести в отдельный класс. Также реализация компаратора, являясь логической частью класса-сущности, может быть его частью и представлена в виде статического внутреннего класса, была популярна до появления перечислений:

```
/* # 14 # класс-сущность с внутренними классами-компараторами # Order.java # */
```

```
package by.epam.learn.entity;
import java.util.Comparator;
public class Order {
    private long orderId;
    private double amount;
    // other code
    public static class IdComparator implements Comparator<Order> {
        @Override
        public int compare(Order o1, Order o2) {
            return Long.compare(o1.orderId, o2.orderId);
        }
    }
    public static class AmountComparator implements Comparator<Order> {
        @Override
        public int compare(Order o1, Order o2) {
            return Double.compare(o1.getAmount(), o2.getAmount());
        }
    }
}
```

Экземпляр компаратора создается обычным для внутренних классов способом:

```
Order.IdComparator comp = new Order.IdComparator();
```

Объявление внутри класса позволяет манипулировать доступом к его функциональности.

Интерфейс **Comparator** не рекомендуется имплементировать самому классу-сущности, для сортировки экземпляров которого он предназначен.

Возможности перечислений и интерфейса **Comparator<T>** позволили отойти от классической модели реализации компаратора и дать возможность перечислению объявить компаратор в качестве поля. Перечисление теперь обрабатывает компаратор:

```
/* # 15 # перечисление с полем-компаратором # OrderComparatorFunctional.java # */
```

```
package by.epam.learn.collection;
import by.epam.learn.entity.Order;
import java.util.Comparator;
public enum OrderComparatorFunctional {
    ID(Comparator.comparingLong(Order::getOrderId)),
    AMOUNT(Comparator.comparingDouble(Order::getAmount));
    private Comparator<Order> comparator;
    OrderComparatorFunctional(Comparator<Order> comparator) {
        this.comparator = comparator;
    }
    public Comparator<Order> getComparator() {
        return comparator;
    }
}
```

Обратиться к такому компаратору при сортировке можно так:

```
orders.sort(OrderComparatorFunctional.AMOUNT.getComparator());
```

где **orders** — список объектов типа **Order**. Списки будут подробно рассмотрены в главе «Коллекции и Stream API».

Возможен вариант полной ассоциации перечисления и компаратора. Для этого достаточно, чтобы класс-перечисление имплементировал интерфейс **Comparator<T>** и для каждого своего элемента предоставил реализацию метода **compare()**:

```
/* # 16 перечисление как компаратор # OrderComparatorClassic.java # */
```

```
package by.epam.learn.collection;
import by.epam.learn.entity.Order;
import java.util.Comparator;
public enum OrderComparatorClassic implements Comparator<Order> {
    ID {
        @Override
        public int compare(Order o1, Order o2) {
            return Long.compare(o1.getId(), o2.getId());
        }
    },
    AMOUNT {
        @Override
        public int compare(Order o1, Order o2) {
            return Double.compare(o1.getAmount(), o2.getAmount());
        }
    }
}
```

Обратиться к этому компаратору при сортировке можно так:

```
orders.sort(OrderComparatorClassic.AMOUNT);
```

Замыкания

Блок кода, представляющий собой лямбда-выражение вместе со значениями локальных переменных и параметров метода, в котором он объявлен, называется замыканием, или *closure*. Объект-функция создается во время исполнения, и применен может быть уже после того как объект, его создавший, прекратит существование. Такая ситуация требует, чтобы переменные, которые использует лямбда-выражение, не могли быть изменены. Значения переменных фиксируются замыканием и изменены быть не могут.

В примере лямбда-выражение использует внешний параметр метода и внешнюю локальную переменную.

```
/* # 17 # замыкание на локальных переменных # FunctionBuilder.java # */
```

```
package by.epam.learn.function.closure;
import java.util.function.Function;
public class FunctionBuilder<T> {
    public static Function<String, Integer> build(String strNum) {
        int count = 1;
        return t -> {
            int res = Integer.valueOf(strNum + t) + count;
            return res;
        };
    }
}
```

Без фигурных скобок выражение примет вид:

```
return t -> Integer.valueOf(strNum + t) + count;
```

При формировании объекта функции **Function**, как возвращаемого значения метода **build()**, значения параметров сохраняются, и функция будет использовать эти зафиксированные значения **strNum** и **count** в тот момент, когда произойдет ее вызов.

```
/* # 18 # вызов метода возвращающего функцию # ClosureMain.java # */
```

```
package by.epam.learn.function.closure;
import java.util.function.Function;
public class ClosureMain {
    public static void main(String[] args) {
        Function<String, Integer> function = FunctionBuilder.build("100");
        int res = function.apply("77");
        System.out.println(res);
    }
}
```

Если попробовать изменить значения **strNum** и **count** в методе, то это вызовет ошибку компиляции при попытке их применения в лямбда-выражении:

```
public static Function<String, Integer> build(String strNum) {
    int count = 1;
    ++count;
    strNum = strNum.concat("1");
    return t -> {
        int res = Integer.valueOf(strNum + t) + count; // compile error
        return res;
    };
}
```

Эти переменные должны иметь константные значения, как если бы они были объявлены как **final**. Отсюда и название — замыкание. Однако существует

возможность обойти это требование и обновлять счетчик с помощью элемента массива:

```
public static Function<String, Integer> build(String strNum) {
    int[] count = {1};
    ++count[0];
    return t -> Integer.valueOf(strNum + t) + ++count[0];
}
```

Такой код не является потокобезопасным.

Замыкания не запрещают использования полей класса как статических, так и нестатических.

```
public class FunctionBuilder<T> {
    static int count = 1;
    public static Function<String, Integer> build(String strNum) {
        count++;
        return t -> Integer.valueOf(strNum + t) + ++count;
    }
}
```

Ошибок компиляции не будет, даже если результат не может быть определен. Все требования по константности локальных переменных существовали и ранее для анонимных классов, но только с появлением лямбда-выражений проблема стала более актуальной.

Ссылки на методы

Более короткая запись лямбда-выражения возможна в случае, если реализации функционального интерфейса необходимо передать уже существующий метод без всяких изменений.

Если задана функция получения идентификатора заказа в виде лямбда:

```
Function<Order, Long> function = o -> o.getId();
```

то можно записать ее в виде ссылки на метод:

```
function = Order::getId;
```

Другой пример на **Consumer**:

```
Consumer<String> consumer = s -> System.out.println(s);
```

что равнозначно записи:

```
consumer = System.out::println;
```

Для статических методов возможно представление:

```
BiFunction<Double, Double, Double> biFunction = Math::hypot;
```

что эквивалентно:

```
biFunction = (x, y) -> Math.hypot(x, y);
```

Оператор видимости «::» отделяет метод от объекта или класса. Существуют три варианта записи:

- **ContainingClass::staticMethodName** — ссылка на статический метод;
- **ContainingObject::instanceMethodName** — ссылка на нестатический метод конкретного объекта;
- **ContainingType::methodName** — ссылка на нестатический метод любого объекта конкретного типа.

Первые два варианта эквивалентны лямбда-выражению с параметрами методами. В третьем варианте первый параметр становится целевым для метода, например:

```
Comparator<Long> comparator = (l1, l2) -> l1.compareTo(l2);
comparator = Long::compareTo;
```

В качестве объекта можно использовать ссылки **this** и **super**.

Кроме ссылки на метод, существуют также и ссылки на конструктор, где в качестве имени метода указывается оператор **new**.

```
Supplier<StringBuilder> supplier1 = StringBuilder::new;
```

Ссылку на конструктор можно применить в реализации фабрики по созданию треугольников:

```
/* # 19 # фабрика с применением ссылки на конструктор # TriangleFactory.java # */
```

```
package by.epam.learn.function.supplier;
import by.epam.learn.entity.RightTriangle;
import java.util.function.Function;
public class TriangleFactory {
    private Function<Double[], RightTriangle> triangle = RightTriangle::new;
    public RightTriangle getTriangle(Double... sides) {
        return triangle.apply(sides);
    }
}
```

Где в вызове **RightTriangle::new**, исходя из объявления **Function**, происходит обращение только к конструктору **RightTriangle(Double[] sides)**. Создание объектов тогда производится следующим образом:

```
/* # 20 # создание объектов фабрикой # FactoryMain.java # */
```

```
package by.epam.learn.function.supplier;
import by.epam.learn.entity.RightTriangle;
public class FactoryMain {
    public static void main(String[] args) {
        TriangleFactory factory = new TriangleFactory();
        RightTriangle triangle = factory.getTriangle(1., 2.);
    }
}
```

```

        System.out.println(triangle);
        RightTriangle triangle1 = factory.getTriangle(3., 4.);
        System.out.println(triangle1);
    }
}

```

В результате будут созданы два объекта:

RightTriangle[sideA=1.0, sideB=2.0]

RightTriangle[sideA=3.0, sideB=4.0]

Вопросы к главе 7

1. Дать определение функционального интерфейса. **default & static** методы. Область применения. Каким образом вызываются?
2. Что такое лямбда-выражение? Его структура.
3. Замыкание. К каким переменным есть доступ у лямбда-выражения?
4. Ссылка на метод. Какие существуют ссылки на методы?
5. Ссылка на конструктор. Как применяется?
6. Интерфейс **Function**. Его назначение. Какие существуют близкие по смыслу интерфейсы и почему можно сделать такой вывод?
7. Интерфейс **UnaryOperator**. Его назначение. Сравнить с интерфейсом **Function**.
8. Интерфейс **Supplier**. Его назначение. Какие существуют близкие по смыслу интерфейсы и почему можно сделать такой вывод?
9. Интерфейс **Predicate**. Его назначение. Какие существуют близкие по смыслу интерфейсы и почему можно сделать такой вывод?
10. Интерфейс **Consumer**. Его назначение. Какие существуют близкие по смыслу интерфейсы и почему можно сделать такой вывод?
11. Интерфейс **Comparator**. Его назначение.
12. Как создать собственный функциональный интерфейс?
13. Как сортировать список с применением лямбда-выражений?
14. Можно ли называть интерфейс функциональным, если при его объявлении не была применена аннотация **@FunctionalInterface**?

Задания к главе 7

Вариант А

1. С помощью каррирования реализовать функцию сложения двух чисел, функцию проверки строки на регулярное выражение, функцию разбиения строки по регулярному выражению.
2. Определить, являются ли слова анаграммами, т.е. можно ли из одного слова составить другое перестановкой букв.

3. Написать класс Пользователь с полями: id, имя, возраст, страна. Создать массив Пользователей. Отсортировать по стране и возрасту. Выбрать всех Пользователей старше заданного возраста, первая буква имени у которых начинается с заданной буквы. Получить максимальный и минимальный элемент в сгруппированном результате по возрасту.
4. Написать функциональный интерфейс с методом, который принимает число и возвращает булево значение. Написать реализацию такого интерфейса в виде лямбда-выражения, которое возвращает **true**, если переданное число делится без остатка на 31.
5. Написать функциональный интерфейс с методом, который принимает две строки и возвращает тоже строку. Написать реализацию такого интерфейса в виде лямбды, которая возвращает ту строку, которая длиннее.
6. Написать функциональный интерфейс с методом, который принимает три дробных числа: a, b, c и возвращает тоже дробное число. Написать реализацию такого интерфейса в виде лямбда-выражения, которое возвращает дискриминант.
7. Написать класс Студент с полями имя, возраст. Создать массив Студентов. Выполнить сортировку по оценке выше 8 баллов и сортировать результат по имени.
8. Вывести количество вхождений заданного слова в тексте соответственно из файла в виде [слово1-2, слово2-3, слово3-0].
9. Вывести коллекцию количества вхождений символа в тексте соответственно из файла.
10. Дано три разных целых числа. Реализовать лямбда-выражение, которое находит наибольшее из этих трех чисел.
11. С помощью лямбда-выражений создать метод, который на вход принимает строку, количество копий N, ограничение на общую длину L. Поставить запятые после каждого слова, сделать N копий, и если слов больше M — не выводить остальные слова.
12. Создать массив целых чисел. Убрать все четные элементы из массива и заполнить в конце нулями до прежнего размера массива.
13. Создать массив целых чисел. Используя лямбда-выражение, отсортировать массив по убыванию.
14. Определить, является ли число элементом Фибоначчи с помощью лямбда-выражений.
15. Создать N пар значений x, y, которые представляют координаты точки на плоскости. Выстроить все точки по увеличению их удаленности от начала координат, и вывести отсортированный список точек на экран в формате: (X:Y).
16. Написать функцию, которая вычисляет сумму списка аргументов произвольной длины с разными типами элементов массива.
17. С помощью лямбда-выражений определить, можно ли из длин сторон a, b, c образовать треугольник?

18. Продemonстрировать работу лямбда-выражения, которое получает входным параметром целое число x и вычисляет количество вхождений заданной цифры в этом числе.
19. Дан предикат **condition** и две функции **ifTrue** и **ifFalse**. Написать метод **ternaryOperator**, который из них построит новую функцию, возвращающую значение функции **ifTrue**, если предикат выполнен, и значение **ifFalse** иначе.
20. С помощью лямбда-выражений вычислить факториал заданного числа.
21. Дан прямоугольный треугольник с катетами a и b . С помощью лямбда-выражения найти радиус вписанной в треугольник окружности.
22. Дана строка. Вернуть строку, где сначала идут гласные, а потом согласные из заданной строки. Гласные и согласные должны быть в отсортированном порядке.
23. Написать программу, которая выводит число прописью.
24. Вывести массив $N \times N$, заполненный по спирали в порядке возрастания.
25. Определить, является ли строка панграммой (использует каждую букву алфавита хотя бы один раз).
26. С помощью генераторов вывести первые N простых чисел.
27. Преобразовать каждый элемент списка, цену без добавленной стоимости в цену с добавленной стоимостью.
28. Дано время в 12-часовом формате в виде строки. Конвертировать время в 24-часовой формат.
29. Дан массив чисел. Построить из этих чисел двоичное дерево поиска и найти глубину этого дерева.
30. Последовательность координат вершин многоугольника задана массивом чисел. Определить, лежит ли точка внутри многоугольника.
31. С применением лямбда-выражения перевернуть входную строку.
32. С помощью лямбда-выражений разработать метод, который на вход получает массив объектов, а возвращает его уже без дубликатов.
33. Написать предикат, выбирающий имена, которые начинаются с заданной буквы.
34. Написать программу, возвращающую значения числа Π , используя лямбда-выражения.
35. Используя фильтр, создать новый массив из строк с числом символов больше заданного.
36. В массиве строк найти все строки, начинающиеся на заданный символ и состоящие из N букв.

Тестовые задания к главе 7

Вопрос 7.1.

Дан фрагмент кода:

```
long result = Arrays.stream(new String[]{"JSE", "JDK", "J"}) // line 1
    .filter(s -> s.length() > 1)
    .filter(s -> s.contains("J"))
    .count();
```

Какое значение примет *result*? (выбрать один)

- a) 0
- b) 2
- c) 3
- d) compilation fails at line 1

Вопрос 7.2.

Дан фрагмент кода:

```
Predicate<String> predicate = s -> {
    int i = 0;
    boolean result = s.contains("JDK");
    System.out.print(i++ + " ");
    return result;
};
Arrays.stream(new String[]{"JRE", "JDK", "JVM", "JSE", "JDK"}).filter(predicate)
    .findFirst().ifPresent(System.out::print);
```

Что будет выведено в консоль?

- a) 0 JDK
- b) 0 0 JDK
- c) 0 1 JDK
- d) 0 0 0 0 0 JDK
- e) 0 1 2 3 4 JDK

Вопрос 7.3.

Дан фрагмент кода:

```
IntStream numbers = new Random().ints(10, 0, 20);
System.out.print(
    //line 1
);
```

Какой оператор должен быть вставлен вместо *line 1*, чтобы был найден максимальный целый элемент потока? (выбрать два)

- a) numbers.max(Integer::max).get()
- b) numbers.max()

- c) numbers.max(Comparator.comparing(n -> n)).get()
- d) numbers.boxed().max(Comparator.comparing(n -> n)).get()
- e) numbers.max(Comparator.comparing(n -> n))

Вопрос 7.4.

Что будет выведено в результате выполнения фрагмента кода? (выбрать один)

```
Stream.of(1).peek(((Consumer<Integer>) (i -> i += 1)).andThen(i -> i += 2))
    .forEach(System.out::print);
Stream.of(1).map(((Function<Integer, Integer>) (i -> i += 1)).andThen(i -> i += 2))
    .forEach(System.out::print);
```

- a) 14
- b) 11
- c) 12
- d) 24
- e) 22

Вопрос 7.5.

Дан фрагмент кода:

```
Stream<String> strings = Arrays.stream(
    new String[]{"Java", "Standard", "Edition", "version", "14"});
System.out.print(
    // line n1
);
```

Какой оператор должен быть вставлен вместо *line 1*, чтобы было определено число строк с длиной меньше чем 4? (выбрать один)

- a) strings.peek(x -> x.length() <= 4).count().get()
- b) strings.filter(x -> x.length() <= 4).count()
- c) strings.filter(x -> x.length() <= 4).mapToInt(Integer::valueOf).count()
- d) strings.map(x -> x.length() <= 4).count()

СТРОКИ

Строка — это застывшая структура данных, и повсюду, куда она передается, происходит значительное дублирование процесса. Это идеальное средство для сокрытия информации.

Алан Дж. Перлис

Строка в языке Java — это основной носитель текстовой информации. В отличие от языка Си это не массив символов типа **char**, а объект соответствующего класса. Системная библиотека Java содержит классы **String**, **StringBuilder** и **StringBuffer**, поддерживающие хранение строк, их обработку и определенные в пакете **java.lang**, подключаемом к приложению автоматически. Эти классы объявлены как **final**, что означает невозможность создания собственных порожденных классов со свойствами строки. Для форматирования и обработки строк применяются также классы **Formatter**, **Pattern**, **Matcher**, **StringJoiner** и другие.

Класс String

Каждая строка, создаваемая с помощью оператора **new**, литерала (заключенная в двойные апострофы) или метода класса, создающего строку, является экземпляром класса **String**. Особенностью объекта класса **String** является то, что его значение не может быть изменено после создания объекта при помощи любого метода класса. Изменение строки всегда приводит к созданию нового объекта в *heap*. Сама объектная ссылка при этом сохраняет прежнее значение и хранится в стеке. Произведенные изменения можно сохранить переинициализируя ссылку.

Класс **String** поддерживает несколько конструкторов, например: **String()**, **String(String original)**, **String(byte[] bytes)**, **String(char[] value)**, **String(char[] value, int offset, int count)**, **String(StringBuffer buffer)**, **String(StringBuilder builder)** и др. Эти конструкторы используются для создания объектов класса **String** на основе их инициализации значениями из массива типа **char**, **byte** и др.

В Java 8 класс **String** был подвержен серьезному изменению внутренней структуры. Вместо массива символов **char** теперь строка хранится в массиве типа **byte**, а ее кодировка в отдельном поле. Изменен алгоритм хеширования, что, как говорит *Oracle*, даст лучшее распределение хэш-кодов,

улучшит производительность основанных на хэшировании коллекций типа **Set** и **Map**.

Когда Java встречает литерал, заключенный в двойные кавычки, автоматически создается объект-литерал типа **String**, на который можно установить ссылку. Таким образом, объект класса **String** можно создать, присвоив ссылке на класс значение существующего литерала:

```
String str1 = "oracle.com";
String str2 = new String("oracle.com");
```

Теперь нет необходимости использовать конструктор **String(String original)** при создании новой строки на основе части другой строки, если новая строка была получена выделением подстроки, например, методом **substring()**. Ранее «обрезанная» часть сохраняла полную строку, что влекло за собой утечки памяти, порой существенные.

Некоторые методы класса **String**:

String concat(String s) или оператор «+» — слияние строк;

boolean equals(Object ob) и **equalsIgnoreCase(String s)** — сравнение строк с учетом и без учета нижнего и верхнего регистра символов соответственно;

int compareTo(String s) и **compareToIgnoreCase(String s)** — лексикографическое сравнение строк с учетом и без учета их регистра. Метод осуществляет вычитание кодов первых различных символов вызывающей и передаваемой строки в метод строк и возвращает целое значение. Метод возвращает значение **0** в случае, когда **equals()** возвращает значение **true**;

boolean contentEquals(CharSequence ob) — сравнение строки и содержимого объекта типа **StringBuffer**, **StringBuilder** и пр.;

boolean matches(String regex) — проверка строки на соответствие регулярному выражению;

String substring(int n, int m) — извлечение из строки подстроки длины **m-n**, начиная с позиции **n**. Нумерация символов в строке начинается с нуля;

String substring(int n) — извлечение из строки подстроки, начиная с позиции **n**;

int length() — определение длины строки;

int indexOf(char ch) — определение позиции символа в строке;

static String valueOf(type v) — преобразование переменной базового типа к строке;

String toUpperCase()/toLowerCase() — преобразование всех символов вызывающей строки в верхний/нижний регистр;

String replace(char c1, char c2) — замена в строке **всех** вхождений первого символа вторым символом;

String replaceAll(String regex, String replacement) — замена в строке **всех** подстрок, соответствующих регулярному выражению, новой строкой, см. также **replaceFirst()**;

String intern() — заносит строку в «пул» литералов и возвращает ее объектную ссылку;

String strip() — удаление всех пробелов в начале и конце строки, более совершенный аналог метода **trim()**, см. также методы **stripLeading()** и **stripTrailing()**;

char charAt(int position) — возвращение символа из указанной позиции (нумерация с нуля);

boolean isEmpty() — возвращает **true**, если длина строки равна 0;

boolean isBlank() — возвращает **true**, если строка пуста или содержит только пробельные символы;

static String join(CharSequence delimiter, CharSequence... elements) — объединение произвольного набора строк (коллекции строк) в одну строку с заданной строкой-разделителем;

char[] getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) — извлечение символов строки в массив символов;

static String format(String format, Object... args), format(Locale l, String format, Object... args) — создание форматированной строки, полученной с использованием формата, локализации и др.;

String[] split(String regex), String[] split(String regex, int limit) — поиск вхождения в строку заданного регулярного выражения-шаблона в качестве разделителя и деление исходной строки в соответствии с этим разделителем на массив строк;

InputStream codePoints() — извлечение символов строки в поток (stream) их кодов;

InputStream chars() — преобразование строки в stream ее символов;

Stream<String> lines() — извлечение строк, разделенных символом перехода на другую строку, в поток (stream) строк.

Во всех случаях вызова методов, изменяющих строку, создается новый объект типа **String**.

Пусть создана строка с пробелами в первых позициях:

```
String str1 = "    " + null + "    4" + 8 + 11;
System.out.println(str1);
str1 = str1.strip(); // new object
System.out.println(str1);
```

Метод **strip()** удалит все пробелы в начале и конце строки, если таковые имеются:

```
null    4811
null    4811
```

Создание новой строки из нескольких:

```
String str3 = String.join("-", "java", "16", "SE");
System.out.println(str3);
```

Будет выведено:

java-16-SE

Демонстрацией работы методов класса служит преобразование строки в массив объектов типа **String** и их сортировка в алфавитном порядке. Ниже рассмотрена сортировка массива строк методом выбора:

```
// # 1 # разбиение и сортировка #

String names = " angela Alena Agnes anette albina Anastasya ALLA ArinA ";
names = names.strip();
String namesArr[] = names.split(" ");
System.out.println(Arrays.toString(namesArr));
for (int j = 0; j < namesArr.length - 1; j++) {
    for (int i = j + 1; i < namesArr.length; i++) {
        if (namesArr[i].compareToIgnoreCase(namesArr[j]) < 0) {
            String temp = namesArr[j];
            namesArr[j] = namesArr[i];
            namesArr[i] = temp;
        }
    }
}
for (String arg : namesArr) {
    if (!arg.isEmpty()) {
        System.out.println(arg);
    }
}
```

Вызов метода **strip()** обеспечивает удаление всех начальных и конечных пробельных символов. В данном случае отсутствие вызова этого метода никак не повлияет на результат выполнения программы. Метод **compareToIgnoreCase()** выполняет лексикографическое сравнение строк между собой по правилам Unicode с игнорированием регистра. Оператор **if(!arg.isEmpty())** не позволяет выводить пустые строки. Метод **isEmpty()** был введен в класс для замены не-экономной проверки на пустую строку оператором **if(str.length() == 0)**.

Возможности интерфейса **Stream** позволяют выполнить все вышеуказанные действия по сортировке строк из примера, приведенного выше, в рамках одного оператора.

```
Arrays.stream(namesArr)
    .filter(s -> !s.isEmpty())
    .sorted(String::compareToIgnoreCase)
    .forEach(System.out::println);
```

Есть несколько возможностей удалить все пробелы в строке, как по краям, так и внутри. При использовании метода **codePoints()** строка преобразуется в **IntStream**, затем из нее убираются все пробелы, и результат выводится в поток вывода:

```
String str = " Java 16 ";
str.codePoints()
    .filter(o -> o != ' ')
    .forEach(o -> System.out.print((char)o));
```

Аналогично при применении объекта **StringBuilder**

```
StringBuilder sb = new StringBuilder(" Java 16 ");
sb.codePoints()
    .filter(o -> o != ' ')
    .forEach(o -> sb.append((char)o));
```

Еще проще эта задача решается методом **replace()**.

При использовании методов класса **String**, изменяющих строку, создается новый объект класса **String**. Сохранить произведенные изменения экземпляра класса **String** можно только с применением оператора присваивания, т.е. установив ссылку на вновь созданный объект. В следующем примере будет подтвержден тезис о неизменяемости экземпляра типа **String**.

```
/* # 2 # передача строки по ссылке # RefString.java */
```

```
package by.epam.learn.strings.change;
public class RefString {
    public static void changeString(String s) {
        s.concat(" Certified");
        s = s.concat(" Certified");
        s += " Certified";
    }
    public static void main(String[] args) {
        String str = new String("Java");
        changeString(str);
        System.out.print(str);
    }
}
```

В результате будет выведена строка:

Java

Поскольку объект был передан по ссылке, любое изменение объекта в методе должно сохраняться и для исходного объекта, так как обе ссылки равноправны. Этого не происходит по той причине, что вызов метода **concat(String s)** приводит к созданию нового объекта.

При создании экземпляра класса **String** путем присваивания его ссылки на литерал, последний помещается в «пул литералов» типа **String**, который теперь перенесен из стека в *heap*. Если в дальнейшем будет создана еще одна ссылка на литерал, эквивалентный ранее объявленному, то будет произведена попытка добавления его в «пул литералов». Так как идентичный литерал там уже существует, то дубликат не может быть размещен, и вторая ссылка будет

ссылаться на существующий литерал. В случае, если литерал является вычисляемым, то компилятор воспринимает литералы **"Java"** и **"Ja" + "va"** как эквивалентные.

```
// # 3 # сравнение ссылок и объектов #
```

```
String s1 = "Java16";
String s2 = "Ja" + "va" + 16;
String s3 = new String("Java16");
String s4 = new String(s1);
System.out.println(s1 + "==" + s2 + " : " + (s1 == s2)); // true
System.out.println(s3 + "==" + s4 + " : " + (s3 == s4)); // false
System.out.println(s1 + "==" + s3 + " : " + (s1 == s3)); // false
System.out.println(s1 + "==" + s4 + " : " + (s1 == s4)); // false
System.out.println(s1 + " equals " + s2 + " : " + s1.equals(s2)); // true
System.out.println(s1 + " equals " + s3 + " : " + s1.equals(s3)); // true
```

В результате будет выведено:

```
Java16==Java16 : true
Java16==Java16 : false
Java16==Java16 : false
Java16==Java16 : false
Java16 equals Java16 : true
Java16 equals Java16 : true
```

Несмотря на то, что одинаковые по значению строковые объекты расположены в различных участках памяти, значения их хэш-кодов совпадают.

Так как в Java все ссылки хранятся в стеке, а объекты — в *heap*, то при создании объектов **s1**, **s2** сначала создается ссылка, а затем этой ссылке устанавливается в соответствие объект. В данной ситуации **s2** ассоциируется с уже существующим литералом, так как объект **s1** уже сделал ссылку на этот литерал. При создании **s3** происходит вызов конструктора, т.е. выделение памяти происходит раньше инициализации, и в этом случае в куче создается новый объект.

Существует возможность сэкономить память и переприсвоить ссылку с объекта на литерал при помощи вызова метода **intern()**.

```
// # 4 # занесение в пул литералов #
```

```
String s1 = "Java";
String s2 = new String("Java");
System.out.println(s1 == s2); // false
s2 = s2.intern();
System.out.println(s1 == s2); // true
```

В данной ситуации ссылка **s1** инициализируется литералом, обладающим всеми свойствами объекта вплоть до вызова методов. Вызов метода **intern()** организует поиск в «пуле литералов» соответствующего значению объекта **s2** литерала

и при положительном результате возвращает ссылку на найденный литерал, а при отрицательном — заносит значение в пул и возвращает ссылку на него.

В Java 8 был добавлен класс **StringJoiner**, основным назначением которого является объединение нескольких строк в одну с заданием разделителя, префикса и суффикса. У класса два конструктора **StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix)** и **StringJoiner(CharSequence delimiter)**.

```
StringJoiner joiner = new StringJoiner(":", "<<", ">>");
String result = joiner.add("blanc").add("rouge").add("blanc").toString();
System.out.println(result);
```

Результатом будет:

<<blanc:rouge:blanc>>

Текстовые блоки упрощают объявление многострочных литералов, содержащих символы, которые требуют экранирования в случае использования внутри строки, двойные кавычки, перенос строки и некоторые другие. Например, если необходимо в виде строки сохранить XML-документ, то для удобочитаемости использовалась бы конкатенация строк, и строка выглядела бы так:

```
String xml = "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"no\"?>\n" +
    "<book>\n" +
    "  <name>Java from EPAM</name>\n" +
    "  <author id=\"777\">Blinov</author>\n" +
    "</book>\n";
```

Три двойные кавычки подряд в начале и конце строки гарантируют ее идентичное сохранение:

```
String xmlBlock = ""
    "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"no\"?>
    <book>
      <name>Java from EPAM</name>
      <author id=\"777\">Blinov</author>
    </book>
    "";
```

StringBuilder и StringBuffer

Классы **StringBuilder** и **StringBuffer** являются «близнецами» и по своему назначению близки к классу **String**, но в отличие от последнего содержимое и размеры объектов классов **StringBuilder** и **StringBuffer** можно динамически изменять в приложении. Основным отличием **StringBuilder** от **StringBuffer** является потокобезопасность последнего. Более высокая скорость обработки есть следствие отсутствия потокобезопасности класса **StringBuilder**. Его следует применять, если не существует вероятности использования объекта в конкурирующих потоках.

С помощью соответствующих методов и конструкторов объекты классов **StringBuffer**, **StringBuilder** и **String** можно преобразовывать друг в друга. Конструктор класса **StringBuffer** (как и **StringBuilder**) может принимать в качестве параметра объект **String** или неотрицательный размер буфера. Объекты этого класса можно преобразовать в объект класса **String** методом **toString()** или с помощью конструктора класса **String**.

Следует обратить внимание на некоторые методы:

void setLength(int newLength) — установка размера буфера;

void ensureCapacity(int minimumCapacity) — установка гарантированно-го минимального размера буфера;

void trimToSize() — сжатие буфера до размеров контента;

int capacity() — возвращение текущего размера буфера;

StringBuffer append(parameters) — добавление к содержимому объекта строкового представления аргумента, который может быть символом, значением базового типа, массивом и строкой;

StringBuffer insert(parameters) — вставка символа, объекта или строки в указанную позицию;

StringBuffer deleteCharAt(int index) — удаление символа; **StringBuffer delete(int start, int end)** — удаление подстроки; **StringBuffer reverse()** — обращение содержимого объекта.

В классе присутствуют также методы, аналогичные методам класса **String**, такие как **replace()**, **substring()**, **charAt()**, **length()**, **getChars()**, **indexOf()** и др.

```
/* # 5 # свойства объекта StringBuffer */
```

```
StringBuilder builder = new StringBuilder();
System.out.println("length = " + builder.length());
System.out.println("capacity = " + builder.capacity());
// builder = "Java"; // error, only for String
builder.append("Java");
System.out.println("content = " + builder);
System.out.println("length = " + builder.length());
System.out.println("capacity = " + builder.capacity());
builder.append("Internationalization");
System.out.println("content = " + builder);
System.out.println("length = " + builder.length());
System.out.println("capacity = " + builder.capacity());
System.out.println("reverse = " + builder.reverse());
```

Результатом выполнения данного кода будет:

```
length = 0
capacity = 16
content = Java
length = 4
capacity = 16
```

```
content = JavaInternationalization
length = 24
capacity = 34
reverse = noitazilanoitanretnJavaJ
```

При создании объекта **StringBuffer** конструктор по умолчанию автоматически резервирует некоторый буфер — объем памяти (16 символов), что в дальнейшем позволяет быстро менять содержимое объекта, оставаясь в границах участка памяти, выделенного под объект. Размер резервируемой памяти при необходимости можно указывать в конструкторе. Если длина строки **StringBuffer** после изменения превышает его размер, то емкость объекта автоматически увеличивается, оставляя при этом некоторый резерв для дальнейших изменений. Методом **reverse()** можно быстро изменить порядок символов в объекте.

Если метод, вызываемый объектом **StringBuilder**, производит изменения в его содержимом, то это не приводит к созданию нового объекта, как в случае объекта **String**, а изменяет текущий объект **StringBuilder**.

```
/* # 6 # изменение объекта StringBuilder # RefStringBuilder.java */
package by.epam.learn.strings.change;
public class RefStringBuilder {
    public static void changeStringBuilder(StringBuilder builder) {
        builder.append(" Certified");
    }
    public static void main(String[] args) {
        StringBuilder str = new StringBuilder("Oracle");
        changeStringBuilder(str);
        System.out.println(str);
    }
}
```

В результате выполнения этого кода будет выведена строка:

Oracle Certified

Объект **StringBuilder** передан в метод **changeStringBuilder()** по ссылке, поэтому все изменения объекта сохраняются и для вызывающего метода.

Для классов **StringBuffer** и **StringBuilder** не переопределены методы **equals()** и **hashCode()**, т.е. сравнить содержимое двух объектов невозможно, следовательно, хэш-коды всех объектов этого типа вычисляются так же, как и для класса **Object**. При идентичном содержимом у двух экземпляров, размеры буфера каждого могут отличаться, поэтому сравнение на эквивалентность объектов представляется неоднозначным.

```
/* # 7 # сравнение объектов StringBuffer и их хэш-кодов # */
StringBuffer sb1 = new StringBuffer();
StringBuffer sb2 = new StringBuffer();
```

```

sb1.append("Java");
sb2.append("Java");
System.out.print(sb1.equals(sb2));
System.out.print(sb1.hashCode() == sb2.hashCode());

```

Результатом выполнения данного кода будет дважды выведенное значение **false**.

Сравнить же содержимое можно следующим образом:

```

sb1.toString().contentEquals(sb2);

```

Регулярные выражения

Регулярные выражения или шаблоны используются для поиска подстроки или строки, соответствующей шаблону в строке, тексте или другом объекте, представляющем последовательность символов. Класс **java.util.regex.Pattern** применяется для создания этого объекта-шаблона. Для определения шаблона применяются специальные синтаксические конструкции. О каждом соответствии можно получить информацию с помощью класса **java.util.regex.Matcher**.

Далее приведены некоторые логические конструкции для задания шаблона. Эти же конструкции применяются не только в Java, но и в других распространенных языках.

При создании регулярного выражения могут использоваться логические операции:

ab	после a следует b
a b	a или b

Если необходимо, чтобы в строке, проверяемой на соответствие, в какой-либо позиции находился один из символов некоторого символического набора, то такой набор (класс символов) можно объявить, используя одну из следующих конструкций:

[abc]	a или b или c
[^abc]	символ, исключая a , b и c
[a-z]	символ между a и z

Кроме стандартных классов символов существуют predefined классы символов:

.	любой символ
\d или \p{Digit}	[0-9]
\D	[^0-9]
\s или \p{Space}	[\t\n\r\b\f]
\S	[^\s]
\w	[0-9_A-Za-z]
\W	[^\w]

<code>\p{Lower}</code>	<code>[a-z]</code>
<code>\p{Upper}</code>	<code>[A-Z]</code>
<code>\p{Alpha}</code>	<code>[a-zA-Z]</code>
<code>\p{Alnum}</code>	<code>[\p{Alpha}\p{Digit}]</code>
<code>\p{Punct}</code>	<code>!"#\$%&'()*+,-./:;<=>?@[\\^_`{}~</code>
<code>\p{Blank}</code>	пробел или табуляция
<code>^</code> или <code>\A</code>	начало строки
<code>\$</code> или <code>\Z</code>	конец строки

Круглые скобки, кроме их логического назначения, также используются для определения групп.

Для определения регулярных выражений недостаточно одних классов символов, т.к. в шаблоне часто нужно указать количество повторений. Для этого существуют квантификаторы.

<code>a?</code>	<code>a</code> один раз или ни разу
<code>a*</code>	<code>a</code> ноль или более раз
<code>a+</code>	<code>a</code> один или более раз
<code>a{n}</code>	<code>a</code> <code>n</code> раз
<code>a{n,}</code>	<code>a</code> <code>n</code> или более раз
<code>a{n,m}</code>	<code>a</code> от <code>n</code> до <code>m</code>

Примером использования квантификатора может служить простейший способ защиты от XSS атаки:

```
String xssString = "<script>alert('hello')</script>";
xssString = xssString.replaceAll("</?script>", "");
```

Выражение выше удаляет теги начала-конца JavaScript-выражения, которое может быть вредоносным на веб-странице.

Существуют еще два типа квантификаторов, которые образованы прибавлением суффикса «?» (слабое или неполное совпадение) или «+» («жадное» или собственное совпадение) к вышеперечисленным квантификаторам. Неполное совпадение соответствует выбору с наименее возможным количеством символов, а собственное — с максимально возможным.

Класс **Pattern** используется для простой обработки строк и объекта хранилища регулярного выражения. Объект класса **Pattern**, в свою очередь, используется для более сложной обработки символьной информации классом **Matcher**, рассмотренном несколько позже.

Некоторые методы класса **Pattern**:

static Pattern compile(String regex) — возвращает **Pattern**, который соответствует **regex**;

static boolean matches(String regex, CharSequence input) — проверяет на соответствие строки **input** шаблону **regex**;

String[] split(CharSequence input) — разбивает на массив строку **input**, учитывая, что разделителем является шаблон;

Stream<String> splitAsStream(CharSequence input) — разбивает в *stream* строку **input**, учитывая, что разделителем является шаблон;

Predicate<String> asPredicate() — возвращает предикат на основе регулярного выражения;

Matcher matcher(CharSequence input) — возвращает **Matcher**, с помощью которого можно находить соответствия в строке **input**.

Методы класса **Pattern** позволяют проверять на соответствие шаблону целую строку и разбивать строку на части, используя шаблон как разделитель. Для определения подстрок, соответствующих шаблону, необходимо использовать класс **Matcher**.

Начальное состояние объекта типа **Matcher** не определено. Должен быть вызван первоначально один из методов поиска. Попытка вызвать какой-либо метод класса для извлечения информации о найденном соответствии приведет к возникновению ошибки **IllegalStateException**. Чтобы начать работу с объектом **Matcher**, необходим вызов одного из следующих методов:

boolean lookingAt() — поиск последовательности символов, начинающейся с начала строки и соответствующей шаблону;

boolean find() или **boolean find(int start)** — определение последовательности символов, соответствующих шаблону, в любом месте строки. Параметр **start** указывает на начальную позицию поиска.

После вызова одного из этих методов можно воспользоваться и методами о состоянии объекта.

Сбросить состояние экземпляра **Matcher** в исходное, для этого применяется метод **reset()** или **reset(CharSequence input)**, который также заменяет на новую последовательность символов для обработки.

Метод **boolean matches()** проверяет, соответствует ли вся информация шаблону. Для замены всех подпоследовательностей символов, удовлетворяющих шаблону, на заданную строку можно применить метод **replaceAll(String replacement)**.

Следующие примеры показывают использование возможностей классов **Pattern** и **Matcher** для поиска, разбора и разбивки строк.

Проверка на соответствие строки шаблону:

```
Pattern pattern = Pattern.compile("x+y");
Matcher matcher = pattern.matcher("xxy");
boolean res = matcher.matches();
System.out.println(res); // printing true
```

Поиск и выбор подстроки, заданной шаблоном:

```
String regex = "\\w{6,}@\\w+\\.\\p{Lower}{2,4}";
String input = "адреса эл.почты:blinov@gmail.com, romanchik@bsu.by!";
Pattern pattern = Pattern.compile(regex);
```

```

Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    System.out.println("e-mail: " + matcher.group());
}

```

В результате будет выведено:

e-mail: blinov@gmail.com
e-mail: romanchik@bsu.by

Разбиение строки на подстроки с применением шаблона в качестве разделителя:

```

String input = "java16;-lambda..[java9var";
Pattern pattern = Pattern.compile("\\d+|\\p{Punct}+");
String[] words = pattern.split(input);
System.out.println(Arrays.toString(words));

```

Результат:

[java, , lambda, java, var]

Создание **stream** с использованием шаблона в качестве разделителя, с последующим удалением из него подпоследовательностей, определяемых предикатом, полученном на основе другого шаблона:

```

Stream<String> stream = pattern.splitAsStream(input);
Pattern pattern1 = Pattern.compile("[^java]");
stream.filter(pattern1.asPredicate()).forEach(System.out::println);

```

Результат:

lambda
var

Еще один пример на обработку строк методами класса **Pattern**. Массив преобразуется в *stream*, затем разбивается на строки, содержащие только числа. Далее *stream* фильтруется, удаляя все элементы, которые не содержат цифр от 2 до 9.

```

/* # 8 # извлечение чисел, содержащих цифры 2-9, из строки
# PatternStreamMain.java */

```

```

package by.epam.learn.string;
import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;
import java.util.stream.Collectors;
public class PatternStreamMain {
    public static void main(String[] args) {
        String[] arrayStr = {"12.9", "44", "the", "7,1", "27..2", "211"};
        List<Integer> list = Arrays.stream(arrayStr)
            .flatMap(Pattern.compile("\\D+").::splitAsStream)

```

```

        .peek((s -> System.out.printf(" %s ", s)))
        .filter(Pattern.compile("[2-9]+").asPredicate())
        .map(Integer::valueOf)
        .collect(Collectors.toList());
    System.out.println("\n" + list);
}
}

```

В результате будет выведено:

```

12 9 44 7 1 27 2 211
[12, 9, 44, 7, 27, 2, 211]

```

В регулярном выражении для более удобной обработки входной последовательности применяются группы, которые помогают выделить части найденной подпоследовательности. В шаблоне они обозначаются скобками «(» и «)». Группы — способ обработки набора символов как одного.

Каждая открывающая скобка слева направо нумерует группу. Выражение ((A)(B(C))) определяет четыре группы:

- ((A)(B(C)))
- (A)
- (B(C))
- (C)

Номера групп начинаются с единицы. Нулевая группа совпадает со всей найденной подпоследовательностью. Далее приведены методы для извлечения информации о группах.

String group() — возвращает всю подпоследовательность, удовлетворяющую шаблону или нулевой группе;

String group(int group) — возвращает конкретную группу по позиции;

int groupCount() — определяет число групп сбора, представленных в сопоставляемом шаблоне. Всегда существует группа 0, представляющая все выражение и не включаемая в счетчик групп;

int start() — возвращает индекс первого символа подпоследовательности, удовлетворяющей шаблону;

int start(int group) — возвращает индекс первого символа указанной группы;

int end() — возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону;

int end(int group) — возвращает индекс последнего символа указанной группы;

boolean hitEnd() — возвращает истину, если был достигнут конец входной последовательности.

Использование групп, а также собственных и неполных квантификаторов.

```

/* # 9 # группы и квантификаторы # GroupMain.java */

```

```

package by.epam.learn.string;
import java.util.regex.Matcher;

```

```

import java.util.regex.Pattern;
public class GroupMain {
    public static void main(String[] args) {
        String base = "java";
        groupView(base, "([a-z]*)([a-z]+)");
        groupView(base, "([a-z]?)([a-z]+)");
        groupView(base, "([a-z]+)([a-z]*)");
        groupView(base, "([a-z]?)([a-z]?");
    }
    private static void groupView(String base, String regExp) {
        Pattern pattern = Pattern.compile(regExp);
        Matcher matcher = pattern.matcher(base);
        if (matcher.matches()) {
            System.out.println("group 1: " + matcher.group(1));
            System.out.println("group 2: " + matcher.group(2));
            System.out.println("main group: " + matcher.group() + " [end]"); //eq. //group(0)
        } else {
            System.out.println("nothing matches");
        }
    }
}

```

Результат работы программы:

```

group 1: jav
group 2: a
main group: java [end]
group 1: j
group 2: ava
main group: java [end]
group 1: java
group 2:
main group: java [end]
nothing matches

```

В первом случае к первой группе относятся все возможные символы, но при этом остается минимальное количество символов для второй группы.

Во втором случае для первой группы выбирается наименьшее количество символов, т.к. используется слабое совпадение.

В третьем случае первой группе будет соответствовать вся строка, а для второй не остается ни одного символа, так как вторая группа использует слабое совпадение.

В четвертом случае строка не соответствует регулярному выражению, т.к. для двух групп выбирается наименьшее количество символов.

Группа 0 всегда одинакова, так как представляет все выражение.

Интернационализация приложения

Класс **java.util.Locale** позволяет учесть особенности региональных представлений алфавита, символов, чисел, дат и проч. Автоматически виртуальная машина использует текущие региональные установки операционной системы, но при необходимости их можно изменять. Для некоторых стран региональные параметры устанавливаются с помощью констант, например: **Locale.US**, **Locale.FRANCE**. Для всех остальных объект **Locale** нужно создавать с помощью конструкторов, например:

```
Locale bel1 = new Locale("be"); // Language
Locale bel2 = new Locale("be", "BY"); // Language & country
```

Определить текущий вариант региональных параметров можно следующим образом:

```
Locale current = Locale.getDefault();
```

Локаль можно установить для текущего экземпляра (*instance*) JVM:

```
Locale.setDefault(Locale.CANADA);
```

Если, например, в ОС установлен регион «Беларусь» или в приложении с помощью **new Locale("be", "BY")**, то следующий код (при выводе результатов выполнения на консоль)

```
current.getCountry(); // region code
current.getDisplayCountry(); // region name
current.getLanguage(); // region language code
current.getDisplayLanguage(); // region language name
```

позволяет получить информацию о регионе и языке в виде:

BY

Беларусь

be

белорусский

Для создания приложений, поддерживающих несколько языков, существует целый ряд решений. Самое логичное из них — дублирование сообщений на разных языках в разных файлах с эквивалентными ключами с последующим извлечением информации на основе значения заданной локали. Данное решение основано на взаимодействии классов **java.util.ResourceBundle** и **Locale**. Класс **ResourceBundle** предназначен для взаимодействия с текстовыми файлами свойств (расширение **properties**). Каждый объект **ResourceBundle** представляет собой набор соответствующих подтипов, которые разделяют одно и то же базовое имя, к которому можно получить доступ через поле **parent**. Следующий список показывает возможный набор соответствующих ресурсов с базовым именем **text**. Символы, следующие за базовым именем, показывают код языка, код страны и тип операционной системы. Например, файл **text_it_ch.properties**

соответствует объекту **Locale**, заданному кодом итальянского языка (**it**) и кодом страны Швейцарии (**ch**).

```
text.properties
text_fr_CA.properties
text_it_CH.properties
text_be_BY.properties
```

Чтобы выбрать определенный объект **ResourceBundle**, необходимо вызвать один из статических перегруженных методов **getBundle(parameters)**. Следующий фрагмент выбирает **text** объекта **ResourceBundle** для объекта **Locale**, который соответствует французскому языку и стране Канада.

```
Locale locale = new Locale("fr", "CA");
ResourceBundle rb = ResourceBundle.getBundle("text", locale);
```

Если объект **ResourceBundle** для заданного объекта **Locale** не существует, то метод **getBundle()** извлечет наиболее общий. Если общее определение файла ресурсов не задано, то метод **getBundle()** генерирует исключительную ситуацию **MissingResourceException**. Чтобы этого не произошло, необходимо обеспечить наличие базового файла ресурсов без суффиксов, а именно: **text.properties** в дополнение к частным случаям.

В файлах свойств информация должна быть организована по принципу:

```
#comments
group1.key10 = value1
group1.key11 = value2
group2.key20 = value3
```

В классе **ResourceBundle** определен ряд методов, в том числе метод **getKeys()**, возвращающий объект **Enumeration**, который применяется для последовательного обращения к элементам. Множество **Set<String>** всех ключей — методом **keySet()**. Конкретное значение по конкретному ключу извлекается методом **String getString(String key)**. Отсутствие запрашиваемого ключа приводит к генерации исключения. Проверить наличие ключа в файле можно методом **boolean containsKey(String key)**. Методы **String getObject(String key)** и **String[] getStringArray(String key)** извлекают объект и массив строк по передаваемому ключу. Файлы **properties** могут содержать пары с одинаковыми ключами.

В следующем примере в зависимости от выбора пользователя известная фраза будет выведена на одном из трех языков.

```
// # 10 # поддержка различных языков # HamletMain.java
```

```
package by.epam.learn.resource;
import java.io.IOException;
import java.util.Locale;
import java.util.ResourceBundle;
```

```
public class HamletMain {
    public static void main(String[] args) {
        System.out.println("1 - eng\n 2 - bel\n any - default");
        char i = 0;
        try {
            i = (char)System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
        String country = "";
        String language = "";
        switch (i) {
            case '1':
                country = "US";
                language = "en";
                break;
            case '2':
                country = "BY";
                language = "be";
        }
        Locale current = new Locale(language, country);
        ResourceBundle rb = ResourceBundle.getBundle("property.text", current);
        String s1 = rb.getString("str1");
        System.out.println(s1);
        String s2 = rb.getString("str2");
        System.out.println(s2);
    }
}
```

Все файлы следует разместить в каталоге **property** в корне проекта на одном уровне с пакетами приложения.

Файл **text_en_US.properties** содержит следующую информацию:

str1 = To be or not to be?

str2 = This is a question.

Файл **text_be_BY.properties**:

str1 = Быць або не быць?

str2 = Вось у чым пытанне.

Файл **text.properties**:

str1 = Быть или не быть?

str2 = Вот в чем вопрос.

Если в результате выполнения приложения на консоль результаты выдаются в нечитаемом виде, то следует изменить кодировку файла или символов.

Для взаимодействия с **properties**-файлами можно создать специальный класс, экземпляр которого позволит не только извлекать информацию по

ключу, но и изменять значение локали, что сделает его удобным для использования при интернационализации приложений.

```
// # 11 # менеджер ресурсов # MessageManager.java

package by.epam.learn.manager;
import java.util.Locale;
import java.util.ResourceBundle;
public enum MessageManager {
    EN(ResourceBundle.getBundle("property.text", new Locale("en", "EN"))),
    BY(ResourceBundle.getBundle("property.text", new Locale("be", "BY")));
    private ResourceBundle bundle;
    MessageManager(ResourceBundle bundle) {
        this.bundle = bundle;
    }
    public String getString(String key) {
        return bundle.getString(key);
    }
}
```

Экземпляров такого класса создается только два, и все приложение пользуется их возможностями.

Качественно разработанное приложение обычно не содержит литералов типа **String**. Все необходимые сообщения хранятся вне системы, в частности, в **properties**-файлах. Это позволяет без перекомпиляции кода безболезненно изменять любое сообщение или информацию, хранящуюся вне классов системы.

Интернационализация чисел

Стандарты представления дат и чисел в различных странах существенно отличаются. Например, в Германии строка «**1.234,567**» воспринимается как «одна тысяча двести тридцать четыре целых пятьсот шестьдесят семь тысячных», для белорусов и французов данная строка просто непонятна и не может представлять число.

Чтобы сделать такую информацию конвертируемой в различные региональные стандарты, применяются возможности класса **java.text.NumberFormat**. Первым делом следует задать или получить текущий объект **Locale**, определяющий стандарты конкретного региона, и создать с его помощью объект форматирования **NumberFormat**:

```
NumberFormat format = NumberFormat.getInstance(new Locale("be", "by"));
```

с конкретными региональными установками или заданными по умолчанию для приложения:

```
NumberFormat.getInstance();
```

Далее для преобразования строки в число и обратно используются методы **Number parse(String source)** и **String format(double number)** соответственно.

В предлагаемом примере производится преобразование строки, содержащей число, в три различных региональных стандарта, а затем одно из чисел преобразуется из одного стандарта в два других.

```
// # 12 # региональные представления чисел # NumberFormatMain.java
```

```
package by.epam.learn.string;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;
public class NumberFormatMain {
    public static void main(String[] args) {
        NumberFormat numberFormat = NumberFormat.getInstance(Locale.GERMAN);
        String source = "1.234,567";
        NumberFormat numberFormatUs = NumberFormat.getInstance(Locale.US);
        NumberFormat numberFormatFr = NumberFormat.getInstance(Locale.FRANCE);
        try {
            double number = numberFormat.parse(source).doubleValue();
            System.out.println(number);
            String resultUs = numberFormatUs.format(number);
            System.out.println("US(number): " + resultUs);
            String resultFr = numberFormatFr.format(number);
            System.out.println("FR(number): " + resultFr);
            double numberUs = numberFormatUs.parse(source).doubleValue();
            System.out.println("DE -> US(parse number): " + numberUs);
            double numberFr = numberFormatFr.parse(source).doubleValue();
            System.out.println("DE -> FR(parse number): " + numberFr);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

Результат работы программы:

```
1234.567
US(number): 1,234.567
FR(number): 1 234,567
DE -> US(parse number): 1.234
DE -> FR(parse number): 1.0
```

Аналогично выполняются переходы от одного регионального стандарта к другому при отображении денежных сумм с добавлением символа валюты.

Интернационализация дат

Учитывая исторически сложившиеся способы отображения даты и времени в различных странах и регионах мира, в языке создан механизм поддержки всех национальных особенностей. Эту задачу решает класс **java.text.DateFormat**.

С его помощью учтены: представления месяцев и дней недели на национальном языке; специфические последовательности в записи даты и часовых поясов; использования различных календарей.

Процесс получения объекта, отвечающего за обработку регионального стандарта даты, похож на создание объекта, отвечающего за национальные представления чисел, а именно:

```
DateFormat format = DateFormat.getDateInstance(DateFormat.MEDIUM, new Locale("be", "by"));
```

или по умолчанию:

```
DateFormat.getDateInstance();
```

Константа **DateFormat.MEDIUM** указывает на то, что будут представлены только дата и время без указания часового пояса. Для указания часового пояса используются константы класса **DateFormat** со значением **LONG** и **FULL**. Константа **SHORT** применяется для сокращенной записи даты, где месяц представлен в виде своего порядкового номера.

Для получения даты в виде строки для заданного региона используется метод **String format(Date date)** в виде:

```
String stringData = format.format(new Date());
```

Метод **Date parse(String source)** преобразовывает переданную в виде строки дату в объектное представление конкретного регионального формата, например:

```
String stringData = "April 9, 2012"; Date date = format.parse(stringData);
```

Класс **DateFormat** содержит большое количество методов, позволяющих выполнять разнообразные манипуляции с датой и временем.

В качестве примера рассмотрено преобразование заданной даты в различные региональные форматы.

```
// # 13 # региональные представления дат # DateFormatMain.java
```

```
package by.epam.learn.string;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.Date;
import java.util.Locale;
public class DateFormatMain {
    public static void main(String[] args) {
        DateFormat dateFormat = DateFormat.getDateInstance(
            DateFormat.FULL, DateFormat.MEDIUM, new Locale("be", "BY"));
        String dateStr = dateFormat.format(new Date());
        System.out.println(dateStr);
        try {
            Date date = dateFormat.parse("аўторак, 17 сакавіка 2020 г., 17:58:17");
            System.out.println(date);
            DateFormat dateFormatCa = DateFormat.getDateInstance(
                DateFormat.SHORT, DateFormat.SHORT, Locale.CANADA);
```

```

        System.out.println(dateFormatCa.format(date));
        DateFormat dateFormatFr = DateFormat.getDateTimeInstance(
            DateFormat.SHORT, DateFormat.SHORT, Locale.FRANCE);
        System.out.println(dateFormatFr.format(date));
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
}
}

```

Результат выполнения:

```

аўторак, 17 сакавіка 2020 г., 17:58:54
Tue Mar 17 17:58:17 MSK 2020
2020-03-17, 5:58 p.m.
17/03/2020 17:58

```

Чтобы получить представление текущей даты во всех возможных региональных стандартах, можно воспользоваться следующим фрагментом кода:

```

Date date = new Date();
Locale[] locales = DateFormat.getAvailableLocales();
for (Locale loc : locales) {
    DateFormat format = DateFormat.getDateInstance(DateFormat.FULL, loc);
    System.out.println(loc.toString() + "---> " + format.format(date));
}

```

В результате будет выведена пара сотен строк, каждая из которых представляет текущую дату в соответствии с региональным стандартом, выводимым перед датой с помощью оператора `loc.toString()`.

API Date\Time

В Java 8 добавлен более совершенный API для работы с датой и временем. Существующие возможности языка по работе с датой\временем, предоставляемые классами `java.util.Date`, `java.util.GregorianCalendar` и другими, выглядят несколько тяжеловесно и не обладают потокобезопасностью. Новые классы находятся в пакете `java.time` и его подпакетах. Большинство классов потокобезопасны и представляют *immutable* объекты.

Основной класс API Date\Time — класс `java.time.Instant`. Представляет время в наносекундах от 1 января 1970 года (UTC), а при отрицательных значениях и любое время до этой даты. В нем объявлено большое число методов. Класс работает с машинным представлением времени. Для манипуляции конкретными датами и временем используются частные вспомогательные классы. Для работы с датами необходим класс `java.time.LocalDate`, для работы со временем — `java.time.LocalTime`, для работы с датой и временем — `java.time.LocalDateTime`.

Класс `Instant` способен выполнять различные манипуляции:

метод `Instant now()`, получение текущей даты\времени,

`Instant parse(CharSequence text)` — парсинг строки в дату\время,

ofEpochMilli(long epochMilli) — преобразование времени в миллисекундах в дату\время:

```
// # 14 # форматы представления дат # InstantMain.java
```

```
package by.epam.learn.time;
import java.time.Instant;
import java.time.LocalDate;
import java.time.ZoneId;
public class InstantMain {
    public static void main(String[] args) {
        Instant instant = Instant.now();// create Instant
        System.out.println(instant);// 2020-03-18T14:48:09.787691700Z
        // parsing string
        instant = Instant.parse("2020-03-17T14:27:32.728500600Z");
        System.out.println(instant);// 2020-03-17T14:27:32.728500600Z
        // current time in millisec
        long millis = System.currentTimeMillis();
        System.out.println(millis); //1584542889801
        //millisec to date time conversion
        instant = Instant.ofEpochMilli(millis);
        System.out.println(instant);// 2020-03-18T14:48:09.801Z
        // conversion to LocalDate
        LocalDate localDate = LocalDate.ofInstant(instant, ZoneId.systemDefault());
        System.out.println(localDate);// 2020-03-18
    }
}
```

Вывод:

```
2020-03-18T14:48:09.787691700Z
2020-03-17T14:27:32.728500600Z
1584542889801
2020-03-18T14:48:09.801Z
2020-03-18
```

Еще один интересный класс **Clock** содержит текущую дату\время и позволяет получить время в миллисекундах и текущее значение объекта класса **Instant**.

```
Clock clock = Clock.systemDefaultZone();
System.out.println(clock.instant());
System.out.println(clock.millis());
```

Результат:

```
2020-03-18T15:13:07.790996900Z
1584544387798
```

Более частные классы **LocalDate**, **LocalTime**, **LocalDateTime** также предоставляют достаточно обширные возможности манипуляции информацией.

Получить текущие дату, время и дату\время можно с помощью статического метода **now()**, который объявлен в каждом из перечисленных классов.

```

LocalDate.now()
LocalTime.now()
LocalDateTime.now()

```

Если вывести результаты в поток вывода, то будет получено:

```

2020-03-18
16:00:21.175954600
2020-03-18T16:00:21.175954600

```

Дату, время и дату\время можно изменять, применяя группу методов **plusXXX()**, **minusXXX()**:

```

LocalDate localDate = LocalDate.now();
System.out.println(localDate);
localDate = localDate.plusDays(2)
                    .minusMonths(1)
                    .plusYears(1);
System.out.println(localDate);

```

Результатом будет:

```

2020-03-18
2021-02-20

```

При необходимости можно задать текущие дату, время и дату\время:

```

LocalDate localDate = LocalDate.of(2024, Month.APRIL, 9);
LocalTime localTime = LocalTime.of(23, 59, 58);
LocalDateTime localDateTime = LocalDateTime.of(localDate, localTime);
LocalDateTime localDateTime1 = LocalDateTime.of(2020, 12, 31, 23, 59, 59);

```

Дату, время и дату\время можно сравнивать на «до» и «после»:

```

LocalDate nowDate = LocalDate.now();
LocalDate afterDate = LocalDate.of(2021, 12, 31);
boolean after = nowDate.isAfter(afterDate); // false
boolean before = nowDate.isBefore(afterDate); // true
System.out.println(after + " " + before);

```

Классы пакета также умеют преобразовывать дату, время и дату\время из представления в виде строки в объект, а также форматировать их с помощью класса **DateTimeFormatter** для последующего вывода, применяя различные паттерны:

```

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("d MM yyyy");
String date = "31 03 2020";
LocalDate localDate = LocalDate.parse(date, formatter);
System.out.println(localDate);

```

Результат при английской локали операционной системы:

```

2020-03-31

```

```
DateTimeFormatter formatter2 = DateTimeFormatter.ofPattern("dd-MMMM-yyyy");
String date2 = "18-March-2020";
LocalDate localDate2 = LocalDate.parse(date2, formatter2);
System.out.println(localDate2); //default, print ISO_LOCAL_DATE
System.out.println(formatter2.format(localDate2));
```

Результат:

2020-03-18

18-March-2020

```
DateTimeFormatter formatter3 = DateTimeFormatter.ofPattern("E, MMM d yyyy");
String date3 = "Fri, May 25 2018";
LocalDate localDate3 = LocalDate.parse(date3, formatter3);
System.out.println(localDate3);
System.out.println(formatter3.format(localDate3));
```

Результат:

2018-05-25

Fri, May 25 2018

```
DateTimeFormatter formatter4 = DateTimeFormatter.ofPattern("EEEE, MMM d, yyyy HH:mm:ss a");
String date4 = "Monday, May 25, 2020 10:30:56 AM";
LocalDateTime localDateTime4 = LocalDateTime.parse(date4, formatter4);
System.out.println(localDateTime4);
System.out.println(formatter4.format(localDateTime4));
```

Результат:

2020-05-25T10:30:56

Monday, May 25, 2020 10:30:56 AM

Класс **java.time.Period** позволяет задавать и манипулировать промежутками в днях между датами:

```
LocalDate before = LocalDate.of(2020, 3, 25);
LocalDate nowDate = LocalDate.now();
System.out.println(nowDate);
Period period = Period.between(nowDate, before);
System.out.println(period);
System.out.println(period.getDays());
```

Результат:

2020-03-18

P7D

7

Класс **java.time.Duration** исполняет ту же роль, что и класс **Period**, но только промежутки измеряются в днях, часах, минутах, секундах:

```
LocalTime time06am = LocalTime.of(6, 0, 0);
LocalTime time23am = LocalTime.of(23, 15, 30);
Duration duration = Duration.between(time06am, time23am);
System.out.println(duration.toHours() + " hours"); // 17
LocalDate newYear = LocalDate.of(2020, 1, 1);
LocalDate today = LocalDate.of(2020, 9, 13);
long daysToToday = ChronoUnit.DAYS.between(newYear, today);
System.out.println(daysToToday + " days"); // 256
```

Результат:

17 hours

256 days

Форматирование информации

Для создания форматированного текстового вывода предназначен класс **java.util.Formatter**. Этот класс обеспечивает преобразование формата, позволяющее выводить числа, строки, время и даты в любом необходимом разработчику виде.

Метод **format()** преобразует переданные в него параметры в строку заданного формата и сохраняет в объекте типа **Formatter**. Аналогичный метод объявлен у классов **PrintStream** и **PrintWriter**. Кроме того, у этих классов объявлен метод **printf()** с параметрами, идентичными параметрам метода **format()**, осуществляющий форматированный вывод в поток. В то время как метод **format()** сохраняет изменения в объекте типа **Formatter**. Таким образом, метод **printf()** автоматически использует возможности класса **Formatter** и подобен функции **printf()** языка C.

Класс **Formatter** преобразует двоичную форму представления данных в форматированный текст. Он сохраняет форматированный текст в буфере, содержимое которого можно получить в любой момент. Можно предоставить классу **Formatter** автоматическую поддержку этого буфера либо задать его явно при создании объекта. Существует возможность сохранения буфера класса **Formatter** в файле.

Для создания объекта класса существует более десяти конструкторов. Ниже приведены наиболее употребляемые:

```
Formatter()
Formatter(Appendable buf)
Formatter(String filename) throws FileNotFoundException
Formatter(File outF) throws FileNotFoundException
Formatter(OutputStream outStrm)
Formatter(PrintStream printStrm)
```

В приведенных образцах **buf** задает буфер для форматированного вывода. Если параметр **buf** равен **null**, класс **Formatter** автоматически размещает объект типа **StringBuilder** для хранения форматированного вывода. Параметр **filename**

задает имя файла, который получит форматированный вывод. Параметр **outF** передает ссылку на открытый файл, в котором будет храниться форматированный вывод. В параметре **outStrm** передается ссылка на поток вывода, который будет получать отформатированные данные. Если используется файл, выходные данные записываются в файл.

Некоторые методы класса:

Formatter format(Locale loc, String fmtString, Object...args) — форматирует аргументы, переданные в аргументе переменной длины **args**, в соответствии со спецификаторами формата, содержащимися в **fmtString**. При форматировании используются региональные установки, заданные в **loc**. Возвращает вызывающий объект. Существует перегруженная версия метода без использования локализации **format(String fmtString, Object...args)**;

Locale locale() — возвращает региональные установки вызывающего объекта;

Appendable out() — возвращает ссылку на базовый объект-приемник для выходных данных;

void flush() — переносит информацию из буфера форматирования и производит запись в указанное место выходных данных, находящихся в буфере. Метод чаще всего используется объектом класса **Formatter**, связанным с файлом;

void close() — закрывает вызывающий объект класса **Formatter**, что приводит к освобождению ресурсов, используемых объектом. После закрытия объекта типа **Formatter** он не может использоваться повторно. Попытка использовать закрытый объект приводит к генерации исключения типа **FormatterClosedException**.

При форматировании используются спецификаторы формата:

Спецификатор формата	Выполняемое форматирование
%a	Шестнадцатеричное значение с плавающей точкой
%b	Логическое (булево) значение аргумента
%c	Символьное представление аргумента
%d	Десятичное целое значение аргумента
%h	Хэш-код аргумента
%e	Экспоненциальное представление аргумента
%f	Десятичное значение с плавающей точкой
%g	Выбирает более короткое представление из двух: %e или %f
%o	Восьмеричное целое значение аргумента
%n	Вставка символа новой строки
%s	Строковое представление аргумента
%t	Время и дата
%x	Шестнадцатеричное целое значение аргумента
%%	Вставка знака %

Также возможны спецификаторы с заглавными буквами: **%A** (эквивалентно **%a**). Форматирование с их помощью обеспечивает перевод символов в верхний регистр.

Форматирование текста по формату **%S, %c**:

```
Formatter formatter = new Formatter();
formatter.format("This %s is about %S %c", "book", "java", '2');
System.out.print(formatter);
```

Будет выведено:

This book is about JAVA 2

Форматирование чисел с использованием спецификаторов **%x, %o, %a, %g**:

```
Formatter formatter = new Formatter();
formatter.format("Hex: %x, Octal: %o", 100, 100);
System.out.println(formatter);
formatter = new Formatter();
formatter.format("%a", 100.001);
System.out.println(formatter);
formatter = new Formatter();
for (double i = 1000; i < 1.0e+10; i *= 100) {
    formatter.format("%g ", i);
    System.out.println(formatter);
}
```

В результате выполнения этого кода будет выведено:

```
Hex: 64, Octal: 144 0x1.90010624dd2f2p6
1000.00
1000.00 100000
1000.00 100000 1.00000e+07
1000.00 100000 1.00000e+07 1.00000e+09
```

Все спецификаторы для форматирования даты и времени могут употребляться для типов **long**, **Long**. В современной практике программирования часто для хранения даты\времени применяется тип **long**. В таблице приведены некоторые из спецификаторов формата времени и даты.

Спецификатор формата	Выполняемое преобразование
%tH	Час (00–23)
%tI	Час (1–12)
%tM	Минуты как десятичное целое (00–59)
%tS	Секунды как десятичное целое (00–59)
%tL	Миллисекунды (000–999)
%tY	Год в четырехзначном формате
%ty	Год в двузначном формате (00–99)

%tB	Полное название месяца («январь»)
%tb или %th	Краткое название месяца («январь»)
%tm	Месяц в двузначном формате (1–12)
%tA	Полное название дня недели («пятница»)
%ta	Краткое название дня недели («пт»)
%td	День в двузначном формате (1–31)
%tR	То же, что и «%tH:%tM»
%tT	То же, что и «%tH:%tM:%tS»
%tr	То же, что и «%tI:%tM:%tS %Tp», где %Tp = (АМ или РМ)
%tD	То же, что и «%tm/%td/%ty»
%tF	То же, что и «%tY-%tm-%td»
%tc	То же, что и «%ta %tb %td %tT %tZ %tY»

Числовое задание точности применяется только в спецификаторах формата **%f**, **%e**, **%g** для данных с плавающей точкой и в спецификаторе **%s** — для строк. Он задает количество выводимых десятичных знаков или символов. Например, спецификатор **%10.4f** выводит число с минимальной шириной поля 10 символов и с четырьмя десятичными знаками. Принятая по умолчанию точность равна шести десятичным знакам.

Примененный к строкам спецификатор точности задает максимальную длину поля вывода. Например, спецификатор **%5.7s** выводит строку длиной не менее пяти и не более семи символов. Если строка длиннее, лишние символы в конце строки отбрасываются.

Ниже приведен пример использования флагов форматирования.

```
Formatter formatter = new Formatter();
formatter.format("|%10.2f|", 123.123); // right alignment
System.out.println(formatter);
formatter = new Formatter();
formatter.format("|%-10.2f|", 123.123); // left alignment
System.out.println(formatter);
formatter = new Formatter();
formatter.format("% (d", -100); // flag use (
System.out.println(formatter);
formatter = new Formatter();
formatter.format("%.2f", 123456789.3456); // flag use ,
System.out.println(formatter);
formatter = new Formatter();
formatter.format("%.4f", 1111.12346789); // precision for numbers
System.out.println(formatter);
```

Результат выполнения:

```
| 123.12|
|123.12 |
```

(100)
123,456,789.35
1111.1235

Шифрование и кодирование строк

В Java реализовано несколько механизмов и алгоритмов шифрования и хэширования данных, например, в пакетах **java.security** и **java.util** соответственно. В качестве основного примера можно предложить задачу о хранении в БД паролей пользователей в зашифрованном виде. Само приложение, получив пароль, шифрует его и сравнивает с соответствующим значением из базы. Хранить пароли в открытом виде считается крайне дурным тоном. Алгоритмов шифрования очень много, как и сторонних библиотек для этой цели. Некоторые стандартные алгоритмы MD5, SHA-1, SHA-256, используются классом **MessageDigest**.

```
/* # 15 # шифрование с MD5 # MessageMain.java */
```

```
package by.epam.learn.md5;
import java.io.UnsupportedEncodingException;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
public class MessageMain {
    public static void main(String[] args) {
        String encrypted = "Pass_1";
        MessageDigest messageDigest = null;
        byte[] bytesEncoded = null;
        try {
            messageDigest = MessageDigest.getInstance("SHA-1"); // only once !
            messageDigest.update(encrypted.getBytes("utf8"));
            bytesEncoded = messageDigest.digest();
        } catch (NoSuchAlgorithmException | UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        BigInteger bigInt = new BigInteger(1, bytesEncoded); //1(sign+) or -1(sign-)
        String resHex = bigInt.toString(16);
        System.out.println(resHex);
    }
}
```

В результате строка будет зашифрована в виде:

fadb2ca605eed0c9a7740cf610b202660e694ec

При данном алгоритме шифрования результат всегда будет иметь одинаковую длину (40 разрядов), независимо от размера исходной строки.

В стандартной Java доступна еще одна схема кодирования **Base64**, преобразующая строку в набор ASCII-кодов и обратно. Предотвращает потери при

передаче данных в информационных системах, которые не поддерживают восьмибитные данные. В классе **Base64** объявлены два внутренних статических класса **Encoder** и **Decoder**, для кодирования и декодирования соответственно.

```
/* # 16 # шифрование с Base64 # Base64EncoderMain.java */
```

```
package by.epam.learn.base;
import java.math.BigInteger;
import java.util.Base64;
public class Base64EncoderMain {
    public static void main(String[] args) {
        String encrypted = "Pass_1";
        Base64.Encoder encoder = Base64.getEncoder();
        byte[] bytesEncoded = encoder.encode(encrypted.getBytes());
        BigInteger bigInt = new BigInteger(1, bytesEncoded);
        String resHex = bigInt.toString(16);
        System.out.println(resHex);
    }
}
```

Размер полученной строки зависит от размера исходной. В результате закодированная строка будет представлена в виде:

5547467a63313878

Для надежной передачи по сети иногда необходимо кодировать и файлы. Существует несколько способов кодирования и декодирования. Метод **fileEncode(String filename)** выполняет кодирование переданного файла.

```
/* # 17 # шифрование файла # Base64EncoderMain.java */
```

```
package by.epam.learn.base;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Base64;
public class Base64FileEncoder {
    public String fileEncode(String filename) throws IOException {
        try (FileInputStream input = new FileInputStream(filename)) {
            Base64.Encoder encoder = Base64.getEncoder();
            try (OutputStream output = encoder.wrap(new FileOutputStream(filename + ".encode"))) {
                int bytes;
                while ((bytes = input.read()) != -1) {
                    output.write(bytes);
                }
            }
        }
        return filename + ".encode";
    }
}
```

Если в файле **data/t.txt** была информация вида:

Hello world 123

То в закодированном файле **data/t.txt.encode** будет:

SGVsbG8gd29ybGQgMTIzDQo=

Метод **fileDecode(String filename)** выполняет декодирование переданного файла.

```
/* # 18 # расшифровка файла # Base64FileDecoder.java */
```

```
package by.epam.learn.base;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Base64;
public class Base64FileDecoder {
    public String fileDecode(String filenameEncode) throws IOException {
        try (FileOutputStream fos = new FileOutputStream(filenameEncode + ".decode")) {
            Base64.Decoder decoder = Base64.getDecoder();
            try(InputStream input = decoder.wrap(new FileInputStream(filenameEncode))) {
                int bytes;
                while ((bytes = input.read()) != -1) {
                    fos.write(bytes);
                }
            }
        }
        return filenameEncode + ".decode";
    }
}
```

```
/* # 19 # # Base64FileMain.java */
```

```
package by.epam.learn.base;
import java.io.IOException;
public class Base64FileMain {
    public static void main(String[] args) {
        String filename = "data/t.txt";
        Base64FileEncoder encoder = new Base64FileEncoder();
        Base64FileDecoder decoder = new Base64FileDecoder();
        try {
            String filenameEncode = encoder.fileEncode(filename);
            String filenameDecode = decoder.fileDecode(filenameEncode);
            System.out.println(filenameDecode + " it's ok");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

В итоге в раскодированном файле `data/t.txt.encode.decode` будет храниться такая же информация, как и в исходном файле до кодирования.

Вопросы к главе 8

1. Как создать объект класса **String**, какие существуют конструкторы класса **String**? Что такое строковый литерал? Что значит «упрощенное создание объекта **String**»?
2. Как реализован класс **String**, какие поля в нем объявлены?
3. Как работает метод **substring()** класса **String**?
4. Можно ли изменить состояние объекта типа **String**? Что происходит при попытке изменения состояния объекта типа **String**? Можно ли наследоваться от класса **String**? Почему строковые объекты *immutable*?
5. Что такое пул литералов? Как строки заносятся в пул литералов? Как занести строку в пул литералов и как получить ссылку на строку, хранящуюся в пуле литералов? В каком отделе памяти хранится пул литералов в Java 1.6 и Java 1.7?
6. В чем отличие объектов классов **StringBuilder** и **StringBuffer** от объектов класса **String**? Какой из этих классов потокобезопасный?
7. Как необходимо сравнивать на равенство объекты классов **StringBuilder** и **StringBuffer** и почему?
8. Что такое **Unicode**? Что такое *code point*? Отличия UTF-8 от UTF-16.
9. Как кодируется символ согласно кодировке UTF-8, UTF-16 и UTF-32?
10. Что такое кодировка? Какие кодировки вы знаете? Как создать строки в различной кодировке?
11. Какие методы класса **String** используются для работы с кодовыми точками? Когда следует их использовать?
12. Что представляет собой регулярное выражение? Что такое метасимволы регулярного выражения? Какие существуют классы символов регулярных выражений? Что такое квантификаторы? Какие существуют логические операторы регулярных выражений?
13. Какие классы Java работают с регулярными выражениями? В каком пакете они расположены?
14. Что такое группы в регулярных выражениях? Как нумеруются группы? Что представляет собой группа номер «0»?
15. Что такое интернационализация и локализация?
16. Что представляет собой локаль в программе? Назначение объектов класса **Locale**? Как получить локаль? Как узнать, какие локали доступны?
17. Какую информацию можно локализовать автоматически, применяя объект класса **Locale**? Как работают классы **NumberFormat** и **DateFormat**?
18. Как можно локализовать приложение, используя класс **ResourceBundle**? Для каких еще целей, кроме локализации, можно применять объекты этого класса?

Задания к главе 8

Вариант А

1. В каждом слове текста k -ю букву заменить заданным символом. Если k больше длины слова, корректировку не выполнять.
2. В тексте каждую букву заменить ее порядковым номером в алфавите. При выводе в одной строке печатать текст с двумя пробелами между буквами, в следующей строке внизу под каждой буквой печатать ее номер.
3. В тексте после буквы Р, если она не последняя в слове, ошибочно напечатана буква А вместо О. Внести исправления в текст.
4. В тексте после k -го символа вставить заданную подстроку.
5. После каждого слова текста, заканчивающегося заданной подстрокой, вставить указанное слово.
6. В зависимости от признака (0 или 1) в каждой строке текста удалить указанный символ везде, где он встречается, или вставить его после k -го символа.
7. Из текста удалить все символы, кроме пробелов, не являющиеся буквами. Между последовательностями подряд идущих букв оставить хотя бы один пробел.
8. Удалить из текста его часть, заключенную между двумя символами, которые вводятся (например, между скобками «(» и «)» или между звездочками «*» и т.п.).
9. Определить, сколько раз повторяется в тексте каждое слово, которое встречается в нем.
10. В тексте найти и напечатать n символов (и их количество), встречающихся наиболее часто.
11. Найти, каких букв, гласных или согласных, больше в каждом предложении текста.
12. В стихотворении найти количество слов, начинающихся и заканчивающихся гласной буквой.
13. Напечатать без повторения слова текста, у которых первая и последняя буквы совпадают.
14. В тексте найти и напечатать все слова максимальной и все слова минимальной длины.
15. Напечатать квитанцию об оплате телеграммы, если стоимость одного слова задана.
16. В стихотворении найти одинаковые буквы, которые встречаются во всех словах.
17. В тексте найти первую подстроку максимальной длины, не содержащую букв.
18. В тексте определить все согласные буквы, встречающиеся не более чем в двух словах.
19. Преобразовать текст так, чтобы каждое слово, не содержащее неалфавитных символов, начиналось с заглавной буквы.
20. Подсчитать количество содержащихся в данном тексте знаков препинания.

21. В заданном тексте найти сумму всех встречающихся цифр.
22. Из кода Java удалить все комментарии (`//`, `/*`, `/**`).
23. Все слова текста встречаются четное количество раз, за исключением одного. Определить это слово. При сравнении слов регистр не учитывать.
24. Определить сумму всех целых чисел, встречающихся в заданном тексте.
25. Из текста удалить все лишние пробелы, если они разделяют два различных знака препинания, и если рядом с ними находится еще один пробел.
26. Строка состоит из упорядоченных чисел от 0 до 100000, записанных подряд без пробелов. Определить, что будет подстрокой от позиции n до m .
27. Определить количество вхождений заданного слова в текст, игнорируя регистр символов и считая буквы «е», «ё», и «и», «й» одинаковыми.
28. Преобразовать текст так, чтобы только первые буквы каждого предложения были заглавными.
29. Заменить все одинаковые рядом стоящие в тексте символы одним символом.
30. Вывести в заданном тексте все слова, расположив их в алфавитном порядке.
31. Подсчитать, сколько слов в заданном тексте начинается с прописной буквы.
32. Подсчитать, сколько раз заданное слово входит в текст.

Вариант В

Создать программу обработки текста учебника по программированию с использованием классов: *Символ*, *Слово*, *Предложение*, *Абзац*, *Лексема*, *Листинг*, *Знак препинания* и др. Во всех задачах с формированием текста заменять табуляции и последовательности пробелов одним пробелом.

Предварительно текст следует разобрать на составные части, выполнить одно из перечисленных ниже заданий и вывести полученный результат.

1. Найти наибольшее количество предложений текста, в которых есть одинаковые слова.
2. Вывести все предложения заданного текста в порядке возрастания количества слов в каждом из них.
3. Найти такое слово в первом предложении, которого нет ни в одном из остальных предложений.
4. Во всех вопросительных предложениях текста найти и напечатать без повторений слова заданной длины.
5. В каждом предложении текста поменять местами первое слово с последним, не изменяя длины предложения.
6. Напечатать слова текста в алфавитном порядке по первой букве. Слова, начинающиеся с новой буквы, печатать с красной строки.
7. Рассортировать слова текста по возрастанию доли гласных букв (отношение количества гласных к общему количеству букв в слове).
8. Слова текста, начинающиеся с гласных букв, рассортировать в алфавитном порядке по первой согласной букве слова.

9. Все слова текста рассортировать по возрастанию количества заданной буквы в слове. Слова с одинаковым количеством расположить в алфавитном порядке.
10. Существует текст и список слов. Для каждого слова из заданного списка найти, сколько раз оно встречается в каждом предложении, и рассортировать слова по убыванию общего количества вхождений.
11. В каждом предложении текста исключить подстроку максимальной длины, начинающуюся и заканчивающуюся заданными символами.
12. Из текста удалить все слова заданной длины, начинающиеся на согласную букву.
13. Отсортировать слова в тексте по убыванию количества вхождений заданного символа, а в случае равенства — по алфавиту.
14. В заданном тексте найти подстроку максимальной длины, являющуюся палиндромом, т.е. читающуюся слева направо и справа налево одинаково.
15. Преобразовать каждое слово в тексте, удалив из него все следующие (предыдущие) вхождения первой (последней) буквы этого слова.
16. В некотором предложении текста слова заданной длины заменить указанной подстрокой, длина которой может не совпадать с длиной слова.

Вариант С

1. Проверить, является ли строка сильным паролем. Пароль считается сильным, если его длина больше либо равна 10 символам, он содержит как минимум одну цифру, одну букву в верхнем и одну букву в нижнем регистре. Пароль может содержать только латинские буквы и/или цифры, а также символ «_».
2. Текст из n^2 символов шифруется по следующему правилу:
 - все символы текста записываются в квадратную таблицу размерности n в порядке слева направо, сверху вниз;
 - таблица поворачивается на 90° по часовой стрелке;
 - 1-я строка таблицы меняется местами с последней, 2-я — с предпоследней и т.д.;
 - 1-й столбец таблицы меняется местами со 2-м, 3-й — с 4-м и т.д.;
 - зашифрованный текст получается в результате обхода результирующей таблицы по спирали по часовой стрелке, начиная с левого верхнего угла.
 Зашифровать текст по указанному правилу.
3. Исключить из текста подстроку максимальной длины, начинающуюся и заканчивающуюся одним и тем же символом.
4. Вычеркнуть из текста минимальное количество предложений так, чтобы у любых двух оставшихся предложений было хотя бы одно общее слово.
5. Осуществить сжатие английского текста, заменив каждую группу из двух или более рядом стоящих символов на один символ, за которым следует количество его вхождений в группу. К примеру, строка *helloworld* должна сжиматься в *hel2owo4rld*.
6. Распаковать текст, сжатый по правилу из предыдущего задания.

7. Определить, удовлетворяет ли имя файла маске. Маска может содержать символы «?» (произвольный символ) и «*» (произвольное количество произвольных символов).
8. Буквенная запись телефонных номеров основана на том, что каждой цифре соответствует несколько английских букв: 2 — ABC, 3 — DEF, 4 — GHI, 5 — JKL, 6 — MNO, 7 — PQRS, 8 — TUV, 9 — WXYZ. Написать программу, которая находит в заданном телефонном номере подстроку максимальной длины, соответствующую слову из словаря.
9. Осуществить форматирование заданного текста с выравниванием по левому краю. Программа должна разбивать текст на строки с длиной, не превосходящей заданного количества символов. Если очередное слово не помещается в текущей строке, его необходимо переносить на следующую.
10. Изменить программу из предыдущего примера так, чтобы она осуществляла форматирование с выравниванием по обоим краям. Для этого добавить дополнительные пробелы между словами.
11. Добавить к программе из предыдущего примера возможность переноса слов по слогам. Предполагается, что есть доступ к словарю, в котором для каждого слова указано, как оно разбивается на слоги.
12. Пусть текст содержит миллион символов, необходимо сформировать из них строку путем конкатенации. Определить время работы кода. Ускорить процесс, используя класс **StringBuilder**.
13. Алгоритм Барроуза-Уиллера для сжатия текстов основывается на преобразовании Барроуза-Уиллера. Оно производится следующим образом: для слова рассматриваются все его циклические сдвиги, которые затем сортируются в алфавитном порядке, после чего формируется слово из последних символов отсортированных циклических сдвигов. К примеру, для слова JAVA циклические сдвиги — это JAVA, AVAJ, VAJA, AJAV. После сортировки по алфавиту получим AJAV, AVAJ, JAVA, VAJA. Значит, результат преобразования — слово VJAA. Реализовать программно преобразование Барроуза-Уиллера для данного слова.
14. Восстановить слово по его преобразованию Барроуза-Уиллера. К примеру, получив на вход VJAA, в результате работы программа должна выдать слово JAVA.
15. В Java код добавить корректные *getter* и *setter*-методы для всех полей данного класса при их отсутствии.
16. В тексте нет слов, начинающихся одинаковыми буквами. Напечатать слова текста в таком порядке, чтобы последняя буква каждого слова совпадала с первой буквой следующего слова. Если все слова нельзя напечатать в таком порядке, найти такую цепочку, состоящую из наибольшего количества слов.
17. Текст шифруется по следующему правилу: из исходного текста выбирается 1, 4, 7, 10-й и т.д. (до конца текста) символы, затем 2, 5, 8, 11-й и т.д. (до конца текста) символы, затем 3, 6, 9, 12-й и т.д. Зашифровать заданный текст.
18. В предложении из n слов первое слово поставить на место второго, второе — на место третьего и т.д., $(n-1)$ -е слово — на место n -го, n -е слово поставить на

место первого. В исходном и преобразованном предложениях между словами должны быть или один пробел, или знак препинания и один пробел.

19. Все слова текста рассортировать в порядке убывания их длин, при этом все слова одинаковой длины рассортировать в порядке возрастания в них количества гласных букв.

Тестовые задания к главе 8

Вопрос 8.1.

Дан код:

```
public class A {
    public static void main(String[] args){
        String str = "Hello";
        System.out.print(str);
        doWork(str);
        System.out.print(str);
    }
    public static void doWork(String value){
        value = null;
    }
}
```

Что будет результатом компиляции и запуска? (выбрать один)

- a) Hello
- b) Hellonull
- c) HelloHello
- d) runtime error

Вопрос 8.2.

Дан код:

```
String s = new String("3");
System.out.println(1 + 2 + s + 4 + 5);
```

В результате при компиляции и запуске будет выведено (выбрать один):

- a) 12345
- b) 3345
- c) 1239
- d) 339
- e) 15
- f) compilation fails

Вопрос 8.3.

Дан фрагмент кода:

```
String s1 = new String("Java");
String s2 = "Java";
```

```
String s3 = new String(s1);
String s4 = "Java";
```

Какие из предложенных операторов дадут результат *true*? (выбрать два)

- a) `s1 == s2`
- b) `s1 == s3`
- c) `s2 == s4`
- d) `s2 == s3`
- e) `s2.equals(s1)`

Вопрос 8.4.

Что будет выведено на консоль при компиляции и выполнении следующих строчек кода? (выбрать один)

```
String[] strings = new String[]{"a", "b", "c"};
int k = 0;
for (String element : strings) {
    strings[k].concat(String.valueOf(k));
    ++k;
}
System.out.print(Arrays.toString(strings));
```

- a) `[a, b, c]`
- b) `[a0, b1, c2]`
- c) `[a1, b2, c3]`
- d) `compilation fails`

Вопрос 8.5.

Дан фрагмент кода:

```
String st = "0";
StringBuffer sb = new StringBuffer("a");
// 1
// 2
System.out.print(st);
System.out.print(sb);
```

Какой из фрагментов кода, будучи вставленным вместо комментария 1 и 2 соответственно, выведет в консоль *01ab*? (выбрать один)

- a) `st = st.concat("1");sb.append("b");`
- b) `st.concat("1");`
- c) `sb.append("b");`
- d) `st = st.concat("1");`
- e) `sb.concat("b");`
- f) `st = st.append("1");`
- g) `sb.append("b");`

ИСКЛЮЧЕНИЯ И ОШИБКИ

Существуют только ошибки.

Аксиома Робертса

Что для одного ошибка, для другого — исходные данные.

Следствие Бермана из аксиомы Робертса

*Никогда не выявляйте в программе ошибки, если не знаете,
что с ними делать.*

Руководство Штейнбаха

Иерархия исключений и ошибок

Исключительные ситуации (исключения) и ошибки возникают во время выполнения программы, когда появившаяся проблема не может быть решена в текущем контексте и невозможно продолжение работы программы. Обычно считается, что исключения и ошибки — тождественные понятия. Примерами «популярных» ошибок являются: попытка индексации вне границ массива, вызов метода на нулевой ссылке или деление на ноль. При возникновении исключения в приложении создается объект, описывающий это исключение. Затем текущий ход выполнения приложения останавливается, и включается механизм обработки исключений. При этом ссылка на объект-исключение передается обработчику исключений, который пытается решить возникшую проблему и продолжить выполнение программы. Если в классе используется метод, в котором может возникнуть проверяемая исключительная ситуация, но не предусмотрена ее обработка, то ошибка возникает еще на этапе компиляции. При создании такого метода программист обязан включить в код метода обработку исключений, которые могут генерироваться в этом методе, или передать обработку исключения на более высокий уровень методу, вызвавшему данный метод. Схема обработки исключения подобна схеме обработки событий.

Исключение не должно восприниматься как нечто вредное, от которого следует избавиться любой ценой. Исключение — это источник дополнительной информации о ходе выполнения приложения. Такая информация позволяет лучше адаптировать код к конкретным условиям его использования, а также на ранней стадии выявить ошибки или защититься от их возникновения в будущем. В противном

случае «подавление» исключений приведет к тому, что о возникшей ошибке никто не узнает или узнает на стадии некорректно обработанной информации. Поиск места возникновения ошибки может быть затруднительным.

Каждой исключительной ситуации поставлен в соответствие некоторый класс, экземпляр которого иницируется при ее появлении. Если подходящего класса не существует, то он может быть создан разработчиком. Все исключения являются наследниками суперкласса **Throwable** и его подклассов **Error** и **Exception** из пакета **java.lang**.

Исключительные ситуации типа **Error** возникают только во время выполнения программы. Такие исключения, связанные с серьезными ошибками, к примеру, с переполнением стека, не подлежат исправлению и не могут обрабатываться приложением. Собственные подклассы от **Error** создавать мало смысла по причине невозможности управления прерываниями. Некоторые классы из иерархии, наследуемые от класса **Error**, приведены на рисунке 9.2.

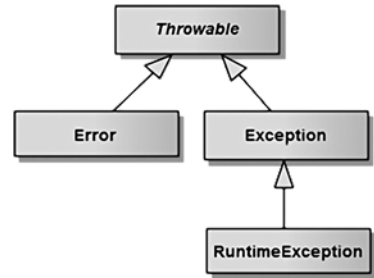


Рис. 9.1. Иерархия основных классов исключений

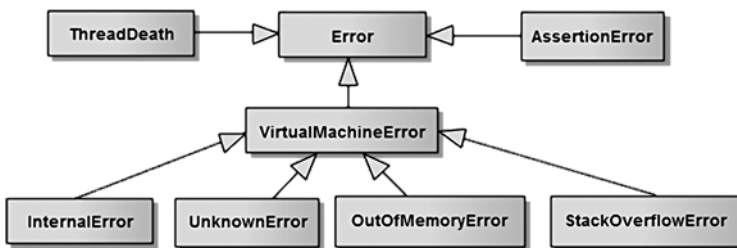


Рис. 9.2. Некоторые классы исключений, наследуемые от класса **Error**

На рисунке 9.3 приведена иерархия классов проверяемых исключений, наследуемых от класса **Exception** при отсутствии в цепочке наследования класса **RuntimeException**. Возможность возникновения проверяемого исключения может быть отслежена еще на этапе компиляции кода. Компилятор проверяет, может ли данный метод генерировать или обрабатывать исключение.

Проверяемые исключения должны быть обработаны в методе, который может их генерировать, или включены в **throws**-список метода для дальнейшей обработки в вызывающих методах.

Во время выполнения могут генерироваться также исключения, которые могут быть обработаны без ущерба для выполнения программы. Список этих исключений приведен на рисунке 9.4. В отличие от проверяемых исключений, класс **RuntimeException** и порожденные от него классы относятся к непроверяемым

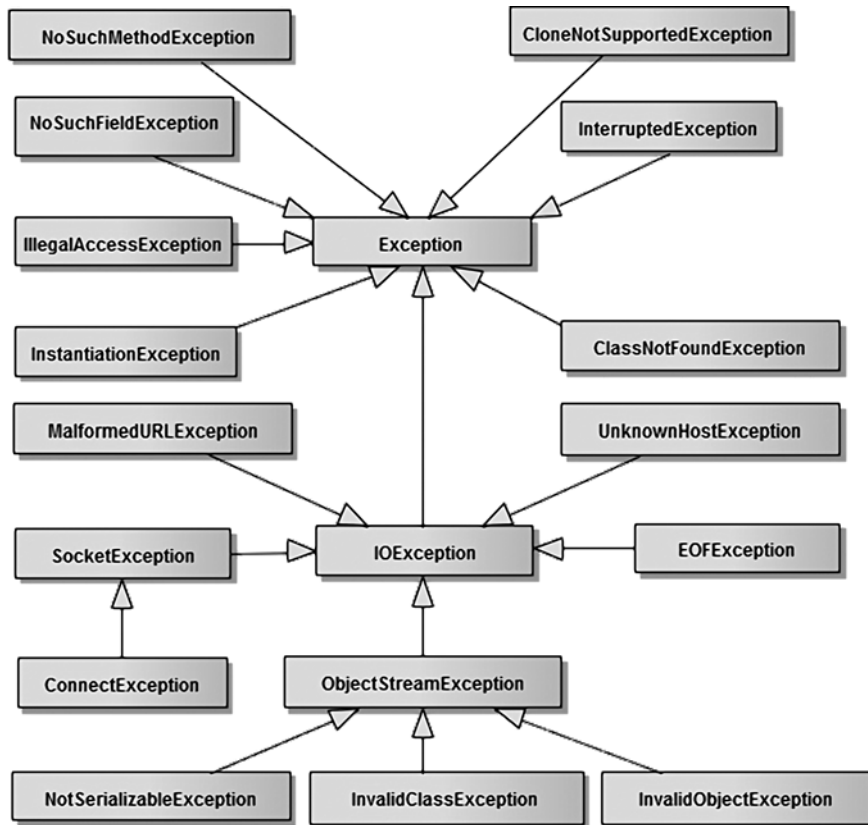


Рис. 9.3. Иерархия классов, проверяемых (checked) исключительных ситуаций

исключениям. Компилятор не проверяет, может ли генерировать и/или обрабатывать метод эти исключения. Исключения типа **RuntimeException** генерируются при возникновении ошибок во время выполнения приложения.

Ниже приведен список часто встречаемых в практике программирования непроверяемых исключений, знание причин возникновения которых необходимо при создании качественного кода.

Почему возникла необходимость деления исключений на проверяемые и непроверяемые? Представим, что следующие ситуации проверяются на этапе компиляции, а именно:

- деление в целочисленных типах вида **a/b** при **b=0** генерирует исключение **ArithmeticException**;
- индексация массивов, строк, коллекций. Выход за пределы такого объекта приводит к исключению **ArrayIndexOutOfBoundsException** и аналогичных;
- вызов метода на ссылке вида **obj.method()**, если **obj** ссылается на **null**.

Исключение	Значение
ArithmeticException	Арифметическая ошибка: деление на ноль и др.
ArrayIndexOutOfBoundsException	Индекс массива находится вне его границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
ConcurrentModificationException	Некорректный способ модификации коллекции
IllegalArgumentException	При вызове метода использован некорректный аргумент
IllegalMonitorStateException	Незаконная операция монитора на разблокированном объекте
IllegalStateException	Среда или приложение находятся в некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ коллекции
NegativeArraySizeException	Попытка создания массива с отрицательным размером
NullPointerException	Недопустимое использование ссылки на null
NumberFormatException	Невозможное преобразование строки в числовой формат
StringIndexOutOfBoundsException	Попытка индексации вне границ строки
MissingResourceException	Отсутствие файла ресурсов properties или имени ресурса в нем
EnumConstantNotPresentException	Несуществующий элемент перечисления
UnsupportedOperationException	Встретилась неподдерживаемая операция

Рис. 9.4. Классы непроверяемых исключений, наследуемых от класса *RuntimeException*

Если бы возможность появления перечисленных исключений проверялась на этапе компиляции, то любая попытка индексации массива или каждый вызов метода требовали бы или блока **try-catch**, или секции **throws**. Такой код был бы практически непригоден для понимания и поддержки, поэтому часть исключений была выделена в группу непроверяемых и ответственность за защиту приложения от последствий их возникновения возложена на программиста.

Способы обработки исключений

Если при возникновении исключения в текущем методе обработчик не будет обнаружен, то его поиск будет продолжен в методе, вызвавшем данный метод, и так далее вплоть до метода **main()** для консольных приложений или другого метода, запускающего соответствующее приложение. Если же и там исключение не будет перехвачено, то JVM выполнит аварийную остановку приложения с вызовом метода **printStackTrace()**, выдающего данные трассировки.

Для проверяемого исключения возможность его генерации отслеживается. Передача обработки вызывающему методу осуществляется с помощью оператора

throws. В конце концов исключение может быть передано в метод **main()**, где и находится крайняя точка обработки. Добавлять оператор **throws** методу **main()** представляется дурным тоном программирования, как безответственное действие программиста, не обращающего никакого внимания на альтернативное выполнение программы.

На практике используется один из двух способов обработки исключений:

- перехват и обработка исключения в блоке **try-catch** метода;
- объявление исключения в секции **throws** метода и передача вызывающему методу (в первую очередь для проверяемых исключений).

Первый подход можно рассмотреть на следующем примере. При преобразовании содержимого строки к числу в определенных ситуациях может возникнуть проверяемое исключение типа **ParseException**. Например:

```
public double parseFromFrance(String numberStr) {
    NumberFormat format = NumberFormat.getInstance(Locale.FRANCE);
    double numFrance = 0;
    try {
        numFrance = format.parse(numberStr).doubleValue();
    } catch (ParseException e) { // checked exception
        // 1. throwing a standard exception, : IllegalArgumentException() – not very good
        // 2. throwing a custom exception, where ParseException as a parameter
        // 3. setting the default value - if possible
        // 4. logging if an exception is unlikely
    }
    return numFrance;
}
```

Исключительная ситуация возникнет в случае, если переданная строка содержит нечисловые символы или не является числом. Генерируется объект исключения, и управление передается соответствующему блоку **catch**, в котором он обрабатывается, иначе блок **catch** пропускается. Блок **try** похож на обычный логический блок. Блок **catch()** похож на метод, принимающий в качестве единственного параметра ссылку на объект-исключение и обрабатывающий этот объект.

Второй подход демонстрируется на этом же примере. Метод может генерировать исключения, которые сам не обрабатывает, а передает для обработки другим методам, вызывающим данный метод. В этом случае метод должен объявить о таком поведении с помощью ключевого слова **throws**, чтобы вызывающий метод мог защитить себя от этих исключений. В вызывающем методе должна быть предусмотрена или обработка этих исключений, или последующая передача соответствующему методу.

При этом объявляемый метод может содержать блоки **try-catch**, а может и не содержать их. Например, метод **parseFrance()** можно объявить:

```
public double parseFrance(String numberStr) throws ParseException {
    NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
    double numFrance = formatFrance.parse(numberStr).doubleValue();
    return numFrance;
}
```

В практическом программировании такой подход допустим для **private**-методов.

Ключевое слово **throws** после имени метода позволяет разобраться с исключениями методов «чужих» классов, код которых отсутствует. Обработать исключение при этом должен будет метод, вызывающий **parseFrance()**:

```
public void doAction() {
    // code here
    try {
        parseFrance(numberStr);
    } catch (ParseException e) {
        // code
    }
}
```

Создание и применение собственных исключений будет рассмотрено ниже. В этой главе.

Обработка нескольких исключений

Если в блоке **try** может быть сгенерировано в разных участках кода несколько типов исключений, то необходимо наличие нескольких блоков **catch**, если только блок **catch** не обрабатывает все типы исключений.

```
/* # 1 # обработка двух типов исключений # */
```

```
public void doAction() {
    try {
        int a = (int) (Math.random() * 2);
        System.out.println("a = " + a);
        int c[] = { 1 / a }; // place of occurrence of exception #1
        c[a] = 71; // place of occurrence of exception #2
    } catch (ArithmeticException e) {
        System.err.println("divide by zero " + e);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("out of bound: " + e);
    } // end try-catch block
    System.out.println("after try-catch");
}
```

Исключение **ArithmeticException** при делении на 0 возникнет при инициализации элемента массива **c[0]** действием **1/a** при **a=0**. В случае **a=1** генерируется исключение «превышение границ массива» при попытке присвоить значение второму элементу массива **c[]**, содержащего только один элемент. Однако пример, приведенный выше, носит чисто демонстративный характер и не является образцом хорошего кода, так как в этой ситуации можно было обойтись простой проверкой аргументов на допустимые значения перед выполнением операций. К тому же генерация и обработка исключения — операция значительно более ресурсоемкая, чем вызов оператора **if** для проверки аргумента. Исключения должны применяться только для обработки исключи-

тельных ситуаций, и если существует возможность обойтись без них, то следует так и поступить.

Подклассы исключений в блоках **catch** должны следовать перед любым из их суперклассов, иначе суперкласс будет перехватывать эти исключения. Например:

```
try { //...
} catch(IllegalArgumentException e) { //...
} catch(PatternSyntaxException e) { // unreachable code
}
```

где класс **PatternSyntaxException** представляет собой подкласс класса **IllegalArgumentException**. Корректно будет просто поменять местами блоки **catch**:

```
try {
} catch(PatternSyntaxException e) { //..
} catch(IllegalArgumentException e) { //..
}
```

На практике иногда возникают ситуации, когда инструкций **catch** несколько, и обработка производится идентичная, например, вывод сообщения об исключении в журнал.

```
try {
    // some code
} catch(NumberFormatException e) {
    e.printStackTrace(); // or Log
} catch(ClassNotFoundException e) {
    e.printStackTrace(); // or Log
} catch(InstantiationException e) {
    e.printStackTrace(); // or Log
}
```

В версии Java 7 появилась возможность объединить все идентичные инструкции в одну, используя для разделения оператор «|».

```
try {
    // some code
} catch(NumberFormatException | ClassNotFoundException | InstantiationException e){
    e.printStackTrace();
}
```

В **catch** не могут находиться исключения из одной иерархической цепочки. Такая запись позволяет избавиться от дублирования кода.

Введено понятие более точной переброски исключений (*more precise rethrow*). Это решение применимо в случае, если обработка возникающих исключений не предусматривается в методе и должна быть передана вызывающему данный метод методу.

До введения этого понятия код выглядел так:

```
public double parseFromFileBefore(String filename)
    throws FileNotFoundException, ParseException, IOException {
    NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
    double numFrance = 0;
    BufferedReader bufferedReader = null;
    try {
        FileReader reader = new FileReader(filename);
        bufferedReader = new BufferedReader(reader);
        String number = bufferedReader.readLine();
        numFrance = formatFrance.parse(number).doubleValue();
    } catch (FileNotFoundException e) {
        throw e;
    } catch (IOException e) {
        throw e;
    } catch (ParseException e) {
        throw e;
    } finally {
        if (bufferedReader != null) {
            bufferedReader.close();
        }
    }
    return numFrance;
}
```

More precise rethrow разрешает записать в единственную инструкцию **catch** более общее исключение, чем может быть генерировано в инструкции **try**, с последующей генерацией перехваченного исключения для его передачи в вызывающий метод.

```
public double parseFile(String filename)
    throws FileNotFoundException, ParseException, IOException {
    NumberFormat formatFrance = NumberFormat.getInstance(Locale.FRANCE);
    double numFrance = 0;
    BufferedReader bufferedReader = null;
    try {
        FileReader reader = new FileReader(filename);
        bufferedReader = new BufferedReader(reader);
        String number = bufferedReader.readLine();
        numFrance = formatFrance.parse(number).doubleValue();
    } catch (final Exception e) { // final – optional
        throw e; // more precise rethrow
    } finally {
        if (bufferedReader != null) {
            bufferedReader.close();
        }
    }
    return numFrance;
}
```

Наличие секции **throws** контролируется компилятором на предмет точного указания списка проверяемых исключений, которые могут быть генерированы в блоке **try-catch**. При возможности возникновения непроверяемых исключений последние в секции **throws** обычно не указываются. Ключевое слово **final** не позволяет подменить экземпляр исключения для передачи за пределы метода. Однако данную конструкцию можно использовать и без **final**.

Операторы **try** можно вкладывать друг в друга. Если у оператора **try** низкого уровня нет раздела **catch**, соответствующего возникшему исключению, поиск будет развернут на одну ступень выше и будут проверены разделы **catch** внешнего оператора **try**.

```
/* # 2 # вложенные блоки try-catch # */
```

```
try { // outer block
    int a = (int) (Math.random() * 2) - 1;
    System.out.println("a = " + a);
    try { // inner block
        int b = 1 / a;
        StringBuilder builder = new StringBuilder(a);
    } catch (NegativeArraySizeException e) {
        System.err.println("invalid buffer size: " + e);
    }
} catch (ArithmeticException e) {
    System.err.println("divide by zero: " + e);
}
```

В результате запуска приложения при **a=0** будет сгенерировано исключение **ArithmeticException**, а подходящий для его обработки блок **try-catch** является внешним по отношению к месту генерации исключения. Этот блок и будет задействован для обработки возникшей исключительной ситуации. Вкладывание блоков **try-catch** друг в друга загромождает код, поэтому такими конструкциями следует пользоваться с осторожностью.

Оператор throw

При разработке кода возникают ситуации, когда в приложении необходимо инициировать генерацию исключения для указания, например, на заведомо ошибочный результат выполнения операции, на некорректные значения параметра метода и др. Для генерации исключительной ситуации и создания экземпляра исключения используется оператор **throw**. В качестве исключения должен быть использован объект подкласса класса **Throwable**, а также ссылки на них.

Общая форма записи инструкции **throw**, генерирующей исключение:

```
throw subclassThrowable;
```

Объект-исключение может уже существовать или создаваться с помощью оператора **new**:

```
throw new IllegalArgumentException();
```

При достижении оператора **throw**, выполнение кода прекращается. Ближайший блок **try** проверяется на наличие соответствующего обработчика **catch**. Если он существует, управление передается ему, иначе проверяется следующий из вложенных операторов **try**. Инициализация объекта-исключения без оператора **throw** никакой исключительной ситуации не вызовет.

В ситуации, когда получение методом достоверной информации критично для выполнения им своей функциональности, у программиста может возникнуть необходимость в генерации исключения, так как метод не может выполнить ожидаемых от него действий, основываясь на некорректных или ошибочных данных. Ниже приведен пример, в котором оператор **throw** генерирует исключение, обрабатываемое виртуальной машиной при выбросе из метода **main()**.

```
/* # 3 # генерация исключений # ActionMain.java # ResourceAction.java
# Resource.java */
```

```
package by.epam.learn.exception;
public class ResourceAction {
    public static void load(Resource resource) {
        if (resource == null || !resource.exists() || !resource.isCreate()) {
            throw new IllegalArgumentException();
            // better custom exception, eg., throw new ResourceException();
        }
        // more code
    }
}

package by.epam.learn.exception;
public class ActionMain {
    public static void main(String[] args) {
        Resource resource = new Resource(); //or Resource resource = null;!!!
        ResourceAction.Load(resource);
    }
}

package by.epam.learn.exception;
public class Resource {
    // fields
    public boolean isCreate() {
        // more code
    }
    public boolean exists() {
        // more code
    }
    public void execute() {
```

```
// more code
}
public void close() {
    // more code
}
}
```

Вызываемый метод **load()** может при отсутствии требуемого ресурса или при аргументе **null** генерировать исключение, перехватываемое обработчиком. В результате экземпляр непроверяемого исключения **IllegalArgumentException** как подкласса класса **RuntimeException** передается обработчику исключений в методе **main()**.

Собственные исключения

Для повышения качества и скорости восприятия кода разработчику следует создать собственное исключение как подкласс класса **Exception**, а затем использовать его при обработке ситуации, не являющейся исключением с точки зрения языка, но нарушающей логику вещей. По соглашению наследник любого класса-исключения должен заканчиваться словом **Exception**.

В случае генерации собственного проверяемого исключения **ResourceException**, компилятор требует обработки объекта исключения в методе или передачи его с помощью инструкции **throws**. Собственное исключение может быть создано как

```
/* # 4 # класс собственного исключения # ResourceException.java */
public class ResourceException extends Exception {
    public ResourceException() {
    }
    public ResourceException(String message, Throwable cause) {
        super(message, cause);
    }
    public ResourceException(String message) {
        super(message);
    }
    public ResourceException(Throwable cause) {
        super(cause);
    }
}
```

У подкласса класса **Exception** обычно определяются минимум три конструктора, два из которых в качестве параметра принимают объект типа **Throwable**, что означает генерацию исключения на основе другого исключения.

Один из параметров — сообщение, которое может быть выведено в поток ошибок; другой — реальное исключение, которое привело к вызову собственного исключения. Этот подход показывает, как можно сохранить дополнительную

информацию внутри пользовательского исключения. Преимущество этого сохранения состоит в том, что если вызываемый метод захочет узнать реальную причину генерации исключения, он всего лишь должен вызвать метод **getCause()**. Это позволяет вызываемому методу решить, нужно ли работать со специфичным исключением или достаточно обработки собственного исключения. Иногда цепочка вложенных исключений может быть достаточно большой, но всегда остается возможность на любом этапе узнать первопричину возникновения цепочки. Тогда метод **load()** с применением собственного исключения **ResourceException** для класса **ResourceAction** будет выглядеть так:

```
public static void load(Resource resource) throws ResourceException {
    if (resource == null || !resource.exists() || !resource.isCreate()) {
        throw new ResourceException();
    }
    // more code
}
```

В этом случае в вызывающем методе обработка исключения будет обязательна.

Если же генерируется стандартное исключение или получены такие значения некоторых параметров, что генерация какого-либо стандартного исключения становится неизбежной немедленно либо сразу по выходе из метода, то следует генерировать собственное исключение.

```
public int loadFile(String filename) throws ResourceException {
    int data;
    try {
        FileReader reader = new FileReader(filename);
        data = reader.read();
    } catch (IOException e) {
        throw new ResourceException(e);
    }
    return data;
}
```

Если метод генерирует исключение с помощью оператора **throw** и при этом блок **catch** в методе отсутствует, то для передачи обработки исключения вызывающему методу тип проверяемого (*checked*) класса исключений должен быть указан в операторе **throws** при объявлении метода.

```
Resource resource = new Resource();
try { // required !!
    ResourceAction.load(resource);
} catch (ResourceException e) {
    System.err.print(e);
}
```

Обязательно передавать перехваченное блоком **catch** исключение в качестве параметра для возможности развертывания стека исключений в момент его обработки.

Если ошибка некритическая, то вместо генерации исключения можно просто записать лог.

Разработчики программного обеспечения стремятся к высокому уровню повторного использования кода, поэтому они постарались предусмотреть и закодировать все возможные исключительные ситуации. При реальном программировании создание собственных классов исключений позволяет разработчику выделить важные аспекты приложения и обратить внимание на детали разработки.

Генерация непроверяемых исключений

Если без каких-то данных приложение не сможет нормально функционировать, значит, есть очень серьезная ошибка, поэтому приложение разумнее остановить, устранить ошибки и перезапустить. Например, файл содержит данные конфигурации пула соединений с СУБД, а по заданному пути файл отсутствует. Обработать такую ошибку в коде приложения невозможно:

```
public void loadFile(String filename) {
    try {
        FileReader reader = new FileReader(filename);
        // more code...
    } catch (FileNotFoundException e) {
        logger.fatal("fatal error: config file not found: " + filename, e);
        throw new RuntimeException("fatal: config file not found: " + filename, e);
    }
}
```

Это единственная разумная ситуация, когда допустимо генерировать непроверяемое исключение. Одновременная запись лога также допустима, чтобы была возможность зафиксировать сообщение о деталях ошибки.

Для исключений, являющихся подклассами класса **RuntimeException** (*unchecked*) и используемых для отображения программных ошибок, при выполнении приложения в объявлении метода секция **throws** может отсутствовать, так как играет только информационную роль.

В секции **throws** можно использовать стандартные исключения только в случае, если этот метод **private**. Такой подход упрощает восприятие кода и не нарушает правила генерации только собственных исключений, так как метод **private** вызывается только методами того же класса и не виден для других классов.

Блок **finally**

Возможна ситуация, при которой нужно выполнить некоторые действия по завершении программы (закрыть поток, освободить соединение с базой данных) вне зависимости от того, произошло исключение или нет. В этом случае используется блок **finally**, который обязательно выполняется после или инструкции **try**, или **catch**. Например:

```
try {
    // code
} catch(OneException e) {
    // code // not required
} catch(TwoException e) {
    // code // not required
} finally {
    // executed after try or after catch */
}
```

Каждому разделу **try** должен соответствовать по крайней мере один раздел **catch** или блок **finally**. Блок **finally** часто используется для закрытия файлов и освобождения других ресурсов, захваченных для временного использования в начале выполнения метода. Код блока выполняется перед выходом из метода даже в том случае, если перед ним были выполнены такие инструкции, как **throws**, **return**, **break**, **continue**.

```
/* # 5 # выполнение блоков finally # ResourceAction.java */
```

```
package by.epam.learn.exception;
public class ResourceAction {
    public void doAction() {
        Resource resource = null;
        try {
            // init resources
            resource = new Resource(); // exception generation possible
            // using resources
            resource.execute(); // exception generation possible
            // resource.close(); // release of resources (incorrect)
        } catch (ResourceException e) {
            // code
        } finally {
            // release of resources (correct)
            if (resource != null) {
                resource.close();
            }
        }
        System.out.print("after finally");
    }
}
```

В методе **doAction()** при использовании ресурсов и генерации исключения осуществляется преждевременный выход из блока **try** с игнорированием всего оставшегося в нем кода, но до выхода из метода обязательно будет выполнен раздел **finally**. Освобождение ресурсов в этом случае произойдет корректно. В следующей главе будет рассмотрен новый способ закрытия ресурсов *autocloseable*.

Наследование и исключения

Создание сложных распределенных систем редко обходится без наследования и обработки исключений. Следует знать два правила для проверяемых исключений при наследовании:

- переопределяемый метод в подклассе не может содержать в инструкции **throws** исключений, не обрабатываемых в соответствующем методе суперкласса;
- конструктор подкласса должен включить в свою секцию **throws** все классы исключений или их суперклассы из секции **throws** конструктора суперкласса, к которому он обращается при создании объекта.

Первое правило имеет непосредственное отношение к расширяемости приложения. Пусть при добавлении в цепочку наследования нового класса его полиморфный метод включил в блок **throws** «новое» проверяемое исключение из другой цепочки наследования или исключение суперкласса текущего. Тогда методы логики приложения, принимающие объект нового класса в качестве параметра и вызывающие данный полиморфный метод, не готовы обрабатывать «новое» исключение, так как ранее в этом не было необходимости. Поэтому при попытке добавления «нового» *checked*-исключения в секцию **throws** полиморфного метода компилятор выдает сообщение об ошибке.

```
/* # 6 # полиморфизм и исключения # */
```

```
package by.epam.learn.exception;
public class Stone {
    public void accept(String data) throws ResourceException {
    }
}
package by.epam.learn.exception;
public class GreenStone extends Stone {
    @Override
    public void accept(String data) {
    }
}
package by.epam.learn.exception;
public class WhiteStone extends Stone {
    @Override
    public void accept(String data) throws ResourceException {
    }
}
```

```

package by.epam.learn.exception;
import java.io.FileWriter;
import java.io.IOException;
public class GreyStone extends Stone {
    @Override
    public void accept(String data) throws IOException {//compile error
        FileWriter writer = new FileWriter("data.txt");
    }
}

package by.epam.learn.exception;
public class StoneService {
    public void buildHouse(Stone stone) {
        try {
            stone.accept("some info");
        } catch (ResourceException e) {
            // handling of ResourceException and its subclasses
            System.err.print(e);
        }
    }
}

```

Если при объявлении метода суперкласса инструкция **throws** присутствует, то в подклассе эта инструкция может вообще отсутствовать или в ней могут быть объявлены только исключения, являющиеся подклассами исключения из секции **throws** метода суперкласса. Второе правило позволяет защитить программиста от возникновения неизвестных ему исключений при создании объекта.

```

/* # 7 # конструкторы и исключения # Resource.java # ConcreteResource.java #
NonConcreteResource.java */

```

```

import java.io.FileNotFoundException;
import java.io.IOException;
class Resource { // old class
    public Resource(String filename) throws FileNotFoundException {
        // more code
    }
}

class ConcreteResource extends Resource { // old class
    public ConcreteResource(String name) throws FileNotFoundException {
        super(name);
        // more code
    }
    public ConcreteResource() throws IOException {
        super("file.txt");
        // more code
    }
}

class NonConcreteResource extends Resource { // new class
    public NonConcreteResource(String name, int mode) { /* compile error */
        super(name);
        // more code
    }
    public NonConcreteResource(String name, int mode, String type)
        throws ParseException { /* compile error */
        super(name);
        // more code
    }
}

```

Если разрешить создание экземпляра в виде:

```
NonConcreteResource inCorrect = new NonConcreteResource("info", 1);
```

то конструктор суперкласса может сгенерировать исключение и никаких предварительных действий по его предотвращению принято не будет.

```
try {
    NonConcreteResource correct = new NonConcreteResource();
} catch(ParseException e) {
    // trace
}
```

В приведенном выше случае компилятор не разрешит создать конструктор подкласса, обращающийся к конструктору суперкласса без корректной инструкции **throws**. Если бы это было возможно, то при создании объекта подкласса класса **NonConcreteResource** не было бы никаких сообщений о возможности генерации исключения, и при возникновении исключительной ситуации ее источник было бы трудно идентифицировать.

Ошибки статической инициализации

Какое значение получит статическая переменная класса, если при ее инициализации будет сгенерировано исключение? Подразумевается, что сразу после инициализации переменной можно воспользоваться. А как это сделать, если инициализация не прошла? Действительно, такой переменной воспользоваться не получится, и все исключения, возникающие в процессе инициализации, получают обертку в виде **ExceptionInInitializerError**, которая останавливает JVM, предоставляя программисту возможность устранить первопричину ошибки непосредственной коррекцией кода или ресурсов приложения.

```
/* # 8 # генерация исключения в статическом поле # IdGenerator.java
```

```
package by.epam.learn.exception;
public class IdGenerator {
    final static int START_COUNTER;
    static {
        START_COUNTER = Integer.parseInt("Y-");
    }
}
```

и попытка запуска кода генерации исключения вида:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
    at by.bsu.exception.Runner.main(Runner.java:10)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.
NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) sun.reflect.
DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at java.lang.reflect.
Method.invoke(Method.java:483)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
Caused by: java.lang.NumberFormatException: For input string: "Y-"
    at ...
```

При попытке в статическом блоке преобразовать недопустимое значение, в число будет сгенерировано исключение **NumberFormatException**, но JVM его обернет в ошибку статической инициализации **ExceptionInInitializerError**. Константа **START_COUNTER** может быть инициализирована только один раз, и второй попытки в данном запуске приложения уже быть не может, и использовать такую переменную не получится, поэтому и генерируется в итоге ошибка, а не исключение. Для инициализации данного поля требуется перезапуск приложения. Та же ошибка будет выброшена и при прямой инициализации статической переменной:

```
static int startCounter = Integer.parseInt("Y-");
```

На практике такие ситуации встречаются достаточно часто, когда при старте приложения необходимо считать конфигурационную информацию, например, содержащую данные для соединения с СУБД:

```
/* # 9 # генерация исключения в статическом поле # DbConfigManager.java */
import java.util.ResourceBundle;
public class DbConfigManager {
    static ResourceBundle bundle = ResourceBundle.getBundle("resources.database");
    static String driverName = bundle.getString("driver.name");
}
```

Ошибка статической инициализации **ExceptionInInitializerError** возникнет как в случае отсутствия файла с указанным именем, так и в случае отсутствия в файле ключа **driver.name**. Только в этом случае основанием для генерации ошибки будет **java.util.MissingResourceException**.

Рекомендации по обработке исключений

Генерация исключения — процесс ресурсоемкий, и слишком частая генерация исключений оказывает влияние на быстродействие. Если генерация исключений необходима, то следует воспользоваться следующими рекомендациями.

- Не обрабатывать конкретное исключение или несколько исключений с использованием в блоке **catch** исключения более общего типа.

```
try {
    // more code here
} catch (Exception e) { // or Throwable
    System.err.println(e);
}
```

Вместо этого следует классифицировать исключения:

```
try {
    // more code here
} catch (NegativeArraySizeException e) {
    System.err.println("invalid buffer size: " + e);
} catch (NumberFormatException e) {
    System.err.println("invalid character in number: " + e);
}
```

- Не оставлять пустыми блоки **catch**. При таком перехвате исключения пользователь не узнает, что исключительная ситуация имела место, и не станет устранять ее причины.

```
try {
    // some code here
} catch (NumberFormatException e) {
}
```

- По возможности не использовать одинаковую обработку различных исключений.

```
try {
    // some code here
} catch (IOException e) {
    e.printStackTrace();
} catch (SAXException e) {
    e.printStackTrace();
}
```

Для замены можно воспользоваться конструкцией *multi-catch* из Java 7 или в каждый блок **catch** помещать уникальную обработку.

- Не создавать класс исключений, эквивалентный по смыслу уже существующему. Такие исключения способны запутать разработчика, например:

```
public class NullPointerException extends Exception {}
```

Исключение похоже на стандартное **NullPointerException** и может обмануть пользователя визуально. Или другой пример:

```
public class EnumNotPresentException extends Exception {}
```

вместо которого следует применить класс **EnumConstantNotPresentException**.

Прежде чем написать свое исключение, необходимо изучить документацию по стандартным исключениям, возможно, там найдется подходящее по смыслу. В обоих случаях непроверяемое исключение было подменено по смыслу на проверяемое, что недопустимо.

- Не создавать избыточное число классов собственных исключений. Прежде чем создавать новый класс исключений, следует подумать, что, возможно, ранее созданный в состоянии его обработать.
- Не использовать исключения в ситуациях, которые могут ввести в заблуждение:

```
public class Coin {
    private int weight;
    public void setWeight(int weight) throws IOException {
        if (weight <= 0) {
            throw new IOException();
        }
        this.weight = weight;
    }
}
```

- Не допускать, чтобы часть обработки ошибки присутствовала в блоке кода, генерирующем исключение:

```
public void setDeduce(double deduce) throws TaxException {
    if (deduce < 0) {
        this.deduce = 0; // unnecessary
        recalculateAmount(); // unnecessary
        System.err.print(DEDUCE_NEGATIVE); // unnecessary
        throw new TaxException("VAT deduce < 0");
    }
    this.deduce = deduce;
    recalculateAmount();
}
```

- Не допускать одновременной записи лога и генерации исключения, кроме непроверяемых исключений, в этом случае запись лога обязательна:

```
public void setDeduce(double deduce) throws TaxException {
    if (deduce < 0) {
        logger.error("VAT deduce is negative");
        throw new TaxException("VAT deduce is negative");
    }
    this.deduce = deduce;
    recalculateAmount();
}
```

- Никогда самостоятельно не генерировать **NullPointerException**, избегать случаев, когда такая генерация возможна в принципе. Проверка значения ссылки на **null** позволяет обойтись без генерации исключения. Если по логике приложения необходимо генерировать исключение, следует использовать, например, **IllegalArgumentException** с соответствующей информацией об ошибке или собственное исключение.
- Никогда самостоятельно не перехватывать **NullPointerException** в блоке **catch**. Избежать этого можно простой проверкой значения ссылки на **null**. Желательно обходиться без перехватов и других непроверяемых исключений.
- Не следует в общем случае в секцию **throws** помещать *unchecked*-исключения.
- В любом случае, если есть возможность не генерировать исключение, следует ею воспользоваться.
- Не рекомендуется вкладывать блоки **try-catch** друг в друга из-за ухудшения читаемости кода.
- При создании собственных исключений следует проводить наследование от класса **Exception** либо от другого проверяемого класса исключений, а не от **RuntimeException**.
- Никогда не генерировать исключения в инструкции **finally**:

```
try {
    // possible throw exception
} finally {
    if (boolean_expression) {
        throw new CustomException();
    }
}
```

При такой генерации исключения никто в приложении не узнает об исключении и, соответственно, не сможет обработать исключение, ранее сгенерированное в блоке **try**, в случае, если оно не было обработано в блоке **catch**. В связи со сказанным никогда не следует использовать в блоке **finally** операторы **return**, **break**, **continue**.

- Во избежание дублирования логов, не следует писать логи из конструкторов классов-исключений:

```
public class CoinTechnicalException extends Exception {
    static Logger logger = LogManager.getLogger();
    public CoinTechnicalException() {
        logger.error(this.printStackTrace());
    }
    public CoinTechnicalException(String message, Throwable cause) {
        super(message, cause);
        logger.error(message, cause);
    }
}
```

Отладочный механизм *assertion*

Борьба за качество программ ведется различными способами. На этапе отладки найти неявные ошибки в функционировании приложения бывает довольно сложно. Механизм *assertion* позволяет проверять предположения о значениях данных в методах приложения. Если механизм отладки включен, то при обнаружении некорректных данных генерируется **AssertionError**, с указанием места их обнаружения. Например, в методе, использующем линейные размеры какой-либо сущности, информация о размере извлекается из файла, и в результате может быть получено отрицательное значение. Далее неверные данные влияют на результат вычисления метода и т.д. Определять такие ситуации позволяет механизм проверочных утверждений (*assertion*). При помощи этого механизма можно сформулировать требования к входным, выходным и промежуточным данным непубличных методов классов в виде некоторых логических условий.

Стандартная попытка обработать ситуацию появления отрицательного размера может выглядеть следующим образом:

```
int size = pool.getPoolSize();
if (size > 0) {
    // more code
} else {
    // fatal error
}
```

Механизм *assertion* позволяет создать код, который будет генерировать **AssertionError** на этапе его отладки в результате проверки постусловия или промежуточных данных в виде:

```
int size = pool.getPoolSize();
assert (size > 0) : "incorrect PoolSize= " + size;
// more code
```

Правописание инструкции **assert**:

```
assert bool_exp : expression;
assert bool_exp;
```

Выражение **bool_exp** может принимать только значение типов **boolean** или **Boolean**, а **expression** — любое значение, которое может быть преобразовано к строке. Если логическое выражение получает значение **false**, то генерируется исключение **AssertionError** и выполнение программы прекращается с выводом на консоль значения выражения **expression** (если оно задано).

Механизм *assertion* хорошо подходит для проверки инвариантов, например, перечислений:

```
enum Mono { WHITE, BLACK }
String str = "WHITE"; // "GRAY"
Mono mono = Mono.valueOf(str);
// more code
switch (mono) {
    case WHITE : // more code
        break;
    case BLACK : // more code
        break
    default :
        assert false : "Colored!";
}
```

Создатели языка не рекомендуют использовать *assertion* при проверке параметров **public**-методов. В таких ситуациях лучше рассматривать возможность генерации исключения одного из типов: **IllegalArgumentException** или собственное исключение. Нет также особого смысла в механизме *assertion* при проверке пограничных значений переменных, поскольку исключительные ситуации генерируются в этом случае без посторонней помощи.

Механизм *assertion* можно включать для отдельных классов или пакетов при запуске виртуальной машины в виде:

```
java -enableassertions RunnerMain
```

или

```
java -ea RunnerMain
```

Для выключения *assertion* применяется **-disableassertions** или **-da**.

Вопросы к главе 9

1. Что для программы является исключительной ситуацией? Какие существуют способы обработки ошибок в программах?
2. Что такое исключение для Java-программы? Что значит «программа генерировала\выбросила исключение»? Привести пример, когда исключения генерируются виртуальной машиной (автоматически) и когда необходимо их генерировать вручную.
3. Привести иерархию классов-исключений, делящую исключения на проверяемые и непроверяемые. В чем особенности проверяемых и непроверяемых исключений?
4. Объяснить работу оператора **try-catch-finally**. Когда данный оператор следует использовать? Сколько блоков **catch** может соответствовать одному блоку **try**?
5. Можно ли вкладывать блоки **try** друг в друга, можно ли вложить блок **try** в **catch** или **finally**? Как происходит обработка исключений, выброшенных внутренним блоком **try**, если среди его блоков **catch** нет подходящего?
6. Что называют стеком операторов **try**? Как работает блок **try** с ресурсами?
7. Указать правило расположения блоков **catch** в зависимости от типов перехватываемых исключений. Может ли перехваченное исключение быть сгенерировано снова, и, если да, то как и кто в этом случае будет обрабатывать повторно сгенерированное исключение? Может ли блок **catch** выбрасывать иные исключения, и если да, то привести пример, когда это может быть необходимо.
8. Когда происходит вызов блока **finally**? Существуют ли ситуации, когда блок **finally** не будет вызван? Может ли блок **finally** выбрасывать исключения? Может ли блок **finally** выполниться дважды?
9. Как генерировать исключение вручную? Объекты каких классов могут быть генерированы в качестве исключений? Можно ли генерировать два исключения одновременно?
10. Объяснить, как работают операторы **throw** и **throws**. В чем их отличия?
11. Объяснить правила реализации секции **throws** при переопределении метода и при описании конструкторов производного класса.
12. Как ведет себя блок **throws** при работе с проверяемыми и непроверяемыми исключениями?
13. Каков будет результат создания объекта, если конструктор при работе генерирует исключительную ситуацию?
14. Нужно ли генерировать исключения, входящие в Java SE? Как создать собственные классы исключений?

Задания к главе 9

Вариант А

В символьном файле находится информация об N числах с плавающей запятой с указанием локали каждого числа отдельно. Прочитать информацию из файла. Проверить на корректность, то есть являются ли числа числами. Преобразовать к числовым значениям и вычислить сумму и среднее значение прочитанных чисел.

Создать собственный класс исключения. Предусмотреть обработку исключений, возникающих при нехватке памяти, отсутствии самого файла по заданному адресу, отсутствии или некорректности требуемой записи в файле, недопустимым значением числа (выходящим за пределы максимально допустимых значений) и т.д.

Тестовые задания к главе 9

Вопрос 9.1.

Дан код:

```
class Quest {
    static void method() throws ArithmeticException {
        int i = 7 / 0;

        try {
            double d = 77.0;
            d /= 0.0;
        } catch (ArithmeticException e) {
            System.out.print("E1");
        } finally {
            System.out.print("Finally ");
        }
    }
}

public static void main(String[] args) {
    try {
        method();
    } catch (ArithmeticException e) {
        System.out.print("E0");
    }
}
```

Каким будет результат компиляции и запуска? (выбрать один)

- a) EOFinally
- b) E0E1
- c) E1Finally
- d) E0
- e) FinallyE0
- f) runtime error

Вопрос 9.2.

Дан код:

```
class ColorException extends Exception {}
class WhiteException extends ColorException {}
abstract class Color {
    abstract void method() throws ColorException;
}
class White extends Color {
    void method() throws WhiteException {
        throw new WhiteException();
    }
    public static void main (String[] args) {
        White white = new White();
        int a, b, c;
        a = b = c = 0;
        try {
            white.method();
            a++;
        } catch (WhiteException e) {
            b++;
        } finally {
            c++;
        }
        System.out.print(a + " " + b + " " + c);
    }
}
```

Какой результат выведется на консоль при попытке компиляции и запуска этого кода? (выбрать один)

- a) 1 1 1
- b) 0 0 1
- c) 0 1 1
- d) 1 0 1
- e) compilation fails

Вопрос 9.3.

Дана иерархия классов:

```
class A{
    A() throws IOException{}
}
class B extends A {}
```

Как следует объявить конструктор класса *B*, чтобы код компилировался без ошибок? (выбрать два)

- a) B() throws IOException{ }
- b) B() throws FileNotFoundException{ }
- c) B(){ }
- d) B() throws IllegalArgumentException{ }
- e) B() throws Exception { }

Вопрос 9.4.

Дан код:

```
class A{  
    void f() throws FileNotFoundException {}  
}  
class B extends A{}
```

Каким образом можно переопределить метод *m()* в классе *B*, не вызвав при этом ошибку компиляции? (выбрать три)

- a) `void m() throws Exception {}`
- b) `void m() throws IOException {}`
- c) `void m(){}`
- d) `void m() throws FileNotFoundException {}`
- e) `void m() throws FileNotFoundException, IOException {}`
- f) `void m() throws ExceptionInInitializerError {}`

ПОТОКИ ВВОДА/ВЫВОДА

Вопрос. Я скачал из интернета файл.
Теперь мне он больше не нужен.
Как закачать его обратно?
Ответ. Вот из-за таких, как ты,
скоро в интернете
все файлы кончатся.

Цитата из сети

Потоки ввода/вывода используются для передачи данных в файловые потоки, на консоль или на сетевые соединения. Потоки представляют собой объекты соответствующих классов. Пакеты ввода/вывода **java.io**, **java.nio** предоставляют пользователю большое число классов и методов и постоянно обновляются.

Байтовые и символьные потоки ввода/вывода

При разработке приложения регулярно возникает необходимость ввода информации из какого-либо источника и хранения результатов. Действия по чтению/записи информации представляют собой стандартный вид деятельности, связанный с передачей и извлечением последовательности байтов из потоков.

Все потоки ввода последовательности байтов являются подклассами абстрактного класса **InputStream**, потоки вывода — подклассами абстрактного класса **OutputStream**. При работе с файлами используются подклассы этих классов, соответственно, **FileInputStream** и **FileOutputStream**, конструкторы которых открывают поток и связывают его с соответствующим физическим файлом. Существуют классы-потоки для ввода массивов байтов, строк, объектов, а также для выбора данных из файлов и сетевых соединений.

Для чтения байта и массива байтов используются реализации абстрактных методов **int read()** и **int read(byte[] b)** класса **InputStream**. Метод **int read()** возвращает **-1**, если достигнут конец потока данных, поэтому возвращаемое значение имеет тип **int**, а не **byte**. При взаимодействии с информационными потоками возможны различные исключительные ситуации, поэтому обработка исключений вида **try-catch** при использовании методов чтения и записи является обязательной.

В классе **FileInputStream** метод **read()** читает один байт из файла, а поток **System.in** как объект подкласса **InputStream** позволяет вводить байт с консоли.

Реализация абстрактного метода **write(int b)** класса **OutputStream** записывает одно значение в поток вывода. Оба эти метода блокируют поток до тех пор, пока байт не будет записан или прочитан. После окончания чтения или записи в поток его всегда следует закрывать с помощью метода **close()** для того, чтобы освободить ресурсы приложения. Класс **FileOutputStream** используется для вывода одного или нескольких байт информации в файл.

Поток ввода связывается с одним из источников данных, в качестве которых могут быть использованы массив байтов, строка, файл, «pipe»-канал, сетевые соединения и др. Набор классов для взаимодействия с перечисленными источниками приведен на рисунках 10.1 и 10.2.

Класс **BufferedInputStream** присоединяет к потоку буфер для упрощения и ускорения следующего доступа.

Для вывода данных в поток используются следующие классы.

Абстрактный класс **FilterOutputStream** используется как шаблон для настройки производных классов. Класс **BufferedOutputStream** присоединяет буфер к потоку для ускорения вывода и ограничения доступа к внешним устройствам.

Начиная с версии 1.2, пакет **java.io** подвергся значительным изменениям. Появились новые классы, которые производят скоростную обработку потоков, хотя и не полностью перекрывают возможности классов предыдущей версии.

Для обработки символьных потоков в формате **Unicode** применяется отдельная иерархия подклассов абстрактных классов **Reader** и **Writer**, которые почти полностью повторяют функциональность байтовых потоков, но являются более актуальными при передаче текстовой информации.

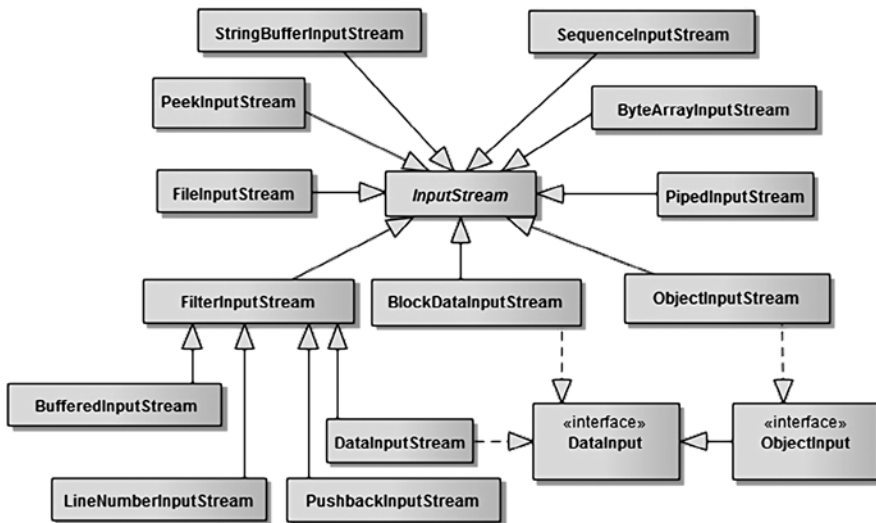


Рис. 10.1. Иерархия классов байтовых потоков ввода

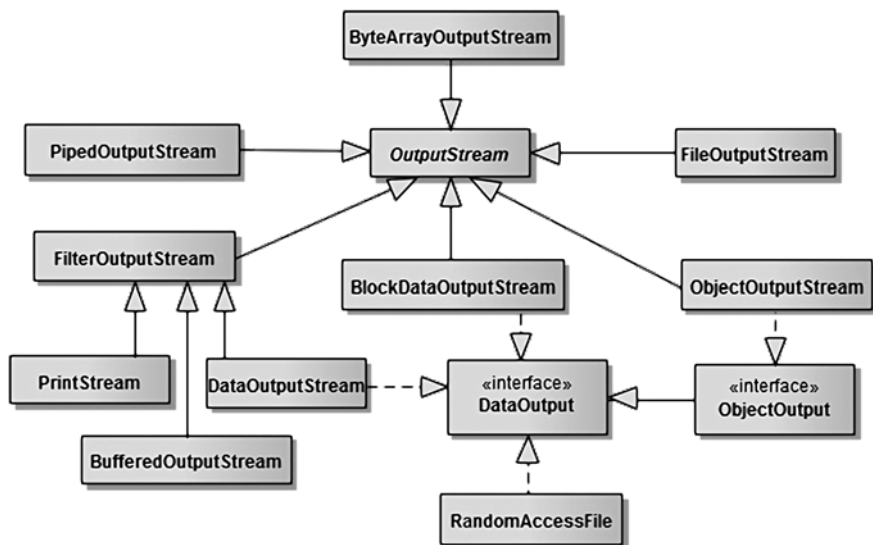


Рис. 10.2. Иерархия классов байтовых потоков вывода

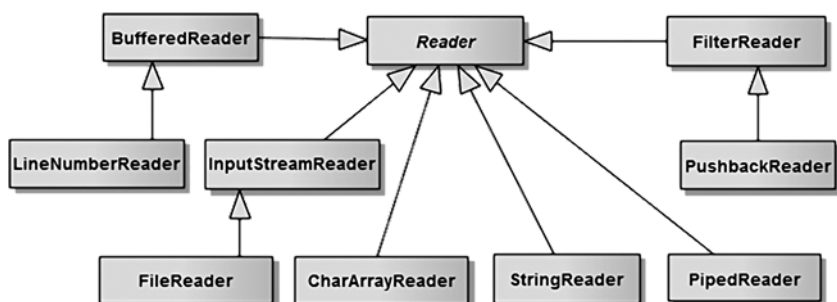


Рис. 10.3. Иерархия символьных потоков ввода

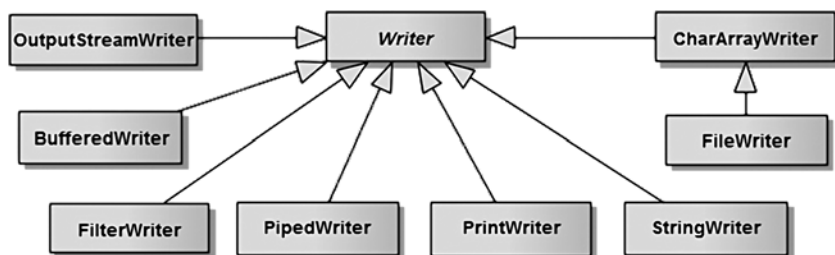


Рис. 10.4. Иерархия символьных потоков вывода

Например, аналогом класса **FileInputStream** является класс **FileReader**. Такой широкий выбор потоков позволяет выбрать наилучший способ записи в каждом конкретном случае.

В примерах по возможности используются способы инициализации для различных семейств потоков ввода/вывода.

```
/* # 1 # чтение одного символа (байта) и массива из потока ввода # InputMain.java */
```

```
package by.epam.learn.io;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Arrays;
public class InputMain {
    public static void main(String[] args) {
        FileInputStream input = null;
        try {
            input = new FileInputStream("data/info.txt");
            int code = input.read();
            System.out.println(code + " char = " + (char)code);
            byte[] arr = new byte[16];
            int numberBytes = input.read(arr);
            System.out.println("numberBytes = " + numberBytes);
            System.out.println(Arrays.toString(arr));
            // input.close(); // wrong
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if(input != null) {
                    input.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Конструктор **FileInputStream(file)** открывает поток **input** и связывает его с файлом, где директория **data** в корне проекта должна существовать.

Если файла по указанному пути не существует, то при попытке инициализации потока с несуществующим файлом будет сгенерировано исключение:

Ошибка файла: java.io.FileNotFoundException: data\info.txt (The system cannot find the file specified)

Если файл существует, то информация из него будет считана по одному символу; и результаты чтения, и количество прочитанных символов будут выведены на консоль.

Для закрытия потока используется метод **close()**. Закрытие потока должно произойти при любом исходе чтения: удачном или с генерацией исключения. Гарантировать закрытие потока может только помещение метода **close()** в блок **finally**. При чтении из потока можно пропустить **n** байт с помощью метода **long skip(long n)**.

Закрытие потока ввода/вывода в блоке **finally** принято как негласное правило и является признаком хорошего кода. Однако данная конструкция выглядит достаточно громоздко. Начиная с Java 7 возможно автоматическое закрытие потоков ввода/вывода без явного вызова метода **close()** и блока **finally**:

```
try (FileReader is = new FileReader(new File("data\\info.txt"))) {
    // code
} catch (IOException e) {
    System.err.println("file error: " + e);
}
```

С этой версии в список интерфейсов, реализуемых практически всеми классами потоков, добавлен интерфейс **AutoCloseable**. Метод **close()** вызывается неявно для всех потоков, открытых в инструкции

```
try(iostream1; iostream2;...; iostreamN)
```

Например:

```
try (
    Writer writer = new FileWriter(fileIn);
    OutputStream out = new FileOutputStream(fileOut)
)
```

Для вывода символа (байта) или массива символов (байтов) в поток используются потоки вывода — объекты подкласса **FileWriter** суперкласса **Writer** или подкласса **FileOutputStream** суперкласса **OutputStream**. В следующем примере для вывода в связанный с файлом поток используется метод **write()**.

```
// # 2 # вывод массива в поток в виде символов и байт # OutMain.java
```

```
package by.epam.learn.io;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class OutMain {
    public static void main(String[] args) {
        try (FileOutputStream output = new FileOutputStream("data/out.txt", true)) {
            output.write(48);
            byte[] value = {65, 67, 100};
            output.write(value);
        } catch (FileNotFoundException e) {
```

```

        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

В результате будет получен файл с набором данных. Если файл **out.txt** в директории **data** не существует, то он будет создан. Директория **data** обязательно должна существовать. При ее отсутствии попытка записи в файл сгенерирует исключительную ситуацию.

File, Path и Files

Для работы с физическими файлами и каталогами (директориями), расположенными на внешних носителях, в приложениях Java используются классы из пакетов **java.io** и **java.nio**.

В Java 7 были добавлены класс **java.nio.file.Files** и интерфейс **java.nio.file.Path**, дублирующие и существенно расширяющие возможности класса **java.io.File**, возможности которого будут рассмотрены ниже.

Интерфейс **Path** представляет более совершенный аналог класса **File**, а класс **Files**, по сути, утилитный класс, содержащий только статические методы для доступа к файлам, директориям, их свойствам и их содержимому.

Получить объект **Path** можно как из объекта **File**:

```

File file = new File("data/info.txt");
Path path = file.toPath();

```

так и прямой инициализацией:

```

Path path1 = Paths.get("data/info.txt");

```

или

```

Path path2 = FileSystems.getDefault().getPath("data/info.txt");

```

Доступ и управление файловой системой, доступной для текущей версии JVM, осуществляют классы **java.nio.file.FileSystem** и **java.nio.file.FileSystems**. Класс **FileSystems** определяет набор статических методов для получения и создания файловых систем. Вызов метода **FileSystems.getDefault()** предоставляет доступ к текущей файловой системе.

Найти все файлы с расширением **.java** из заданной директории **src** с помощью метода **Files.find(Path start, int maxDepth, BiPredicate<Path, BasicFileAttributes> matcher, FileVisitOption... options)** позволит следующий фрагмент кода:

```

Path path = FileSystems.getDefault().getPath("src");
if (Files.exists(path) && Files.isDirectory(path)) {
    int maxDepth = 5;

```

```
try (Stream<Path> streamDir = Files.find(path, maxDepth,
    (p, a) -> String.valueOf(p).endsWith(".java"))) {
    long counter = streamDir
        .map(String::valueOf)
        .peek(System.out::println)
        .count();
    System.out.println("found: " + counter);
} catch (IOException e) {
    e.printStackTrace();
}
}
```

В результате будут выведены на консоль все файлы с расширением **.java** из указанной директории и всех поддиректорий с глубиной погружения 5, а также подсчитано их общее число.

Метод **Files.walk(Path start, int maxDepth)** позволяет получить список всех файлов и подкаталогов указанного каталога:

```
Path start = Paths.get("src");
int maxDepth = 5;
try (Stream<Path> pathStream = Files.walk(start, maxDepth)) {
    pathStream.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

При работе с файлами и директориями\каталогами лучше воспользоваться возможностями пакета **java.nio.file**, так как его классы позволяют учитывать очень широкий набор свойств объектов.

Класс **java.io.File** обладает достаточно ограниченной функциональностью. Класс **File** служит для хранения и обработки в качестве объектов каталогов и имен файлов. Этот класс не содержит методы для работы с содержимым файла, но позволяет манипулировать такими свойствами файла, как право доступа, дата и время создания, путь в иерархии каталогов, создание, удаление файла, изменение его имени и каталога и т.д.

Объект класса **File** создается одним из способов:

```
File file = new File("\\com\\file.txt");
File dir = new File("c:/jdk/src/java/io");
File file1 = new File(dir, "File.java");
File file2 = new File("c:\\com", "file.txt");
```

В первом случае создается объект, соответствующий файлу, во втором — подкаталогу. Третий и четвертый случаи практически идентичны. Для создания объекта указывается каталог и имя файла.

При создании объекта класса **File** любым из конструкторов компилятор не выполняет проверку на существование физического файла с заданным путем.

Существует разница между разделителями, употребляющимися при записи пути к файлу: для системы *Unix* — «/», а для *Windows* — «\». Для случаев,

когда неизвестно, в какой системе будет выполняться код, предусмотрены специальные поля в классе **File**:

```
public static final String separator;
public static final char separatorChar;
```

С помощью этих полей можно задать путь, универсальный в любой системе:

```
File file = new File(File.separator + "com" + File.separator + "data.txt");
```

Также предусмотрен еще один тип разделителей для директорий:

```
public static final String pathSeparator;
public static final char pathSeparatorChar;
```

К примеру, для ОС *Unix* значение **pathSeparator** принимает значение «:», а для ОС MS-DOS — «;».

Некоторые возможности класса **File** представлены в следующем примере:

```
/* # 3 # работа с файловой системой: FileMain.java */
```

```
package by.epam.learn.io;
import java.io.*;
import java.time.Instant;
public class FileMain {
    public static void main(String[] args) {
        File file = new File("data" + File.separator + "info.txt");
        if (file.exists() && file.isFile()) {
            System.out.println("Path " + file.getPath());
            System.out.println("Absolute Path " + file.getAbsolutePath());
            System.out.println("Size " + file.length());
            System.out.println("Dir " + file.getParent());
            file.delete();
            try {
                file.createNewFile();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        File dir = new File("data");
        if (dir.exists() && dir.isDirectory()) {
            for (File current : dir.listFiles()) {
                long millis = current.lastModified();
                Instant date = Instant.ofEpochMilli(millis);
                System.out.println(current.getPath() + "\t" + current.length() + "\t" + date);
            }
            File root = File.listRoots()[0];
            System.out.printf("\n%s %d from %d free bytes", root.getPath(), root.getUsableSpace(),
                                root.getTotalSpace());
        }
    }
}
```

В результате файл **data\info.txt** будет очищен, а на консоль выведено:

Path data\info.txt

Absolute Path C:\Users\Ihar_Blinou\IdeaProjects\Learn1\data\info.txt

Size 0

Dir data

data\info.txt 0 2019-03-14T17:27:39.978

data\out.dat 90 2019-03-12T16:50:37.822

data\out.txt 21 2019-03-12T15:38:37.312

data\res.txt 105 2019-03-13T15:49:19.761

C:\ 50,595,426,304 from 255,820,034,048 free bytes

У каталога как объекта класса **File** есть дополнительное свойство — просмотр списка имен файлов с помощью методов **list()**, **listFiles()**, **listRoots()**.

Чтение из потока

В отличие от Java 1.1 в языке Java 1.2 для ввода используется не байтовый, а символьный поток. В этой ситуации для ввода используется подкласс **BufferedReader** абстрактного класса **Reader** и методы **read()** и **readLine()** для чтения символа и строки соответственно. Этот поток для организации чтения из файла лучше всего инициализировать объектом класса **FileReader** в виде:

```
new BufferedReader(new FileReader(new File("data\\res.txt")));
```

Возможности по чтению информации из файла были существенно расширены в Java 7 и Java 8 с появлением класса **Files** и методов чтения в **Stream**.

Чтение из файла можно произвести следующим образом:

```
// # 4 # чтение строк из файла # ReadStringMain.java
```

```
package by.epam.learn.io;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class ReadStringMain {
    public static void main(String[] args) {
        String stringLines = "";
        try (BufferedReader reader = new BufferedReader(new FileReader("data\\res.txt"))){
            String tmp;
            while ((tmp = reader.readLine()) != null) { //java 2
                stringLines += tmp;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

        System.out.println(stringLines);
    }
}

```

Чтение производится по одной строке за каждую итерацию цикла. Строки в цикле суммируются. Когда метод **readLine()** встречает символ конца файла, то возвращает **null**.

В Java 7 в классе **Files** появился метод **readAllLines()**, считывающий сразу весь файл в список строк.

```
// # 5 # чтение строк из файла как в java 7 # ReadStringMain1.java
```

```

package by.epam.learn.io;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
public class ReadStringMain1 {
    public static void main(String[] args) {
        String dirName = "data";
        String filename = "res.txt";
        Path path = FileSystems.getDefault().getPath(dirName, filename);
        try { // java7
            List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);
            System.out.println(lines);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Появление Java 8 добавило новые методы по чтению строк файла в объект **Stream**, с возможностью дальнейшего преобразования в строку или коллекцию.

В класс **BufferedReader** добавлен метод **lines()** чтения файла в **Stream**:

```
// # 6 # чтение строк из файла как в java 8 # ReadStringMain2.java
```

```

package by.epam.learn.io;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class ReadStringMain2 {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("data\\res.txt")));
        Stream<String> stream = reader.lines()) { // java 8
            String lines = stream.collect(Collectors.joining());

```

```

        System.out.println(lines);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Результаты чтения были преобразованы в строку.

В класс **Files** добавлен статический метод **newBufferedReader(Path path)**, создающий объект **BufferedReader**.

// # 7 # чтение строк из файла как в java 8 # ReadStringMain3.java

```

package by.epam.learn.io;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;
public class ReadStringMain3 {
    public static void main(String[] args) {
        Path path = Paths.get("data\\res.txt");
        try (Stream<String> stream = Files.newBufferedReader(path).lines()) {
            stream.forEach(System.out::println);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

В тот же класс **Files** добавлен статический метод **lines()**, возвращающий **Stream** из строк.

// # 8 # чтение строк из файла как в java 8 # ReadStringMain4.java

```

package by.epam.learn.io;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class ReadStringMain4 {
    public static void main(String[] args) {
        Path path = Paths.get("data\\res.txt");
        try (Stream<String> streamLines = Files.lines(path)) {
            String result = streamLines.collect(Collectors.joining());
            System.out.println(result);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Предопределенные стандартные потоки

Система вывода языка Java содержит стандартные потоки вывода значений простых типов, объектов и ошибок в удобочитаемом виде. Класс **System** пакета **java.lang** содержит поля **out**, **err** — ссылки на объекты байтового потока **PrintStream**, объявленные со спецификаторами **public static**, являющиеся стандартными потоками ввода, вывода и вывода ошибок соответственно.

Эти потоки связаны с консолью, но могут быть переназначены на другое устройство.

Для назначения вывода текстовой информации в произвольный поток следует использовать символьный поток **PrintWriter**, являющийся подклассом абстрактного класса **Writer**. Кроме того, невозможно оборачивать символьный поток байтовым, в то время как обратное возможно.

Для вывода информации в файл или в любой другой поток стоит организовать следующую последовательность инициализации потоков с помощью класса **PrintWriter**:

```
new PrintWriter(new BufferedWriter(new FileWriter(new File("text\\data.txt"))));
```

В итоге класс **PrintWriter** выступает классом-оберткой для класса **BufferedWriter**, так же, как и класс **BufferedReader** для **FileReader**. По окончании работы с потоками закрывать следует только самый последний класс. В данном случае — **PrintWriter**. Все остальные, в него обернутые, будут закрыты автоматически.

В приведенном ниже примере демонстрируется вывод в файл строк и чисел с плавающей точкой.

```
// # 9 # буферизованный вывод в файл # PrintMain.java
```

```
package by.epam.learn.io;
import java.io.*;
public class PrintMain {
    public static void main(String[] args) {
        double[] values = {1.10, 1.2};
        try(PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter("data\\r.txt")))){
            for (double value: values) {
                writer.printf("Java %.2g%n", value);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Для вывода данных в файл в текстовом формате использовался фильтрованный поток вывода **PrintWriter** и метод **printf()**. После соединения этого потока

с дисковым файлом посредством символьного потока **BufferedWriter** и удобного средства записи в файл **FileWriter**, становится возможной запись текстовой информации с помощью обычных методов **println()**, **print()**, **printf()**, **format()**, **write()**, **append()**, что и показано в следующем фрагменте кода:

```
double[] values = {14.10, 17};
try(PrintStream stream = new PrintStream(new FileOutputStream("data\\res.txt"))){
    for (double value: values) {
        stream.printf("Java %.2g%n", value);
        System.setOut(stream);
        System.out.printf("%.2g%n", value);
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Сериализация объектов

Кроме данных базовых типов, в поток можно отправлять объекты классов целиком в байтовом представлении для передачи клиентскому приложению, а также для хранения в файле или базе данных.

Процесс преобразования объектов в потоки байтов для хранения называется сериализацией. Процесс извлечения объекта из потока байтов называется десериализацией. Существует два способа сделать объект сериализуемым.

Для того, чтобы объекты класса могли быть подвергнуты процессу сериализации, этот класс должен имплементировать интерфейс **java.io.Serializable**. Все подклассы такого класса также будут сериализованы. Многие стандартные классы реализуют этот интерфейс. Этот процесс заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора **static** или **transient**. Спецификаторы **transient** и **static** означают, что поля, помеченные ими, не могут быть предметом сериализации, но существует различие в десериализации. Так, поле со спецификатором **transient** после десериализации получает значение по умолчанию, соответствующее его типу (объектный тип всегда инициализируется по умолчанию значением **null**), а поле со спецификатором **static** получает значение по умолчанию в случае отсутствия в области видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта. На поля, помеченные как **final**, ключевое слово **transient** не действует.

Интерфейс **Serializable** не имеет методов, которые необходимо реализовать, поэтому его использование ограничивается упоминанием при объявлении класса. Все действия в дальнейшем производятся по умолчанию. Для записи объектов в поток необходимо использовать класс **ObjectOutputStream**. После этого достаточно вызвать метод **writeObject(Object ob)** этого класса для сериализации объекта **ob** и пересылки его в выходной поток данных. Для чтения используются, соответственно, класс **ObjectInputStream** и его метод **readObject()**,

возвращающий ссылку на класс **Object**. После этого следует преобразовать полученный объект к нужному типу.

Необходимо знать, что при использовании **Serializable** десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор сериализуемого класса при этом не вызывается, но вызываются все конструкторы суперклассов в заданной последовательности до класса, имплементирующего **Serializable**.

```
/* # 10 # сериализуемый класс # Student.java */
```

```
package by.epam.learn.io;
import java.io.Serializable;
import java.util.StringJoiner;
public class Student implements Serializable {
    static String faculty = "ММФ";
    private String name;
    private int id;
    private transient String password;
    private static final long serialVersionUID = 2L;
    public Student(String name, int id, String password) {
        this.name = name;
        this.id = id;
        this.password = password;
    }
    @Override
    public String toString() {
        return new StringJoiner(", ", Student.class.getSimpleName() + "[", "]")
            .add("name='" + name + "'").add("id=" + id)
            .add("password='" + password + "'").toString();
    }
}
```

```
/* # 11 # запись сериализованного объекта в файл # SerializationMain.java */
```

```
package by.epam.learn.io;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
public class SerializationMain {
    public static void main(String[] args) {
        try(ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream("data/o.dat"))) {
            Student student = new Student("Janka", 555777, "VKL_1410");
            System.out.println(student);
            output.writeObject(student);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Объект **student** будет сериализован в файл, а в консоль выведено:

Student[name='Janka', id=555777, password='VKL_1410']

```

/* # 12 # десериализация объекта из файла # DeSerializationMain.java */

package by.epam.learn.io;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
public class DeSerializationMain {
    public static void main(String[] args) {
        Student.faculty = "GEO";
        try(ObjectInputStream input = new ObjectInputStream(new FileInputStream("data/out.dat"))) {
            Student student = (Student)input.readObject();
            System.out.println(student);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

В результате выполнения данного кода в консоль будет выведено:

Student[name='Janka', id=555777, password='null']

В итоге поля **name** и **id** нового объекта **student** сохранили значения, которые им были присвоены до записи в файл. Поле **password** со спецификатором **transient** получило значение по умолчанию, соответствующее типу (объектный тип всегда инициализируется по умолчанию значением **null**). Поле **faculty**, помеченное как статическое, получает то значение, которое имеет это поле на текущий момент, то есть при создании объекта **student** поле получило при инициализации значение **MMF**, а затем значение статического поля было изменено на **GEO**. Если же объекта данного типа не было в области видимости, а значение статического поля не было задано, то статическое поле также получает значение по умолчанию.

Если полем класса является ссылка на другой тип, то необходимо, чтобы агрегированный тип также реализовывал интерфейс **Serializable**, иначе при попытке сериализации объекта такого класса будет сгенерировано исключение **NotSerializableException**.

```

/* # 13 # класс, сериализация которого невозможна # Student.java */

public class Student implements Serializable {
    static String faculty;
    private int id;
    private String name;
    private transient String password;
}

```

```
private Address addr = new Address(); // non-serializable field
private static final long serialVersionUID = 2L;
// more code
}
```

Если класс **Address** не имплементирует интерфейс **Serializable**, а объявлен как:

```
public class Address {
}
```

то при таком объявлении класса **Address** сериализация объекта класса **Student** невозможна.

При сериализации объекта класса, реализующего интерфейс **Serializable**, учитывается порядок объявления полей в классе. Поэтому при изменении порядка, имен и типов полей или добавлении новых полей в класс структура информации, содержащейся в сериализованном объекте, будет серьезно отличаться от новой структуры класса. Поэтому десериализация может пройти некорректно. Этим обусловлена необходимость добавления программистом в каждый класс, реализующий интерфейс **Serializable**, поля **private static final long serialVersionUID** на стадии разработки класса. Это поле содержит уникальный идентификатор версии класса. Оно задается программистом или вычисляется по содержимому класса — полям, их порядку объявления, методам, их порядку объявления. Для этого применяются специальные программы-генераторы UID.

Это поле записывается в поток при сериализации класса. Это тот случай, когда **static**-поле сериализуется.

При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают, инициируется исключение **java.io.InvalidClassException**. Соответственно, при любом изменении в первую очередь полей класса значение поля **serialVersionUID** должно быть изменено программистом или генератором.

Если набор полей класса и их порядок жестко определены, методы класса могут меняться. В этом случае сериализации и десериализации ничего не угрожает.

Вместо реализации интерфейса **Serializable** можно реализовать **Externalizable**, который содержит два метода: **writeExternal(ObjectOutput out)** и **readExternal(ObjectInput in)**.

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию о состоянии экземпляра должен сам класс. Для этого в нем должны быть переопределены методы **writeExternal()** и **readExternal()** интерфейса **Externalizable**. Эти методы должны обеспечить сохранение состояния, описываемого полями самого класса и его суперкласса.

При восстановлении **Externalizable**-объекта экземпляр создается путем вызова конструктора без аргументов, после чего вызывается метод **readExternal()**, поэтому необходимо проследить, чтобы в классе был конструктор по умолчанию. Для сохранения состояния вызываются методы **ObjectOutput**, с помощью

которых можно записать как примитивные, так и объектные значения. Для корректной работы в соответствующем методе **readExternal()** эти значения должны быть считаны в том же порядке.

Для чтения и записи в поток значений отдельных полей объекта используются методы внутренних классов: **ObjectInputStream.GetField** и **ObjectOutputStream.PutField**.

Сериализация в XML

С понятием сериализации связан еще один важный термин, называемый **JavaBeans**.

Какие требования должен выполнять класс, чтобы называться **JavaBean**-классом?

- объявление класса со спецификатором **public**;
- объявление **public**-конструктора без параметров (по умолчанию);
- объявление инкапсулированных полей;
- объявление корректных *get*-еров и *set*-еров для каждого нестатического поля;
- возможность сохранения объекта целиком — сериализация.

До выхода Java 1.4 единственным способом сериализации объекта была имплементация интерфейса **Serializable**. В Java 1.4 понятие сериализации было расширено возможностью сохранения объекта в XML-файле с помощью метода **writeObject(Object obj)** класса **java.beans.XMLEncoder**, что само по себе напоминает механизм сериализации класса **ObjectOutputStream**. Десериализация легко выполняется методом **Object readObject()** класса **java.beans.XMLDecoder**.

Чтобы класс **Student** мог быть подвергнут XML-сериализации, он должен быть объявлен в виде:

```
/* # 14 # класс для сериализации в XML-файл # Student.java */
```

```
package by.epam.learn.io;
import java.io.Serializable;
import java.util.StringJoiner;
public class Student implements Serializable {
    static String faculty = "MMF";
    private String name;
    private int id;
    private transient String password;
    private static final long serialVersionUID = 2L;
    public Student(){}
    public Student(String name, int id, String password) {
        this.name = name;
        this.id = id;
        this.password = password;
    }
}
```

```

public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
@Override
public String toString() {
    return new StringJoiner(", ", Student.class.getSimpleName() + "[", "]")
        .add("name='" + name + "'").add("id=" + id)
        .add("password='" + password + "'").toString();
}
}

```

Статические поля не сериализуются. Поля со спецификатором **transient** — сериализуются в XML.

Реализация процесса приведена в следующем примере:

```
/* # 15 # запись сериализованного объекта в XML-файл # XmlSerializeMain.java */
```

```

package by.epam.learn.io;
import java.beans.XMLDecoder;
import java.beans.XMLEncoder;
import java.io.*;
public class XmlSerializeMain {
    public static void main(String[] args) {
        try (XMLEncoder xmlEncoder = new XMLEncoder(new BufferedOutputStream(
            new FileOutputStream("data\\serial.xml")))) {
            Student student = new Student("Janka", 555777, "VKL_1410");
            xmlEncoder.writeObject(student);
            xmlEncoder.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        try (XMLDecoder xmlDecoder = new XMLDecoder(new BufferedInputStream(
            new FileInputStream("data/serial.xml")))) {
            Student student = (Student)xmlDecoder.readObject();
            System.out.println(student);
        }
    }
}

```

```

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
}

```

Объект в результате в XML-файле сохраняется в виде:

```

<?xml version="1.0" encoding="UTF-8"?>
<java version="11" class="java.beans.XMLDecoder">
  <object class="by.epam.learn.io.Student">
    <void property="id">
      <int>555777</int>
    </void>
    <void property="name">
      <string>Janka</string>
    </void>
    <void property="password">
      <string>VKL_1410</string>
    </void>
  </object>
</java>

```

с указанием класса, который может выполнить десериализацию, пути и имени сериализуемого класса, имен, типов и значений полей объекта этого класса.

Класс Scanner

Объект класса **java.util.Scanner** принимает форматированный объект или ввод из потока и преобразует его в двоичное представление. При вводе могут использоваться данные из консоли, файла, строки или любого другого источника, реализующего интерфейсы **Readable**, **InputStream** или **ReadableByteChannel**. На настоящий момент класс **Scanner** предлагает наиболее удобный и полный интерфейс для извлечения информации практически из любых источников.

Некоторые конструкторы класса:

```

Scanner(String source)
Scanner(InputStream source)
Scanner(Path source)
Scanner(Path source, String charset)

```

где **source** — источник входных данных, а **charset** — кодировка источника.

Объект класса **Scanner** читает лексемы из источника, указанного в конструкторе, например, из строки или файла. Лексема — это набор символов, выделенный набором разделителей (по умолчанию пробельными символами). В случае ввода из консоли следует определить объект:

```

Scanner console = new Scanner(System.in);

```

После создания объекта его используют для ввода информации в приложение, например, лексемы и строки,

```
String str1 = console.next();
String str2 = console.nextLine();
```

или типизированной лексемы, например, целого числа,

```
if(console.hasNextInt()) {
    int number = console.nextInt();
}
```

В классе **Scanner** определены группы методов, проверяющих данные заданного типа на доступ для ввода. Для проверки наличия произвольной лексемы используется метод **hasNext()**. Проверка конкретного типа производится с помощью одного из методов **boolean hasNextType()** или **boolean hasNextType(int radix)**, где **radix** — основание системы счисления. Например, вызов метода **hasNextInt()** возвращает **true**, только если следующая входящая лексема — целое число. Если данные указанного типа доступны, они считываются с помощью одного из методов *Type nextType()*. Произвольная лексема считывается методом **String next()**. После извлечения любой лексемы текущий указатель устанавливается перед следующей лексемой.

В качестве примера можно рассмотреть форматированное чтение из файла **scan.txt**, содержащего информацию следующего вида:

2 Java 1,6 true 1.7

```
// # 16 # разбор текстового файла # ScannerMain.java
```

```
package by.epam.learn.io;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;
public class ScannerMain {
    public static void main(String[] args) {
        String filename = "data\\scan.txt";
        try(Scanner scan = new Scanner(new FileReader(filename))) {
            while (scan.hasNext()) {
                if (scan.hasNextInt()) {
                    System.out.println(scan.nextInt() + " :int");
                } else if (scan.hasNextBoolean()) {
                    System.out.println(scan.nextBoolean() + " :boolean");
                } else if (scan.hasNextDouble()) {
                    System.out.println(scan.nextDouble() + " :double");
                } else {
                    System.out.println(scan.next() + " :String");
                }
            }
        }
    }
}
```

```

    } catch (FileNotFoundException e) {
        System.err.println(e);
    }
}
}

```

В результате выполнения кода при белорусских региональных установках операционной системы будет выведено:

2 :int

Java :String

1.6:double

true :boolean

1.7:String

Процедура проверки типа реализована методами типа **hasNextType()**. Такой подход предпочтителен из-за отсутствия возможности возникновения исключительной ситуации, так как ее обработка требует на порядок больше ресурсов, чем нормальное течение программы.

Объект класса **Scanner** определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода **useDelimiter(Pattern pattern)** или **useDelimiter(String regex)**, где **pattern** и **regex** содержит набор разделителей в виде регулярного выражения. Применение метода **useLocale(Locale loc)** позволяет задавать правила чтения информации, принятые в заданной стране или регионе.

```

/* # 17 # применение разделителей и локалей # ScannerDelimiterMain.java */

```

```

package by.epam.learn.io;
import java.util.Locale;
import java.util.Scanner;
public class ScannerDelimiterMain {
    public static void main(String[] args) {
        double sum = 0.0;
        String numbersStr = "1,3;2,0; 8,5; 4,8;9,0; 1; 10;";
        Scanner scan = new Scanner(numbersStr)
            .useLocale(Locale.FRANCE) // change to Locale.US
            .useDelimiter(";\\s*");
        while (scan.hasNext()) {
            if (scan.hasNextDouble()) {
                sum += scan.nextDouble();
            } else {
                System.out.println(scan.next());
            }
        }
        System.out.printf("Sum = " + sum);
        scan.close();
    }
}

```

В результате выполнения программы будет выведено:

Sum = 36.6

Если заменить **Locale** на американскую, то результат будет иным, так как представление чисел с плавающей точкой отличается.

Использование шаблона «`;\s*`» указывает объекту класса **Scanner**, что символ «`;`» и ноль или более пробелов следует рассматривать как разделитель.

Метод **String findInLine(Pattern pattern)** или **String findInLine(String pattern)** ищет заданный шаблон в текущей строке текста. Если шаблон найден, соответствующая ему подстрока извлекается из строки ввода. Если совпадений не найдено, то возвращается **null**.

Методы **String findWithinHorizon(Pattern pattern, int count)** и **String findWithinHorizon(String pattern, int count)** производят поиск заданного шаблона в ближайших **count** символах. Можно пропустить образец с помощью метода **skip(Pattern pattern)**.

Если в строке ввода найдена подстрока, соответствующая образцу **pattern**, метод **skip()** просто перемещается за нее в строке ввода и возвращает ссылку на вызывающий объект. Если подстрока не найдена, метод **skip()** генерирует исключение **NoSuchElementException**.

Архивация

Для хранения классов языка Java и связанных с ними ресурсов в языке Java используются сжатые архивные **jar**-файлы.

Для работы с архивами в спецификации Java существуют два пакета — **java.util.zip** и **java.util.jar**, соответственно для архивов **zip** и **jar**. Различие форматов **jar** и **zip** заключается только в расширении архива **zip**. Пакет **java.util.jar** аналогичен пакету **java.util.zip**, за исключением реализации конструкторов и метода **void putNextEntry(ZipEntry e)** класса **JarOutputStream**. Ниже будет рассмотрен только пакет **java.util.jar**. Чтобы переделать все ниже представленные примеры на использование **zip**-архива, достаточно всюду в коде заменить **Jar** на **Zip**.

Пакет **java.util.jar** позволяет считывать, создавать и изменять файлы форматов **jar**, а также вычислять контрольные суммы входящих потоков данных. Класс **JarEntry** (подкласс **ZipEntry**) используется для предоставления доступа к записям **jar**-файла. Некоторые методы класса:

void setMethod(int method) — устанавливает метод сжатия записи;

void setSize(long size) — устанавливает размер несжатой записи;

long getSize() — возвращает размер несжатой записи;

long getCompressedSize() — возвращает размер сжатой записи.

У класса **JarOutputStream** существует возможность записи данных в поток вывода в **jar**-формате. Он переопределяет метод **write()** таким образом, чтобы

любые данные, записываемые в поток, предварительно подвергались сжатию. Основными методами класса являются:

void setLevel(int level) — устанавливает уровень сжатия. Чем больше уровень сжатия, тем медленней происходит работа с таким файлом;

void putNextEntry(ZipEntry e) — записывает в поток новую **jar**-запись.

Этот метод переписывает данные из экземпляра **JarEntry** в поток вывода;

void closeEntry() — завершает запись в поток **jar**-записи и заносит дополнительную информацию о ней в поток вывода;

void write(byte b[], int off, int len) — записывает данные из буфера **b**, начиная с позиции **off**, длиной **len** в поток вывода;

void finish() — завершает запись данных **jar**-файла в поток вывода без закрытия потока.

К примеру, необходимо архивировать файлы в указанной директории. Если директория содержит другие директории, то их файлы также должны архивироваться, но на глубину не выше значения **maxDepth** равное **10**, задаваемое в методе **walk()**.

```

/* # 18 # создание jar-архива # PackJar.java */

package by.epam.learn.io;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;
import java.util.jar.JarEntry;
import java.util.jar.JarOutputStream;
import java.util.stream.Collectors;
import java.util.zip.Deflater;
public class PackJar {
    private String jarFileName;
    public final static int BUFFER = 2_048;
    public PackJar(String jarFileName) throws FileNotFoundException {
        if(!jarFileName.endsWith(".jar")) {
            throw new FileNotFoundException(jarFileName + " incorrect archive name");
        }
        this.jarFileName = jarFileName;
    }
    public void pack(String dirName) throws FileNotFoundException {
        Path dirPath = Paths.get(dirName);
        if (Files.notExists(dirPath) || !Files.isDirectory(dirPath)) {
            throw new FileNotFoundException(dirPath + " not found");
        }
        List<Path> listFilesToJar = null;
        try {
            listFilesToJar = Files.walk(dirPath, 10).filter(

```

```

        f -> !Files.isDirectory(f)).collect(Collectors.toList());
    } catch (IOException e) {
        e.printStackTrace();
    }
    Path[] temp = {};
    Path[] filesToJar = listFilesToJar.toArray(temp);
    // actually archiving
    try (FileOutputStream outputStream = new FileOutputStream(jarFileName);
        JarOutputStream jarOutputStream = new JarOutputStream(outputStream)) {
        byte[] buffer = new byte[BUFFER];
        jarOutputStream.setLevel(Deflater.DEFAULT_COMPRESSION);
        for (int i = 0; i < filesToJar.length; i++) {
            String file = filesToJar[i].toString();
            jarOutputStream.putNextEntry(new JarEntry(file));
            try (FileInputStream in = new FileInputStream(file)) {
                int len;
                while ((len = in.read(buffer)) > 0) {
                    jarOutputStream.write(buffer, 0, len);
                }
                jarOutputStream.closeEntry();
            } catch (FileNotFoundException e) {
                System.err.println("File not found " + e);
            }
        }
    } catch (IllegalArgumentException e) {
        System.err.println("incorrect data " + e);
    } catch (IOException e) {
        System.err.println("I/O error " + e);
    }
}
}

```

Расширить класс до архивации файлов из вложенных директорий относительно просто.

Класс **JarFile** обеспечивает гибкий доступ к записям, хранящимся в **jar**-файле. Это достаточно эффективный способ, поскольку доступ к данным осуществляется гораздо быстрее, чем при считывании каждой отдельной записи. Единственным недостатком является то, что доступ может осуществляться только для чтения. Метод **entries()** извлекает все записи из **jar**-файла. Этот метод возвращает список экземпляров **JarEntry** — по одной для каждой записи в **jar**-файле. Метод **getEntry(String name)** извлекает запись по имени. Метод **getInputStream()** создает поток ввода для записи. Этот метод возвращает поток ввода, который может использоваться приложением для чтения данных записи.

```
/* # 19 # запуск процесса архивации # PackMain.java */
```

```

package by.epam.learn.io;
import java.io.FileNotFoundException;
public class PackMain {

```

```
public static void main(String[] args) {
    String dirName = "name_of_directory";
    try {
        PackJar packJar = new PackJar("example.jar");
        packJar.pack(dirName);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

В результате выполнения кода будет создан архивный файл **example.jar**, размещенный в корне проекта.

Класс **JarInputStream** читает данные в **jar**-формате из потока ввода. Он переопределяет метод **read()** таким образом, чтобы любые данные, считываемые из потока, предварительно распаковывались.

Теперь следует распаковать файл из архива и разместить по заданному пути, к которому добавится исходный путь к файлам.

```
/* # 20 # чтение jar-архива # UnPackJar.java */
```

```
package by.epam.learn.io;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;
public class UnPackJar {
    private Path destinationPath;
    // buffer size when unpacking
    public static final int BUFFER = 2_048;
    public void unpack(String destinationDirectory, String nameJar) {
        try (JarFile jarFile = new JarFile(nameJar)) {
            jarFile.stream().forEach(entry -> {
                String entryname = entry.getName();
                System.out.println("Extracting: " + entry);
                destinationPath = Paths.get(destinationDirectory, entryname);
                // create directory structure
                destinationPath.getParent().toFile().mkdirs();
                // unpack the record, if it is not a directory
                if (!entry.isDirectory()) {
                    writeFile(jarFile, entry);
                }
            });
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
    private void writeFile(JarFile jar, JarEntry entry) {
        int currentByte;
        byte data[] = new byte[BUFFER];
        try (BufferedInputStream bufferedInput = new BufferedInputStream(jar.getInputStream(entry));
            FileOutputStream fileOutput = new FileOutputStream(destinationPath.toString());
            BufferedOutputStream bufferedOutput = new BufferedOutputStream(fileOutput, BUFFER)) {
            // write the file to disk
            while ((currentByte = bufferedInput.read(data, 0, BUFFER)) > 0) {
                bufferedOutput.write(data, 0, currentByte);
            }
            bufferedOutput.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

При указании пути к архивному файлу и пути, по которому требуется разместить разархивированные файлы, следует указывать либо абсолютный путь, либо путь относительно корневой директории проекта.

```
/* # 21 # запуск процесса архивации # UnPackMain.java */
```

```

package by.epam.learn.io;
public class UnPackMain {
    public static void main(String[] args) {
        // Location and archive name
        String nameJar = "example.jar";
        // directory to which the files will be unpacked
        String destinationPath = "c:\\tmp\\";
        new UnPackJar().unpack(destinationPath, nameJar);
    }
}

```

На консоль будет выведен список разархивированных файлов, например:

```

Extracting: data\inf\serial.xml
Extracting: data\info.txt
Extracting: data\message.properties
Extracting: data\messages2.properties
Extracting: data\out.dat
Extracting: data\out.txt
Extracting: data\res.txt
Extracting: data\thread.txt

```

Вопросы к главе 10

1. Что такое поток данных? Какие потоки данных существуют в Java? Привести иерархию потоков ввода-вывода в Java.
2. Какие классы байтовых потоков ввода-вывода существуют?
3. Какие классы символьных потоков ввода-вывода существуют?
4. Как работают методы **read()** и **write()** базовых классов иерархии символьных и байтовых потоков? Сравнить.
5. Для чего используются классы **BufferedInputStream**, **BufferedOutputStream**, **BufferedReader**, **BufferedWriter**?
6. Для чего используются классы **FilterInputStream**, **FilterOutputStream**, **FilterReader**, **FilterWriter**?
7. Для чего применяются классы **InputStreamReader** и **OutputStreamWriter**?
8. Что такое упаковка (*wrapping*) потоков?
9. Какие существуют предопределенные потоки ввода-вывода в Java? Кто эти потоки создает, и кто их закрывает?
10. Что такое сериализация, для чего нужна, когда применяется? Правила сериализации объектов.
11. Что такое десериализация? Правила десериализации объектов.
12. Будет ли повторно сериализоваться уже сериализованный объект?
13. Что получится при десериализации, если при сериализации сохраняемые объекты имели ссылки на одни и те же объекты?
14. Как происходит десериализация? Вызывается ли конструктор при десериализации? Как десериализуются объекты, созданные от классов, у которых базовые классы несериализуемые?
15. Как сериализовать и десериализовать объект? Какие классы и интерфейсы для этого необходимо использовать? Какое статическое поле сериализуется?
16. Что происходит при сериализации/десериализации объекта-синглтона. Как правильно сериализовать синглтон?
17. Ключевое слово **transient**, для чего нужно?
18. Возможно ли сохранить объект не в байт-код, а в XML-файл?

Задания к главе 10

Вариант А

В следующих заданиях требуется ввести последовательность строк из текстового потока и выполнить указанные действия. При этом могут рассматриваться два варианта:

- каждая строка состоит из одного слова;
- каждая строка состоит из нескольких слов.

Имена входного и выходного файлов, а также абсолютный путь к ним могут быть введены как параметры командной строки или храниться в файле.

1. В каждой строке найти и удалить заданную подстроку.
2. В каждой строке стихотворения найти и заменить заданную подстроку на подстроку иной длины.
3. В каждой строке найти слова, начинающиеся с гласной буквы.
4. Найти и вывести слова текста, для которых последняя буква одного слова совпадает с первой буквой следующего слова.
5. Найти в строке наибольшее число цифр, следующих подряд.
6. В каждой строке стихотворения Максима Богдановича подсчитать частоту повторяемости каждого слова из заданного списка и вывести эти слова в порядке возрастания частоты повторяемости.
7. В каждом слове повести Владимира Короткевича «Дикая охота короля Стаха» заменить первую букву слова на прописную.
8. Определить частоту повторяемости букв и слов в стихотворении Адама Мицкевича.

Вариант В

Выполнить задания из варианта В гл. 4, сохраняя объекты приложения в одном или нескольких файлах с применением механизма сериализации. Объекты могут содержать поля, помеченные как **static**, а также **transient**. Для изменения информации и извлечения информации в файле создать специальный класс-коннектор с необходимыми для выполнения этих задач методами.

Вариант С

При выполнении следующих заданий для вывода результатов создавать новую директорию и файл средствами класса **File**.

1. Создать и заполнить файл случайными целыми числами. Отсортировать содержимое файла по возрастанию.
2. Прочитать текст Java-программы и все слова **public** в объявлении атрибутов и методов класса заменить на слово **private**.
3. Прочитать текст Java-программы и записать в другой файл в обратном порядке символы каждой строки.
4. Прочитать текст Java-программы и в каждом слове длиннее двух символов все строчные символы заменить прописными.
5. В файле, содержащем фамилии студентов и их оценки, записать прописными буквами фамилии тех студентов, которые имеют средний балл более 7.
6. Файл содержит символы, слова, целые числа и числа с плавающей запятой. Определить все данные, тип которых вводится из командной строки.
7. Из файла удалить все слова, содержащие от трех до пяти символов, но при этом из каждой строки должно быть удалено только максимальное четное количество таких слов.
8. Прочитать текст Java-программы и удалить из него все лишние пробелы и табуляции, оставив только необходимые для разделения операторов.

9. Из текста Java-программы удалить все виды комментариев.
10. Прочитать строки из файла и поменять местами первое и последнее слова в каждой строке.
11. Ввести из текстового файла, связанного с входным потоком, последовательность строк. Выбрать и сохранить m последних слов в каждой из последних n строк.
12. Из текстового файла ввести последовательность строк. Выделить отдельные слова, разделяемые пробелами. Написать метод поиска слова по образцу-шаблону. Вывести найденное слово в другой файл.
13. Сохранить в файл, связанный с выходным потоком, записи о телефонах и их владельцах. Вывести в файл записи, телефоны в которых начинаются на k и на j .
14. Входной файл содержит совокупность строк. Строка файла содержит строку квадратной матрицы. Ввести матрицу в двумерный массив (размер матрицы найти). Вывести исходную матрицу и результат ее транспонирования.
15. Входной файл хранит квадратную матрицу по принципу: строка представляет собой число. Определить размерность. Построить двумерный массив, содержащий матрицу. Вывести исходную матрицу и результат ее поворота на 90° по часовой стрелке.
16. В файле содержится совокупность строк. Найти номера строк, совпадающих с заданной строкой. Имя файла и строка для поиска — аргументы командной строки. Вывести строки файла и номера строк, совпадающих с заданной.

Тестовые задания к главе 10

Вопрос 10.1.

Для чего используется метод **flush()** класса **OutputStream**? (выбрать один)

- a) очистка выходного буфера с последующим завершением операции вывода данных;
- b) закрытие выходного потока;
- c) установка минимально возможного размера выходного буфера;
- d) вывод выходного буфера.

Вопрос 10.2.

Объектом какого класса является статическое поле **out** класса **System**? (выбрать один)

- a) **DataOutputStream**
- b) **OutputStream**
- c) **BufferedPrintStream**
- d) **BufferedOutputStream**
- e) **PrintStream**

Вопрос 10.3.

Дан фрагмент кода:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter number: ");
// line 1
```

Какой из операторов следует вставить вместо *line 1*, чтобы из консоли было прочитано число? (выбрать один)

- a) `int number = Integer.valueOf(reader.readLine());`
- b) `int number = reader.read();`
- c) `int number = new Scanner(reader.nextInt());`
- d) `int number = String.parseInt(reader.readLine());`
- e) ни один из перечисленных.

Вопрос 10.4.

Дан код:

```
class Quest {
    public static void main(String[] args) throws IOException {
        Path source = Paths.get("datares/data/mail.properties");
        Path destination = Paths.get("datares/mail.properties");
        Files.copy(source, destination);
    }
}
```

Каким будет результат запуска кода, если файл *mail.properties* существует в директории *datares/data* ? (выбрать один)

- a) генерация исключения `NoSuchFileException`;
- b) генерация исключения `FileAlreadyExistsException`;
- c) генерация исключения `FileNotFoundException`;
- d) код выполнится корректно, но никаких изменений произведено не будет;
- e) файл *mail.properties* будет скопирован в директорию *datares*.

Вопрос 10.5.

Дан фрагмент кода. Выбрать одно корректное утверждение:

```
BufferedWriter b1 = new BufferedWriter(new File("data.txt")); // 1
BufferedWriter b2 = new BufferedWriter(new FileWriter("data.txt")); // 2
BufferedWriter b3 = new BufferedWriter(new PrintWriter("data.txt")); // 3
BufferedWriter b4 = new BufferedWriter(new BufferedWriter(b3)); // 4
```

- a) все строки скомпилируются корректно;
- b) все строки скомпилируются некорректно;
- c) ошибка компиляции в строке 1;
- d) ошибка компиляции в строке 2;
- e) ошибка компиляции в строке 3;
- f) ошибка компиляции в строке 4.

КОЛЛЕКЦИИ И STREAM API

Программирование заставило дерево зацвести.

Алан Дж. Перлис

Общие определения

Коллекции — это хранилища или контейнеры, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных структур данных, поддерживающих три основные операции:

- добавление нового элемента в коллекцию;
- удаление элемента из коллекции;
- изменение элемента в коллекции.

В качестве других операций могут быть реализованы следующие: заменить, просмотреть элементы, подсчитать их количество и др.

Для работы с коллекциями разработчиками был создан **Collection Framework**. Применение коллекций обуславливается возросшими объемами обрабатываемой информации. Когда счет используемых объектов идет на сотни тысяч или миллионов, массивы не обеспечивают ни должной скорости, ни экономии ресурсов.

Примером коллекции является стек (структура LIFO — *Last In First Out*), в котором всегда удаляется объект, вставленный последним. Для очереди (структура FIFO — *First In First Out*) используется другое правило удаления: всегда удаляется элемент, вставляемый первым. В абстрактных типах данных существует несколько видов очередей: двусторонние очереди, кольцевые очереди, обобщенные очереди, в которых запрещены повторяющиеся элементы. Стеки и очереди могут быть реализованы как на базе массива, так и на базе связанного списка.

Коллекции в языке Java объединены в библиотеке классов **java.util** и представляют собой контейнеры для хранения и манипулирования объектами. До появления Java 2 эта библиотека содержала классы только для работы с простейшими структурами данных: **Vector**, **Stack**, **Hashtable**, **BitSet**, а также интерфейс **Enumeration** для работы с элементами этих классов. Коллекции, появившиеся в Java 2, представляют собой общую технологию хранения и доступа к объектам. Скорость обработки коллекций повысилась по сравнению с предыдущей версией языка за счет отказа от их потокобезопасности. Поэтому, если объект коллекции может быть доступен из различных потоков, что наиболее естественно для распределенных приложений, возможно использование коллекции из Java 1.

В Java 5 в новом пакете **java.util.concurrent** появились ограниченно потоко-безопасные коллекции, гарантирующие более высокую производительность в многопоточной среде для конкурирующих потоков.

Так как в коллекциях при практическом программировании хранится набор ссылок на объекты одного типа, следует обезопасить коллекцию от появления ссылок на другие, не разрешенные логикой приложения типы. Такие ошибки при использовании нетипизированных коллекций выявляются на стадии выполнения, что повышает трудозатраты на исправление и верификацию кода. Поэтому, начиная с версии Java SE 5, коллекции стали типизированными или *generic*.

Более удобным стал механизм работы с коллекциями, а именно:

- предварительное сообщение компилятору о типе ссылок, которые будут храниться в коллекции, при этом проверка осуществляется на этапе компиляции;
- отсутствие необходимости постоянно преобразовывать возвращаемые по ссылке объекты (тип **Object**) к требуемому типу.

Структура коллекций характеризует способ, с помощью которого программы Java обрабатывают группы объектов. Так как **Object** — суперкласс для всех классов, то в коллекции можно хранить объекты любого типа, кроме базовых. Коллекции — это динамические массивы, связанные списки, деревья, множества, хэш-таблицы, стеки, очереди.

Интерфейсы коллекций:

Map<K, V> — карта отображения вида «ключ-значение»;

Collection<E> — основной интерфейс коллекций, вершина иерархии коллекций **List**, **Set**. Также наследует интерфейс **Iterable<E>**;

List<E> — специализирует коллекции для обработки упорядоченного набора элементов;

Set<E> — множество, содержащее уникальные элементы;

Queue<E> — очередь, где элементы добавляются в один конец списка, а извлекаются из другого конца.

Все классы коллекций реализуют интерфейсы **Serializable**, **Cloneable** (кроме **WeakHashMap**).

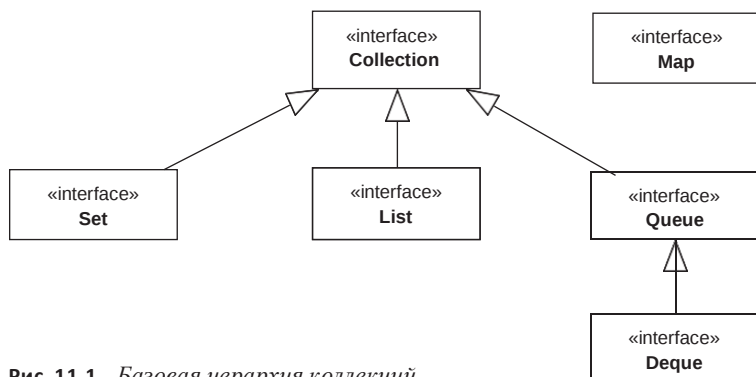


Рис. 11.1. Базовая иерархия коллекций

В интерфейсе **Collection<E>** определены методы, которые работают на всех коллекциях:

boolean add(E obj) — добавляет **obj** к вызывающей коллекции и возвращает **true**, если объект добавлен, и **false**, если **obj** уже элемент коллекции;

boolean remove(Object obj) — удаляет **obj** из коллекции;

boolean addAll(Collection<? extends E> c) — добавляет все элементы коллекции к вызывающей коллекции;

void clear() — удаляет все элементы из коллекции;

boolean contains(Object obj) — возвращает **true**, если вызывающая коллекция содержит элемент **obj**;

boolean equals(Object obj) — возвращает **true**, если коллекции эквивалентны;

boolean isEmpty() — возвращает **true**, если коллекция пуста;

int size() — возвращает количество элементов в коллекции;

Object[] toArray() — копирует элементы коллекции в массив объектов;

<T> T[] toArray(T a[]) — копирует элементы коллекции в массив объектов определенного типа.

Появление **Stream API** обусловило возникновение методов для создания потоков объектов и работы с функциональными интерфейсами:

default Stream<E> stream() — преобразует коллекцию в *stream* объектов;

default Stream<E> parallelStream() — преобразует коллекцию в параллельный *stream* объектов. Повышает производительность при работе с очень большими коллекциями на многоядерных процессорах;

default boolean removeIf(Predicate<? super E> filter) — удаляет все элементы коллекции в зависимости от условия.

Метод **void forEach(Consumer<? super T> action)**, **Iterator<T> iterator()** унаследован от интерфейса **Iterable<T>**.

При работе с элементами коллекции применяются интерфейсы: **Iterator<E>**, **ListIterator<E>**, **Map.Entry<K, V>** — для перебора коллекции и доступа к объектам коллекции.

Интерфейс **Iterator<E>** строит объект, обеспечивающий доступ к элементам коллекции. К этому типу относится объект, возвращаемый методом **iterator()**. Такой объект позволяет осуществлять навигацию по содержимому коллекции последовательно, элемент за элементом. Позиции итератора условно располагаются в коллекции между элементами. В коллекции, состоящей из *N* элементов, существует *N*+1 позиций итератора.

Методы интерфейса **Iterator<E>**, представляющего собой одну из реализаций дизайн-паттерна с одноименным названием:

boolean hasNext() — проверяет наличие следующего элемента, а в случае его отсутствия (завершения коллекции) возвращает **false**. Итератор при этом остается неизменным;

E next() — возвращает ссылку на объект, на который указывает итератор, и передвигает текущий указатель на следующий, предоставляя доступ

К следующему элементу. Если следующий элемент коллекции отсутствует, то метод **next()** генерирует исключение **NoSuchElementException**;

void remove() — удаляет объект, возвращенный последним вызовом метода **next()**. Если метод **next()** до вызова **remove()** не вызывался, то будет сгенерировано исключение **IllegalStateException**;

void forEachRemaining(Consumer<? super E> action) — выполняет действие над каждым оставшимся необработанным элементом коллекции.

Интерфейс **Map.Entry** предназначен для извлечения ключей и значений карты с помощью методов **K getKey()** и **V getValue()** соответственно. Вызов метода **V setValue(V value)** заменяет значение, ассоциированное с текущим ключом.

ArrayList

Интерфейс **List<E>** реализует класс **ArrayList<E>** — динамический массив объектных ссылок.

Иерархия наследования следующая:

```
java.util.AbstractCollection<E>
    java.util.AbstractList<E>
        java.util.ArrayList<E>
```

В классе объявлены конструкторы:

```
ArrayList()
ArrayList(Collection <? extends E> c)
ArrayList(int capacity)
```

Практически все методы класса являются реализацией абстрактных методов из суперклассов и интерфейсов или дефолтными методами интерфейсов **Collection**, **List**, **Iterable**. Методы интерфейса **List<E>** позволяют вставлять и удалять элементы из позиций, указываемых через отсчитываемый от нуля индекс:

E get(int index) — возвращает элемент в виде объекта из позиции **index**, представляет собой одно из главных достоинств класса из-за скорости выполнения;

void add(int index, E element) — вставляет **element** в позицию, указанную в **index**;

E remove(int index) — удаляет объект из позиции **index**;

E set(int index, E element) — заменяет объект в позиции **index**, возвращает при этом удаляемый элемент;

boolean addAll(int index, Collection<? extends E> c) — вставляет в вызывающий список все элементы коллекции **c**, начиная с позиции **index**;

int indexOf(Object ob) — возвращает индекс указанного объекта;

default void sort(Comparator<? super E> c) — сортирует список на основе компаратора;

List<E> subList(int fromIndex, int toIndex) — извлекает часть коллекции в указанных границах;

static <E> List<E> copyOf(Collection <? extends E> coll) — создает немодифицируемый список на основе передаваемой коллекции.

Удаление и добавление элементов для такой коллекции представляет ресурсоемкую задачу, поэтому объект **ArrayList<E>** лучше всего подходит для хранения списков с малым числом подобных действий. С другой стороны, навигация по списку осуществляется очень быстро, поэтому операции поиска производятся за более короткое время.

```
/* # 1 # создание параметризованной коллекции # ListMain.java */
package by.epam.learn.collection;
import java.util.ArrayList;
import java.util.List;
public class ListMain {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        // List<int> integers = new ArrayList<>(); // compile error
        list.add(5);
        list.add(7);
        list.add(42);
        list.add(null); // you can add null to the list
        list.add(50);
        list.add(77);
        System.out.println(list);
        list.add(5, 87); // adding to position number 5
        System.out.println(list);
        list.remove(2); // removal from position number 2
        System.out.println(list);
    }
}
```

В результате будет выведено:

[5, 7, 42, null, 50, 77]
 [5, 7, 42, null, 50, 87, 77]
 [5, 7, null, 50, 87, 77]

В данной ситуации не создается новый класс для каждого конкретного типа, и сама коллекция не меняется, просто компилятор снабжается информацией о типе элементов, которые могут храниться в **list**. При этом параметром коллекции может быть только объектный тип. Попытка добавления\удаления элемента с номером, выходящим за пределы текущего размера списка, приведет к исключительной ситуации.

Указывать тип следует при создании ссылки, иначе будет позволено добавлять объекты всех типов. На этом основан принцип типобезопасности, обеспечиваемый параметризацией коллекций. Пусть в системе онлайн-торговли

используется понятие **Order**. Заказы могут собираться в списки, например, список заказов за день. Возможна ситуация, когда по каким-либо причинам в список заказов будет добавлен экземпляр товара или просто строка. В этой ситуации даже свести баланс действительно сделанных заказов простым определением размера списка будет невозможно. Придется извлекать каждый объект, определять его тип и проч.

```
/* # 2 # некорректная(raw) и корректная коллекции # UncheckMain.java */
package by.epam.learn.collection;
import by.epam.learn.entity.Order;
import java.util.ArrayList;
import java.util.List;
public class UncheckMain {
    public static void main(String[] args) {
        // raw type
        List raw = List.of(new Order(231, 12.f),
            new ArrayList(),
            new Order(217, 17.f),
            "Unitas");
        // type casting required
        Order or1 = (Order) raw.get(0);
        ArrayList or2 = (ArrayList)raw.get(1);
        Order or3 = (Order) raw.get(2);
        String or4 = (String) raw.get(3);
        raw.forEach(System.out::println);
        List<Order> orders = List.of(new Order(231, 12.f),
            new Order(389, 29.f),
            // new ArrayList(), compile error
            new Order(217, 17.f));
        orders.forEach(System.out::println);
    }
}
```

В результате будет выведено:

```
Order{orderId=231, amount=12.0}
[]
Order{orderId=217, amount=17.0}
Unitas
Order{orderId=231, amount=12.0}
Order{orderId=389, amount=29.0}
Order{orderId=217, amount=17.0}
```

где класс **Order** представляет собой сущность в следующем виде:

```
/* # 3 # класс-сущность используется здесь и далее для наполнения коллекций
# Order.java */
```

```
package by.epam.learn.entity;
public class Order extends Entity {
    private long orderId;
    private double amount;
    public Order() {
    }
    public Order(long orderId, double amount) {
        this.orderId = orderId;
        this.amount = amount;
    }
    public long getOrderId() {
        return orderId;
    }
    public void setOrderId(long orderId) {
        this.orderId = orderId;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Order order = (Order) o;
        if (orderId != order.orderId) return false;
        return Double.compare(order.amount, amount) == 0;
    }
    public int hashCode() {
        int result;
        long temp;
        result = (int) (orderId ^ (orderId >>> 32));
        temp = Double.doubleToLongBits(amount);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        return result;
    }
    public String toString() {
        final StringBuilder sb = new StringBuilder("Order{");
        sb.append("orderId=").append(orderId).append(", amount=").append(amount).append('}');
        return sb.toString();
    }
}
```

```
/* # 4 # супер-класс всех сущностей системы # Entity.java */
```

```
package by.epam.learn.entity;
import java.io.Serializable;
public abstract class Entity implements Serializable, Cloneable {
}
```

Чтобы параметризация коллекции была полной, необходимо указывать параметр и при объявлении ссылки, и при создании объекта оператором *даймонд* «<>». Интерфейс `List<E>` был дополнен в Java 8 несколькими методами: `void sort(Comparator<? super E> c)`, методом `copyOf()`, серией статических перегруженных методов `of()` с параметрами и без. Метод `of()` выполняет задачу создания коллекции, с которой нельзя выполнить операции по добавлению/удалению элементов. Этот метод добавляет еще один способ создания немодифицируемой коллекции, в частности, для поддержки принципа инкапсуляции. Метод `copyOf()` создает немодифицируемую коллекцию на основе модифицируемой исходной.

Пусть существует класс с полем в виде списка:

```
/* # 5 # класс, агрегирующий список объектов # OrderList.java */
```

```
package by.epam.learn.collection;
import by.epam.learn.entity.Order;
import java.util.ArrayList;
import java.util.List;
public class OrderList {
    private List<Order> orders;
    public OrderList() {
        this.orders = new ArrayList<Order>();
    }
    public OrderList(List<Order> orders) {
        this.orders = orders;
    }
    public List<Order> getOrders() {
        return orders;
    }
}
```

Метод `getOrders()` дает пользователю доступ к объекту `orders`. Но есть одно «но». Тот, кто получит список, может спокойно добавлять, удалять объекты в список, и эти изменения коснутся списка внутри объекта `OrderList`. После этого нет смысла говорить об инкапсуляции — ее просто не существует в этом варианте. Решение довольно простое: метод `getOrders()` должен возвращать неизменяемый список:

```
public List<Order> getOrders() {
    return List.copyOf(orders);
}
```

или

```
public List<Order> getOrders() {
    Order[] array = {};
    return List.of(orders.toArray(array));
}
```

Список, полученный обеими реализациями метода **getOrders()**, нельзя модифицировать.

Итераторы

Интерфейс **Iterator<E>** используется для последовательного доступа к элементам коллекции. Классическое использование итератора выглядит так:

```
/* # 6 # классическое итерирование коллекции # IteratorMain.java */
```

```
package by.epam.learn.collection;
import by.epam.learn.entity.Order;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class IteratorMain {
    public static void main(String[] args) {
        List<Order> orders = new ArrayList<Order>() {
            {
                add(new Order(231, 12.));
                add(new Order(389, 29.));
                add(new Order(747, 32.));
                add(new Order(517, 18.));
                add(new Order(414, 17.));
                add(new Order(777, 10.));
            }
        };
        Iterator<Order> iterator = orders.iterator();
        while (iterator.hasNext()) {
            Order order = iterator.next();
            System.out.println(order);
        }
    }
}
```

Альтернативное применение итератора с применением метода **foreach()** интерфейса **Iterable** намного проще:

```
orders.forEach(System.out::println);
```

Ниже приведен пример поиска заказов с помощью **while**, **if** и итератора, сумма которых превышает заданную, с удалением всех объектов, не удовлетворяющих условию, и назначением всем оставшимся заказам скидки.

```
/* # 7 # поиск, удаление, изменение элементов списка. Классический способ # */
```

```
final int controlAmount = 20;
final int discountPercent = 10;
Iterator<Order> iterator = orders.iterator();
while (iterator.hasNext()) {
    Order current = iterator.next();
    if (current.getAmount() < controlAmount) {
        iterator.remove();
        continue;
    }
    current.setAmount(current.getAmount() * (100 - discountPercent) / 100.0);
}
System.out.println(orders);
```

где **orders** уже наполненный список заказов.

Методы **removeIf(Predicate<? super T> filter)** и **foreach(Consumer<? super T> action)** позволяют переписать код уже без ветвлений.

```
/* # 8 # поиск, удаление, изменение элементов списка.
Функциональные интерфейсы # */
```

```
orders.removeIf(o -> o.getAmount() <= controlAmount);
orders.forEach(o -> o.setAmount(o.getAmount() * (100 - discountPercent)/100.0));
orders.forEach(System.out::println);
```

Для доступа к элементам списка может также использоваться специализированный интерфейс **ListIterator<E>**, который позволяет получить доступ сразу в необходимую позицию списка вызовом метода **listIterator(int index)** на объекте списка. Интерфейс **ListIterator<E>** расширяет интерфейс **Iterator<E>**, предназначен для обработки списков и их вариаций. Наличие методов **E previous()**, **int previousIndex()** и **boolean hasPrevious()** обеспечивает обратную навигацию по списку. Метод **int nextIndex()** возвращает номер следующего итератора. Метод **void add(E e)** позволяет вставлять элемент в список текущей позиции. Вызов метода **void set(E e)** производит замену текущего элемента списка на объект, передаваемый методу в качестве параметра.

Внесение изменений в коллекцию методами коллекции после извлечения итератора гарантирует генерацию исключения **ConcurrentModificationException** из пакета **java.util** при попытке использовать итератор позднее модификации коллекции, а именно:

```
List<Order> orders = new ArrayList<>();
// list filling
Iterator<Order> iterator = orders.iterator();
orders.add(new Order(12, 12.1f)); // or orders.remove(0);
while (iterator.hasNext()) { // generation exception
    System.out.println(iterator.next());
}
```

Параллельная или конкурентная модификация коллекции методами самой коллекции и ее итератором приводит к неразрешимой проблеме и заканчивается генерацией исключения практически всегда. Проблема заключается в том, что итератор извлек N число позиций, а коллекция изменила число своих экземпляров, и итератор перестал соответствовать коллекции. Если модификацию коллекции и работу с итератором нужно выполнять из различных потоков, то для решения такой задачи используются ограниченно потокобезопасные решения для коллекций из пакета **java.util.concurrent**.

При создании класса, содержащего коллекцию элементов, возможны два способа: агрегация коллекции в качестве поля класса или отношение **HAS-A** и наследование от класса, представляющего коллекцию, или отношение **IS-A**.

Последнее встречается на практике значительно реже.

Класс **OrderType** тогда может выглядеть следующим образом:

```
/* # 9 # класс, агрегирующий список, с реализацией интерфейса Iterable
# OrderType.java*/
```

```
package by.epam.learn.collection;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class OrderType implements Iterable<String> {
    private int orderId;
    private List<String> currencyNames = new ArrayList<>();/* SEK, DKK, NOK, CZK,
                                                                GBP, EUR, PLN */

    public OrderType(int orderId) {
        this.orderId = orderId;
    }
    public List<String> getCurrencyNames() {
        return List.copyOf(currencyNames);
    }
    // delegated method
    public boolean add(String e) {
        return currencyNames.add(e);
    }
    @Override
    public Iterator<String> iterator() {
        return currencyNames.iterator();
    }
}
```

Отношение **IS-A** записывается еще проще, но теряется имя **currencyNames**. Класс **OrderType** теперь сам является списком.

```
/* # 10 # класс, наследующий список # OrderType.java*/
```

```
package by.epam.learn.collection;
import java.util.ArrayList;
```

```
public class OrderType extends ArrayList<String> {
    private int orderId;
    public OrderType(int orderId) {
        this.orderId = orderId;
    }
}
```

Stream API

Преобразование коллекции в *stream* для последующей обработки позволяет не только избавиться от циклов, условных переходов и прямого извлечения итератора, но все действия, выполненные в примере #8, можно произвести в одном операторе. Код становится более коротким, но менее интуитивно понятным, особенно на первый взгляд.

```
/* # 11 # поиск, удаление, изменение элементов списка с помощью stream # */
```

```
List<Order> orderList = orders.stream()
    .filter(o -> o.getAmount() <= controlAmount)
    .map(o -> { o.setAmount(o.getAmount() * (100 - discountPercent) / 100.0);
        return o; })
    .collect(Collectors.toList());
```

Выполнение данного оператора никак не изменит исходный список **orders**, в то время как два предыдущих способа изменяют внутреннее состояние списка **orders**. Отсутствие влияния на исходную коллекцию является еще одной дополнительной возможностью, предоставляемой **Stream API**.

Интерфейс **java.util.stream.Stream<T>** — поток объектов для преобразования коллекций, массивов. В потоке не хранятся элементы операции, он не модифицирует источник, а формирует в ответ на действие новый поток. Операции в потоке не выполняются до момента, пока не потребуется получить конечный результат выполнения.

Stream нельзя воспринимать как просто поток ввода\вывода. Этот поток создается на основе коллекции\массива, элементы которой переходят в состояние информационного ожидания действия, переводящего поток в следующее состояние до достижения терминальной цели, после чего поток прекращает свое существование.

Невозможно на одном и том же состоянии объекта *stream* вызвать метод его обработки повторно:

```
/* # 12 # ошибочное использование stream # */
```

```
Stream<String> stream = strings.stream();
stream.findFirst();
stream.filter(String::isBlank).findAny();
```

В результате будет сгенерировано исключение **IllegalStateException**.

Получить *stream* из коллекции можно методом **stream()** интерфейса **Collection**, а из массива методом **Stream.of(T...values)**.

```
List<String> strings = List.of("as a the d on and".split("\\s+"));
strings.stream();
```

Некоторые *промежуточные* методы преобразования потоков объектов:

filter(Predicate<? super T> predicate) — выбор элементов из потока на основании работы предиката в новый поток. Отбрасываются все элементы, не удовлетворяющие условию предиката:

```
/* # 13 # фильтрация элементов списка # */
```

```
strings.stream()
    .filter(s -> s.length() < 2)
    .forEach(s -> System.out.print(s + " "));
```

В результате останутся две строки, чья длина меньше чем 2 символа:

a d

map(Function<? super T, ? extends R> mapper) — изменение всех элементов потока, применяя к каждому элементу функцию **mapper**. Тип параметризации потока может изменяться, если типизация **T** и **R** относится к различным классам:

```
/* # 14 # изменение всех элементов списка # */
```

```
strings.stream()
    .map(s -> s.length())
    .forEach(s -> System.out.print(s + " "));
```

Функция в методе **map()** получит строку и возвратит ее длину:

2 1 3 1 2 3

```
strings.stream()
    .map(String::toUpperCase)
    .forEach(s -> System.out.print(s + " "));
```

Функция в методе **map()** получит строку и возвратит ее, преобразовав все символы в верхний регистр:

AS A THE D ON AND

flatMap(Function<T, Stream<R>> mapper) — преобразовывает один объект, как правило составной, в объект более простой структуры, например, массив в строку, список в объект, список списков в один список:

```
/* # 15 # преобразование списка # */
```

```
OrderType type1 = new OrderType(771);
type1.add("SEK");
type1.add("DKK");
```

```

type1.add("NOK");
type1.add("EUR");
OrderType type2 = new OrderType(779);
type2.add("SEK");
type2.add("PLN");
type2.add("CZK");
type2.add("EUR");
List<OrderType> orderTypes = List.of(type1, type2);
List<String> currencyList =
    orderTypes.stream()
        .map(o -> o.getCurrencyNames()) // -> Stream<List<String>>
        .flatMap(o -> o.stream())       // -> Stream<String>
        .collect(Collectors.toList());
System.out.println(currencyList);

```

В результате будет получен объединенный список на основе двух списков:

[SEK, DKK, NOK, EUR, SEK, PLN, CZK, EUR]

peek(Consumer<T> consumer) — возвращает поток, содержащий все элементы исходного потока. Используется для просмотра элементов в текущем состоянии потока. Можно использовать для записи логов:

```

strings.stream()
    .map(String::length)
    .peek(s -> System.out.print(s + "-"))
    .map(n -> n + 100)
    .forEach(s -> System.out.print(s + " "));

```

Объект в *stream* обрабатывается сразу через все промежуточные и терминальные функции:

2-102 1-101 3-103 1-101 2-102 3-103

sorted(Comparator<T> comparator) и sort() — сортировка в новый поток:

```
/* # 16 # сортировка списка # */
```

```

strings.stream()
    .sorted()
    .forEach(s -> System.out.print(s + " "));

```

Сортировка с использованием собственного компаратора класса **String**:

a and as d on the

```

strings.stream()
    .sorted((s1, s2) -> s1.length() - s2.length())
    //.sorted(Comparator.comparingInt(String::length)) // identically
    .forEach(s -> System.out.print(s + " "));

```

Сортировка с использованием заданного компаратора, сортирующего по длине строк:

as d as on the and

limit(long maxSize) — ограничивает выходящий поток заданным в параметре значением;

```
strings.stream()
    .limit(3)
    .forEach(s -> System.out.print(s + " "));
```

Выведет в поток только первые три элемента:

as a the

skip(long n) — не включает в выходной поток первые **n** элементов исходного потока;

```
strings.stream()
    .skip(4)
    .forEach(s -> System.out.print(s + " "));
```

Пропустит первые четыре элемента:

on and

distinct() — удаляет из потока все идентичные элементы;

```
List<String> stringsDouble = List.of("the and the and the and".split("\\s+"));
stringsDouble.stream()
    .distinct()
    .forEach(s -> System.out.print(s + " "));
```

Будет выведено:

the and

Некоторые *терминальные* методы сведения потока к результату. Результатом может быть новая коллекция, объект некоторого класса, число. Промежуточные операции обязательно должны завершаться терминальными, иначе они не выполнятся, так как просто не имеют смысла.

void forEach(Consumer<T> action) — выполняет действие над каждым элементом потока. Чтобы результат действия сохранялся, реализация *action* должна предусматривать фиксацию результата в каком-либо объекте или потоке вывода;

Optional<T> findFirst() — находит первый элемент в потоке;

```
String firstStr = strings.stream()
    .filter(s -> s.matches("a\\w*"))
    .findFirst()
    .orElse("none");
System.out.println(firstStr);
```

Результат: **as**

Фильтр выбрал в потоке все элементы, удовлетворяющие условию предиката, а метод **findFirst()** нашел первый элемент.

Optional<T> findAny() — находит элемент в потоке;

```
String anyStr = strings.stream()
    .filter(s -> s.matches("an[a-z]"))
    .findAny()
    .orElse("none");
System.out.println(anyStr);
```

Результат: **and**

Фильтр выбрал в потоке все элементы, удовлетворяющие условию предиката, а метод **findAny()** нашел один из элементов.

long count() — возвращает число элементов потока;

```
long count = strings.stream()
    .filter(s -> s.matches("a\\w*"))
    .count();
System.out.println(count);
```

Результат: **3**

Фильтр выбрал в потоке все элементы, удовлетворяющие условию предиката, а метод **count()** вернул их число.

boolean allMatch(Predicate<T> predicate) — возвращает истину, если все элементы *stream* удовлетворяют условию предиката;

```
boolean res1 = strings.stream()
    .allMatch(s -> s.length() < 5); // true
```

boolean anyMatch(Predicate<T> predicate) — возвращает истину, если хотя бы один элемент *stream* удовлетворяет условию предиката;

```
boolean res2 = strings.stream()
    .anyMatch(s -> s.startsWith("a")); // true
```

boolean noneMatch(Predicate<T> predicate) — возвращает истину, если ни один элемент *stream* не удовлетворяет условию предиката;

```
boolean res3 = strings.stream()
    .noneMatch(s -> s.endsWith("z")); // true
```

Optional<T> reduce(T identity, BinaryOperator<T> accumulator) — сводит все элементы потока к одному результирующему объекту, завернутому в оболочку **Optional**;

```
int sumLength = strings.stream()
    .map(s -> s.length())
    .reduce(0, (n1, n2) -> n1 + n2);
```

Результат: **12**

Поток строк преобразуется в поток их длин и метод **reduce()** вычисляет сумму всех длин строк.

```
int sum = strings.stream()
    .reduce(0, (identity, v) -> v.length() + identity, Integer::sum);
```

То же самое действие, но без промежуточного преобразования в *stream* целых чисел.

<R, A> R collect(Collector<? super T, A, R> collector) — собирает элементы в коллекцию или объект другого типа;

```
Map<String, Integer> map = Arrays.stream("as a the d on and".split("\\s+"))
    .collect(Collectors.toMap(s -> s, s -> s.length()));
```

В результате будет получен объект **map** вида:

```
{the=3, a=1, as=2, d=1, and=3, on=2}
```

Разбиение строки на *stream* подстрок. Создание **Map**, где ключом является сама подстрока, а значением ее длина.

Optional<T> min(Comparator<T> comparator) — находит минимальный элемент;

```
String min = strings.stream()
    .min(Comparator.comparingInt(s -> s.charAt(s.length() - 1)))
    .orElse("none");
System.out.println(min);
```

Результат: **a**

Поиск строки с минимальным значением кода его последнего символа.

Optional<T> max(Comparator<T> comparator) — находит максимальный элемент.

```
String max = strings.stream()
    .max(Comparator.comparingInt(Action::sumCharCode))
    .orElse("empty");
System.out.println(max);
```

Поиск строки с максимальной суммой кодов его символов.

Результат: **the**

где класс **Action** и его метод подсчета суммы кодов — символов строки:

```
class Action {
    public static int sumCharCode(String str) {
        return str.codePoints().reduce(0, (v1, v2) -> v1 + v2);
    }
}
```

Источники Stream API:

collection.stream(),
Arrays.stream(int[] array),

```
Files.walk(Path path),
Files.list(Path path),
bufferedReader.lines(),
Stream.iterate(T seed, UnaryOperator<T> f),
Stream.generate(Supplier<? extends T> s),
Stream.of(T...t),
Stream.ofNullable(T t),
Stream.empty(),
Stream.concat(Stream<? extends T> a, Stream<? extends T> b),
string.lines(),
string.codePoints(),
string.chars(),
random.ints(),
random.doubles(),
random.longs() и другие.
```

Алгоритмы сведения Collectors

Методы класса `java.util.stream.Collectors` представляют основные возможности, позволяя произвести обработку *stream* к результату, будь то число, строка или коллекция. Каждый статический метод класса создает экземпляр **Collector** для передачи методу `collect()` интерфейса **Stream**, который и выполнит действия по преобразованию *stream* алгоритмом, содержащимся в экземпляре **Collector**.

```
List<String> strings = List.of("as a the d on and".split("\\s+"));
```

`toCollection(Supplier<C> collectionFactory)`, `toList()`, `toSet()` — преобразование в коллекцию;

```
// accumulate lengths of strings into a List
List<Integer> listLengths = strings.stream()
    .map(String::length)
    .collect(Collectors.toList());
System.out.println(listLengths);
```

Результат: [2, 1, 3, 1, 2, 3]

Преобразование в список с применением ссылки на конструктор:

```
List<Character> listFirstSymb = strings.stream()
    .map(s -> s.charAt(0))
    .collect(Collectors.toCollection(ArrayList::new));
System.out.println(listFirstSymb);
```

Результат: [a, a, t, d, o, a]

`joining(CharSequence delimiter)` — обеспечивает конкатенацию строк с заданным разделителем;

```
// convert strings to upperCase and concatenate them, separated by colon
String result = strings.stream()
    .map(String::toUpperCase)
    .collect(Collectors.joining(" : "));
System.out.println(result);
```

Результат: AS : A : THE : D : ON : AND

mapping(Function<? super T,? extends U> mapper,Collector<? super U,A,R> downstream) — позволяет преобразовать элементы одного типа в элементы другого типа;

```
// converting a list of strings to a list of codes of their first characters
List<Integer> listCode = strings.stream()
    .collect(Collectors.mapping(s -> (int) s.charAt(0), Collectors.toList()));
System.out.println(listCode);
```

Результат: [97, 97, 116, 100, 111, 97]

```
// converting a list of strings to a list of codes of their first characters and
// finding the maximum value
int max = strings.stream()
    .collect(Collectors.mapping(s -> (int) s.charAt(0),
        Collectors.maxBy(Integer::compareTo)).orElse(-1);
System.out.println(max);
```

Результат: 116

minBy(Comparator<? super T> c)/maxBy(Comparator<? super T> c) — коллектор для нахождения минимального или максимального элемента в потоке;

```
// search for the first line with the minimum length
String minLex = strings.stream()
    .collect(Collectors.minBy(String::compareTo))
    .orElse("none"); System.out.println(minLex);
```

Результат: a

filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream) — выполняет фильтрацию элементов;

```
// selection of flow elements with a length greater than 1
List<String> lists = strings.stream()
    .collect(Collectors.filtering(s -> s.length() > 1, Collectors.toList()));
System.out.println(lists);
```

Результат: [as, the, on, and]

counting() — позволяет посчитать количество элементов потока;

```
// counting the number of stream elements
long counter = strings.stream()
    .collect(Collectors.counting());
System.out.println(counter);
```

Результат: 6

summingInt(ToIntFunction<? super T> mapper) — выполняет суммирование элементов. Существуют версии для **Long** и **Double**;

```
// compute sum of lengths of all strings
int length = strings.stream()
    .collect(Collectors.summingInt(String::length));
System.out.println(length);
```

Результат: 12

averagingInt(ToIntFunction<? super T> mapper) — вычисляет среднее арифметическое элементов потока. Существуют версии для **Long** и **Double**;

```
// average value of stream line lengths
Double averageLength = strings.stream()
    .collect(Collectors.averagingDouble(String::length));
System.out.println(averageLength);
```

Результат: 2.0

reducing(T identity, BinaryOperator<T> op) — коллектор, осуществляющий редукцию (сведение) элементов на основании заданного бинарного оператора;

```
// compute sum of code first chars each strings
int sumCodeFirstChars = strings.stream()
    .map(s -> (int)s.charAt(0))
    .collect(Collectors.reducing(0, (a, b) -> a + b));
System.out.println(sumCodeFirstChars);
```

Результат: 618

reducing(U identity, Function<? super T, ? extends U> mapper, BinaryOperator<U> op) — аналогичное предыдущему действие. В примере ниже производится умножение всех длин строк, но без предварительного преобразования *stream* к другому типу;

```
int productLength = strings.stream()
    .collect(Collectors.reducing(1, s -> s.length(), (s1, s2) -> s1 * s2));
System.out.println(productLength);
```

Результат: 36

groupingBy(Function<? super T, ? extends K> classifier) — коллектор группировки элементов потока;

```
// group strings by length
Map<Integer, List<String>> byLength = strings.stream()
    .collect(Collectors.groupingBy(String::length));
System.out.println(byLength);
```

Результат: {1=[a, d], 2=[as, on], 3=[the, and]}

```
// compute sum of common length by string length
Map<Integer, Integer> sumChars = strings.stream()
```

```
.collect(Collectors.groupingBy(String::length, Collectors.summingInt(String::length)));
System.out.println(sumChars);
```

Результат: {1=2, 2=4, 3=6}

partitioningBy(Predicate<? super T> predicate) — коллектор разбиения элементов потока;

```
// partition strings into length less than 2 and all the rest
Map<Boolean, List<String>> boolLength = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() < 2));
System.out.println(boolLength);
```

Результат: {false=[as, the, on, and], true=[a, d]}

Метасимвол в коллекциях

Метасимволы используются при параметризации коллекций для расширения возможностей самой коллекции и обеспечения ее типобезопасности. Например, если параметр метода предыдущего примера изменить с **List<Order>** на **List<? extends Order>**, то в метод можно будет передавать коллекции, параметризованные любым допустимым типом, а именно классом **Order** и любым его подклассом, что невозможно при записи без анонимного символа.

Но в методе нельзя будет добавить к коллекции новый элемент, пусть даже и допустимого типа, так как компилятору неизвестен заранее тип параметризации списка.

```
List<Order> action(List<? extends Order> orders) {
    // orders.add(new Order(231, 12.f)); // compile error
    orders.remove(0); // can be deleted
}
```

Объясняется это тем, что список, передаваемый в качестве параметра метода, может быть параметризован классом **DiscountOrder**, а в методе будет совершаться попытка добавления экземпляра самого класса **Order**, как показано выше, что недопустимо определением параметризации самого списка, передаваемого в метод, а именно:

```
action(new ArrayList<DiscountOrder>());
```

где

```
public class DiscountOrder extends Order {
    public DiscountOrder(int orderId, float amount) {
        super(orderId, amount);
    }
}
```

Поэтому добавление к спискам, параметризованным метасимволом с применением **extends**, запрещено всегда.

Параметризация **List<? super Order>** утверждает, что параметр метода или возвращаемое значение может получить список типа **Order** или любого из его суперклассов, в то же время разрешает добавлять туда экземпляры класса **Order** и любых его подклассов.

```
List<? super Order> action(List<? super Order> orders) {
    orders.add(new Order(231, 12.f));
    return orders;
}
```

В этом случае к списку, возвращенному методом, можно будет добавлять экземпляры класса **Order** и его подклассов.

Класс **LinkedList** и интерфейс **Queue**

Коллекция **LinkedList<E>** реализует возможности связанного списка.

```
java.util.AbstractCollection<E>
    java.util.AbstractList<E>
    java.util.AbstractSequentialList<E>
    java.util.LinkedList<E>
```

Реализует, кроме интерфейсов, указанных при описании **ArrayList**, также интерфейсы **Queue<E>** и **Deque<E>**.

Связанный список хранит ссылки на объекты отдельно вместе со ссылками на следующее и предыдущее звенья последовательности, поэтому часто называется двунаправленным списком.

Самый быстрый метод класса **add(E element)**. Главным же достоинством класса является скорость работы метода **remove()** на **Iterator**, после получения его из **LinkedList**. Также очень быстро работает метод **add(E element)** на **ListIterator**.

Операция удаления из начала и конца списка выполняется достаточно быстро, в отличие от операций поиска и извлечения.

При тестировании на списке из десяти тысяч элементов **LinkedList** быстрее, чем **ArrayList**, при добавлении в середину списка методом **add()** в 2 раза, а в начало или конец примерно в 40 раз. Вставки и удаления элементов из **LinkedList** происходят за постоянное время, в том числе и с использованием итераторов, в то же время вставка\удаление элемента в **ArrayList** приводит к сдвигу всех элементов после позиции добавления\удаления, а в случае, если базовый массив хранения переполняется, то еще и сам массив увеличивается в полтора раза с копированием старого массива в новый. Список **ArrayList**, в свою очередь, быстрее при вызове метода **get(index)** примерно в 50 раз. Происходит это вследствие того, что определение позиции в списке производится за конкретный интервал времени, не зависящий от размера списка, при поиске же индекса в **LinkedList** время поиска пропорционально размеру списка.

Список **LinkedList** занимает большой объем памяти за счет необходимости хранения ссылок на соседние объекты, что следует учитывать при создании списков больших размеров. Список **LinkedList** занимает от 3,5 до 5 раз больше памяти нежели аналогичный список **ArrayList**.

В этом классе объявлены методы, позволяющие манипулировать им как очередью, двунаправленной очередью и т.д. Двунаправленный список, кроме обычного, имеет особый «нисходящий» итератор, позволяющий двигаться от конца списка к началу, и извлекается методом **descendingIterator()**.

Для манипуляций с первым и последним элементами списка в **LinkedList<E>** реализованы методы:

void addFirst(E ob), void addLast(E ob) — добавляющие элементы в начало и конец списка;

E getFirst(), E getLast() — извлекающие элементы;

E removeFirst(), E removeLast() — удаляющие и извлекающие элементы;

E removeLastOccurrence(E elem), E removeFirstOccurrence(E elem) — удаляющие и извлекающие элемент, первый или последний раз встречаемый в списке.

Класс **LinkedList<E>** реализует интерфейс **Queue<E>**, что позволяет списку придать свойства очереди. В компьютерных науках очередь — структура данных, в основе которой лежит принцип FIFO (*first in, first out*). Элементы добавляются в конец и вынимаются из начала очереди. Но существует возможность не только добавлять и удалять элементы, также можно просмотреть, что находится в очереди. К тому же методы интерфейса **Queue<E>** по манипуляции первым и последним элементами такого списка **E element(), boolean offer(E o), E peek(), E poll(), E remove()** работают немного быстрее, чем соответствующие методы класса **LinkedList<E>**.

Методы интерфейса **Queue<E>**:

boolean add(E o) — вставляет элемент в очередь, но если же очередь полностью заполнена, то генерирует исключение **IllegalStateException**;

boolean offer(E o) — вставляет элемент в очередь, если возможно;

E element() — возвращает, но не удаляет головной элемент очереди;

E peek() — возвращает, но не удаляет головной элемент очереди, возвращает **null**, если очередь пуста;

E poll() — возвращает и удаляет головной элемент очереди, возвращает **null**, если очередь пуста;

E remove() — возвращает и удаляет головной элемент очереди.

Методы **element()** и **remove()** отличаются от методов **peek()** и **poll()** тем, что генерируют исключение **NoSuchElementException**, если очередь пуста.

```
Queue<Order> queue = new LinkedList<>();
```

Создается очередь простым присваиванием списка **LinkedList** ссылке типа **Queue**.

```
/* # 17 # двунаправленный список и очередь # QueueMain.java */
```

```
package by.epam.learn.collection;
import java.util.LinkedList;
import java.util.Queue;
public class QueueMain {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>(){
            {
                this.offer("Jeans");
            }
        };
        queue.add("Dress");
        queue.offer("Gloves");
        queue.offer("T-Shirt");
        queue.stream()
            .filter(s -> !s.endsWith("s"))
            .forEach(System.out::println);
        System.out.println("_____");
        queue.forEach(System.out::println);
    }
}
```

В результате исходная очередь не изменится и будет выведено:

T-Shirt

Jeans

Dress

Gloves

T-Shirt

Если заменить применение *stream* на

```
queue.removeIf(s -> s.endsWith("s"));
```

то действие будет произведено с изменением исходной коллекции.

При всей схожести списков **ArrayList** и **LinkedList** существуют серьезные отличия, которые необходимо учитывать при использовании коллекций в конкретных задачах. Если необходимо осуществлять быструю навигацию по списку, то следует применять **ArrayList**, так как перебор элементов в **LinkedList** осуществляется на порядок медленнее. С другой стороны, если требуется часто добавлять и удалять элементы из списка, то уже класс **LinkedList** обеспечивает значительно более высокую скорость переиндексации. То есть если коллекция формируется в начале процесса и в дальнейшем используется только для доступа к информации, то применяется **ArrayList**, если же коллекция подвергается изменениям на всем протяжении функционирования приложения, то выгоднее **LinkedList**.

Интерфейс **Queue<E>** также реализует класс **PriorityQueue<E>**. Иерархия наследования представлена в виде:

```
java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
        java.util.PriorityQueue<E>
```

В такую очередь элементы добавляются не в порядке «живой очереди», а в порядке, определяемом в классе, экземпляры которого содержатся в очереди. Сам порядок следования задан реализацией интерфейсов **Comparator<E>** или **Comparable<E>**.

По умолчанию компаратор размещает элементы в очереди в порядке возрастания, который можно изменить на убывание, если компаратору задать **reverseOrder()**.

```
/* # 18 # создание приоритетной очереди # PriorityMain.java */
```

```
package by.epam.learn.collection;
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
public class PriorityMain {
    public static void main(String[] args) {
        Queue<String> prior = new PriorityQueue<>(Comparator.reverseOrder());
        prior.offer("J");
        prior.offer("A");
        prior.offer("V");
        prior.offer("A");
        prior.offer("1");
        prior.offer("4");
        while(!prior.isEmpty()) {
            System.out.println(prior.poll());
        }
    }
}
```

Результатом будет:

```
V
J
A
A
4
1
```

Если порядок в классе не определен, а именно, не реализован ни один из указанных интерфейсов, то генерируется исключительная ситуация **ClassCastException** при добавлении второго элемента в очередь. При добавлении первого элемента компаратор не вызывается, так как сравнивать объект не с кем.

```
PriorityQueue<Order> orders = new PriorityQueue();
orders.add(new Order(546, 53.)); // ok
orders.add(new Order(146, 17.)); // runtime error
```

С другой стороны, класс **String** реализует интерфейс **Comparable<E>**, тогда будет:

```
PriorityQueue<String> orders = new PriorityQueue<String>();
orders.add(new String("Oracle")); // ok
orders.add(new String("Google")); // ok
```

Если попытаться заменить тип **String** на **StringBuilder** или **StringBuffer**, то создать очередь **PriorityQueue** так просто не удастся, как и в случае добавления объектов класса **Order**. Решением такой задачи будет создание необходимого класса-компаратора или класса-оболочки с полем типа **StringBuilder** и реализацией интерфейса **Comparable<T>**.

```
/* # 19 # пользовательский класс, объект которого может быть добавлен
в очередь PriorityQueue и множество TreeSet # StringBuilderWrapper.java */
```

```
package by.epam.learn.collection;
public class StringBuilderWrapper implements Comparable<StringBuilder> {
    private StringBuilder content = new StringBuilder();
    @Override
    public int compareTo(StringBuilder o) {
        return content.toString().compareTo(o.toString());
    }
}
```

Предлагаемое решение универсально для любых пользовательских типов. Применимо для создания упорядоченных множеств вида **TreeSet<T>**, которые используют **Comparable** или **Comparator** для сортировки при добавлении экземпляра в множество.

Интерфейс Deque и класс ArrayDeque

Интерфейс **Deque** определяет «двунаправленную» очередь и, соответственно, методы доступа к первому и последнему элементам двусторонней очереди. Реализацию этого интерфейса можно использовать для моделирования стека. Методы обеспечивают удаление, вставку и обработку элементов. Каждый из этих методов существует в двух формах. Одни методы создают исключительную ситуацию в случае неудачного завершения, другие возвращают какое-либо из значений (**null** или **false** в зависимости от типа операции). Вторая форма добавления элементов в очередь сделана специально для реализаций **Deque**, имеющих ограничение по размеру. В большинстве реализаций операции добавления заканчиваются успешно. Методы **addFirst()**, **addLast()** вставляют элементы в начало и в конец очереди соответственно. Метод **add()** унаследован от интерфейса **Queue** и абсолютно аналогичен методу **addLast()** интерфейса **Deque**. Объявить двуконечную очередь на основе связанного списка можно, например, следующим образом:

```
Deque<String> deque = new LinkedList<>();
```

Класс **ArrayDeque**<E> быстрее, чем **Stack**<E>, если используется как стек, и быстрее, чем **LinkedList**<E>, если используется в качестве очереди.

Простое применение в виде стека и очереди:

```
/* # 20 # двуконечная очередь как стек и как очередь # ArrayDequeMain.java */

package by.epam.learn.collection;
import java.util.ArrayDeque;
import java.util.Deque;
public class ArrayDequeMain {
    public static void main(String[] args) {
        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);
        while (!stack.isEmpty()){
            System.out.println(stack.pop()); // Like as stack
        }
        Deque<Integer> queue = new ArrayDeque<>();
        queue.offer(11);
        queue.offer(22);
        queue.offer(33);
        while(!queue.isEmpty()){
            System.out.println(queue.poll()); // Like as queue
        }
    }
}
```

Будет выведено:

```
3
2
1
11
22
33
```

Множества

Интерфейс **Set**<E> объявляет поведение коллекции, не допускающей дублирования элементов. Интерфейс **SortedSet**<E> наследует **Set**<E> и объявляет поведение набора, отсортированного в возрастающем порядке, заранее определенном для класса. Интерфейс **NavigableSet**<E> существенно облегчает поиск элементов, например, расположенных рядом с заданным.

Класс **HashSet**<E> наследуется от абстрактного суперкласса **AbstractSet**<E> и реализует интерфейс **Set**<E>, используя хэш-таблицу для хранения коллекции.

```
java.util.AbstractCollection<E>
    java.util.AbstractSet<E>
        java.util.HashSet<E>
```

Ключ хэш-код используется в качестве индекса хэш-таблицы для доступа к объектам множества, что значительно ускоряет процессы поиска, добавления и извлечения элемента. Скорость указанных процессов становится заметной для коллекций с большим количеством элементов. Множество **HashSet** не является сортированным. В таком множестве могут храниться элементы с одинаковыми хэш-кодами в случае, если эти элементы не эквивалентны при сравнении. Для грамотной организации **HashSet** стоит следить, чтобы реализации методов **equals()** и **hashCode()** соответствовали правилам.

Конструкторы класса:

```
HashSet()
HashSet(Collection <? extends E> c)
HashSet(int capacity)
HashSet(int capacity, float loadFactor),
```

где **capacity** — число ячеек для хранения хэш-кодов.

```
/* # 21 # создание множества # HashSetMain.java */
```

```
package by.epam.learn.collection;
import java.util.HashSet;
public class HashSetMain {
    public static void main(String[] args) {
        HashSet<String> hashSet = new HashSet<>();
        hashSet.add("8Y");
        hashSet.add("2Y");
        hashSet.add("2Y");
        hashSet.add("8Y");
        hashSet.add("6Y");
        hashSet.add("5Y");
        hashSet.add("Y-");
        hashSet.stream()
            .peek(System.out::print)
            .forEach(s -> System.out.println(" " + s.hashCode()));
    }
}
```

При выводе видно, что объекты в **HashSet** хранятся в произвольном порядке относительно значений хэш-кодов.

```
2Y 1639
5Y 1732
6Y 1763
Y- 2804
8Y 1825
```

```

/* # 22 # использование множества для вывода всех уникальных слов из файла
# HashSetWordsMain.java */

```

```

package by.epam.learn.collection;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.HashSet;
import java.util.Scanner;
import java.util.TreeSet;
public class HashSetWordsMain {
    public static void main(String[] args) {
        HashSet<String> words = new HashSet<>(100_000);
        Scanner scan;
        try {
            scan = new Scanner(new File("data\\uladzimir_arlou.txt"));
            scan.useDelimiter("[^А-я]+");
            while (scan.hasNext()) {
                String word = scan.next();
                words.add(word.toLowerCase());
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        System.out.println(words);
        TreeSet<String> treeSet = new TreeSet<>(words);
        System.out.println(treeSet);
    }
}

```

Класс **TreeSet<E>** для хранения объектов использует бинарное (красно-черное) дерево. С этим алгоритмом желательно ознакомиться самостоятельно. Иерархия наследования **TreeSet**:

```

java.util.AbstractCollection<E>
    java.util.AbstractSet<E>
        java.util.TreeSet<E>

```

При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что класс реализует интерфейс **SortedSet**, где правило сортировки добавляемых элементов определяется в самом классе, сохраняемом в множестве, который в большинстве случаев реализует интерфейс **Comparable**. Обработка операций удаления и вставки объектов происходит несколько медленнее, чем в хэш-множествах, где при любом числе элементов время этих операций постоянно.

Конструкторы класса:

```

TreeSet()
TreeSet(Collection <? extends E> c)
TreeSet(Comparator <? super E> c)
TreeSet(SortedSet <E> s)

```

Класс **TreeSet<E>** содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов **E first()** и **E last()**. Методы **subSet(E from, E to)**, **tailSet(E from)** и **headSet(E to)** предназначены для извлечения определенной части множества. Метод **Comparator <? super E> comparator()** возвращает объект **Comparator**, используемый для сортировки объектов множества или **null**, если выполняется обычная сортировка.

```
/* # 23 # создание и преобразование множеств и их методы # HashSetMain.java */
```

```
package by.epam.learn.collection;
import java.util.Set;
import java.util.TreeSet;
public class HashSetMain {
    public static void main(String[] args) {
        Set<String> set = Set.of("2Y", "8Y", "6Y", "5Y", "Y-");
        System.out.println(set);
        TreeSet<String> treeSet = new TreeSet<>(set);
        treeSet.add("Y-");
        System.out.println(treeSet);
        System.out.println(treeSet.last() + " " + treeSet.first());
    }
}
```

В результате может быть выведено:

```
[2Y, 5Y, 6Y, Y-, 8Y]
[2Y, 5Y, 6Y, 8Y, Y-]
Y- 2Y
```

Множество инициализируется списком и сортируется сразу же в процессе создания. С помощью итератора элемент может быть найден и удален из множества. Для множества, состоящего из обычных строк, используется лексикографическая сортировка, задаваемая реализацией интерфейса **Comparable**, поэтому метод **comparator()** возвращает **null**.

EnumSet

Абстрактный класс **EnumSet<E extends Enum<E>>** наследуется от абстрактного класса **AbstractSet**.

```
java.util.AbstractCollection<E>
    java.util.AbstractSet<E>
        java.util.EnumSet<E>
```

Класс специально реализован для работы с типами **enum**. Все элементы такой коллекции должны принадлежать единственному типу **enum**, определенному явно или неявно. Внутренне множество представимо в виде вектора битов, обычно единственного **long**. Множества нумераторов поддерживают перебор по

диапазону из нумераторов. Скорость выполнения операций над таким множеством очень высока, даже если в ней участвует большое количество элементов.

Основное назначение множества **EnumSet<E>** — это выделение подмножества из полного набора элементов перечисления. Сами способы создания объекта этого множества указывают на эту особенность. Создать объект этого класса можно только с помощью статических методов. Метод **EnumSet<E> noneOf(Class<E> elemType)** создает пустое множество нумерованных констант с указанным типом элемента, метод **allOf(Class<E> elementType)** создает множество нумерованных констант, содержащее все элементы указанного типа. Метод **of(E first, E... rest)** создает множество, первоначально содержащее указанные элементы. С помощью метода **complementOf(EnumSet<E> s)** создается множество, содержащее все элементы, которые отсутствуют в указанном множестве. Метод **range(E from, E to)** создает множество из элементов, содержащихся в диапазоне, определенном двумя элементами. При передаче указанным методам в качестве параметра **null** будет сгенерирована исключительная ситуация **NullPointerException**.

```
/* # 24 # класс-множество # Country.java */
```

```
package by.epam.learn.collection;
public enum Country {
    ARMENIA, BELARUS, INDIA, KAZAKHSTAN, POLAND, UKRAINE
}
```

```
/* # 25 # использование множества enum-типов # EnumSetCountryMain.java */
```

```
package by.epam.learn.collection;
import java.util.EnumSet;
import static by.epam.learn.collection.Country.*;
public class EnumSetCountryMain {
    public static void main(String[] args) {
        EnumSet<Country> asiaCountries = EnumSet.of(ARMENIA, INDIA, KAZAKHSTAN);
        String nameCountry = "Belarus";
        Country current = Country.valueOf(nameCountry.toUpperCase());
        if (asiaCountries.contains(current)) {
            System.out.print(current + " is in Asia");
        } else {
            System.out.print(current + " is not in Asia");
        }
    }
}
```

Карты отображений

Карта отображений — это объект, который хранит пару «ключ–значение». Поиск объекта (значения) облегчается по сравнению с множествами за счет

того, что его можно найти по уникальному ключу. Уникальность объектов-ключей должна обеспечиваться переопределением методов `hashCode()` и `equals()` или реализацией интерфейсов `Comparable`, `Comparator` пользовательским классом. Классы карт отображений:

AbstractMap<K, V> — реализует интерфейс **Map<K, V>**, является супер-классом для всех перечисленных карт отображений;

HashMap<K, V> — использует хэш-таблицу для работы с ключами;

TreeMap<K, V> — использует дерево, где ключи расположены в виде дерева поиска в определенном порядке;

WeakHashMap<K, V> — позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости приложения;

LinkedHashMap<K, V> — образует дважды связанный список ключей. Этот механизм эффективен, только если превышен коэффициент загруженности карты при работе с кэш-памятью и др.

Для класса **IdentityHashMap<K, V>** хэш-коды объектов-ключей вычисляются методом **System.identityHashCode()** по адресу объекта в памяти, в отличие от обычного значения **hashCode()**, вычисляемого сугубо по содержанию самого объекта.

Интерфейсы карт:

Map<K, V> — отображает уникальные ключи и значения;

Map.Entry<K, V> — описывает пару «ключ–значение»;

SortedMap<K, V> — содержит отсортированные ключи и значения;

NavigableMap<K, V> — добавляет новые возможности навигации и поиска по ключу.

Интерфейс **Map<K, V>** содержит следующие методы:

V get(Object obj) — возвращает значение, связанное с ключом **obj**. Если элемент с указанным ключом отсутствует в карте, то возвращается значение **null**;

V put(K key, V value) — помещает ключ **key** и значение **value** в вызывающую карту. При добавлении в карту элемента с существующим ключом, произойдет замена текущего элемента новым. При этом метод возвратит заменяемый элемент;

default V putIfAbsent(K key, V value) — помещает ключ **key** и значение **value** в вызывающую карту. При добавлении в карту элемента с существующим ключом, замена не произойдет;

default V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) — помещает ключ **key** и вычисляет значение **value** при добавлении в вызывающую карту;

default V computeIfAbsent(K key, Function<? super K, ? super V> mappingFunction) — помещает ключ **key** и значение **value** в вызывающую карту, если пары с таким ключом не существует, если ключ существует, то замена не производится;

default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) — заменяет значение **value** в вызывающей карте, если ключ с таким значением существует, если же пары с таким ключом не существует, то вставка пары не производится;

void putAll(Map <? extends K, ? extends V> m) — помещает карту **m** в вызывающую карту;

V remove(Object key) — удаляет пару «ключ–значение» по ключу **key**;

void clear() — удаляет все пары из вызываемой карты;

boolean containsKey(Object key) — возвращает **true**, если вызывающая карта содержит **key** как ключ;

boolean containsValue(Object value) — возвращает **true**, если вызывающая карта содержит **value** как значение;

Set<K> keySet() — возвращает множество ключей;

Set<Map.Entry<K, V>> entrySet() — возвращает множество, содержащее значения карты в виде пар «ключ–значение»;

Collection<V> values() — возвращает коллекцию, содержащую значения карты;

static <K, V> Map<K, V> copyOf(Map <? extends K, ? extends V> map) — копирует исходную карту в немодифицируемую новую карту;

static <K, V> Map<K, V> of(parameters) — перегруженный метод для создания неизменяемых карт на основе переданных в метод параметров;

default void forEach(BiConsumer<? super K, ? super V> action) — выполняет действие над каждым элементом **Map**.

В коллекциях, возвращаемых тремя последними методами, можно только удалять элементы, добавлять нельзя. Данное ограничение обуславливается параметризацией возвращаемого методами значения.

Интерфейс **Map.Entry<K, V>** представляет пару «ключ–значение» и содержит следующие методы:

K getKey() — возвращает ключ текущего входа;

V getValue() — возвращает значение текущего входа;

V setValue(V obj) — устанавливает значение объекта **obj** в текущем входе.

В примере показаны способы создания хэш-карты и доступа к ее элементам.

```
/* # 26 # создание hashmap # HashMapMain.java */
```

```
package by.epam.learn.collection;
import java.util.HashMap;
import java.util.Map;
public class HashMapMain {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Jeans", 40); // adding a pair
        map.put("T-Shirt", 35);
        map.put("Gloves", 42);
    }
}
```

```

map.compute("Shoes", (k,v) -> 77); // adding a pair
System.out.println(map);
// replacing value if key exists
map.computeIfPresent("Shoes", (k,v) -> v + k.length());
System.out.println(map);
map.computeIfAbsent("Shoes", v -> 11);
// adding a pair if the key is missing
map.computeIfAbsent("Shoes_2", v -> 11);
System.out.println(map);
}
}

```

{Gloves=42, T-Shirt=35, Jeans=40, Shoes=77}

{Gloves=42, T-Shirt=35, Jeans=40, Shoes=82}

{Shoes_2=11, Gloves=42, T-Shirt=35, Jeans=40, Shoes=82}

```
/* # 27 # создание хэш-карты и замена элемента по ключу # MapEntryMain.java */
```

```

package by.epam.learn.collection;
import java.util.*;
public class MapEntryMain {
    public static void main(String[] args) {
        HashMap<String, Integer> hashMap = new HashMap<String, Integer>();
        hashMap.put("Пряник", 5);
        hashMap.put("Кефир", 1);
        hashMap.put("Хлеб", 1);
        hashMap.putIfAbsent("Хлеб", 2); // replacement will not happen
        hashMap.putIfAbsent("Молоко", 5);
        hashMap.computeIfAbsent("Сырок", v -> 3); // adding a pair
        hashMap.computeIfPresent("Сырок", (k, v) -> 4); // replacement a value
        hashMap.computeIfAbsent("Сырок", v -> 144); // replacement will not happen
        System.out.println(hashMap);
        hashMap.put("Пряник", 4); // replacement or addition in the absence of a key
        System.out.println(hashMap + "after replacing the element");
        System.out.println(hashMap.get("Хлеб") + " - found by key 'Хлеб'");
        Set<Map.Entry<String, Integer>> entrySet = hashMap.entrySet();
        System.out.println(entrySet);
        entrySet.stream()
            .forEach(e -> System.out.println(e.getKey() + " : " + e.getValue()));
        Set<Integer> values = new HashSet<Integer>(hashMap.values());
        System.out.println(values);
    }
}

```

{Хлеб=1, Сырок=4, Пряник=5, Кефир=1, Молоко=5}

**{Хлеб=1, Сырок=4, Пряник=4, Кефир=1, Молоко=5}after replacing the element
1 - found by key 'Хлеб'**

[Хлеб=1, Сырок=4, Пряник=4, Кефир=1, Молоко=5]

Хлеб : 1

Сырок : 4
 Пряник : 4
 Кефир : 1
 Молоко : 5
 [1, 4, 5]

Метод **put(K key, V value)** не проверяет наличия пары с таким же ключом, как и у добавляемой пары, а просто заменяет значение по ключу на новое. Если критично наличие всех ранее добавленных элементов, следует использовать метод **putIfAbsent(K key, V value)**, который может добавлять пару ключ–значение только в том случае, если в карте не содержится ключа со значением, идентичным ключу в параметре метода, или же перед добавлением пары выполнять проверку на наличие идентичных ключей вида:

```
if(!hashMap.containsKey("Пряник") ) {
    // replacement will not happen if the key exists
    hashMap.put("Пряник", 4);
}
```

Класс **EnumMap<K extends Enum<K>, V>** в качестве ключа может принимать только объекты, принадлежащие одному типу **enum**, который должен быть определен при создании коллекции. Специально организован для обеспечения максимальной скорости доступа к элементам коллекции.

```
/* # 28 # создание EnumMap # EnumMapCountryMain.java */
```

```
package by.epam.learn.collection;
import java.util.EnumMap;
public class EnumMapCountryMain {
    public static void main(String[] args) {
        EnumMap<Country, Integer> map = new EnumMap<Country, Integer>(Country.class);
        map.put(Country.POLAND, 8);
        map.put(Country.UKRAINE, 1);
        map.put(Country.BELARUS, 0);
        map.forEach((k, v) -> System.out.println(k + " " + v));
    }
}
```

BELARUS 0
POLAND 8
UKRAINE 1

Класс **WeakHashMap<K, V>** хорошо работает в ситуациях, когда в процессе работы с коллекцией некоторые объекты должны из нее гарантированно удаляться, но моменты необходимости удаления и само удаление пары из коллекции могут отстоять друг от друга по времени.

Пусть существует некоторый набор заказов. Заказ может находиться в состоянии «обработан/необработан». Исходно коллекция содержит необработанные

заказы. Как только заказ изменяет статус на «обработан», его следует удалить из коллекции. Разрешать всем пользователям коллекции удалять заказы, то есть модифицировать коллекцию не очень разумно по причинам безопасности. Коллекция **WeakHashMap** сама будет заботиться об удалении неактуальных объектов, утративших ссылку на внешний ключ.

```
/* # 29 # карта со слабыми ссылками # CurrentOrders.java */
```

```
package by.epam.learn.collection;
import by.epam.learn.entity.Key;
import by.epam.learn.entity.Order;
import java.util.WeakHashMap;
public class CurrentOrders {
    private WeakHashMap<Key, Order> orders = new WeakHashMap<>();
    public Order put(Key key, Order value) {
        return orders.put(key, value);
    }
    public Order get(Object key) {
        return orders.get(key);
    }
    public int size() {
        return orders.size();
    }
}
```

где класс **Key** содержит информацию об уникальном номере заказа и текущем статусе заказа.

```
/* # 30 # ключ для карты # Key.java */
```

```
package by.epam.learn.entity;
public class Key {
    private int keyUnique;
    private boolean isProcessed;
    public Key(int keyUnique) {
        this.keyUnique = keyUnique;
    }
    public boolean isProcessed() {
        return isProcessed;
    }
    public void setProcessed(boolean processed) {
        isProcessed = processed;
    }
}
```

Чтобы сохранять ссылки на ключи, для них будет создан список. После этого значение **isProcessed** у ключа будет изменено, он будет удален из списка ключей и, как следствие, ключ из **WeakHashMap** утратит ссылку и станет целью для удаления «сборщиком мусора».

```

/* # 31 # демонстрация автоматического удаления из карты слабых ссылок
# CurrentOrdersMain.java */

```

```

package by.epam.learn.collection;
import by.epam.learn.entity.Key;
import by.epam.learn.entity.Order;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class CurrentOrdersMain {
    public static void main(String[] args) throws InterruptedException {
        CurrentOrders orders = new CurrentOrders();
        List<Key> keys = new ArrayList<>();
        keys.add(new Key(100));
        keys.add(new Key(220));
        keys.add(new Key(770));
        orders.put(keys.get(0), new Order(77, 10d));
        orders.put(keys.get(1), new Order(65, 54d));
        orders.put(keys.get(2), new Order(41, 93d));
        keys.get(1).setProcessed(true);
        final int size = keys.size();
        Iterator<Key> iterator = keys.iterator();
        while (iterator.hasNext()) {
            Key ordersKey = iterator.next();
            if (ordersKey.isProcessed()) {
                iterator.remove();
            }
        }
        System.out.println(orders.size());
        System.gc();
        Thread.sleep(1_000);
        System.out.println(orders.size());
    }
}

```

Будет выведено:

3
2

Удаление происходит не мгновенно, а только когда сработает «сборщик мусора». И после этого число заказов уменьшится с 3 до 2.

Унаследованные коллекции

В ряде распределенных приложений, например, с использованием сервлетов, до сих пор применяются коллекции более медленные в обработке, но при этом потокобезопасные (*thread-safety*), существовавшие в языке Java с момента

его создания, а именно карта **Hashtable<K, V>**, список **Vector<E>** и итератор **java.util.Enumeration<E>**. Все они также были параметризованы и могут быть обработаны с помощью *stream*, но сохраняют безопасность от одновременного доступа из конкурирующих потоков.

Класс **Hashtable<K, V>** реализует интерфейс **Map**, обладает также несколькими специфичными, по сравнению с другими коллекциями, методами:

Enumeration<V> elements() — возвращает итератор для значений карты;

Enumeration<K> keys() — возвращает итератор для ключей карты.

```
/* # 32 # создание и обработка legacy хэш-таблицы # HashtableMain.java */
```

```
package by.epam.learn.collection;
import java.util.Enumeration;
import java.util.Hashtable;
public class HashtableMain {
    public static void main(String[] args) {
        Hashtable<String, Integer> table = new Hashtable<>();
        table.put("Jeans", 40); // adding a pair
        table.put("T-Shirt", 35);
        table.put("Gloves", 42);
        table.compute("Shoes", (k,v) -> 77); // adding a pair
        System.out.println(table);
        Enumeration<String> keys = table.keys();
        while (keys.hasMoreElements()) {
            System.out.println(keys.nextElement());
        }
        Enumeration<Integer> values = table.elements();
        while (values.hasMoreElements()) {
            System.out.println(values.nextElement());
        }
    }
}
```

В результате в консоль будет выведено:

{Jeans=40, Gloves=42, Shoes=77, T-Shirt=35}

Jeans

Gloves

Shoes

T-Shirt

40

42

77

35

Принципы работы с коллекциями, в отличие от их структуры, со сменой версий языка существенно не изменились. Внутренняя организация **HashMap** в Java 8 претерпела важные изменения, необходимые к самостоятельному изучению.

```
/* # 33 # создание и обработка legacy списка # VectorMain.java */
```

```
package by.epam.learn.collection;
import java.util.Enumeration;
import java.util.Vector;
public class VectorMain {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>(777);
        vector.add("java");
        vector.add("epam");
        vector.add(1, null);
        vector.addAll(vector);
        System.out.println(vector);
        vector.removeIf(e -> e==null);
        vector.replaceAll(String::toUpperCase);
        System.out.println(vector);
        long size = vector.stream().count();
        System.out.println(size);
        Enumeration<String> enumeration = vector.elements();
        while(enumeration.hasMoreElements()) {
            System.out.printf("%s ", enumeration.nextElement());
        }
    }
}
```

[java, null, epam, java, null, epam]

[JAVA, EPAM, JAVA, EPAM]

4

JAVA EPAM JAVA EPAM

Класс **Properties** предназначен для хранения карты свойств, где и имена, и значения являются экземплярами класса **String**. Объект этого класса используется многими технологиями для хранения и передачи свойств конфигурации и пр. Значения пары можно загружать из файла и сохранять в файле.

```
/* # 34 # создание экземпляра и файла properties # PropertiesStoreMain.java */
```

```
package by.epam.learn.collection;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
public class PropertiesStoreMain {
    public static void main(String[] args) {
        Properties props = new Properties();
        try {
            props.setProperty("db.driver", "com.mysql.cj.jdbc.Driver");
            //props.setProperty("db.url", "jdbc:mysql://127.0.0.1:3306/testphones");
            props.setProperty("user", "root");
            props.setProperty("password", "pass");
        }
    }
}
```

```

        props.setProperty("poolsize", "5");
        props.store(new FileWriter("datares/base.properties"), "No Comment's");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

Директория **datares** должна существовать. Объект **FileWriter** не создает промежуточные директории. В результате в файле **base.properties** будет размещена следующая информация:

```

#No Comment's
#Wed July 15 18:32:32 MSK 2020
db.driver=com.mysql.cj.jdbc.Driver
user=root
password=pass
poolsize=5

```

Символ «=» служит по умолчанию разделителем ключа и значения в файле **properties**, также в этом качестве можно использовать символ «:». Эти два специальных символа при записи в файл в качестве части ключа или значения получают впереди символ «\», чтобы в дальнейшем при чтении не быть воспринятым как разделитель. Тогда для вызова

```
props.setProperty("db.url", "jdbc:mysql://127.0.0.1:3306/testphones");
```

В файл будет записано:

```
db.url=jdbc:mysql\://127.0.0.1\:3306/testphones
```

При чтении символ «\» будет проигнорирован.
Извлечь информацию из файла достаточно просто.

```
/* # 35 # загрузка файла properties в экземпляр # PropertiesLoadDemo.java */
```

```

package by.epam.learn.collection;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;
public class PropertiesLoadDemo {
    public static void main(String[] args) {
        Properties props = new Properties();
        try {
            props.load(new FileReader("datares\\base.properties"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        String dbUrl = props.getProperty("db.url");
    }
}

```

```
// following two names are missing in the file
String maxIdle = props.getProperty("maxIdle"); // maxIdle = null
// value "20" will be assigned to the key if it is not found in the file
String maxActive = props.getProperty("maxActive", "20");
System.out.println("dbUrl: " + dbUrl);
System.out.println("maxIdle: " + maxIdle);
System.out.println("maxActive: " + maxActive);
}
}
```

В результате будет выведено:

```
dbUrl: jdbc:mysql://127.0.0.1:3306/testphones
maxIdle: null
maxActive: 20
```

В веб-приложении поток ввода для чтения файла **properties** создается следующим образом:

```
this.getClass().getClassLoader().getResourceAsStream("datares//base.properties")
```

Алгоритмы класса Collections

Класс **java.util.Collections** содержит большое количество статических методов, предназначенных для манипулирования коллекциями.

<T> void copy(List<? super T> dest, List<? extends T> src) — копирует все элементы из одного списка в другой;

boolean disjoint(Collection<?> c1, Collection<?> c2) — возвращает **true**, если коллекции не содержат одинаковых элементов;

<T> List<T> emptyList(), <K, V> Map<K, V> emptyMap(), <T> Set<T> emptySet() — возвращают пустой список, карту отображения и множество соответственно;

<T> void fill(List<? super T> list, T obj) — заполняет список заданным элементом;

int frequency(Collection<?> c, Object o) — возвращает количество вхождений в коллекцию заданного элемента;

<T> boolean replaceAll(List<T> list, T oldVal, T newVal) — заменяет все заданные элементы новыми;

void reverse(List<?> list) — «переворачивает» список;

void rotate(List<?> list, int distance) — сдвигает список циклически на заданное число элементов;

void shuffle(List<?> list) — перетасовывает элементы списка;

singleton(T o), singletonList(T o), singletonMap(K key, V value) — создают множество, список и карту отображения, позволяющие добавлять только один элемент;

<T extends Comparable<? super T>> void sort(List<T> list),
<T> void sort(List<T> list, Comparator<? super T> c) — сортировка списка естественным порядком и с использованием **Comparable** или **Comparator** соответственно;

void swap(List<?> list, int i, int j) — меняет местами элементы списка, стоящие на заданных позициях;

<T> List<T> unmodifiableList(List<? extends T> list) — возвращает ссылку на список с запрещением его модификации. Аналогичные методы есть для всех коллекций.

```
/* # 36 # применение некоторых алгоритмов # AlgorithmMain.java */
```

```
package by.epam.learn.collection;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class AlgorithmMain {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList();
        Collections.addAll(list, 1, 2, 3, 4, 5);
        Collections.shuffle(list);
        System.out.println(list);
        Collections.sort(list);
        System.out.println(list);
        Collections.reverse(list);
        System.out.println(list);
        Collections.rotate(list, 3);
        System.out.println(list);
        System.out.println("min: " + Collections.min(list));
        System.out.println("max: " + Collections.max(list));
        List<Integer> singletonList = Collections.singletonList(777);
        System.out.println(singletonList);
        //singletonList.add(21); // runtime error
    }
}
```

В результате будет выведено:

```
[5, 3, 4, 1, 2]
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
[3, 2, 1, 5, 4]
min: 1
max: 5
[777]
```

Вопросы к главе 11

1. Назвать основные интерфейсы коллекций. Какие бывают коллекции?
2. В чем особенности разных видов коллекций? Когда и какие коллекции следует применять?
3. Сравнить **ArrayList** и **LinkedList**.
4. Сравнить **HashMap** и **Hashtable**.
5. Как устроены **HashSet**, **TreeMap**, **TreeSet**.
6. Принцип работы и реализации **HashMap**. Изменения **HashMap** в java 8.
7. Чем отличается **ArrayList** от **Vector**?
8. Особенности интерфейса **Set**.
9. Как добавляются объекты в **HashSet**?
10. Какими способами можно отсортировать коллекцию? Привести три способа.
11. Как правильно удалить элемент из коллекции при итерации в цикле?
12. Как правильно удалить элемент из **ArrayList** (или другой коллекции) при поиске этого элемента в цикле?
13. Коллекции из пакета **java.util.concurrent**. Их особенности.
14. Что происходит при добавлении в **ArrayList** нового элемента и как это реализовано?
15. Метод для преобразования потокобезопасной коллекции в потокобезопасную.
16. Написать метод, в котором проверяется **HashMap** на наличие в нем некоторого значения, и его извлечения, если такого значения нет, надо добавить значение с пустой строкой и ее вернуть. Написать код, чтобы он был как можно более эффективным.
17. Какие коллекции более «быстрые» — *legacy* (**Vector**, **Hashtable**) или из пакета **java.util.concurrent**?
18. Если в коллекцию часто добавлять элементы и удалять, какую лучше использовать? Почему? Как они устроены?
19. Как быстро получить копию коллекции? Записать код преобразования.
20. Чем **Stream** отличается от коллекции?
21. Промежуточные и терминальные операции в *stream*.
22. Методы: **map()** vs **flatMap()** в *stream*.
23. Что такое потоковая обработка данных?

Задания к главе 11

Вариант А

1. Ввести строки из файла, записать в список. Вывести строки в файл в обратном порядке.
2. Ввести число, занести его цифры в стек. Вывести число, у которого цифры идут в обратном порядке.

3. Создать список из элементов каталога и его подкаталогов.
4. Занести стихотворения одного автора в список. Провести сортировку по возрастанию длин строк.
5. Задать два стека, поменять информацию местами.
6. Определить множество на основе множества целых чисел. Создать методы для определения пересечения и объединения множеств.
7. Списки, стеки или очереди $T(1..n)$ и $U(1..n)$ содержат результаты n -измерений тока и напряжения на неизвестном сопротивлении R . Найти приближенное число R методом наименьших квадратов.
8. С использованием множества выполнить попарное суммирование произвольного конечного ряда чисел по следующим правилам: на первом этапе суммируются попарно рядом стоящие числа, на втором этапе суммируются результаты первого этапа и т.д. до тех пор, пока не останется одно число.
9. Сложить два многочлена заданной степени, если коэффициенты многочленов хранятся в объекте **HashMap**.
10. Умножить два многочлена заданной степени, если коэффициенты многочленов хранятся в различных списках.
11. Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные — в начало списка.
12. Ввести строки из файла, записать в список **ArrayList**. Выполнить сортировку строк, используя метод **sort()** из класса **Collections**.
13. Задана строка, состоящая из символов «(», «)», «[», «]», «{», «}». Проверить правильность расстановки скобок. Использовать стек.
14. Задан файл с текстом на английском языке. Выделить все различные слова. Слова, отличающиеся только регистром букв, считать одинаковыми. Использовать класс **HashSet**.
15. Задан файл с текстом на английском языке. Выделить все различные слова. Для каждого слова подсчитать частоту его встречаемости. Слова, отличающиеся регистром букв, считать различными. Использовать класс **HashMap**.
16. Заполнить **HashMap** 10 объектами **<Integer, String>**. Найти строки у которых *ключ* > 5 . Если *ключ* = 0, вывести строки через запятую. Перемножить все ключи, где длина строки > 5 .
17. Написать функцию, которая получала бы итераторы на начало и конец отсортированного **List** и заданный символ. Возвращать функция должна начало и конец диапазона, строки в котором начинаются с заданного символа.

Вариант В

1. В кругу стоят N человек, пронумерованных от 1 до N . При ведении счета по кругу вычеркивается каждый второй человек, пока не останется один. Составить две программы, моделирующие процесс. Одна из программ должна использовать класс **ArrayList**, а вторая — **LinkedList**. Какая из двух программ работает быстрее? Почему?

2. Задан список целых чисел и некоторое число X . Не используя вспомогательных объектов и методов сортировки и не изменяя размера списка, переставить элементы списка так, чтобы сначала шли числа, не превосходящие X , а затем числа, больше X .
3. Написать программу, осуществляющую сжатие английского текста. Построить для каждого слова в тексте оптимальный префиксный код по алгоритму Хаффмена. Использовать класс **PriorityQueue**.
4. Реализовать класс **Graph**, представляющий собой неориентированный граф. В конструкторе класса передается количество вершин в графе. Методы должны поддерживать быстрое добавление и удаление ребер.
5. На базе коллекций реализовать структуру хранения чисел с поддержкой следующих операций:
 - добавление/удаление числа;
 - поиск числа, наиболее близкого к заданному (т.е. модуль разницы минимален).
6. Реализовать класс, моделирующий работу N -местной автостоянки. Машина подъезжает к определенному месту и едет вправо, пока не встретится свободное место. Класс должен поддерживать методы, обслуживающие приезд и отъезд машины.
7. Во входном файле хранятся две разреженные матрицы — A и B . Построить циклически связанные списки CA и CB , содержащие ненулевые элементы соответственно матриц A и B . Просматривая списки, вычислить: а) сумму $S = A + B$; б) произведение $P = A \times B$.
8. Во входном файле хранятся наименования некоторых объектов. Построить список Z , элементы которого содержат наименования и шифры данных объектов, причем элементы списка должны быть упорядочены по возрастанию шифров. Затем «сжать» список Z , удаляя дублирующие наименования объектов.
9. Во входном файле расположены два набора положительных чисел; между наборами стоит отрицательное число. Построить два списка $C1$ и $C2$, элементы которых содержат соответственно числа 1-го и 2-го набора таким образом, чтобы внутри одного списка числа были упорядочены по возрастанию. Затем объединить списки $C1$ и $C2$ в один упорядоченный список, изменяя только значения полей ссылочного типа.
10. Во входном файле хранится информация о системе главных автодорог, связывающих г. Полоцк с другими городами Беларуси. Используя эту информацию, построить дерево, отображающее систему дорог республики, а затем, продвигаясь по дереву, определить минимальный по длине путь из г. Полоцка в другой заданный город. Предусмотреть возможность сохранения дерева в виртуальной памяти.
11. Один из способов шифрования данных, называемый «двойным шифрованием», заключается в том, что исходные данные при помощи некоторого

- преобразования последовательно шифруются на некоторые два ключа — K1 и K2. Разработать и реализовать эффективный алгоритм, позволяющий находить ключи K1 и K2 по исходной строке и ее зашифрованному варианту. Проверить, оказался ли разработанный способ действительно эффективным, протестировав программу для случая, когда оба ключа являются 20-битными (время ее работы не должно превосходить одной минуты).
12. На плоскости задано N точек. Вывести в файл описания всех прямых, которые проходят более чем через одну точку из заданных. Для каждой прямой указать, через сколько точек она проходит. Использовать класс **HashMap**.
 13. На клетчатой бумаге нарисован круг. Вывести в файл описания всех клеток, целиком лежащих внутри круга, в порядке возрастания расстояния от клетки до центра круга. Использовать класс **PriorityQueue**.
 14. На плоскости задано N отрезков. Найти точку пересечения двух отрезков, имеющую минимальную абсциссу. Использовать класс **TreeMap**.
 15. На клетчатом листе бумаги закрашена часть клеток. Выделить все различные фигуры, которые образовались при этом. Фигурой считается набор закрашенных клеток, достижимых друг из друга при движении в четырех направлениях. Две фигуры являются различными, если их нельзя совместить поворотом на угол, кратный 90 градусам, и параллельным переносом. Используйте класс **HashSet**.
 16. Дана матрица из целых чисел. Найти в ней прямоугольную подматрицу, состоящую из максимального количества одинаковых элементов. Использовать класс **ArrayDeque**.
 17. Реализовать структуру «черный ящик», хранящую множество чисел и имеющую внутренний счетчик K , изначально равный нулю. Структура должна поддерживать операции добавления числа в множество и возвращение K -го по минимальности числа из множества.
 18. На прямой гоночной трассе стоит N автомобилей, для каждого из которых известны начальное положение и скорость. Определить, сколько произойдет обгонов.
 19. На прямой гоночной трассе стоит N автомобилей, для каждого из которых известны начальное положение и скорость. Вывести первые K обгонов.

Тестовые задания к главе 11

Вопрос 11.1.

Дан класс:

```
public record Order(long orderId, double amount) {
}
```

и фрагмент кода:

```
List<Order> orders = List.of(new Order(1, 50), new Order(5, 70), new Order(7, 50));
orders.stream()
    .collect(Collectors.groupingBy(Order::amount))
    .forEach((source, r) -> System.out.print(source + " "));
```

Что будет результатом? (выбрать один)

- a) 70.0 50.0
- b) 50.0 70.0 50.0
- c) 1, 50.0 5, 70.0 7, 50.0
- d) 5, 70.0 7, 50.0
- e) ничего не будет выведено

Вопрос 11.2.

Дан класс:

```
class Order {
    long orderId;
    double amount;
    public Order(long orderId, double amount) {
        this.orderId = orderId;
        this.amount = amount;
    }
    public String toString() {
        return orderId + ", " + amount ;
    }
}
```

и фрагмент кода:

```
List<Order> orders = Arrays.asList(new Order(1, 50), new Order(5, 70), new Order(7, 70));
Order order = orders.stream()
    .reduce(new Order(4, 0), (p1, p2) -> new Order(p1.orderId, p1.amount += p2.amount));
System.out.print(order);
```

Что будет результатом? (выбрать один)

- a) 4, 0.0
- b) 4, 190.0
- c) 17, 0.0
- d) 17, 190.0

Вопрос 11.3.

Дан класс:

```
class Order {
    long orderId;
    double amount;
```

```

public Order(long orderId, double amount) {
    this.orderId = orderId;
    this.amount = amount;
}
public String toString() {
    return orderId + ", " + amount ;
}
}

```

и фрагмент кода:

```

List<Order> orders = Arrays.asList(new Order(1, 50), new Order(5, 70), new Order(7, 70));
orders.stream()
    .reduce((p1, p2) -> p1.amount > p2.amount ? p1 : p2)
    .ifPresent(System.out::println);

```

Что будет результатом? (выбрать один)

- a) ничего не будет выведено
- b) 5, 70.0
- c) 1, 50.0
- d) 7, 70.0

Вопрос 11.4.

Дан класс:

```

public record Item(String name, double price) {
}

```

и фрагмент кода:

```

List<Item> items = List.of(new Item("Jeans", 20), new Item("Socks", 10), new Item("Jacket", 30));
var value = items.stream()
    .filter(s -> s.name().endsWith("s"))
    .mapToDouble(Item::price)
    .average()
    .getAsDouble();
System.out.print(value);

```

Что будет результатом? (выбрать один)

- a) compilation fails
- b) 0.0
- c) 15.0
- d) 20.0

Вопрос 11.5.

Дан фрагмент кода:

```
Map<String, Integer> map = new HashMap<>();
map.compute("y", (k, v) -> v==null ? 1 : 0);
map.compute("z", (k, v) -> v==null ? 2 : 0);
map.computeIfPresent("z", (k, v) -> v!=null ? 3 : 0);
map.computeIfAbsent("y", v -> v!=null ? 4 : 0);
System.out.println(map.values());
```

Что будет результатом? (выбрать один)

- a) [4, 3]
- b) [1, 3]
- c) [2, 4]
- d) [2, 4]
- e) runtime error

Вопрос 11.6.

Дан фрагмент кода:

```
List<Integer> integers = List.of(1, 2, 3, 1, 7);
boolean res = integers.stream()
    .// line 1
System.out.print(res);
```

Какой фрагмент кода следует вставить вместо комментария *line 1*, чтобы в консоль было выведено значение **true**? (выбрать один)

- a) noneMatch(i -> i == 2);
- b) allMatch(i -> i == 2);
- c) anyMatch(i -> i == 2);
- d) findFirst().

ПОТОКИ ВЫПОЛНЕНИЯ

Соседняя очередь всегда движется быстрее.

Наблюдение Этторе

*Как только вы перейдете в другую очередь,
ваша начнет двигаться быстрее.*

Наблюдение О'Брайена

Класс Thread и интерфейс Runnable

К большинству современных распределенных приложений (Rich Client) и веб-приложений (Thin Client) выдвигаются требования одновременной поддержки многих пользователей, каждому из которых выделяется отдельный поток, а также разделения и параллельной обработки информационных ресурсов. Потоки — средство, которое помогает организовать одновременное выполнение нескольких задач, каждой в независимом потоке. Потоки представляют собой экземпляры классов, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существует три способа создания и запуска потока: на основе расширения класса **Thread**, реализации интерфейсов **Runnable** или **Callable**.

// # 1 # расширение класса Thread # WalkThread.java

```
package by.epam.learn.thread;
public class WalkThread extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 7; i++) {
                System.out.println("Walk " + i);
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } finally {
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

При реализации интерфейса **Runnable** необходимо определить его единственный абстрактный метод **run()**. Например:

```
/* # 2 # реализация интерфейса Runnable # TalkThread.java */

package by.epam.learn.thread;
public class TalkThread implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 7; i++) {
                System.out.println("Talk -->" + i);
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } finally {
            System.out.println(Thread.currentThread().getName());
        }
    }
}

package by.epam.learn.thread;
public class BaseThreadMain {
    public static void main(String[] args) {
        WalkThread walk = new WalkThread(); // new thread object
        Thread talk = new Thread(new TalkThread()); // new thread object
        talk.start(); // start of thread
        walk.start(); // start of thread
        // TalkThread t = new TalkThread(); just an object, not a thread
        // t.run(); or talk.run();
        // method will execute, but thread will not start!
    }
}
```

Запуск двух потоков для объектов классов **WalkThread** непосредственно и **TalkThread** через инициализацию экземпляра **Thread** приводит к выводу строк: **Walk n** и **Talk -->n**. Порядок вывода, как правило, различен при нескольких запусках приложения.

В конце работы каждого потока происходит вызов **Thread.currentThread().getName()**, обеспечивающий вывод на консоль имени потока, в котором произошел вызов. В данном случае это будут строки **Thread-1** и **Thread-2**. Имена даются потокам по умолчанию либо с помощью метода **setName(String name)** или конструктора потока. Статический метод **currentThread()** дает доступ к потоку, в котором он вызван.

Интерфейс **Runnable** не имеет метода **start()**, а только единственный метод **run()**. Поэтому для запуска такого потока, как **TalkThread**, следует создать

экземпляр класса **Thread** с передачей экземпляра **TalkThread** его конструктору. Однако при прямом вызове метода **run()** поток не запустится, выполнится только тело самого метода.

Жизненный цикл потока

При выполнении программы объект класса **Thread** может быть в одном из четырех основных состояний: «новый», «работоспособный», «неработоспособный» и «пассивный». При создании потока он получает состояние «новый» (**NEW**) и не выполняется. Для перевода потока из состояния «новый» в состояние «работоспособный» (**RUNNABLE**) следует выполнить метод **start()**, который вызывает метод **run()** — основной метод потока.

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного класса-перечисления **Thread.State**:

NEW — поток создан, но еще не запущен;

RUNNABLE — поток выполняется;

BLOCKED — поток блокирован;

WAITING — поток ждет окончания работы другого потока;

TIMED_WAITING — поток некоторое время ждет окончания другого потока или просто в ожидании истечения времени;

TERMINATED — поток завершен.

Получить текущее значение состояния потока можно вызовом метода **getState()**.

Поток переходит в состояние «неработоспособный» в режиме ожидания (**WAITING**) вызовом методов **join()**, **wait()** или методов ввода/вывода, которые предполагают задержку. Для задержки потока на некоторое время

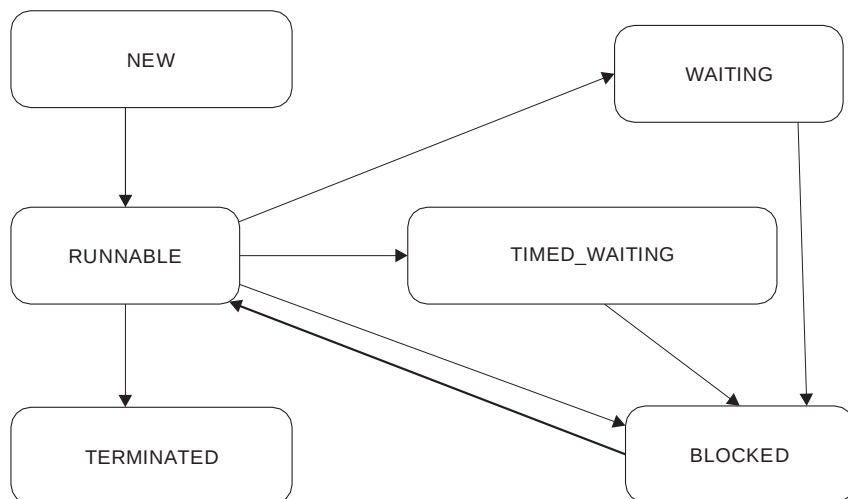


Рис. 12.1. Состояния потока

(в миллисекундах) можно перевести его в режим ожидания по времени (**TIMED_WAITING**) с помощью методов **sleep(long millis)**, **join(long timeout)** и **wait(long timeout)**. Вернуть потоку работоспособность после вызова метода **suspend()** можно методом **resume()** (*deprecated*-метод), а после вызова метода **wait()** — методами **notify()** или **notifyAll()**.

Когда поток просыпается, ему необходимо изменить состояние монитора объекта, на котором проходило ожидание. Для этого поток переходит в состояние **BLOCKED** и только после этого возвращается в работоспособное состояние.

Поток переходит в «пассивное» состояние (**TERMINATED**), если вызваны методы **interrupt()**, **stop()** (*deprecated*-метод) или метод **run()** завершил выполнение, и запустить его повторно уже невозможно. После этого, чтобы запустить поток, необходимо создать новый объект потока. Метод **interrupt()** успешно завершает поток, если он находится в состоянии «работоспособный». Если же поток в этот момент неработоспособен, например, находится в состоянии **TIMED_WAITING**, то метод инициирует исключение **InterruptedException**. Чтобы это не происходило, следует предварительно вызвать метод **isInterrupted()**, который проверит возможность завершения работы потока. При разработке не следует использовать методы принудительной остановки потока, так как возможны проблемы с закрытием ресурсов и другими внешними объектами.

Deprecated-методы класса **Thread**: **suspend()**, **resume()**, **stop()** и некоторые другие категорически запрещены к использованию, так как они не являются в полной мере потокобезопасными.

Перечисление TimeUnit

Представляет различные единицы измерения времени. В **TimeUnit** реализован ряд методов по преобразованию между единицами измерения и по управлению операциями ожидания в потоках в этих единицах. Используется для информирования методов, работающих со временем, о том, как интерпретировать заданный параметр времени.

Перечисление **TimeUnit** может представлять время в семи размерностях-значениях: **NANOSECONDS**, **MICROSECONDS**, **MILLISECONDS**, **SECONDS**, **MINUTES**, **HOURS**, **DAYS**.

Кроме методов преобразования единиц времени, представляют интерес методы управления потоками:

void timedWait(Object obj, long timeout) — выполняет метод **wait(long time)** для объекта **obj** класса **Object**, используя заданные единицы измерения;

void timedJoin(Thread thread, long timeout) — выполняет метод **join(long time)** на потоке **thread**, используя заданные единицы измерения;

void sleep(long timeout) — выполняет метод **sleep(long time)** класса **Thread**, используя заданные единицы измерения. Например:

```
TimeUnit.MINUTES.sleep(42);
```

Интерфейс Callable

В альтернативной системе управления потоками разработан механизм исполнителей, функции которого заключаются в запуске отдельных потоков и их групп, а также в управлении ими: принудительной остановке, контроле числа работающих потоков и планирования их запуска.

Интерфейс **Callable<V>** представляет поток, возвращающий значение вызывающему потоку. Определяет один метод **V call() throws Exception**, в код реализации которого и следует поместить решаемую задачу. Результат выполнения метода **V call()** может быть получен после окончания работы через экземпляр класса **Future<V>**, методами **V get()** или **V get(long timeout, TimeUnit unit)**. Эти методы останавливают выполнение потока, в котором они вызваны, поэтому вызывать их следует в момент, когда закончится выполнение потока **Callable**. Определить этот интервал затруднительно, и вместо ускорения работы приложения можно получить обратный результат. Перед извлечением результатов работы потока **Callable** можно проверить, завершилась ли задача успешно или была отменена, методами **isDone()** и **isCancelled()** соответственно.

Пусть стоит задача посчитать сумму значений элементов целочисленного списка с помощью интерфейса **Callable**. Решить эту задачу с помощью Stream API очень просто:

```
List<Integer> listInt = List.of(1, 10, 100, 1_000, 10_000);
int sum1 = listInt.stream().mapToInt(x -> x).sum();
// or
int sum2 = listInt.stream().reduce(0, (x, y) -> x + y);
System.out.println(sum1 + " " + sum2); //output: 11111 11111
```

Но если список состоит из сотен миллиардов элементов, то решать такую задачу следует с применением потоков.

```
/* # 3 # поток с возвращением результата # ActionCallable.java */
```

```
package by.epam.learn.thread.call;
import java.util.List;
import java.util.concurrent.Callable;
public class ActionCallable implements Callable<Integer> {
    private List<Integer> integers;
    public ActionCallable(List<Integer> integers) {
        this.integers = integers;
    }
    @Override
    public Integer call() {
        int sum = 0;
        for (int number : integers) {
            sum += number;
        }
        return sum;
    }
}
```

В Java 5 добавлен механизм управления заданиями, основанный на возможностях интерфейса **Executor** и его наследников **ExecutorService**, **ScheduledExecutorService**, включающих организацию запуска потоков и их групп, а также способы их планирования, управления, отслеживания прогресса и завершения асинхронных задач. Все эти задачи можно было сделать и раньше, но это выполнялось либо самим программистом, либо с привлечением сторонних библиотек. Поэтому был определен наиболее распространенный набор задач и реализован в возможностях **ExecutorService**.

Класс **ExecutorService** методом **execute(Runnable task)** запускает традиционные потоки, методы же **submit(Callable<T> task)** и **submit(Runnable task)** запускают потоки как с возвращаемым значением, так и классические. Несколько потоков можно запустить методом **invokeAll(Collection<? extends Callable<T>> tasks)**. Метод **shutdown()** прекращает действие самого исполнителя после того, как все запущенные им ранее потоки отработают, и не даст запустить новые, сгенерировав при этом исключение **RejectedExecutionException**. Метод **shutdownNow()** останавливает работу сервиса и удаляет все запущенные на объекте **ExecutorService** задачи-потоки.

```
/* # 4 # запуск потока объектом ExecutorService # ActionMain.java */
```

```
package by.epam.learn.thread.call;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
public class ActionMain {
    public static void main(String[] args) {
        List<Integer> list = IntStream.range(0, 1_000)
            .boxed()
            .collect(Collectors.toList());
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Integer> future = executor.submit(new ActionCallable(list));
        executor.shutdown();// stops service but not thread
        // executor.submit(new Thread());/* attempt to start will throw
            an exception */
        // executor.shutdownNow(); // stops service and all running threads
        try {
            System.out.println(future.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Результатом будет:

499500

При использовании **ExecutorService** метод **shutdown()** обязателен к вызову, иначе приложение не завершит свою работу.

Вызов метода **boolean awaitTermination(long timeout, TimeUnit unit)** останавливает поток, в котором вызван, и по истечении времени возвращает **true**, если все потоки, запущенные объектом **ExecutorService**, завершили свою работу, или **false** — если нет. Статические методы класса **Executors** определяют правила запуска потоков: **newSingleThreadExecutor()** позволяет исполнителю запускать только один поток, **newFixedThreadPool(int numThreads)** — число потоков не более, чем указано в параметре **numThreads**, ставя другие потоки в очередь ожидания окончания уже запущенных потоков, **newScheduledThreadPool()** — запуск по расписанию.

Механизм Fork\Join

В Java 7 появилась еще одна имплементация **ExecutorService** класс планировщик потоков **ForkJoinPool** с поддержкой аппаратного параллелизма для запуска задач, реализующих функционал абстрактного класса **ForkJoinTask**, но также способный запускать потоки, не являющиеся **ForkJoinTask**. Планировщик **ForkJoinPool** распределяет задания между запущенными потоками. Реализуется стратегия «разделяй и властвуй»: очень большая задача делится на более мелкие подзадачи, которые, в свою очередь, могут делиться на еще более мелкие, рекурсивно, до тех пор, пока не будет получена маленькая задача, которая уже может решаться непосредственно. Далее весь пул мелких задач выполняется параллельно на процессорных ядрах. Результаты каждой задачи «объединяются» при получении общего результата. Все работающие потоки, если для них нет заданий, пытаются найти и выполнить задачи, созданные другими активными потоками. Этот механизм еще называют «воровство работы», что позволяет использовать ограниченное число потоков более производительного по сравнению с другими реализациями **ExecutorService**. Все потоки, созданные **ForkJoinPool**, запущены как потоки-демоны.

Алгоритм стратегии можно условно представить в виде:

```
if (размер_задачи < пороговое_значение) {
    1. решить задачу напрямую, так как она мелкая
} else {
    1. разбить задачу на подзадачи
    2. рекурсивно решить каждую задачу
    3. объединить результаты
}
```

Механизм **Fork\Join** упрощает создание потоков, а также использует несколько процессоров, за счет чего сокращается время вычислений и повышается производительность приложения в целом при решении крупных задач, таких как обработка больших массивов данных. Этот механизм позволяет

управлять большим числом задач и относительно небольшим числом потоков, создаваемых и контролируемых в **ForkJoinPool**.

Абстрактный класс **ForkJoinTask** представляет основные **final**-методы:

ForkJoinTask <V> fork() — отправляет задачу для асинхронного выполнения. Этот метод возвращает текущий объект **ForkJoinTask**, а вызывающий его поток продолжает работать;

V join() — ожидает выполнения задачи и возвращает результат;

V invoke() — объединяет **fork()** и **join()** в одном вызове. Он запускает задачу, ожидает ее завершения и возвращает результат. Существует статический метод **invokeAll()** для запуска нескольких задач одновременно.

Класс **RecursiveTask<V>** для задач, которые возвращают значения, и по своей сути аналогичен интерфейсу **Callable<E>** с его методом **E call()**. Определен абстрактный метод **protected V compute()**, не предназначенный для внешних вызовов и содержащий алгоритм задачи и условия создания и запуска подзадач.

```
/* # 5 # fork/join для задач с возвращаемым значением # SumRecursiveTask.java
# ForkJoinTaskMain.java */
```

```
package by.epam.learn.thread.fork;
import java.util.List;
import java.util.concurrent.RecursiveTask;
public class SumRecursiveTask extends RecursiveTask<Long> {
    private List<Long> longList;
    private int begin;
    private int end;
    public static final long THRESHOLD = 10_000;
    public SumRecursiveTask(List<Long> longList) {
        this(longList, 0, longList.size());
    }
    private SumRecursiveTask(List<Long> longList, int begin, int end) {
        this.longList = longList;
        this.begin = begin;
        this.end = end;
    }
    @Override
    protected Long compute() {
        int length = end - begin;
        long result = 0;
        if (length <= THRESHOLD) {
            for (int i = begin; i < end; i++) {
                result += longList.get(i);
            }
        } else {
            int middle = begin + length / 2;
            SumRecursiveTask taskLeft = new SumRecursiveTask(longList, begin, middle);
            taskLeft.fork(); // run async
```

```

        SumRecursiveTask taskRight = new SumRecursiveTask(longList, middle, end);
        taskRight.fork();//or compute()
        Long leftSum = taskLeft.join();
        Long rightSum = taskRight.join();
        result = leftSum + rightSum;
    }
    return result;
}
}
package by.epam.learn.thread.fork;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.stream.Collectors;
import java.util.stream.LongStream;
public class ForkJoinTaskMain {
    public static void main(String[] args) {
        int end = 1_000_000;
        List<Long> numbers = LongStream.range(0, end)
            .boxed()
            .collect(Collectors.toList());
        ForkJoinTask<Long> task = new SumRecursiveTask(numbers);
        long result = new ForkJoinPool().invoke(task);
        System.out.println(result);
    }
}

```

Результат:

499999500000

При слишком низком отношении значения порога **THRESHOLD** к общему числу обрабатываемых данных, создается слишком много разветвлений потоков, поэтому возникает опасность переполнения стека виртуальной машины.

Возможности **Stream API** позволяют переписать реализацию метода **compute()** с применением лямбда выражений:

```

protected Long compute() {
    int length = end - begin;
    long result = 0;
    if (length <= THRESHOLD) {
        for (int i = begin; i < end; i++) {
            result += longList.get(i);
        }
    } else {
        int middle = begin + length / 2;
        List<SumRecursiveTask> tasks = new ArrayList<>();
        tasks.add(new SumRecursiveTask(longList, begin, middle));
        tasks.add(new SumRecursiveTask(longList, middle, end));
    }
}

```

```
tasks.stream().forEach(RecursiveTask::fork);
result = tasks.stream()
    .map(RecursiveTask::join)
    .reduce((r1, r2) -> r1 + r2)
    .orElse(0L);
}
return result;
}
```

При необходимости можно одну задачу разбивать не на две, а на большее число подзадач.

Класс **RecursiveAction** применяется для задач, которые не возвращают значений, и по своей сути аналогичен интерфейсу **Runnable** с его методом **void run()**. Определен абстрактный метод **protected void compute()**, также не предназначенный для внешних вызовов и определяющий алгоритм решения задачи и разбиения на более мелкие.

```
/* # 6 # fork/join для задач без возвращаемого значения # UnaryAction.java
# UnaryActionMain.java */
```

```
package by.epam.learn.thread.fork;
import java.util.List;
import java.util.concurrent.RecursiveAction;
import java.util.function.UnaryOperator;
public class UnaryAction<T> extends RecursiveAction {
    private List<T> subjectList;
    private UnaryOperator<T> operator;
    private int begin;
    private int end;
    private static final int THRESHOLD = 100000;
    public UnaryAction(List<T> subjectList, UnaryOperator<T> operator, int begin, int end) {
        this.operator = operator;
        this.subjectList = subjectList;
        this.begin = begin;
        this.end = end;
    }
    public UnaryAction(List<T> subjectList, UnaryOperator<T> operator) {
        this(subjectList, operator, 0, subjectList.size());
    }
    @Override
    protected void compute() {
        if (end - begin < THRESHOLD) {
            System.out.printf("from %d, to %d - thread %s\n",
                begin, end, Thread.currentThread().getName());
            for (int i = begin; i < end; i++) {
                subjectList.set(i, operator.apply(subjectList.get(i)));
            }
        } else {
```

```

        int oneThird = (begin + end) / 3;
        int middle = (begin + end) / 2;
        invokeAll(new UnaryAction<T>(subjectList, operator, begin, middle),
                  new UnaryAction<T>(subjectList, operator, middle, end));
    }
}
}
package by.epam.learn.thread.fork;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
public class UnaryActionMain {
    public static void main(String[] args) {
        List<Double> numbers = IntStream.range(1, 1_000_000)
            .asDoubleStream()
            .boxed()
            .parallel()
            .peek(n -> System.out.printf("%5.2f; ", n))
            .collect(Collectors.toList());
        new UnaryAction<>(numbers, d -> Math.sqrt(d)).invoke();
        numbers.stream().forEach(r -> System.out.printf("%7.4f %n ", r));
    }
}

```

Параллелизм

В Java 8 вместе со **Stream API** было введено понятие параллелизма, которое использует **ForkJoinPool** неявно. Задача обработки будет выполняться параллельно вызовом метода **parallel()** сразу же после создания *stream* или создавая его непосредственно методом **parallelStream()**.

```
/* # 7 # явный параллелизм # ParallelMain.java */
```

```

package by.epam.learn.thread.call;
import java.util.stream.LongStream;
public class ParallelMain {
    public static void main(String[] args) {
        long result;
        result = LongStream.range(0, 1_000_000_000)
            .boxed()
            .parallel()
            .map(x -> x / 7)
            .peek(v -> System.out.println(Thread.currentThread().getName()))
            .reduce((x,y)-> x + (int) (3 * Math.sin(y)))
            .get();
        System.out.println(result);
    }
}

```

В процессе выполнения приложение выводит в консоль имена потоков обработки в виде:

ForkJoinPool.commonPool-worker-5

ForkJoinPool.commonPool-worker-11

Если убрать вызов метода **parallel()**, то в процессе выполнения будет выводиться только имя основного потока **main**, и никакого разделения на подзадачи не произойдет.

Пользовательский пул потоков, созданный на основе **Runnable** или **Callable**, непосредственно сделать выполняемым параллельно не существует возможности, однако его можно обернуть в **ForkJoinPool**.

```
/* # 8 # неявный параллелизм # ActionParallelMain.java */
```

```
package by.epam.learn.thread.call;
import java.util.concurrent.*;
import java.util.stream.IntStream;
public class ActionParallelMain {
    public static void main(String[] args) {
        long sec = System.currentTimeMillis();
        Callable<Integer> task = () -> IntStream.range(0, 1_000_000_000)
            .boxed()
            .parallel()
            .map(x -> x / 3)
            .peek(th -> System.out.println(Thread.currentThread().getName()))
            .reduce((x, y) -> x + (int)(3 * Math.sin(y)))
            .get();
        ForkJoinPool pool = new ForkJoinPool(8);// 8 processors
        try {
            int result = pool.submit(task).get();
            System.out.println(result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
        System.out.println((System.currentTimeMillis() - sec) / 1000.);
    }
}
```

Запуск кода покажет, что используются потоки вида:

ForkJoinPool-1-worker-21

ForkJoinPool-1-worker-7

Таким образом, параллельный поток объявляет **ForkJoin** в качестве родительского, а не обычного потока. Вследствие чего **ForkJoinPool.commonPool** не используется.

Параллельные потоки, запущенные в разных частях приложения, будут конкурировать между собой, поэтому следует ограничивать размеры пула.

Timer и поток TimerTask

Некоторые виды задач требуют периодичности выполнения. Например, рассылка почтовых уведомлений или сообщений об истечении срока действия пароля, лицензии. Уведомления о совершении периодических платежей типа абонентской платы, квартплаты и др. Запуск на выполнение задач в заданное время или с определенной периодичностью представляет собой типичное использование потоков.

В задаче контроля и устранения утечек из пулов соединений с БД поток-таймер может выполнять несколько различных задач. Основная — проверка целостности пула: если число активных соединений не совпадает с заявленным размером пула, то таймер может добавить новые соединения. Если стратегия пула зависит от его загруженности, то таймер может управлять числом активных соединений, а именно: закрывать часть соединений при низкой нагрузке на БД и увеличивать их число при повышении нагрузки, не выходя при этом за предельные значения размеров пула.

Для запуска потоков по расписанию в пакете **java.util** — абстрактный класс **TimerTask**, имплементирующий интерфейс **Runnable**. Планировкой выполнения потоков занимается класс **java.util Timer**, методы которого работают с объектом **TimerTask**.

Для запуска потоков используются методы **schedule(params)** и **scheduleAtFixedRate(params)**.

```
/* # 9 # поток, запускаемый через определенный промежуток времени
# TimerCounter.java # TimerMain.java */
```

```
package by.epam.learn.thread.timertask;
import java.util.TimerTask;
import java.util.concurrent.TimeUnit;
public class TimerCounter extends TimerTask {
    private static int i;
    @Override
    public void run() {
        System.out.print(++i);
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("\t" + i);
    }
}
package by.epam.learn.thread.timertask;
import java.util.Timer;
public class TimerMain {
```

```
public static void main(String[] args) {
    Timer timer = new Timer();
    timer.schedule(new TimerCounter(), 100, 3000);
}
}
```

Поток **TimerCounter** стартует через 100 миллисекунд и будет запускаться на выполнение каждые три секунды.

Прервать выполнение задания можно методом **cancel()** класса **Timer**. Удалить все прерванные задания из очереди таймера — методом **purge()**.

Управление приоритетами

Потоку можно назначить приоритет от **1** (константа **Thread.MIN_PRIORITY**) до **10** (**Thread.MAX_PRIORITY**) с помощью метода **setPriority(int newPriority)**. Получить значение приоритета потока можно с помощью метода **getPriority()**.

```
// # 10 # потоки с приоритетами # PriorityMain.java
```

```
package by.epam.learn.thread;
public class PriorityMain {
    public static void main(String[] args) {
        Thread walkMin = new Thread(new WalkThread(), "Min");
        Thread talkMax = new Thread(new TalkThread(), "Max");
        walkMin.setPriority(Thread.MIN_PRIORITY);
        talkMax.setPriority(Thread.MAX_PRIORITY);
        talkMax.start();
        walkMin.start();
    }
}
```

Для успешной демонстрации работы с приоритетами в классах **WalkThread** и **TalkThread**, следует увеличить число итераций, например, с 7 до 777, а также убрать задержки по времени.

Поток с более высоким приоритетом в данном случае, как правило, монополизует вывод на консоль.

Управление потоками

Приостановить (задержать) выполнение потока можно с помощью метода **static void sleep(int millis)** класса **Thread**. Поток переходит в состояние **TIMED_WAITING**. Иной способ состоит в вызове метода **static void yield()**, который отдает квант времени другому потоку, не мешая самому потоку работать и предоставляя возможность виртуальной машине переместить его в конец очереди других потоков.

Метод **join()** блокирует работу потока, в котором он вызван до тех пор, пока не будет закончено выполнение вызывающего метод потока или не истечет время ожидания при обращении к методу **join(long timeout)**. Такие же результаты позволяет получить замена метода **join()** класса **Thread** на метод **timedJoin(Thread thread, long timeout)** перечисления **TimeUnit**.

```
// # 11 # задержка потока методом join() # JoinThread.java # JoinMain.java

package by.bsu.learn.thread;
import java.util.concurrent.TimeUnit;
class JoinThread extends Thread {
    public void run() {
        System.out.println("Start");
        try {
            TimeUnit.MILLISECONDS.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("End");
    }
}

public class JoinMain {
    public static void main(String[] args) {
        new Thread(() -> {
            System.out.println("start 1");
            try {
                TimeUnit.MILLISECONDS.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("end 1");
        }).start();
        JoinThread thread = new JoinThread();
        thread.start();
        try {
            thread.join(); // or join(100)
            // or //TimeUnit.MILLISECONDS.timedJoin(thread, 100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("end of " + Thread.currentThread().getName());
    }
}
```

Возможно, будет выведено:

```
start 1
Start
End
end of main
end 1
```

Несмотря на вызов метода **join()** для потока **thread**, поток, стартовавший с помощью лямбда-выражения, будет независимо работать, в отличие от потока **main**, который сможет продолжить свое выполнение только по завершении потока **thread**.

Если вместо метода **join()** без параметров использовать версию **join(long timeout)**, то поток **main** будет остановлен только на указанный промежуток времени. При вызове **thread.join(5)** вывод будет другим:

```
start 1
Start
end of main
End
end 1
```

Вызов статического метода **yield()** для исполняемого потока должен приводить к приостановке потока на некоторый квант времени, чтобы другие потоки могли выполнять свои действия. В некоторых ситуациях поток может отдавать квант времени самому себе или вообще ничего не делать. Например, в случае потока с высоким приоритетом после обработки части пакета данных, когда следующая еще не готова, стоит уступить часть времени другим потокам. Польза **yield()** и приоритетов потока в оптимизации выполнения и взаимодействия потоков может оказаться не просто сомнительной, но и просто отрицательной. Если требуется надежная остановка потока, то следует применить другой способ.

```
// # 12 # задержка потока на квант времени # YieldMain.java
```

```
package by.epam.learn.thread;
public class YieldMain {
    public static void main(String[] args) {
        new Thread(() -> {
            System.out.println("start 1");
            Thread.yield();
            Thread.yield();
            System.out.println("end 1");
        }).start();
        new Thread(() -> {
            System.out.println("start 2");
            System.out.println("end 2");
        }).start();
    }
}
```

В результате может быть выведено:

```
start 1
start 2
end 2
end 1
```

Активизация метода **yield()** в коде метода **run()** первого объекта потока приводит к тому, что, возможно, первый поток будет остановлен на некоторый квант времени, что даст возможность другому потоку запуститься и выполнить свой код.

Потоки-демоны

Потоки-демоны работают в фоновом режиме вместе с программой, но представляют собой функциональность, которая не является важной для основной логики программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения, а его деятельность заключается в косвенном обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон. С помощью метода **setDaemon(boolean value)**, вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод **boolean isDaemon()** позволяет определить, является ли указанный поток демоном или нет.

Пусть существует следующий класс-поток:

```
/* # 13 # запуск и выполнение потока-демона # SimpleThread.java
# DaemonMain.java */
```

```
package by.epam.learn.thread;
import java.util.concurrent.TimeUnit;
public class SimpleThread extends Thread {
    public void run() {
        try {
            if (isDaemon()) {
                System.out.println("start of daemon thread");
                TimeUnit.MILLISECONDS.sleep(10);
            } else {
                System.out.println("start of normal thread");
            }
        } catch (InterruptedException e) {
            System.err.print(e);
        } finally {
            if (!isDaemon()) {
                System.out.println("normal thread completion");
            } else {
                System.out.println("daemon thread completion");
            }
        }
    }
}

package by.epam.learn.thread;
public class DaemonMain {
    public static void main(String[] args) {
        SimpleThread normal = new SimpleThread();
        SimpleThread daemon = new SimpleThread();
        daemon.setDaemon(true);
```

```

    daemon.start();
    normal.start();
    System.out.println("end of main");
}
}

```

В результате компиляции и запуска, возможно, будет выведено:

end of main

start of daemon thread

start of normal thread

normal thread completion

Поток-демон (из-за вызова метода **sleep(10)**) не успел выполнить свой код до завершения основного потока приложения, связанного с методом **main()**. Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода **main()**, не обращая внимания на то, что поток-демон еще работает. Если уменьшать время задержки потока-демона, то он может успеть завершить свое выполнение до окончания работы основного потока.

Потоки и исключения

В процессе функционирования потоки являются в общем случае независимыми друг от друга. Прямым следствием такой независимости будет корректное продолжение работы потока **main** после аварийной остановки запущенного из него потока после генерации исключения.

```
/* # 14 # генерация исключения в созданном потоке # ThreadExceptionMain.java */
```

```

package by.epam.learn.thread;
import java.util.concurrent.TimeUnit;
public class ThreadExceptionMain {
    public static void main(String[] args) {
        new Thread(()-> {
            if(Boolean.TRUE) {
                throw new RuntimeException();
            }
            System.out.println("end of Thread");
        }).start();
        try {
            TimeUnit.MILLISECONDS.sleep(20);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("end of main thread");
    }
}

```

«Основной» поток избавлен от необходимости обрабатывать исключения в порожденных потоках.

Верно и обратное: если «основной» поток прекратит свое выполнение из-за необработанного исключения, то это никак не скажется на работоспособности порожденного им потока.

```
/* # 15 # генерация исключения в потоке main # ThreadExceptionMain2.java */
```

```
package by.epam.learn.thread;
import java.util.concurrent.TimeUnit;
public class ThreadExceptionMain2 {
    public static void main(String[] args) {
        new Thread(() -> {
            try {
                TimeUnit.MILLISECONDS.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("end of Thread");
        }).start();
        try {
            TimeUnit.MILLISECONDS.sleep(20);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if(Boolean.TRUE) {
            throw new RuntimeException();
        }
        System.out.println("end of main thread");
    }
}
```

Атомарные типы и модификатор **volatile**

Все данные приложения находятся в основном хранилище данных. При запуске нового потока создается копия хранилища и именно ею пользуется этот поток. Изменения, произведенные в копии, могут не сразу находить отражение в основном хранилище, и наоборот. Для получения актуального значения следует прибегнуть к синхронизации. Наиболее простым приемом будет объявление поля класса с модификатором **volatile**. Данный модификатор вынуждает потоки производить действия по фиксации изменений достаточно быстро. То есть другой заинтересованный поток, скорее всего, получит доступ к уже измененному значению. Для базовых типов до 32 бит этого достаточно. При использовании со ссылкой на объект синхронизировано будет только значение самой ссылки, а не объект, на который она ссылается. Синхронизация ссылки будет эффективной в случае, если она указывает на перечисление, так как все элементы перечисления существуют

В единственном экземпляре. Решением проблемы с доступом к одному экземпляру из разных потоков является блокирующая синхронизация. Модификатор **volatile** обеспечивает неблокирующую синхронизацию.

Существует целая группа классов пакета **java.util.concurrent.atomic**, обеспечивающая неблокирующую синхронизацию. Атомарные классы созданы для организации неблокирующих структур данных. Классы атомарных переменных **AtomicInteger**, **AtomicLong**, **AtomicBoolean**, **AtomicReference** и др. фактически являются оболочкой для **volatile**, расширяют ее нотацию значений, полей и элементов массивов. Все атомарные классы являются изменяемыми, в отличие от соответствующих им классов-оболочек. При реализации классов пакета использовались эффективные атомарные инструкции машинного уровня, которые доступны на современных процессорах. В некоторых ситуациях могут применяться варианты внутреннего блокирования.

Экземпляры классов, например, **AtomicInteger** и **AtomicReference**, предоставляют доступ к единственной переменной соответствующего типа. Каждый класс также обеспечивает набор методов для этого типа. В частности, класс **AtomicInteger** — атомарные методы инкремента и декремента. Инструкции при доступе и обновлении атомарных переменных, в общем, следуют правилам для **volatile**.

Не стоит классы атомарных переменных использовать как замену соответствующих классов-оболочек.

Пусть имеется некоторая торговая площадка, представленная классом **Market**, работающая в непрерывном режиме и информирующая о разнонаправленных изменениях биржевого индекса (поле **index** типа **AtomicLong**) дважды за один цикл с интервалом до 500 миллисекунд. Изменения поля **index** фиксируются методом **addAndGet(long delta)** атомарного добавления переданного значения к текущему.

```
/* # 16 # класс с атомарным полем # Market.java */
```

```
package by.epam.learn.thread.atomic;
import java.util.Random;
import java.util.concurrent.atomic.AtomicLong;
public class Market extends Thread {
    private AtomicLong index;
    private Random generator = new Random();
    public Market(AtomicLong index) {
        this.index = index;
    }
    public AtomicLong getIndex() {
        return index;
    }
    @Override
    public void run() {
        try {
            while (true) {
                index.addAndGet(generator.nextInt(21) - 10);
            }
        }
    }
}
```

```

        Thread.sleep(generator.nextInt(500));
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

```

Объявлен класс **Broker**, запрашивающий значение поля **index** с заданным интервалом в миллисекундах для всех объектов этого класса.

```
/* # 17 # получатель значения атомарного поля # Broker.java */
```

```

package by.epam.learn.thread.atomic;
import java.util.concurrent.TimeUnit;
public class Broker extends Thread {
    private static Market market;
    private static final int PAUSE_IN_MILLIS = 500;
    public static void initMarket(Market market) {
        Broker.market = market;
    }
    @Override
    public void run() {
        try {
            while (true) {
                System.out.println("Current index: " + market.getIndex());
                TimeUnit.MILLISECONDS.sleep(PAUSE_IN_MILLIS);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Количество экземпляров класса **Broker** может быть любым, и они постоянно с заданным интервалом запрашивают текущее значение **index**.

```
/* # 18 # запуск потоков изменения атомарного поля и его отслеживания
несколькими потоками # AtomicMain.java */
```

```

package by.epam.learn.thread.atomic;
import java.util.concurrent.atomic.AtomicLong;
public class AtomicMain {
    private static final int NUMBER_BROKERS = 30;
    public static void main(String[] args) {
        Market market = new Market(new AtomicLong(100));
        Broker.initMarket(market);
        market.start();
        for (int i = 0; i < NUMBER_BROKERS; i++) {
            new Broker().start();
        }
    }
}

```

Атомарность поля обеспечивает получение экземплярами класса **Broker** идентичных текущих значений поля **index**.

Для сравнения атомарных и неатомарных следует заменить объявление

```
public class Market extends Thread {
    private AtomicLong index;
```

на

```
public class Market extends Thread {
    private Long index;
```

и оценить результаты работы приложения.

Методы **synchronized**

Нередко возникает ситуация, когда несколько потоков имеют доступ к некоторому объекту, проще говоря, пытаются использовать общий ресурс и начинают мешать друг другу. Более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в объект (поток, файл и т.д.). Для контролирования процесса записи может использоваться разделение ресурса с применением ключевого слова **synchronized**.

В качестве примера будет рассмотрен процесс записи информации в файл двумя конкурирующими потоками. В методе **main()** класса **SynchroMain** создаются два потока. В этом же методе создается экземпляр класса **CommonResource**, содержащий поле типа **FileWriter**, связанное с файлом на диске. Экземпляр **CommonResource** передается в качестве параметра обоим потокам. Первый поток записывает строку методом **writing()** в экземпляр класса **CommonResource**. Второй поток также пытается сделать запись строки в тот же самый объект **CommonResource**. Во избежание одновременной записи такие методы объявляются как **synchronized**. Синхронизированный метод изолирует объект, после чего он становится недоступным для других потоков. Изоляция снимается, когда поток полностью выполнит соответствующий метод. Другой способ снятия изоляции — вызов метода **wait()** из изолированного метода — будет рассмотрен позже.

В примере продемонстрирован вариант синхронизации файла для защиты от одновременной записи информации в файл двумя различными потоками.

```
/* # 19 # синхронизация записи информации в файл # UseFileThread.java
# CommonResource.java # SynchroMain.java */
```

```
package by.epam.learn.thread;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;
import java.util.concurrent.TimeUnit;
public class CommonResource implements AutoCloseable {
```

```

private FileWriter fileWriter;
public CommonResource(String file) throws IOException {
    fileWriter = new FileWriter(file, true);
}
public synchronized void writing(String info, int i) {
    try {
        fileWriter.append(info + i);
        System.out.print(info + i);
        TimeUnit.MILLISECONDS.sleep(new Random().nextInt(500));
        fileWriter.append("->" + info.charAt(0) + i + " ");
        System.out.print("->" + info.charAt(0) + i + " ");
    } catch (IOException | InterruptedException e) {
        System.err.print(e);
    }
}
@Override
public void close() throws IOException {
    if (fileWriter != null) {
        fileWriter.close();
    }
}
}
package by.epam.learn.thread;
public class UseFileThread extends Thread{
    private CommonResource resource;
    public UseFileThread(String name, CommonResource resource) {
        super(name);
        this.resource = resource;
    }
    public void run() {
        for (int i = 0; i < 5; i++) {
            resource.writing(this.getName(), i); // synchronized method call
        }
    }
}
package by.epam.learn.thread;
import java.io.IOException;
import java.util.concurrent.TimeUnit;
public class SynchroMain {
    public static void main(String[] args) {
        try(CommonResource resource = new CommonResource("data\\thread.txt")) {
            UseFileThread thread1 = new UseFileThread("First", resource);
            UseFileThread thread2 = new UseFileThread("Second", resource);
            thread1.start();
            thread2.start();
            TimeUnit.SECONDS.sleep(5);
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

В результате в файл будет, например, выведено:

**First0->F0 Second0->S0 First1->F1 Second1->S1 Second2->S2
Second3->S3 First2->F2 Second4->S4 First3->F3 First4->F4**

где видно, что метод **writing()** всегда отрабатывает полностью.

Код построен таким образом, что при отключении синхронизации метода **writing()**, в случае его вызова одним потоком, другой поток может вклиниться и произвести запись своей информации, несмотря на то, что метод не завершил запись, инициированную первым потоком.

Вывод в этом случае может быть, например, следующим:

**First0Second0->S0 Second1->S1 Second2->F0 First1->S2 Second3->
F1 First2->S3 Second4->F2 First3->F3 First4->S4 ->F4**

Инструкция **synchronized**

Синхронизировать объект можно не только при помощи методов с соответствующим модификатором, но и при помощи синхронизированного блока кода. В этом случае происходит блокировка объекта, указанного в инструкции **synchronized(reference){/*code*/}**, и он становится недоступным для других синхронизированных методов и блоков.

Такая синхронизация позволяет не создавать класс-оболочку для класса, к которому требуется организовать синхронизированный доступ. Также блок синхронизации позволяет сузить область синхронизации, т.е. вывести за пределы синхронизации код, в ней не нуждающийся. Обычные методы на синхронизацию внимания не обращают, поэтому ответственность за грамотную блокировку объектов ложится на программиста.

```
/* # 20 # блокировка объекта потоком # SynchroBlockMain.java */
```

```
package by.epam.learn.thread;
import java.util.concurrent.TimeUnit;
public class SynchroBlockMain {
    static int counter;
    public static void main(String[] args) {
        StringBuilder info = new StringBuilder();
        new Thread(() -> {
            synchronized (info) {
                do {
                    info.append('A');
                    System.out.println(info);
                    try {
                        TimeUnit.MILLISECONDS.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        })
```

```

    } while (counter++ < 2) ;
}
}).start();
new Thread(() -> {
    synchronized (info) {
        while(counter++ < 6) {
            info.append('Z');
            System.out.println(info);
        }
    }
}).start();
}
}

```

В результате компиляции и запуска, скорее всего (например, второй поток может заблокировать объект первым), будет выведено:

```

A
AA
AAA
AAAZ
AAAZZ
AAAZZZ

```

Один из потоков блокирует объект, и до тех пор, пока он не закончит выполнение блока синхронизации, в котором производится изменение значения объекта, ни один другой поток не может вызвать синхронизированный блок для этого объекта.

Если в коде убрать синхронизацию объекта **info**, то вывод будет другим, так как другой поток сможет получить доступ к объекту и изменить его раньше, чем первый закончит выполнение цикла.

Монитор

Контроль за доступом к объекту-ресурсу обеспечивает понятие монитора. Монитор экземпляра может иметь только одного владельца. При попытке конкурирующего доступа к объекту, чей монитор имеет владельца, желающий заблокировать объект-ресурс поток должен подождать освобождения монитора этого объекта, только после этого завладеть им и начать использование объекта-ресурса. Каждый экземпляр любого класса имеет монитор. *Final*-методы **wait()**, **wait(long inmillis)**, **notify()**, **notifyAll()** класса **Object** корректно срабатывают только на экземплярах, чей монитор уже кем-то захвачен. Статический **synchronized** метод захватывает монитор экземпляра класса **Class**, того класса, на котором он вызван. Существует в единственном экземпляре. Нестатический **synchronized** метод захватывает монитор экземпляра класса, на котором он вызван.

Механизм wait\notify

Эти *final*-методы не могут переопределяться и используются только в исходном виде. Вызываются только внутри синхронизированного блока или метода на объекте, монитор которого захвачен текущим потоком. Попытка обращения к данным методам вне синхронизации или на несинхронизированном объекте (со свободным монитором) приводит к генерации исключительной ситуации **IllegalMonitorStateException**. В примере #21 рассмотрено взаимодействие методов **wait()** и **notify()** при освобождении и возврате блокировки в **synchronized** блоке. Эти методы используются для управления потоками в ситуации, когда необходимо задать определенную последовательность действий без повторного запуска потоков.

Метод **wait()**, вызванный внутри синхронизированного блока или метода, останавливает выполнение текущего потока и освобождает от блокировки захваченный объект. Возвратить блокировку объекта потоку можно вызовом метода **notify()** для одного потока или **notifyAll()** для всех потоков. Если ожидающих потоков несколько, то после вызова метода **notify()** невозможно определить, какой поток из ожидающих потоков заблокирует объект. Вызов может быть осуществлен только из другого потока, заблокировавшего, в свою очередь, тот же самый объект.

Проиллюстрировать работу указанных методов можно с помощью примера, когда инициализация полей и манипуляция их значениями производится в различных потоках.

```
/* # 21 # взаимодействие wait() и notify() # Payment.java # PaymentMain.java */
```

```
package by.epam.learn.thread;
import java.util.Scanner;
public class Payment {
    private int amount;
    public synchronized void doPayment() {
        try {
            System.out.println("Start payment");
            while (amount <= 0) {
                this.wait();
            }
            // payment code
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Payment is closed");
    }
    public synchronized void init() {
        System.out.println("Init amount:");
        amount = new Scanner(System.in).nextInt();
        this.notify();
    }
}
```

```

    }
}
package by.epam.learn.thread;
import java.util.concurrent.TimeUnit;
public class PaymentMain {
    public static void main(String[] args) {
        Payment payment = new Payment();
        new Thread(() -> payment.doPayment()).start();

        try {
            TimeUnit.MILLISECONDS.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        payment.init();
    }
}

```

В результате компиляции и запуска при вводе корректного значения для инициализации поля **amount** будет запущен процесс проведения платежа.

Start payment

Init amount:

777

Payment is closed

Задержки потоков методом **sleep()** используются для точной демонстрации последовательности действий, выполняемых потоками. Если же в коде приложения убрать все блоки синхронизации, а также вызовы методов **wait()** и **notify()**, то результатом вычислений, скорее всего, будет ноль, так как вычисление будет произведено до инициализации полей объекта.

Однократный вызов метода **notify()** позволит захватить блокировку только одному потоку, остановленному методом **wait()**. Если к этому моменту методом **wait()** были остановлены другие потоки, то они так и останутся в состоянии **WAITING**.

Чтобы продемонстрировать эту ситуацию достаточно запустить несколько потоков **Payment**.

```

Payment payment = new Payment();
for (int i = 0; i < 5; i++) {
    new Thread(() -> payment.doPayment()).start();
}
try {
    TimeUnit.MILLISECONDS.sleep(100);
} catch (InterruptedException e) {
    e.printStackTrace();
}
payment.init();

```

В консоль тогда будет выведено:

```
Start payment
Start payment
Start payment
Start payment
Start payment
Init amount:
55
Payment is closed
```

То есть после вызова **notify()** блокировку перехватит только один случайно выбранный поток, а оставшиеся четыре так и останутся в режиме ожидания, из которого их уже не вывести.

Решить эту проблему достаточно легко, заменив вызов **notify()** в методе **init()** на вызов метода **notifyAll()**, это приведет к тому, что в момент вызова уведомления о возможном снятии блокировки получают все потоки, остановленные методом **wait()** на этом объекте. Разблокировка будет происходить не одномоментно, а последовательно в случайном порядке по мере завершения работы синхронизированных методов.

Решение, заключенное в классе **Payment** взаимодействием методов **doPayment()** и **init()**, представляет собой стандартное решение **Producer\Consumer**, которое можно представить в виде:

```
/* # 22 # поставщик/потребитель # ProducerConsumer.java */

public class ProducerConsumer {
    private boolean ready;
    public synchronized void consume() {
        while(!ready) {
            try {
                wait();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        ready = false;
    }
    public synchronized void produce() {
        ready = true;
        notify();
    }
}
```

Интерфейс `Lock` как альтернатива `synchronized`

Синхронизация ресурса ключевым словом **`synchronized`** накладывает достаточно жесткие правила на освобождение этого ресурса.

Дополнительные гибкие реализации моделей синхронизации представляют:

- интерфейс **`Lock`**, поддерживающий ограниченные ожидания снятия блокировки, прерываемые попытки блокировки, очереди блокировки и установку ожидания снятия нескольких блокировок посредством интерфейса **`Condition`**;
- класс семафор **`ReentrantLock`**, добавляющий ранее не существовавшую функциональность по отказу от попытки блокировки объекта с возможностью многократного повторения запроса на блокировку и отказа от нее;
- класс **`ReentrantReadWriteLock`** позволяет изменять объект только одному потоку, а читать в это время — нескольким.

Интерфейс **`Lock`** расширяет возможности блокирующей синхронизации. Появилась возможность провести проверку возможности блокировки, установить время ожидания блокировки и определить условия ее прерывания.

В том время как **`synchronized`** метод блокирует объект, на котором вызван, методы интерфейса **`Lock`** блокируют сам объект **`Lock`**.

Интерфейс также оптимизирует работу JVM с процессами конкурентирования за освобождаемые ресурсы.

Интерфейс **`Lock`** представляет методы:

`void lock()` — получает блокировку экземпляра **`ReentrantLock`**. Если экземпляр заблокирован другим потоком, то поток отключается и бездействует до освобождения экземпляра;

`void unlock()` — освобождает блокировку экземпляра **`Lock`**. Если текущий поток не является обладателем блокировки, генерируется исключение **`IllegalMonitorStateException`**.

Шаблонное применение этих методов после объявления экземпляра **`locking`** класса **`ReentrantLock`**, реализации интерфейса **`Lock`**:

```
try {
    locking.lock();
    // some code here
} finally {
    locking.unlock();
}
```

Данная конструкция копирует функциональность блока **`synchronized`**. Гибкость классу предоставляют другие методы:

`boolean tryLock()` — получает блокировку экземпляра **`Lock`**, если она свободна, и возвращает **`true`**. Если же блокировка выполнена другим потоком, то метод сразу возвращает **`false`**, поэтому этот метод обычно используется с оператором **`if`**, чтобы, например, отказаться от попытки блокировки и обойти участок кода:

```
if (locking.tryLock()) {
    try {
        // some code here
    } finally {
        locking.unlock();
    }
}
```

boolean tryLock(long timeout, TimeUnit unit) — получает блокировку экземпляра **Lock**, если она свободна, и возвращает **true**. Если экземпляр блокирован другим потоком, то метод приостанавливает поток на время **timeout**, и если блокировка становится возможна в течение этого интервала, то поток ее получает, если же блокировка недоступна, то метод возвращает **false**.

Метод **lock()** и оба метода **tryLock()** при успешном получении блокировки в одном и том же потоке увеличивают счетчик блокировок на единицу, что требует для освобождения блокировки вызова метода **unlock()**.

Интерфейс **Condition** предназначен для управления блокировкой. Ссылку на экземпляр можно получить только из объекта типа **Lock** методом **newCondition()**. Расширение возможностей происходит за счет методов **await()** и **signal()**, функциональность которых подобна действию методов **wait()** и **notify()** класса **Object**.

Теперь можно попробовать переписать код класса **Payment**, заменяя возможности классической синхронизации на методы интерфейсов **Lock** и **Condition**.

```
/* # 23 # реализация выполнения платежа с применением Lock # PaymentLock.java */
```

```
package by.epam.learn.thread;
import java.util.Scanner;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
public class PaymentLock {
    private int amount;
    private ReentrantLock lock = new ReentrantLock(true);
    private Condition condition = lock.newCondition();
    public void doPayment() {
        try {
            System.out.println("Start payment(lock)");
            lock.lock();
            while (amount <= 0) {
                condition.await();
            }
            // payment code here
            System.out.println("Payment(lock) is closed");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

```

    }
    public void init() {
        try {
            lock.lock();
            System.out.println("Init amount: ");
            amount = new Scanner(System.in).nextInt();
        } finally {
            condition.signal();
            lock.unlock();
        }
    }
}

```

```
/* # 24 # запуск процессов доступа к ресурсу # PaymentLockMain.java */
```

```

package by.epam.learn.thread;
import java.util.concurrent.TimeUnit;
public class PaymentLockMain {
    public static void main(String[] args) {
        final PaymentLock payment = new PaymentLock();
        new Thread(() -> payment.doPayment()).start();
        try {
            TimeUnit.MILLISECONDS.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        payment.init();
        System.out.println("the end");
    }
}

```

Интерфейс **ReentrantReadWriteLock** предоставляет еще одну возможность: позволяет выполнять задачи на «чтение» в блоке кода, ограниченном *lock\unlock* несколькими потоками одновременно, но только одному потоку выполнять задачу на «запись». Под «чтением» здесь подразумеваются действия, совместное выполнение которых не повлияет друг на друга. То есть, если несколько потоков захотят получить доступ на «чтение», то они его получают. Если в этот момент доступ попытается получить поток на «запись», то он будет остановлен до окончания работы всех потоков на «чтение». Объекты блокировки на «чтение» и «запись» создаются следующим образом:

```

ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
Lock readLock = lock.readLock();
Lock writeLock = lock.writeLock();

```

При использовании на этих объектах вызываются обычные методы управления блокировками **lock()** и прочие.

Пусть решается задача по вычислению расстояния от точки на плоскости до начала координат. Точка всего одна. Разные потоки вызывают методы вычисления

расстояния и изменения координат точки. Вычисления расстояния могут производиться одновременно несколькими потоками, а изменением координат — только один.

```
/* # 25 # работа с объектом на чтение/запись # Point.java # PointManager.java  
# PointThread.java # PointLockMain.java*/
```

```
package by.epam.learn.thread.rwlock;
import java.util.StringJoiner;
public class Point {
    private double x;
    private double y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double getX() {
        return x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double getY() {
        return y;
    }
    public void setY(double y) {
        this.y = y;
    }
    @Override
    public String toString() {
        return new StringJoiner(", ", "[", "] ")
            .add("x=" + x).add("y=" + y).toString();
    }
}

package by.epam.learn.thread.rwlock;
import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class PointManager {
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();
    public double length(Point point) {
        double length = 0;
        String threadName = Thread.currentThread().getName();
        try {
            readLock.lock();
            System.out.println(" Read begin: " + threadName);
```

```

        TimeUnit.MILLISECONDS.sleep(50);
        length = Math.hypot(point.getX(), point.getY());
        TimeUnit.MILLISECONDS.sleep(50);
        System.out.printf("Read end: %16s %s %5.2f %n", threadName,
                           point, length);

    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        readLock.unlock();
    }
    return length;
}

public void randomChangePoint(Point point) {
    String threadName = Thread.currentThread().getName();
    try {
        writeLock.lock();
        System.out.println("writeLock begin: " + threadName + point);
        TimeUnit.MILLISECONDS.sleep(50);
        point.setX(point.getX() + (5 - new Random().nextInt(10)) / 2.0);
        point.setY(point.getY() + (5 - new Random().nextInt(10)) / 2.0);
        TimeUnit.MILLISECONDS.sleep(50);
        System.out.println(" writeLock end: " + threadName + point);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        writeLock.unlock();
    }
}

}

package by.epam.learn.thread.rwlock;
public class PointThread extends Thread {
    private PointManager pointManager;
    private boolean writeStatus;
    private Point point;
    public PointThread(PointManager pointManager, Point point, boolean writeStatus) {
        this.pointManager = pointManager;
        this.point = point;
        this.writeStatus = writeStatus;
    }
    @Override
    public void run() {
        if (writeStatus) {
            pointManager.randomChangePoint(point);
        } else {
            pointManager.length(point);
        }
    }
}

package by.epam.learn.thread.rwlock;
import java.util.Random;

```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class PointLockMain {
    public static void main(String[] args) {
        PointManager pointManager = new PointManager();
        Random rand = new Random();
        ExecutorService service = Executors.newFixedThreadPool(15);
        Point point = new Point(1, -1);
        for (int i = 0; i < 15; i++) {
            service.submit(new PointThread(pointManager, point, rand.nextBoolean()));
        }
        service.shutdown();
    }
}
```

В результате может быть выведено:

```
writeLock begin: pool-1-thread-2 [x=1.0, y=-1.0]
writeLock end: pool-1-thread-2 [x=2.5, y=-0.5]
Read begin: pool-1-thread-1
Read begin: pool-1-thread-3
Read begin: pool-1-thread-4
Read end: pool-1-thread-1 [x=2.5, y=-0.5] 2.55
Read end: pool-1-thread-3 [x=2.5, y=-0.5] 2.55
Read end: pool-1-thread-4 [x=2.5, y=-0.5] 2.55
writeLock begin: pool-1-thread-9 [x=2.5, y=-0.5]
writeLock end: pool-1-thread-9 [x=3.0, y=1.5]
Read begin: pool-1-thread-6
Read begin: pool-1-thread-5
Read begin: pool-1-thread-8
Read begin: pool-1-thread-7
Read end: pool-1-thread-6 [x=3.0, y=1.5] 3.35
Read end: pool-1-thread-5 [x=3.0, y=1.5] 3.35
Read end: pool-1-thread-8 [x=3.0, y=1.5] 3.35
Read end: pool-1-thread-7 [x=3.0, y=1.5] 3.35
writeLock begin: pool-1-thread-10 [x=3.0, y=1.5]
writeLock end: pool-1-thread-10 [x=1.5, y=4.0]
writeLock begin: pool-1-thread-11 [x=1.5, y=4.0]
writeLock end: pool-1-thread-11 [x=3.5, y=3.0]
writeLock begin: pool-1-thread-12 [x=6.0, y=4.5]
writeLock end: pool-1-thread-12 [x=5.0, y=4.5]
Read begin: pool-1-thread-13
Read begin: pool-1-thread-14
Read end: pool-1-thread-14 [x=5.0, y=4.5] 6.73
Read end: pool-1-thread-13 [x=5.0, y=4.5] 6.73
```

writeLock begin: pool-1-thread-15 [x=5.0, y=4.0]

writeLock end: pool-1-thread-15 [x=6.0, y=4.5]

Семафор

В Java существуют способы управления группами потоков, объединенные общим названием семафор, а именно: такие классы-барьеры синхронизации, как:

CountDownLatch — заставляет потоки ожидать завершения заданного числа операций, по окончании чего все ожидающие потоки освобождаются;

Semaphore — предлагает потоку ожидать завершения действий в других потоках;

CyclicBarrier — предлагает нескольким потокам ожидать момента, когда они все достигнут какой-либо точки, после чего барьер снимается;

Phaser — барьер, контракт которого является расширением возможностей **CyclicBarrier**, а также частично согласовывается с возможностями **CountDownLatch**.

Семафор **java.util.concurrent.Semaphore** позволяет управлять доступом к ресурсам или просто работой потоков на основе запрещений-разрешений. Семафор всегда устанавливается на предельное положительное число потоков, одновременное функционирование которых может быть разрешено в определенном участке кода. При превышении предельного числа все желающие работать потоки будут приостановлены до освобождения семафора одним из работающих по его разрешению потоков. Уменьшение счетчика доступа производится методами **void acquire()** и его оболочки **boolean tryAcquire()**. Оба метода занимают семафор, если он свободен. Если же семафор занят, то метод **tryAcquire()** возвращает ложь и пропускает поток дальше, что позволяет при необходимости отказаться от дальнейшей работы потоку, который не смог получить семафор. Метод **acquire()** при невозможности захвата семафора остановит поток до тех пор, пока другой поток не освободит семафор. Метод **boolean tryAcquire(long timeout, TimeUnit unit)** возвращает ложь, если время ожидания превышено, т.е. за указанное время поток не получил от семафора разрешение работать и пропускает поток дальше. Метод **release()** освобождает семафор и увеличивает счетчик на единицу. Простое стандартное взаимодействие методов **acquire()** и **release()** демонстрирует следующий фрагмент.

```
try {
    semaphore.acquire();
    // some code
} catch (InterruptedException e) {
    e.printStackTrace();
} finally {
    semaphore.release();
}
```

Методы **acquire()** и **release()** в общем случае могут и не вызываться в одном методе кода. Тогда за корректное и своевременное возвращение семафора будет

ответственен разработчик. Метод **acquire()** не пропустит поток до тех пор, пока счетчик семафора имеет значение ноль.

Методы **acquire()**, **tryAquire()** и **release()** имеют перегруженную версию с параметром типа **int**. В такой метод можно передать число, на которое изменится значение счетчика семафора при успешном выполнении метода, в отличие от методов без параметров, которые всегда изменяют значение счетчика только на единицу.

Для демонстрации работы семафора предлагается задача о пуле ресурсов с ограниченным числом, в данном случае аудиоканалов, и значительно бóльшим числом клиентов, желающих воспользоваться одним из каналов. Каждый клиент получает доступ к каналу, причем пользоваться можно только одним каналом. Если все каналы заняты, то клиент ждет в течение заданного интервала времени. Если лимит ожидания превышен, генерируется исключение и клиент уходит, так и не воспользовавшись услугами пула.

Класс **ChannelPool** объявляет семафор и очередь из каналов. В методе **getResource()** производится запрос к семафору, и в случае успешного его прохождения метод ищет в списке свободный канал, изменяет его статус на «занят» и выдает его в качестве возвращаемого значения метода. Метод **releaseResource()** изменяет статус экземпляра на «свободен», тем самым делая его доступным для клиентов, и освобождает семафор.

Вместо списков в качестве хранения ресурсов могут использоваться очереди, тогда реализация принципов пула предоставляет возможность повторного использования объектов в ситуациях, когда создание нового объекта — дорогостоящая процедура с точки зрения задействованных для этого ресурсов виртуальной машины. Поэтому при возможности следует объект после использования не уничтожать, а вернуть его в так называемый «пул объектов» для повторного использования. Данная стратегия широко используется при организации пула соединений с базой данных. Реализаций организации пулов существует достаточно много с различающимися способами извлечения и возврата объектов, а также способа контроля за объектами и за заполняемостью пула. Поэтому выбрать какое-либо решение как абсолютно лучшее для всех случаев невозможно.

```
// # 26 # пул ресурсов # ChannelPool.java
```

```
package by.epam.learn.thread.semaphore;
import java.util.*;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ChannelPool<T> extends Channel<T> {
    private final static int POOL_SIZE = 5;
    private Semaphore semaphore = new Semaphore(POOL_SIZE, true);
    private ArrayList<T> resources = new ArrayList<>();
```

```

public ChannelPool(List<T> source) {
    resources.addAll(source);
}
public T getResource(Client client, long maxWaitMillis) throws ResourceException {
    try {
        if (semaphore.tryAcquire(maxWaitMillis, TimeUnit.MILLISECONDS)) {
            for (T resource : resources) {
                if (!resource.isBusy()) {
                    resource.setBusy(true);
                    System.out.println("Client #" + client.getId()
                        + " took channel " + resource);
                    return resource;
                }
            }
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    throw new ResourceException("timed out " + maxWaitMillis);
}
public void releaseResource(Client client, T resource) {
    resource.setBusy(false); // change the status of the resource
    System.out.println("Client #" + client.getId() + ": " + resource + " --> released");
    semaphore.release();
}
}

```

Объявление поля **busy** класса **Channel** как **volatile** в данной задаче необходимо, так как возможен одновременный проход семафора через метод **tryAcquire()** двумя и более потоками и, соответственно, эти потоки могут получить один и тот же объект из списка в качестве нужного им ресурса, что противоречит постановке задачи: один ресурс для одного клиента.

Класс **AudioChannel** предлагает простейшее описание канала и его использования.

```
// # 27 #класс Channel, разделяемый ресурс # Channel.java # AudioChannel.java
```

```

package by.epam.learn.thread.semaphore;
public abstract class Channel {
    private volatile boolean busy;
    public boolean isBusy() {
        return busy;
    }
    public void setBusy(boolean busy) {
        this.busy = busy;
    }
    public abstract void using();
}

```

```
package by.epam.learn.thread.semaphore;
import java.util.Random;
import java.util.concurrent.TimeUnit;
public class AudioChannel extends Channel {
    private int channelId;
    public AudioChannel(int id) {
        this.channelId = id;
    }
    public int getChannelId() {
        return channelId;
    }
    public void setChannelId(int id) {
        this.channelId = id;
    }
    @Override
    public void using() {
        try {
            // using channel
            TimeUnit.MILLISECONDS.sleep(new Random().nextInt(500));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("AudioChannel{");
        sb.append("id=").append(channelId).append(", busy=").append(isBusy()).append('}');
        return sb.toString();
    }
}
```

Исключение **ResourceException** желательно в такого рода задачах, чтобы точно описать возникающую проблему при работе ресурса, используемого конкурирующими потоками.

Класс **Client** представляет поток, запрашивающий доступ к ресурсу из пула, использующий его некоторое время и освобождающий его для повторного использования.

```
// # 28 # поток, обрабатывающий ресурс # Client.java
```

```
package by.epam.learn.thread.semaphore;
public class Client extends Thread {
    private ChannelPool<AudioChannel> pool;
    public Client(ChannelPool<AudioChannel> pool) {
        this.pool = pool;
    }
    public void run() {
        AudioChannel channel = null;
```

```

try {
    channel = pool.getResource(this, 500); // change to 100
    channel.using();
} catch (ResourceException e) {
    System.err.println("Client #" + this.getId() + " lost ->" + e.getMessage());
} finally {
    if (channel != null) {
        pool.releaseResource(this, channel);
    }
}
}
}

```

Класс **SemaphoreMain** демонстрирует работу пула ресурсов аудиоканалов. При заполнении очереди каналов в данном решении необходимо следить, чтобы число каналов, передаваемых списком в конструктор класса **ChannelPool**, совпадало со значением константы **POOL_SIZE** этого же класса, которая используется для инициализации семафора.

```
// # 29 # создание и использование пула # SemaphoreMain.java
```

```

package by.epam.learn.thread.semaphore;
import java.util.List;
import java.util.ArrayList;
public class SemaphoreMain {
    public static void main(String[] args) {
        List<AudioChannel> audioChannels = new ArrayList<AudioChannel>() {
            {
                this.add(new AudioChannel(771));
                this.add(new AudioChannel(883));
                this.add(new AudioChannel(550));
                this.add(new AudioChannel(337));
                this.add(new AudioChannel(442));
            }
        };
        ChannelPool<AudioChannel> pool = new ChannelPool<>(audioChannels);
        for (int i = 0; i < 20; i++) {
            new Client(pool).start();
        }
    }
}

```

Результатом может быть вывод:

```

Client #15 took channel AudioChannel{id=771, busy=true}
Client #14 took channel AudioChannel{id=883, busy=true}
Client #18 took channel AudioChannel{id=550, busy=true}
Client #17 took channel AudioChannel{id=337, busy=true}
Client #16 took channel AudioChannel{id=442, busy=true}
Client #16: AudioChannel{id=442, busy=false} --> released

```

```

Client #25 took channel AudioChannel{id=442, busy=true}
Client #14: AudioChannel{id=883, busy=false} --> released
Client #19 took channel AudioChannel{id=883, busy=true}
Client #25: AudioChannel{id=442, busy=false} --> released
Client #22 took channel AudioChannel{id=442, busy=true}
Client #15: AudioChannel{id=771, busy=false} --> released
Client #23 took channel AudioChannel{id=771, busy=true}
Client #23: AudioChannel{id=771, busy=false} --> released
Client #24 took channel AudioChannel{id=771, busy=true}
Client #24: AudioChannel{id=771, busy=false} --> released
Client #20 took channel AudioChannel{id=771, busy=true}
Client #17: AudioChannel{id=337, busy=false} --> released
Client #26 took channel AudioChannel{id=337, busy=true}
Client #18: AudioChannel{id=550, busy=false} --> released
Client #21 took channel AudioChannel{id=550, busy=true}
Client #20: AudioChannel{id=771, busy=false} --> released
Client #27 took channel AudioChannel{id=771, busy=true}
Client #29 lost ->timed out 500
Client #33 lost ->timed out 500
Client #31 lost ->timed out 500
Client #28 lost ->timed out 500
Client #30 lost ->timed out 500
Client #32 lost ->timed out 500
Client #26: AudioChannel{id=337, busy=false} --> released
Client #19: AudioChannel{id=883, busy=false} --> released
Client #22: AudioChannel{id=442, busy=false} --> released
Client #21: AudioChannel{id=550, busy=false} --> released
Client #27: AudioChannel{id=771, busy=false} --> released

```

Часть потоков не дождалась доступа к ресурсу за заданное время, и по его истечении были сгенерированы исключения.

Барьер

Многие задачи могут быть разделены на подзадачи и выполняться параллельно. По достижении некоторой данной точки всеми параллельными потоками подводится итог и определяется общий результат. Если определена задача задерживать заданное число потоков до достижения ими определенной точки синхронизации, то используются классы-барьеры. После того, как все потоки достигли этой самой точки, они будут разблокированы и смогут продолжать выполнение.

Класс **CyclicBarrier** определяет минимальное число потоков, которое может быть остановлено барьером. Кроме этого, барьер сам может быть проинициализирован потоком, который будет запускаться при снятии барьера. Методы

int await() и **int await(long timeout, TimeUnit unit)** останавливают поток, использующий барьер, до тех пор, пока число потоков не достигнет заданного числа в классе-барьере. Метод **await()** возвращает порядковый номер достижения потоком барьерной точки. Метод **boolean isBroken()** проверяет состояние барьера. Метод **reset()** сбрасывает состояние барьера к моменту инициализации. Метод **int getNumberWaiting()** позволяет определить число ожидаемых барьером потоков до его снятия. Экземпляр **CyclicBarrier** можно использовать повторно.

Процесс проведения аукциона подразумевает корректное использование класса **CyclicBarrier**. Класс **Auction** определяет список участников-клиентов и размер барьера. В аукционе участвуют все объявленные клиенты. Чтобы приложение работало корректно, необходимо, чтобы размер списка участников совпадал с размером барьера. Сам барьер инициализируется потоком определения победителя торгов, который запустится после того, как все предложения будут объявлены и потоки участников будут остановлены методом **await()**. Если потоков будет запущено больше, чем размер барьера, то «лишние» предложения могут быть не учтены при вычислении победителя, если же потоков будет меньше, то приложение окажется в состоянии *deadlock*. Для предотвращения подобных ситуаций следует использовать метод **await()** с параметрами.

```
// # 30 # определение барьера и действия по его окончании # Auction.java
```

```
package by.epam.learn.thread.barrier;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.concurrent.CyclicBarrier;
public class Auction {
    private List<Participant> participants;
    public static CyclicBarrier barrier;
    public Auction(int numberParticipant) {
        participants = new ArrayList<>();
        this.barrier = new CyclicBarrier(numberParticipant, () -> Auction.this.defineWinner());
    }
    public boolean add(Participant e) {
        return participants.add(e);
    }
    public Participant defineWinner() {
        Participant winner = Collections.max(participants,
                                              Comparator.comparingInt(Participant::getCurrentLotPrice));
        System.out.println("Participant #" + winner.getBidId() + ", price:"
                           + winner.getCurrentLotPrice() + " win!");
        winner.setCash(winner.getCash() - winner.getCurrentLotPrice());
        return winner;
    }
}
```

Класс **Participant** определяет клиента и его предложение цены на лот аукциона и запрашивает барьер, после которого, в случае победы, клиент либо заплатит за лот, либо будет продолжать работать дальше.

```
// # 31 # поток, использующий барьер # Participant.java
```

```
package by.epam.learn.thread.barrier;
import java.util.Random;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;
public class Participant extends Thread {
    private Integer participantId;
    private int currentLotPrice;
    private int cash;
    private CyclicBarrier barrier = Auction.barrier;
    public Participant(int id, int currentLotPrice, int cash) {
        this.participantId = id;
        this.currentLotPrice = currentLotPrice;
        this.cash = cash;
    }
    public Integer getBidId() {
        return participantId;
    }
    public int getCurrentLotPrice() {
        return currentLotPrice;
    }
    public int getCash() {
        return cash;
    }
    public void setCash(int cash) {
        this.cash = cash;
    }
    @Override
    public void run() {
        try {
            System.out.println("Participant " + participantId
                               + " specifies a price. (cash = " + cash + ")");
            TimeUnit.MILLISECONDS.sleep(new Random().nextInt(2500));
            int delta = new Random().nextInt(20); /* determine the level of
                                                    price increase */
            currentLotPrice += delta;
            System.out.println("Auction Participant " + participantId + " : " + currentLotPrice);
            this.barrier.await(); // stop at the barrier
            System.out.println("Participant " + participantId
                               + " Continue to work...(cash = " + cash + ")");
        } catch (BrokenBarrierException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
// # 32 # создание аукциона и его запуск # AuctionMain.java
```

```
package by.epam.learn.thread.barrier;
import java.util.Random;
public class AuctionMain {
    public static void main(String[] args) {
        int numberParticipant = 5;
        Auction auction = new Auction(numberParticipant);
        int startPrice = 50;
        System.out.println("startPrice =" + startPrice);
        for (int num = 0; num < numberParticipant; num++) {
            int cash = 100 + new Random().nextInt(50);
            Participant participant = new Participant(num, startPrice, cash);
            auction.add(participant);
            participant.start();
        }
    }
}
```

Результаты работы аукциона:

```
startPrice =50
Participant 3 specifies a price. (cash = 146)
Participant 2 specifies a price. (cash = 110)
Participant 1 specifies a price. (cash = 140)
Participant 0 specifies a price. (cash = 139)
Participant 4 specifies a price. (cash = 142)
Auction Participant 3 : 63
Auction Participant 4 : 60
Auction Participant 1 : 64
Auction Participant 2 : 57
Auction Participant 0 : 69
Participant #0, price:69 win!
Participant 4 Continue to work...(cash = 142)
Participant 0 Continue to work...(cash = 70)
Participant 3 Continue to work...(cash = 146)
Participant 2 Continue to work...(cash = 110)
Participant 1 Continue to work...(cash = 140)
```

CountDownLatch или «защелка»

Еще один вид барьера представляет класс **CountDownLatch**. Экземпляр класса инициализируется начальным значением числа ожидающих снятия «защелки» потоков. В отличие от **CyclicBarrier**, метод **await()** просто останавливает поток без каких-либо изменений значения счетчика. Значение счетчика снижается вызовом метода **countDown()**, т.е. «защелка» сдвигается на единицу.

Когда счетчик обнулится, барьеры, поставленные методом **await()**, снимаются для всех ожидающих разрешения потоков. Крайне желательно, чтобы метод **await()** был вызван раньше, чем метод **countDown()**. Последнему безразлично, вызывался метод **await()** или нет, счетчик все равно будет уменьшен на единицу. Если счетчик равен нулю, то «лишние» вызовы метода **countDown()** будут проигнорированы.

Демонстрацией возможностей класса **CountDownLatch** может служить задача с аукционом. Отличие от предыдущей постановки заключается в том, что участник аукциона может отказаться от участия в нем, в данном случае по причине недостатка средств на счете, и продолжить свою работу независимо от потока, в котором проводится аукцион.

```
// # 33 # поток-аукцион # AuctionLatch.java
```

```
package by.epam.learn.thread.latch;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.stream.Collectors;

public class AuctionLatch extends Thread {
    private List<ParticipantLatch> participants;
    public static CountDownLatch latchEndAuction = new CountDownLatch(1);
    public static CountDownLatch latchAuctionBegin;
    public AuctionLatch(int numbersParticipant) {
        latchAuctionBegin = new CountDownLatch(numbersParticipant);
        participants = new ArrayList<>();
    }
    public boolean add(ParticipantLatch e) {
        return participants.add(e);
    }
    @Override
    public void run() {
        try {
            System.out.println("waiting for participants to bet...");
            latchAuctionBegin.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        ParticipantLatch win = defineWinner();
        latchEndAuction.countDown();
    }
    private ParticipantLatch defineWinner() {
        ParticipantLatch winner =
            Collections.max(participants
                .stream()
                .filter(o -> !o.isLost())
                .collect(Collectors.toList()),
```

```

        Comparator.comparingInt(ParticipantLatch::getCurrentLotPrice));
    System.out.println("Participant #" + winner.getParticipantId()
        + ", price:" + winner.getCurrentLotPrice() + " win!");
    winner.setCash(winner.getCash() - winner.getCurrentLotPrice());
    return winner;
}
}

```

// # 34 # участник аукциона # ParticipantLatch.java

```

package by.epam.learn.thread.latch;
import java.util.Random;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
public class ParticipantLatch extends Thread {
    private Integer participantId;
    private int currentLotPrice;
    private int cash;
    private CountDownLatch latchEndBid = AuctionLatch.latchEndAuction;
    private CountDownLatch latchAuctionBegin = AuctionLatch.latchAuctionBegin;
    private boolean lost;
    public ParticipantLatch(int id, int lotPrice, int cash) {
        this.participantId = id;
        this.currentLotPrice = lotPrice;
        this.cash = cash;
    }
    public Integer getParticipantId() {
        return participantId;
    }
    public int getCurrentLotPrice() {
        return currentLotPrice;
    }
    public int getCash() {
        return cash;
    }
    public void setCash(int cash) {
        this.cash = cash;
    }
    public boolean isLost() {
        return lost;
    }
    @Override
    public void run() {
        try {
            if(cash < currentLotPrice) {
                lost = true;
                latchAuctionBegin.countDown();
                System.out.println("participant #" + participantId
                    + " lost because (cash = " + cash + ") < (price = "
                    + currentLotPrice + ")");
            }
        }
    }
}

```

```

        return;
    }
    System.out.println("participant " + participantId
        + " specifies a price. (cash = " + cash + ")");
    TimeUnit.MILLISECONDS.sleep(new Random().nextInt(2500));
    // determining the level of price increase
    int delta = new Random().nextInt(10);
    currentLotPrice += delta;
    currentLotPrice = currentLotPrice < cash ? currentLotPrice : cash;
    System.out.println("made a bet Participant " + participantId + " : " + currentLotPrice);
    latchAuctionBegin.countDown();
    this.latchEndBid.await();
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("participant #" + participantId
    + " Continue to work...(cash = " + cash + ") ");
}
}

```

```

/* # 35 # формирование и запуск аукциона # LatchMain.java */

```

```

package by.epam.learn.thread.latch;
import java.util.Random;
public class LatchMain {
    public static void main(String[] args) {
        int numbersParticipant = 5;
        AuctionLatch auction = new AuctionLatch(numbersParticipant);
        int startPrice = 100;
        auction.start();
        for (int num = 0; num < numbersParticipant; num++) {
            int cash = 100 + new Random().nextInt(10);
            ParticipantLatch participant = new ParticipantLatch(num, startPrice, cash);
            auction.add(participant);
            participant.start();
        }
    }
}

```

В результате будет выведено:

```

waiting for participants to bet...
participant 2 specifies a price. (cash = 110)
participant 0 specifies a price. (cash = 115)
participant 3 specifies a price. (cash = 116)
participant #4 lost because (cash = 109) < (price = 110)
participant #1 lost because (cash = 102) < (price = 110)
made a bet Participant 0 : 111

```

made a bet Participant 2 : 110
 made a bet Participant 3 : 114
 Participant #3, price:114 win!
 participant #2 Continue to work...(cash = 110)
 participant #0 Continue to work...(cash = 115)
 participant #3 Continue to work...(cash = 2)

В этом и предыдущих заданиях не полностью выполнены все условия по обеспечению корректной работы приложений во всех возможных ситуациях, возникающих по бизнес-логике приложения и в условиях многопоточности. Например, в этом приложении будет генерироваться исключительная ситуация, если все участники откажутся от аукциона, поэтому необходимо добавить соответствующую проверку.

Можно рассматривать данный пример в качестве каркаса для построения работоспособного приложения, когда проводится несколько сессий по продаже лотов на аукционе.

Deadlock

При работе с блокировками ресурсов потоков может возникать ситуация *deadlock*. То есть потоки в своем взаимодействии остановились и никаким образом не могут продолжить свое выполнение.

Наглядный пример взаимной блокировки состоит в следующем: поток № 1 заблокировал объект А, поток № 2 заблокировал объект В. Далее поток № 1 пытается получить доступ к объекту В и блокируется, так как объект В уже занят потоком № 2. В свою очередь, поток № 2 пытается получить доступ к объекту А и блокируется, так как объект А уже занят потоком № 1. В итоге оба потока заблокировали друг друга.

```
/* # 36 # deadlock для двух потоков и двух объектов # DeadLockMain.java */
```

```
package by.epam.learn.thread;
public class DeadlockMain {
    public static void main(String[] args) {
        InviteAction invite1 = new InviteAction("first");
        InviteAction invite2 = new InviteAction("second");
        new Thread(() -> invite1.invite(invite2)).start();
        new Thread(() -> invite2.invite(invite1)).start();
    }
}

package by.epam.learn.thread;
public class InviteAction {
    private String name;
    public InviteAction(String name) {
        this.name = name;
    }
}
```

```
public synchronized void invite(InviteAction obj) {
    System.out.println(name + " invites " + obj.name.toUpperCase());
    obj.action(); // deadlock
}
public synchronized void action() {
    System.out.println(name + " action");
}
}
```

Приложение выведет:

first invites SECOND
second invites FIRST

и остановится.

Обмен блокировками

Ситуации с необходимостью обмена объектами при их взаимном блокировании возникают, для решения разработан класс **Exchanger**, предоставляющий возможность безопасного обмена объектами, в том числе и синхронизированными. Функционал обмена представляет метод **Texchange(Tob)**. Возвращаемый параметр метода — объект, который будет принят из другого потока, передаваемый параметр **ob** метода — собственный объект потока, который помещается в буфер обмена и будет отдан другому потоку. Процесс обмена завершится успешно только в случае, если два потока вызовут метод **exchange()** на одном и том же объекте **Exchanger**.

Класс **ItemAction** представляет возможность изменения цены и описания товара. В одном методе производится изменение цены товаров, в другом — изменение его описания. Объект **Exchanger** позволяет эти действия развести по двум независимым потокам.

```
/* # 37 # действие, содержащее Exchanger # ItemAction.java */
```

```
package by.epam.learn.thread.exchanger;
import java.util.concurrent.Exchanger;
public class ItemAction {
    private static Exchanger<Item> exchanger = new Exchanger<>();
    public void doActionPrice(Item item, float discount) {
        try {
            item.setPrice(item.getPrice() * discount);
            item = exchanger.exchange(item);
            item.setPrice(item.getPrice() * discount);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

public void doActionDescription(Item item, String addDescription) {
    try {
        item.setDescription(item.getDescription() + addDescription);
        item = exchanger.exchange(item);
        item.setDescription(item.getDescription() + addDescription);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
}

```

```
/* # 38 # класс-сущность товар # Item.java */
```

```

package by.epam.learn.thread.exchanger;
public class Item {
    private int itemId;
    private double price;
    private String description;
    public Item(int itemId, double price, String description) {
        this.itemId = itemId;
        this.price = price;
        this.description = description;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    @Override
    public String toString() {
        return String.format("id=%d, price=%4.2f, description=%s", itemId, price, description);
    }
}
}

```

```
/* # 39 # процесс обмена # ExchangeMain.java */
```

```

package by.epam.learn.thread.exchanger;
import java.util.concurrent.TimeUnit;
public class ExchangeMain {
    public static void main(String[] args) {
        ItemAction action = new ItemAction();
        Item o1 = new Item(101, 5.0, "Tie");
        Item o2 = new Item(171, 7.0, "Gloves");
    }
}

```

```

System.out.println(o1 + "\n" + o2);
new Thread(() -> action.doActionPrice(o1, 0.9f)).start();
new Thread(() -> action.doActionDescription(o2, " with discount")).start();
try {
    TimeUnit.SECONDS.timedJoin(Thread.currentThread(), 1);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println(o1 + "\n" + o2);
}
}

```

В результате будет получено:

```

id=101, price=5.00, description=Tie
id=171, price=7.00, description=Gloves
id=101, price=4.50, description=Tie with discount
id=171, price=6.30, description=Gloves with discount

```

Блокирующие очереди

Возможности синхронизации коллекций существовали и ранее. Практически это означало, что синхронизированные экземпляры блокировались целиком, хотя необходимо это было далеко не всегда. Например, поток, изменяющий одну часть объекта **Hashtable**, блокировал работу других потоков, которые хотели бы просто прочесть, ничего не изменяя, совсем другую часть этого объекта. Поэтому введение дополнительных возможностей, связанных с синхронизацией потоков и блокировкой ресурсов, довольно логично. Усовершенствованные для задач многопоточности аналоги существующих ранее коллекции находятся в пакете **java.util.concurrent**. Ограниченно потокобезопасные коллекции **ConcurrentHashMap**, **ConcurrentLinkedQueue**, **ConcurrentLinkedDeque**, позволяющие нескольким потокам выполнять одновременные безопасные действия с коллекцией.

Другая группа коллекций представляет блокирующие очереди **BlockingQueue** и **BlockingDeque**, гарантирующие остановку потока, запрашивающего элемент из пустой очереди до появления в ней элемента, доступного для извлечения, а также блокирующего поток, пытающийся вставить элемент в заполненную очередь до тех пор, пока в очереди не освободится позиция.

Реализации интерфейсов **BlockingQueue** и **BlockingDeque** предлагают методы по добавлению/извлечению элементов с задержками, а именно:

void put(E e) — добавляет элемент в очередь. Если очередь заполнена, то ожидает, пока освободится место;

boolean offer(E e, long timeout, TimeUnit unit) — добавляет элемент в очередь. Если очередь заполнена, то ожидает время **timeout**, пока освободится место; если за это время место не освободилось, то возвращает **false**, не выполнив действия по добавлению;

boolean offer(E e) — добавляет элемент в очередь. Если очередь заполнена, то возвращает **false**, не выполнив действия по добавлению;

E take() — извлекает и удаляет элемент из очереди. Если очередь пуста, то ожидает, пока там появится элемент;

E poll(long timeout, TimeUnit unit) — извлекает и удаляет элемент из очереди. Если очередь пуста, то ожидает время **timeout**, пока там появится элемент, если за это время очередь так и осталась пуста, то возвращает **null**;

E poll() — извлекает и удаляет элемент из очереди. Если очередь пуста, то возвращает **null**.

Интерфейс **BlockingDeque** добавляет набор методов по работе с началом и концом очереди.

Все методы **BlockingQueue** для наглядности можно расположить в виде таблицы, как это и сделано в документации Java:

	<i>Throws exception</i>	<i>Special value</i>	<i>Blocks</i>	<i>Times out</i>
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	<i>not applicable</i>	<i>not applicable</i>

Максимальный размер очереди должен быть задан при ее создании, а именно, все конструкторы класса **ArrayBlockingQueue** принимают в качестве одного из параметров **capacity** — длину очереди. Пусть объявлена очередь из двух элементов. Изначально очередь пуста. В первом потоке производится попытка добавления трех элементов. Два добавятся успешно, а при попытке добавления третьего поток будет остановлен до появления свободного места в очереди. Только когда второй поток извлечет один элемент и освободит место, первый поток получит возможность добавить свой элемент.

```
/* # 40 # добавление\удаление в блокирующую очередь # BlockingQueueMain.java */
```

```
package by.epam.learn.thread;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
public class BlockingQueueMain {
    public static void main(String[] args) {
        BlockingQueue<String> queue = new ArrayBlockingQueue<String>(2);
        new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                    queue.put("Java" + i);
                    System.out.println("Element " + i + " added");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(() -> {
            try {
                System.out.println("Element " + queue.take() + " took");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

В результате вероятно будет выведено:

```
Element 0 added
Element 1 added
Element Java0 took
Element 2 added
```

CopyOnWriteArrayList

Коллекции **CopyOnWriteArrayList** и **CopyOnWriteArraySet** копируют свое содержимое при попытке его изменения, причем ранее полученный итератор будет корректно продолжать работать с исходным набором данных.

Эти коллекции предназначены для обработки своих элементов в условиях конкурентного доступа. Например, если в одном потоке извлекается итератор из коллекции и элементы просто отправляются в поток вывода, и одновременно в другом потоке элементы в эту же коллекцию добавляются, то эти изменения станут доступны и первому потоку без генерации исключения, как это произошло бы в случае применения обычного **ArrayList**. Такая реализация алгоритма работы с коллекцией позволяет исключить возможность возникновения исключения **ConcurrentModificationException**.

```
/* # 41 # конкурентная обработка списка # CopyOnWriteMain.java */
```

```
package by.epam.learn.thread.copyonwrite;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.Collectors;
public class CopyOnWriteMain {
    public static void main(String[] args) {
        List<Integer> temp = new Random()
            .ints(5, 0, 10)
            .boxed()
            .collect(Collectors.toList());

        List<Integer> newList = new ArrayList<>();
        CopyOnWriteArrayList<Integer> copyList = new CopyOnWriteArrayList<>(temp);
        System.out.printf("%17s: %s%n ", "copyList before", temp);
        // ArrayList<Integer> list = new ArrayList<>(temp);
        new Thread(() -> { // thread # 1
            try {
                TimeUnit.MILLISECONDS.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Iterator<Integer> iterator = copyList.iterator();
            while (iterator.hasNext()) {
                Integer current = iterator.next();
                newList.add(current);
            }
            System.out.printf("%16s: %s%n ", "newList Th #1", newList);
        }).start();
        new Thread(() -> { // thread # 2
            for (int i = 0; i < 10; i++) {
                try {
                    TimeUnit.MILLISECONDS.sleep(10); // change to 100
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                copyList.addIfAbsent(i);
            }
        }).start();
    }
}
```

```

try {
    TimeUnit.SECONDS.timedJoin(Thread.currentThread(), 1);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.printf("%16s: %s%n ", "copyList Th #2", copyList);
System.out.printf("%16s: %s%n ", "newList Th #1", newList);
}
}

```

Результатом может быть:

```

copyList before: [9, 0, 9, 2, 4]
newList Th #1: [9, 0, 9, 2, 4, 1, 3]
copyList Th #2: [9, 0, 9, 2, 4, 1, 3, 5, 6, 7, 8]
newList Th #1: [9, 0, 9, 2, 4, 1, 3]

```

Смысл приложения в следующем. Изначально создается список **temp**. На основе этого списка создается объект **CopyOnWriteArrayList** с именем **copyList** и выводится в первой строке вывода. Затем стартуют два потока. Первый поток занят извлечением с помощью итератора объектов из **copyList** и добавлением их в другой список **newList**. Второй поток в это же время добавляет в список **copyList** недостающие цифры в интервале от 0 до 9. Первый поток может закончить свою работу по переносу элементов из **copyList** в **newList** раньше, чем второй успеет добавить недостающий элемент в **copyList**.

Замена объявления **CopyOnWriteArrayList** на **ArrayList** приведет к генерации хорошо известного исключения **ConcurrentModificationException**.

Phaser

Более сложное поведение этого синхронизатора **Phaser** напоминает поведение **CyclicBarrier**, однако число участников синхронизации может меняться. Участвующие стороны сначала должны зарегистрироваться *phaser*-объектом. Регистрация осуществляется с помощью методов **register()**, **bulkRegister(int parties)** или подходящего конструктора. Выход из синхронизации *phaser*-объектом производит метод **arriveAndDeregister()**, причем выход из числа синхронизируемых сторон может быть и в случае, когда поток завершил выполнение, и в случае, когда поток все еще выполняется. Основным назначением класса **Phaser** является синхронизация потоков, для выполнения которых требуется разбить их на отдельные этапы (фазы), а эти фазы, в свою очередь, необходимо синхронизовать. **Phaser** может как задержать поток, пока другие потоки не достигнут конца текущей фазы методом **arriveAndAwaitAdvance()**, так и пропустить поток, отметив лишь окончание какой-либо фазы методом **arrive()**.

Вопросы к главе 12

1. Что такое поток выполнения? Состояния потока.
2. Как создать поток? Какими тремя способами можно создать поток, запустить его, прервать (завершить, убить)?
3. Жизненный цикл потока.
4. Как выполнить набор команд в отдельном потоке?
5. Как работают методы **wait()** и **notify()/notifyAll()**? Каков будет результат, если на ресурсе вызвать метод **notify()**, не вызвав до этого соответствующий ему **wait()**?
6. Чем отличается работа метода **wait()** с параметром и без параметра?
7. Как работает метод **yield()**? Чем отличаются методы **Thread.sleep()** и **Thread.yield()**?
8. Как можно осуществить приостановку/возобновление работы потоков?
9. Чем отличаются методы **Thread.sleep()** и **object.wait()**?
10. Как работает метод **join()**?
11. Зачем нужны потоки-демоны? Как создать поток-демон? Когда следует применять потоки-демоны?
12. На что влияет приоритет потока? Как потоку установить приоритет?
13. Привести пример, как можно обработать непроверяемые исключения, генерированные в потоке.
14. Что такое синхронизация? Зачем она нужна? Для чего нужно ключевое слово **synchronized**? Какие методы синхронизации существуют? Какими средствами достигается?
15. Что такое монитор объекта? Кто и как с ним может работать? Объяснить, что значит поток, взявший монитор, может взять его повторно?
16. Отличия работы **synchronized** от **Lock**?
17. Есть ли у **Lock** механизм, аналогичный механизму **wait/notify** у **synchronized**?
18. Что такое **deadlock**? Нарисовать схему, как это происходит. Как избежать этой ситуации?
19. Объяснить, зачем была введена библиотека **java.util.concurrent**? Что она содержит?
20. Привести примеры синхронизированных коллекций и их назначение из пакета **java.util.concurrent**.
21. Классы **Semaphore**, **CyclicBarrier**, **CountDownLatch**. Чем похожи на **Lock** и чем от него отличаются?
22. Что такое механизм **Executors** в пакете **java.util.concurrent**? Зачем он нужен?
23. Что представляет собой пул потоков? Привести пример самостоятельно написанного пула потоков.
24. Написать **deadlock**, придумать примеры с использованием **synchronized**, **AtomicInteger**.
25. По каким объектам синхронизируются статические и нестатические методы?

26. Что такое атомарная операция? Какие операции в Java являются атомарными? Перед какими полями имеет смысл использовать модификатор **volatile**? Как изменится поведение программы, если перед полями использовать модификатор **volatile**?
27. Сравнить возможности **volatile** и классов пакета **java.util.concurrent.atomic**.
28. Дан массив из N элементов. Создать N потоков, которые принимают по числу из массива, обрабатывают и возвращают обратно. Собрать все обработанные числа обратно в массив.

Задания к главе 12

Вариант А

Разработать многопоточное приложение. Использовать возможности, предоставляемые пакетом **java.util.concurrent**. Не использовать слово *synchronized*. Все сущности, желающие получить доступ к ресурсу, должны быть потоками.

1. **Порт.** Корабли заходят в порт для разгрузки/загрузки контейнеров. Число контейнеров, находящихся в текущий момент в порту и на корабле, должно быть неотрицательным и превышающим заданную грузоподъемность судна и вместимость порта. В порту работает несколько причалов. У одного причала может стоять один корабль. Корабль может загружаться у причала, разгружаться или выполнять оба действия.
2. **Маленькая библиотека.** Доступны для чтения несколько книг. Одинаковых книг в библиотеке нет. Некоторые выдаются на руки, некоторые только в читальный зал. Читатель может брать на руки и в читальный зал несколько книг.
3. **Автостоянка.** Доступно несколько машиномест. На одном месте может находиться только один автомобиль. Если все места заняты, то автомобиль не станет ждать больше определенного времени и уедет на другую стоянку.
4. **CallCenter.** В организации работает несколько операторов. Оператор может обслуживать только одного клиента, остальные должны ждать своей очереди. Клиент может положить трубку и перезвонить еще раз через некоторое время.
5. **Автобусные остановки.** На маршруте несколько остановок. На одной остановке может останавливаться несколько автобусов одновременно, но не более заданного числа.
6. **Свободная касса.** В ресторане быстрого обслуживания есть несколько касс. Посетители стоят в очереди в конкретную кассу, но могут перейти в другую очередь при уменьшении или исчезновении там очереди.
7. **Тоннель.** В горах существует два железнодорожных тоннеля, по которым поезда могут двигаться в обоих направлениях. По обоим концам тоннеля

- собралось много поездов. Обеспечить безопасное прохождение тоннелей в обоих направлениях. Поезд можно перенаправить из одного тоннеля в другой при превышении заданного времени ожидания на проезд.
8. **Банк.** Имеется банк с кассирами, клиентами и их счетами. Клиент может снимать/пополнять/переводить/оплачивать/обменивать денежные средства. Кассир последовательно обслуживает клиентов. Поток-наблюдатель следит, чтобы в кассах всегда были наличные, при скоплении денег более определенной суммы, часть их переводится в хранилище, при истощении запасов наличных происходит пополнение из хранилища.
 9. **Аукцион.** На торги выставляется несколько лотов. Участники аукциона делают заявки. Заявку можно корректировать в сторону увеличения несколько раз за торги одного лота. Аукцион определяет победителя и переходит к следующему лоту. Участник, не заплативший за лот в заданный промежуток времени, отстраняется на несколько лотов от торгов.
 10. **Биржа.** На торгах брокеры предлагают акции нескольких фирм. На бирже совершаются действия по купле-продаже акций. В зависимости от количества проданных-купленных акций их цена изменяется. Брокеры предлагают к продаже некоторую часть акций. От активности и роста-падения котировок акций изменяется индекс биржи. Биржа может приостановить торги при резком падении индекса.
 11. **Аэропорт.** Посадка/высадка пассажиров может осуществляться через конечное число терминалов и наземным способом через конечное число трапов. Самолеты бывают разной вместимости и дальности полета. Организовать функционирование аэропорта, если пунктов назначения 4–6, и зон дальности 2–3.

Тестовые задания к главе 12

Вопрос 12.1.

Дан класс:

```
class GoThread implements Runnable {
    public void run() {
        System.out.println("running...");
    }
}
```

Укажите правильные варианты создания объекта потока, который можно запустить вызовом метода **start()**? (выбрать один)

- a) **new Thread().new GoThread();**
- b) **new Runnable(new GoThread());**
- c) **new Thread(new GoThread());**
- d) **new GoThread().**

Вопрос 12.2.

Сколько независимых друг от друга потоков будет запущено в цикле? (выбрать один)

```
for(int i = 0, j = 10; i <= j; i++, --j) {
    if(i < j - 2) {
        new Thread(new Thread()).start();
    }
}
```

- a) 0
- b) 1
- c) 2
- d) 3
- e) 4
- f) 5
- g) compilation fails

Вопрос 12.3.

Дан код:

```
class WaitMain {
    public static void main(String [] args) {
        System.out.print("0");
        synchronized(args){
            System.out.print("1");
            try {
                args.wait();
            }
            catch(InterruptedException e){}
        }
        System.out.print("2");
    }
}
```

Каким будет результат компиляции и запуска программы? (выбрать один)

- a) 012 и закончит выполнение
- b) 01 и не закончит выполнение
- c) 01 и закончит выполнение
- d) 02 и не закончит выполнение
- e) 02 и закончит выполнение
- f) runtime error: генерация `IllegalMonitorStateException` при вызове `wait()`

Вопрос 12.4.

Дан фрагмент кода:

```
AtomicInteger i = new AtomicInteger(0);
Runnable runnable = () -> System.out.print(i.incrementAndGet());
new Thread(runnable).start();
new Thread(runnable).start();
new Thread(runnable).start();
new Thread(runnable).start();
```

Каким будет результат компиляции и запуска кода в методе **main()**? (выбрать один)

- a) 4444
- b) 0000
- c) 0123 в любой последовательности
- d) 1234 в любой последовательности

Вопрос 12.5.

Дан фрагмент кода:

```
Runnable runnable = () -> System.out.print("R");
Callable<String> callable = () -> "C"; // line 1
ExecutorService service = Executors.newSingleThreadExecutor();
service.submit(runnable); // line 2
Future<String> future = service.submit(callable);
System.out.println(future.get());
service.shutdown();
```

Каким будет результат компиляции и запуска кода в методе **main()**? (выбрать один)

- a) RC
- b) CR
- c) R и генерирует исключение
- d) C и генерирует исключение
- e) compilation fails в строке *line 1*, некорректная реализация интерфейса
- f) compilation fails в строке *line 2*, запуск *runnable* невозможен методом **submit()**

Java DataBase Connectivity

*Пространство — иллюзия,
дисковое пространство — тем более.*

Драйверы, соединения и запросы

API JDBC (*Java DataBase Connectivity*) — стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и СУБД. Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов. В настоящий момент JDBC выделены три типа драйверов:

1. Драйвер, представляющий собой частично библиотеку Java, работающий через *native* библиотеки для взаимодействия с клиентом СУБД.
2. Драйвер только на основе Java, работающий по сетевому и независимому от СУБД протоколу, который, в свою очередь, подключается к клиенту СУБД.
3. Сетевой драйвер, состоящий только из библиотеки Java, работающий напрямую с клиентом СУБД.

Если приложение выполняется на сервере, который не предполагает установки клиента СУБД, то выбор производится между вторым и третьим типами. Причем третий тип работает напрямую с СУБД по ее протоколу, поэтому можно предположить, что драйвер третьего типа будет более эффективным с точки зрения производительности.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных, для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Последовательность действий для выполнения первого запроса.

1. *Подключение библиотеки с классом-драйвером базы данных.*

Дополнительно требуется подключить к проекту библиотеку, содержащую драйвер, поместив ее предварительно в папку **lib** приложения или сервера приложений.

mysql-connector-java-[version]-bin.jar для СУБД MySQL,

ojdbc[version].jar для СУБД Oracle.

2. *Установка соединения с БД.*

До появления JDBC 4.0 объект драйвера СУБД для консольных приложений нужно было регистрировать с помощью вызова:

```
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver()); // for MySQL
DriverManager.registerDriver(new oracle.jdbc.OracleDriver()); // for Oracle
```

или создавать явно

```
Class.forName("com.mysql.cj.jdbc.Driver");
Class.forName("oracle.jdbc.OracleDriver");
```

В настоящее время в большинстве случаев в этом нет необходимости, так как экземпляр драйвера загружается автоматически при попытке получения соединения с БД.

Для установки соединения с БД вызывается один из перегруженных статических методов **getConnection()** класса **java.sql.DriverManager**. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Метод возвращает объект **java.sql.Connection**. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов.

```
Connection connection = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/testphones", "root", "pass");
Connection connection = DriverManager.getConnection(
    "jdbc:oracle:thin:@//localhost:1521:testphones", "system", "pass");
```

В результате будет возвращен объект **Connection** и создано одно установленное соединение с БД, именуемой **testphones**. Класс **DriverManager** предоставляет средства для управления набором драйверов баз данных. С помощью метода **getDrivers()**, **Stream<Driver> drivers()** можно получить список всех доступных драйверов.

3. Создание объекта для передачи запросов.

После создания объекта **Connection** и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект **java.sql.Statement**, создаваемый вызовом метода **createStatement()** класса **Connection**.

```
Statement statement = connection.createStatement();
```

Объект класса, реализующего интерфейс **Statement**, используется для прямого выполнения SQL-запроса. Могут применяться также объекты классов **PreparedStatement** и **CallableStatement** для выполнения подготовленных запросов и хранимых процедур. Оба этих интерфейса наследуют возможности интерфейса **Statement**.

Метод **createStatement(int resultSetType, int resultSetConcurrency)** позволяет установить условия прокрутки и изменения объекта **ResultSet**.

Параметр **resultSetType** со значением **ResultSet.TYPE_FORWARD_ONLY** позволяет продвигаться по объекту только от начала к концу и выставляется по умолчанию. Значение **ResultSet.TYPE_SCROLL_INTENSIVE** разрешает навигацию в обе стороны и не учитывает изменения от других пользователей.

ResultSet.TYPE_SCROLL_SENSITIVE разрешает навигацию в обе стороны и учитывает изменения от других пользователей после того, как **ResultSet** был получен.

Значение **ResultSet.CONCUR_READ_ONLY** параметра **resultSetConcurrency** позволяет только читать результат и устанавливается по умолчанию. Значение **ResultSet.CONCUR_UPDATABLE** создает **ResultSet** с возможностью изменения данных.

4. *Выполнение запроса.*

Созданный объект **Statement** можно использовать для выполнения запросов SQL, передавая их в один из методов:

ResultSet executeQuery(String sql) — выполняет запросы **SELECT**. Результаты выборки из базы помещаются в объект **ResultSet**:

int executeUpdate(String sql) — выполняет запросы, изменяющие состояние базы **INSERT**, **UPDATE**, **DELETE**. Возвращает количество строк, задействованных запросом;

boolean execute(String sql) — применяется для выполнения произвольных запросов;

int[] executeBatch() — выполняет *batch*-команды, т.е. группу запросов, как один запрос

```
// extract all data from the phonebook table
```

```
ResultSet resultSet = statement.executeQuery("SELECT idphonebook, lastname, phone FROM phonebook");
```

5. *Обработка результатов запроса* на выборку данных производится методами интерфейса **ResultSet**, где самыми распространенными являются **next()**, **first()**, **previous()**, **last()**, **beforeFirst()**, **afterLast()**, **isFirst()**, **isLast()**, **absolute(int i)** — методы навигации по строкам таблицы результатов, группа методов по доступу к информации по номеру позиции в записи вида **String getString(int pos)**, а также аналогичные методы, начинающиеся с **getType(int pos)** (**int getInt(int pos)**, **float getFloat(int pos)** и др.) и **updateType()**. Среди них следует выделить методы **getClob(int pos)** и **getBlob(int pos)**, позволяющие извлекать из полей таблицы специфические объекты (*Character Large Object*, *Binary Large Object*), которые могут быть, например, графическими или архивными файлами.

Следует обратить внимание, что счет позиций в **ResultSet** начинается с «1», а не с «0», как в коллекциях и массивах.

Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его имени в строке результатов методами типа **int getInt(String columnLabel)**, **String getString(String columnLabel)**, **Object getObject(String columnLabel)** и подобными им.

Обновляемый набор данных позволяет обновлять, изменять и **ResultSet**, и информацию в таблице базы данных: **updateRow()**, **insertRow()**, **updateString()** и др.

При первом вызове метода **next()** указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение **false**.

6. Закрытие соединения.

```
connection.close(); // closes also Statement & ResultSet
```

Когда база больше не нужна, соединение должно быть закрыто. Для того, чтобы правильно пользоваться приведенными методами, программисту как минимум требуется знать SQL, способ организации конкретной БД, типы полей БД и др.

7. Выгрузка драйверов.

По завершении работы приложения следует выгрузить или deregистрировать драйвер:

```
DriverManager.getDrivers().asIterator().forEachRemaining(driver -> {
    try {
        DriverManager.deregisterDriver(driver);
    } catch (SQLException e) {
        // Log
    }
});
```

СУБД MySQL

СУБД MySQL совместима с JDBC и будет применяться для создания учебных баз данных. Версия СУБД может быть загружена с сайта **www.mysql.com**. Для корректной установки необходимо следовать инструкциям мастера. В процессе установки следует создать администратора СУБД с именем **root** и паролем, например, **pass**. Если планируется разворачивать реально работающее приложение, стоит исключить тривиальных пользователей сервера БД, иначе злоумышленники могут получить полный доступ к БД. Для запуска следует использовать команду из папки **/mysql/bin**:

```
mysqld -nt -standalone
```

Если не появится сообщение об ошибке, то СУБД MySQL запущена. Для создания базы данных и ее таблиц используются команды языка SQL.

Соединение и запрос

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем **testphones** и одной таблицей **PHONEBOOK**. Таблица должна содержать три поля: числовое (первичный ключ) — **IDPHONEBOOK**, символьное — **LASTNAME** и числовое — **PHONE** и несколько занесенных записей.

IDPHONEBOOK	LASTNAME	PHONE
1	Руденко	7756544
2	Artukevich	6861880

При создании таблицы следует задавать кодировку UTF-8, поддерживающую хранение любых символов.

Приложение, осуществляющее простейший запрос на выбор всей информации из таблицы:

```
/* # 1 # соединение с БД и простой запрос # SimpleJdbcMain.java */
```

```
package by.epam.learn.db;
import by.epam.learn.entity.Abonent;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
public class SimpleJdbcMain {
    public static void main(String[] args) {
        try {
            DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
        } catch (SQLException e) {
            e.printStackTrace();
        }
        String url = "jdbc:mysql://localhost:3306/testphones";
        Properties prop = new Properties();
        prop.put("user", "root");
        prop.put("password", "pass");
        prop.put("autoReconnect", "true");
        prop.put("characterEncoding", "UTF-8");
        prop.put("useUnicode", "true");
        prop.put("useSSL", "true");
        prop.put("useJDBCCompliantTimezoneShift", "true");
        prop.put("useLegacyDatetimeCode", "false");
        prop.put("serverTimezone", "UTC");
        prop.put("serverSslCert", "classpath:server.crt");
        try (Connection connection = DriverManager.getConnection(url, prop);
            Statement statement = connection.createStatement()) {
            String sql = "SELECT idphonebook, lastname, phone FROM phonebook";
            ResultSet resultSet = statement.executeQuery(sql);
            List<Abonent> abonents = new ArrayList<>();
            while (resultSet.next()) {
                int id = resultSet.getInt(1);
                String name = resultSet.getString(2);
                int number = resultSet.getInt("phone");
                abonents.add(new Abonent(id, name, number));
            }
            System.out.println(abonents);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

В простом приложении достаточно контролировать закрытие соединения, так как незакрытое или «провисшее» соединение снижает быстродействие СУБД. Объект **ResultSet** также нуждается в закрытии, но его автоматически закрывает метод **close()** интерфейса **Statement** при закрытии или механизм **AutoCloseable**.

Класс **Abonent**, используемый приложением для хранения информации, извлеченной из БД, выглядит очень просто:

```
/* # 2 # классы с информацией # Entity.java # Abonent.java */

package by.epam.learn.entity;
import java.io.Serializable;
public abstract class Entity implements Serializable, Cloneable {
}
package by.epam.learn.entity;
public class Abonent extends Entity {
    private int id;
    private String name;
    private int phone;
    public Abonent() {
    }
    public Abonent(int id, String name, int phone) {
        this.id = id;
        this.name = name;
        this.phone = phone;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getPhone() {
        return phone;
    }
    public void setPhone(int phone) {
        this.phone = phone;
    }
    public String toString() {
        final StringBuilder sb = new StringBuilder("Abonent{");
        sb.append("id=").append(id).append(", name=").append(name).append('\n');
        sb.append(", phone=").append(phone).append('}');
        return sb.toString();
    }
}
```

Параметры соединения можно задавать: с помощью прямой передачи значений в коде класса, а также с помощью файлов **properties**, **json**, **yaml** или **xml**. Окончательный выбор производится в зависимости от конфигурации проекта.

Чтение параметров соединения с базой данных и получение соединения следует вынести в отдельный класс. Пусть класс **ConnectorCreator** использует файл ресурсов **database.properties**, в котором хранятся, как правило, такие параметры подключения к БД, как логин, пароль доступа и др.

```
db.driver=com.mysql.cj.jdbc.Driver
user      = root
password = pass
poolsize = 32
db.url = jdbc:mysql://localhost:3306/testphones
useUnicode = true
encoding = UTF-8
useSSL = true
useJDBCCompliantTimezoneShift = true
useLegacyDatetimeCode = false
serverTimezone = UTC
serverSslCert = classpath:server.crt
```

Код класса **ConnectionCreator** может выглядеть следующим образом:

```
/* # 3 # создание соединений с БД # ConnectionCreator.java */

package by.epam.learn.db;
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
public class ConnectionCreator {
    private static final Properties properties = new Properties();
    private static final String DATABASE_URL;
    static {
        try {
            properties.load(new FileReader("datares\\database.properties"));
            String driverName = (String) properties.get("db.driver");
            Class.forName(driverName);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace(); // fatal exception
        }
        DATABASE_URL = (String) properties.get("db.url");
    }
    private ConnectionCreator() {}
    public static Connection createConnection() throws SQLException {
        return DriverManager.getConnection(DATABASE_URL, properties);
    }
}
```

В таком случае получение соединения с БД сведется к вызову:

```
Connection connection = ConnectionCreator.createConnection();
```

Класс **ConnectorCreator** лучше сделать обычным синглтоном.

Добавление и изменение записи

Объект **ResultSet** позволяет вставлять запись в базу данных без дополнительных запросов. Объект **Statement** нужно создавать с разрешением на изменение в базе данных.

```
try (Connection connection = DriverManager.getConnection(url, prop);
    Statement statement = connection.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
    ResultSet resultSet = statement.executeQuery(
        "SELECT idphonebook, lastname, phone FROM phonebook");
    resultSet.moveToInsertRow(); // insert row
    resultSet.updateInt(1, 77);
    resultSet.updateString(2, "Bahdanovich");
    resultSet.updateInt(3, 222322);
    resultSet.insertRow();
    resultSet.moveToCurrentRow();
} catch (SQLException e) {
    e.printStackTrace();
}
}
```

Изменения в базу данных также легко вносятся с помощью возможностей **ResultSet**:

```
while (resultSet.next()) {
    int id = resultSet.getInt(1);
    if (id == 2) {
        resultSet.updateInt("phone", 550055); // update row
        resultSet.updateRow();
    }
}
```

В результате у записи с **IDPHONEBOOK=2** номер телефона будет заменен как в **ResultSet**, так и в базе данных.

Метаданные

Существует целый ряд методов интерфейсов **ResultSetMetaData** и **DatabaseMetaData** для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД. Для записей подобных методов нет.

Получить объект **ResultSetMetaData** можно следующим образом:

```
ResultSetMetaData rsMetaData = resultSet.getMetaData();
```

Некоторые методы интерфейса **ResultSetMetaData**:

int getColumnCount() — возвращает число столбцов набора результатов объекта **ResultSet**;

String getColumnName(int column) — возвращает имя указанного столбца;

String getColumnName(int column) — возвращает тип данных указанного столбца.

Если добавить следующий код к примеру #1

```
System.out.println("ColumnCount: " + rsMetaData.getColumnCount());
System.out.println("ColumnName: " + rsMetaData.getColumnName(1));
System.out.println("ColumnType: " + rsMetaData.getColumnTypeName(1));
System.out.println("isAutoIncrement: " + rsMetaData.isAutoIncrement(1));
System.out.println("ColumnName: " + rsMetaData.getColumnName(2));
System.out.println("ColumnType: " + rsMetaData.getColumnTypeName(2));
System.out.println("isAutoIncrement: " + rsMetaData.isAutoIncrement(2));
```

то в консоль будет выведено:

ColumnCount: 3

ColumnName: idphonebook

ColumnType: INT

isAutoIncrement: true

ColumnName: lastname

ColumnType: VARCHAR

isAutoIncrement: false

Получить объект **DatabaseMetaData** можно следующим образом:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
```

Некоторые методы обширного интерфейса **DatabaseMetaData**:

String getDatabaseProductName() — возвращает название СУБД;

String getDatabaseProductVersion() — номер версии СУБД;

String getDriverName() — имя драйвера JDBC;

String getUserName() — имя пользователя БД;

String getURL() — местонахождение источника данных;

ResultSet getTables() — набор типов таблиц, доступных для данной БД.

Если добавить следующий код к примеру #1

```
System.out.println("DatabaseName: " + dbMetaData.getDatabaseProductName());
System.out.println("DatabaseVersion: " + dbMetaData.getDatabaseProductVersion());
System.out.println("UserName: " + dbMetaData.getUserName());
System.out.println("URL: " + dbMetaData.getURL());
```

то в консоль будет выведено:

DatabaseName: MySQL
DatabaseVersion: 5.7.15-log
UserName: root@localhost
URL: jdbc:mysql://localhost:3306/testphones

Подготовленные запросы и хранимые процедуры

Для представления запросов существует еще два типа объектов **PreparedStatement** и **CallableStatement**. Объекты первого типа используются при выполнении часто повторяющихся запросов SQL. Такой оператор предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов. Второй интерфейс используется для выполнения хранимых процедур, созданных средствами самой СУБД.

При использовании **PreparedStatement** невозможен *sql injection attacks*. То есть, если существует возможность прямой передачи в запрос информации в виде строки, то следует использовать для выполнения такого запроса объект **PreparedStatement**.

Для подготовки SQL-запроса, в котором отсутствуют конкретные параметры, используется метод **prepareStatement(String sql)** интерфейса **Connection**, возвращающий объект **PreparedStatement**.

```
String sql = "INSERT INTO phonebook(idphonebook, lastname, phone) VALUES(?, ?, ?)";
PreparedStatement statement = connection.prepareStatement(sql);
```

Установка входных значений конкретных параметров этого объекта производится с помощью методов **setString(int index, String x)**, **setInt(int index, int x)** и подобных им, после чего и осуществляется непосредственное выполнение запроса методами **int executeUpdate()**, **ResultSet executeQuery()**.

```
/* # 4 # подготовка запроса на добавление информации # PreparedMain.java */
```

```
package by.epam.learn.db;
import java.sql.*;
public class PreparedMain {
    public static void main(String[] args) {
        try(Connection connection = ConnectionCreator.createConnection()){
            String sql = "INSERT INTO phonebook(idphonebook, lastname, phone) VALUES (?, ?, ?)";
            PreparedStatement statement = connection.prepareStatement(sql);
            statement.setInt(1, 43);
            statement.setString(2, "Skaryna");
            statement.setInt(3, 990077);
            int rowsUpdate = statement.executeUpdate();
            System.out.println(rowsUpdate);
        }
    }
}
```

```

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

Так как данный оператор предварительно подготовлен, то он выполняется быстрее обычных операторов, ему соответствующих. Оценить преимущества во времени можно, выполнив большое число повторяемых запросов с предварительной подготовкой запроса и без нее, и сравнив время выполнения.

Как известно, таблицы базы данных обычно содержат столбец, помеченный как первичный ключ, значение которого уникально у каждой записи таблицы. Для первичных ключей разработаны различные механизмы автогенерации уникального значения. При добавлении записи в базу данных разработчику можно не знать об этих правилах, и тогда запрос на добавление данных не должен содержать информации о первичном ключе. Значение первичного ключа для этой записи будет сгенерировано базой данных автоматически.

Значение же сгенерированного ключа может понадобиться сразу же. Выполнять для его получения запрос **SELECT** неэкономично. Метод **getGeneratedKeys()** возвратит значение ключа. Объект **PreparedStatement** должен быть создан с параметром **Statement.RETURN_GENERATED_KEYS**.

```

String sql = "INSERT INTO phonebook(lastname, phone) VALUES (?, ?)";

PreparedStatement statement = connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
statement.setString(1, "Kalinouski");
statement.setInt(2, 186300);
statement.executeUpdate();
ResultSet resultSet = statement.getGeneratedKeys();
if(resultSet.next()) {
    int key = resultSet.getInt(1);
    System.out.println(key);
}

```

Механизм автогенерации также удобен во избежание ошибки дублирования ключа. Пользователь может пытаться добавить запись с уже существующим в базе данных значением. Или возможна неточность порядка значения. Например, все значения ключей в базе двадцатизначные, а пользователь добавил двузначное значение. Ошибки это не вызовет, но внешне такие ключи будут выглядеть несогласованно.

Интерфейс **CallableStatement** имеет более широкие возможности, чем интерфейс **PreparedStatement**, поэтому обеспечивает выполнение хранимых процедур.

Хранимая процедура — это, в общем случае, именованная последовательность команд SQL, рассматриваемых как единое целое, и выполняющаяся в адресном пространстве процессов СУБД, которую можно вызвать извне (в зависимости от политики доступа используемой СУБД). В данном случае

хранимая процедура будет рассматриваться в более узком смысле как последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных. Для создания объекта **CallableStatement** вызывается метод **prepareCall()** объекта **Connection**.

Интерфейс **CallableStatement** позволяет исполнять хранимые процедуры, которые находятся непосредственно в БД. Одна из особенностей этого процесса в том, что **CallableStatement** способен обрабатывать не только входные (IN) параметры, но выходящие (OUT) и смешанные (INOUT) параметры. Тип выходного параметра должен быть зарегистрирован с помощью метода **registerOutParameter()**. После установки входных и выходных параметров вызываются методы **execute()**, **executeQuery()** или **executeUpdate()**.

Пусть в СУБД MySQL существует хранимая процедура **findlastname**, которая по уникальному номеру телефона для каждой записи в таблице **phonebook** будет возвращать соответствующее ему имя:

```
CREATE DEFINER=`root`@`localhost`
PROCEDURE `findlastname`(IN p_phone INT, OUT p_lastname VARCHAR(40))
BEGIN
SELECT lastname INTO p_lastname FROM phonebook WHERE phone = p_phone;
END
```

Тогда для получения имени через вызов данной процедуры необходимо исполнить java-код вида:

```
final String SQL = "{call findlastname (?, ?)}";
CallableStatement statement = connection.prepareCall(SQL);
statement.setInt(1, 654321);
statement.registerOutParameter(2, java.sql.Types.VARCHAR);
statement.execute();
String lastName = statement.getString(2);
```

В JDBC также существует механизм *batch*-команд, который позволяет запустить на исполнение в БД массив запросов SQL, как одну единицу.

```
/* # 5 # выполнение массива запросов # BatchMain.java */
```

```
package by.epam.learn.db;
import java.sql.*;
import java.util.Arrays;
public class BatchMain {
    public static void main(String[] args) {
        Connection connection = null;
        try {
            connection = ConnectionCreator.createConnection();
            connection.setAutoCommit(false); // turn off autocommit
            Statement statement = connection.createStatement();
            statement.addBatch("INSERT INTO phonebook VALUES (92, 'Sapega', 112211)");
```

```

statement.addBatch("INSERT INTO location VALUES (201, 'Minsk')");
statement.addBatch("INSERT INTO location VALUES (202, 'polotesk')");
// submit a batch of update commands for execution
int[] updateCounts = statement.executeBatch();
connection.commit();
System.out.println(Arrays.toString(updateCounts));
} catch (SQLException e) {
    try {
        if(connection != null) {
            connection.rollback();
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
} finally {
    try {
        if(connection != null) { // turn on autocommit
            connection.setAutoCommit(true);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    try {
        if(connection != null) {
            connection.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
}

```

Чтобы запустить это приложение, необходимо в базу данных **testphones** добавить таблицу **location** со столбцами **idcity** и **city**.

Удалить одну из команд нельзя, можно лишь очистить полностью методом **clearBatch()**.

Если используется объект **PreparedStatement**, *batch*-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров.

Метод **executeBatch()** интерфейса **PreparedStatement** возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из *batch*-команды.

Пусть существует список объектов типа **Abonent** со стандартным набором геттеров и сеттеров для каждого из его полей, и необходимо внести их значения в БД. Многократное выполнение методов **execute()** или **executeUpdate()** становится неэффективным, и в данном случае лучше использовать схему *batch*-команд:

```

try {
    List<Abonent> abonents = new ArrayList<>();
    // filling abonents
    PreparedStatement statement = connection.prepareStatement("INSERT INTO phonebook VALUES(?,?,?)");
    for (Abonent abonent : abonents) {
        statement.setInt(1, abonent.getId());
        statement.setString(2, abonent.getName());
        statement.setInt(3, abonent.getPhone());
        statement.addBatch();
    }
    int[] updateCounts = statement.executeBatch();
} catch (SQLException throwables) {
    throwables.printStackTrace();
}

```

Транзакции

При проектировании информационных систем возникают ситуации, когда сбой в системе или какой-либо ее периферийной части может привести к утрате информации или к финансовым потерям. Простейшим примером может служить ситуация с перечислением денег с одного счета на другой. Если сбой произошел в тот момент, когда операция снятия денег с одного счета уже произведена, а операция зачисления на другой счет еще не произведена, то система, допускающая такие ситуации, должна быть признана не отвечающей требованиям заказчика. Или должны выполняться обе операции, или не выполняться вовсе. Такие две операции трактуют как одну и называют транзакцией.

Транзакция, или деловая операция, определяется как единица работы, обладающая свойствами ACID:

- *Атомарность* — две или более операций, выполняются все или не выполняются ни одна.
- *Согласованность* — при возникновении сбоя система возвращается в состояние до начала неудавшейся транзакции. Если транзакция завершается успешно, то проверка согласованности удостоверяется в успешном завершении всех операций транзакции.
- *Изолированность* — во время выполнения транзакции все объекты-сущности, участвующие в ней, должны быть синхронизированы.
- *Долговечность* — все изменения, произведенные с данными во время транзакции, обязательно сохраняются, например, в базе данных, что позволяет восстанавливать систему.

Для фиксации результатов работы SQL-операторов, логически выполняемых в рамках некоторой транзакции, используется SQL-оператор **COMMIT**. В API JDBC эта операция выполняется по умолчанию после каждого вызова методов **executeQuery()** и **executeUpdate()**. Если же необходимо сгруппировать запросы

и только после этого выполнить операцию **COMMIT**, сначала вызывается метод **setAutoCommit(boolean param)** интерфейса **Connection** с параметром **false**, в результате выполнения которого текущее соединение с БД переходит в режим неавтоматического подтверждения операций. После этого выполнение любого запроса на изменение информации в таблицах базы данных не приведет к необратимым последствиям, пока операция **COMMIT** не будет выполнена непосредственно. Подтверждает выполнение SQL-запросов метод **commit()** интерфейса **Connection**, в результате действия которого все изменения таблицы производятся как одно логическое действие. Если же транзакция не выполнена, то методом **rollback()** отменяются действия всех запросов SQL, начиная от последнего вызова **commit()**. В следующем примере информация добавляется в таблицу в режиме действия транзакции, подтвердить или отменить действия которой можно, снимая или добавляя комментарий в строках вызова методов **commit()** и **rollback()**.

Схематически транзакцию можно представить в виде:

```
try(Connection connection = /* take connection */) {
    connection.setAutoCommit(false); // 1
    Statement statement = connection.createStatement();
    statement.executeUpdate("some update/insert/delete/select query_1"); // 2
    statement.executeUpdate("some update/insert/delete/select query_2"); // 2
    connection.commit(); // 3a
} catch (SQLException e) {
    connection.rollback(); // 3b
    //log or throw
} finally {
    connection.setAutoCommit(true); // 4
}
```

Цифрами обозначена последовательность действий.

Для транзакций существует несколько типов чтения:

- грязное чтение (*dirty reads*) происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;
- неповторяющееся чтение (*nonrepeatable reads*) происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;
- фантомное чтение (*phantom reads*) происходит, когда транзакция А считывает все строки, удовлетворяющие **WHERE**-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие **WHERE**-условию, уже вместе с новой строкой или недосчитавшись старой.

JDBC удовлетворяет уровням изоляции транзакций, определенным в стандарте SQL:2003.

Уровни изоляции транзакций определены в виде констант интерфейса **Connection** (по возрастанию уровня ограничения):

- **TRANSACTION_NONE** — информирует о том, что драйвер не поддерживает транзакции;
- **TRANSACTION_READ_UNCOMMITTED** — позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, неповторяющееся и фантомное чтения;
- **TRANSACTION_READ_COMMITTED** — означает, что любое изменение, сделанное в транзакции, не видно вне ее, пока она не сохранена, что предотвращает грязное чтение, но разрешает неповторяющееся и фантомное;
- **TRANSACTION_REPEATABLE_READ** — запрещает грязное и неповторяющееся чтение, но фантомное разрешено;
- **TRANSACTION_SERIALIZABLE** — определяет, что грязное, неповторяющееся и фантомное чтения запрещены.

Метод **boolean supportsTransactionIsolationLevel(int level)** интерфейса **DatabaseMetaData** определяет, поддерживается ли заданный уровень изоляции транзакций.

В свою очередь, методы интерфейса **Connection** определяют доступ к уровню изоляции:

int getTransactionIsolation() — возвращает текущий уровень изоляции;

void setTransactionIsolation(int level) — устанавливает необходимый уровень.

Точки сохранения

Точки сохранения, представляемые классом **java.sql.Savepoint**, дают дополнительный контроль над транзакциями, привязывая изменения СУБД к конкретной точке в области транзакции. Фактически это транзакция внутри транзакции. Установкой точки сохранения обозначается логическая точка внутри транзакции, которая может быть использована для отката данных. Подтвердить **Savepoint** невозможно, все точки сохранения автоматически подтверждаются с подтверждением всей транзакции. Таким образом, если произойдет ошибка, можно вызвать метод **rollback(Savepoint point)** для отмены всех изменений, которые были сделаны после точки сохранения. Метод **boolean supportsSavepoints()** интерфейса **DatabaseMetaData** используется для того, чтобы определить, поддерживают ли точки сохранения сама СУБД и ее драйвер JDBC. Методы **setSavepoint(String name)** и **setSavepoint()** возвращают объект **Savepoint** интерфейса **Connection** и используются для установки именованной или неименованной точки сохранения во время текущей транзакции. При этом новая транзакция будет начата, если в момент вызова **setSavepoint()** не будет активной транзакции.

Savepoint используется при сложном взаимодействии с базой данных в пределах одного логического действия операции. Например, покупка авиабилета. На первой странице заполняется форма с данными покупателя, на второй — условия багажа и страховки, на третьей — внесение информации о банковской карте. Если же пользователь решил вернуться на шаг назад, то с применением **Savepoint** можно отменить только этот шаг, не отменяя все действия целиком.

Data Access Object

При создании информационной системы выявляются некоторые слои, которые отвечают за взаимодействие различных частей приложения. Связь с базой данных является важной частью любой системы, поэтому всегда выделяется часть кода, ответственная за передачу запросов в базу данных и обработку полученных от нее ответов.

Общее определение шаблона DAO трактует его как прослойку между приложением и СУБД. DAO инкапсулирует и абстрагирует бизнес-сущности системы и отображает реляционные данные в БД и обратно. Можно трактовать DAO как введение ORM-технологий, представленных в настоящее время Hibernate, JPA, EclipseLink и др.

DAO определяет общие способы использования соединения с БД, моменты его открытия и закрытия или извлечения и возвращения в пул. В общем случае DAO можно определять таким образом, чтобы была возможность подмены одной модели базы данных другой. Например, реляционную заменить на объектную или, что проще, MySQL на Oracle или Derby. В практическом программировании такие глобальные задачи ставятся крайне редко, поэтому будет приведено несколько способов организации взаимодействия с БД, отличающихся уровнем использования соединения с БД и организацией работы с бизнес-сущностями. В общем случае DAO определяет только стандартные CRUD-методы.

Вершина иерархии DAO представляет собой класс или интерфейс с описанием набора методов, которые будут использоваться при взаимодействии с таблицей или группой таблиц. Как правило, это методы выбора, поиска сущности по признаку, добавление, удаление и замена информации.

```
/* # 6 # общие методы взаимодействия с моделью данных # BaseDao.java */
```

```
package by.epam.learn.dao;
import by.epam.learn.entity.Entity;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
public interface BaseDao <K, T extends Entity> {
    List<T> findAll() throws DaoException;
    T findEntityById(K id) throws DaoException;
```

```

boolean delete(T t) throws DaoException;
boolean delete(K id) throws DaoException;
boolean create(T t) throws DaoException;
T update(T t) throws DaoException;
default void close(Statement statement) {
    try {
        if (statement != null) {
            statement.close();
        }
    } catch (SQLException e) {
        // Log
    }
}
default void close(Connection connection) {
    try {
        if (connection != null) {
            connection.close(); // or connection return code to the pool
        }
    } catch (SQLException e) {
        // Log
    }
}
}

```

Набор методов может варьироваться в зависимости от логики приложения. Параметр **K** описывает, как правило, ключ в таблице, так как крайне редко таблица, содержащая описание сущности, обходится без первичного ключа. Параметр **Entity** определяет общую бизнес-сущность, от которой наследуются все бизнес-сущности системы. В класс включены методы возвращения соединения в пул или его закрытия, а также закрытия экземпляра **Statement** в виде.

Все абстрактные методы в своей сигнатуре содержат **throws DaoException**, генерация которого будет происходить в случае возникновения **SQLException**, которое, в свою очередь, не может быть обработано в слое DAO и передается на уровень вне DAO, чтобы вызывающий метод был уведомлен об ошибке, и уже разработчик принимал решение о реакции на это исключение.

```
/* # 7 # класс-исключение для слоя DAO # DaoException.java */
```

```

package by.epam.learn.dao;
public class DaoException extends Exception {
    public DaoException() {
    }
    public DaoException(String message) {
        super(message);
    }
    public DaoException(String message, Throwable cause) {
        super(message, cause);
    }
}

```

```

    public DaoException(Throwable cause) {
        super(cause);
    }
}

```

DAO. Уровень метода

На уровне конкретного бизнес-объекта создается интерфейс, наследующий **BaseDao** с сигнатурами методов, характерных только для этой сущности, определять которые в более общем классе не имеет смысла из-за узкой области применения. В данном случае это абстрактный метод **List<Abonent> findAbonentByLastName(String patternName)**.

```

package by.epam.learn.dao;
import by.epam.learn.entity.Abonent;
import java.util.List;
public interface AbonentDao extends BaseDao <Long, Abonent> {
    List<Abonent> findAbonentByLastname(String patternName) throws DaoException;
}

```

Реализация на уровне метода предполагает использование соединения для выполнения единственного запроса, т.е. соединение будет получено из пула в начале работы метода и возвращено по его окончании, что в общем случае не является экономным решением. Часть методов может остаться нереализованной.

В простейшем варианте пул соединений применяться не будет.

```

/* # 8 # реализация взаимодействия с моделью данных # AbonentDaoImpl.java */

```

```

package by.epam.learn.dao.impl;
import by.epam.learn.dao.AbonentDao;
import by.epam.learn.dao.DaoException;
import by.epam.learn.db.ConnectionCreator;
import by.epam.learn.entity.Abonent;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
public class AbonentDaoImpl implements AbonentDao {
    private static final String SQL_SELECT_ALL_ABONENTS =
        "SELECT idphonebook, lastname,phone FROM phonebook";
    private static final String SQL_SELECT_ABONENT_BY_LASTNAME =
        "SELECT idphonebook, phone FROM phonebook WHERE lastname=?";
    @Override
    public List<Abonent> findAbonentByLastname(String namePattern) throws DaoException {
        List<Abonent> abonents = new ArrayList<>();
        Connection connection = null;
        PreparedStatement statement = null;
    }
}

```

```

try {
    connection = ConnectionCreator.createConnection();
    statement = connection.prepareStatement(SQL_SELECT_ABONENT_BY_LASTNAME);
    statement.setString(1, namePattern);
    ResultSet resultSet = statement.executeQuery();
    while(resultSet.next()){
        Abonent abonent = new Abonent();
        abonent.setId(resultSet.getInt("idphonebook"));
        abonent.setPhone(resultSet.getInt("phone"));
        abonent.setName(resultSet.getString("lastname"));
        abonents.add(abonent);
    }
} catch (SQLException e) {
    throw new DaoException(e);
} finally {
    close(statement);
    close(connection);
}
return abonents;
}

@Override
public List<Abonent> findAll() throws DaoException {
    List<Abonent> abonents = new ArrayList<>();
    Connection connection = null;
    Statement statement = null;
    try {
        connection = ConnectionCreator.createConnection();
        statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery(SQL_SELECT_ALL_ABONENTS);
        while(resultSet.next()){
            Abonent abonent = new Abonent();
            abonent.setId(resultSet.getInt("idphonebook"));
            abonent.setPhone(resultSet.getInt("phone"));
            abonent.setName(resultSet.getString("lastname"));
            abonents.add(abonent);
        }
    } catch (SQLException e) {
        throw new DaoException(e);
    } finally {
        close(statement);
        close(connection);
    }
    return abonents;
}

@Override
public Abonent findEntityById(Long id) throws DaoException {
    throw new UnsupportedOperationException();
}

@Override
public boolean delete(Abonent abonent) throws DaoException {

```

```

        throw new UnsupportedOperationException();
    }
    @Override
    public boolean delete(Long id) throws DaoException {
        throw new UnsupportedOperationException();
    }
    @Override
    public boolean create(Abonent abonent) throws DaoException {
        throw new UnsupportedOperationException();
    }
    @Override
    public Abonent update(Abonent abonent) throws DaoException {
        throw new UnsupportedOperationException();
    }
}

```

SQL-запросы размещаются в статических константах класса. В редких случаях запросы могут храниться в **xml** или **properties** файлах.

DAO. Уровень слоя

На практике чаще всего возникает необходимость при выполнении запроса пользователя обращаться сразу к нескольким реализациям DAO и использовать при этом единственное соединение с БД. Соединение с БД создается или извлекается из пула после создания экземпляров DAO, передается в эти экземпляры, а затем закрывается или возвращается в пул после выполнения всех обращений к БД в пределах одного метода бизнес-логики.

Реализация на уровне слоя подразумевает использование одного соединения к базе данных для вызова нескольких методов различных DAO классов. Это решение позволяет экономично организовать транзакцию на единственном соединении.

Вершиной иерархии DAO в качестве поля может объявлять само соединение к СУБД или его оболочка:

```

/* # 9 # DAO с полем Connection без прямой инициализации # AbstractDAO.java */
package by.epam.learn.daologic;
import by.epam.learn.entity.Entity;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
public abstract class AbstractDao <T extends Entity> {
    protected Connection connection; // or WrapperConnection or ProxyConnection
    // constructors
    public abstract List<T> findAll();
    public abstract T findEntityById(long id);
    public abstract boolean delete(long id);
}

```

```

public abstract boolean delete(T entity);
public abstract boolean create(T entity);
public abstract T update(T entity);
public void close(Statement statement) {
    try {
        if (statement != null) {
            statement.close();
        }
    } catch (SQLException e) {
        // Log
    }
}
public void setConnection(Connection connection) {
    this.connection = connection;
}
}

```

Класс-оболочка соединения может определяться следующим образом:

```

package by.epam.learn.daologic;
import java.sql.Connection;
public class WrapperConnection {
    private Connection connection;
    // constructors, methods
}

```

Реализация с классом-оберткой для соединения, инкапсулирующим процесс создания соединения, резко затрудняет попадание в пул «диких» соединений, созданных разработчиком вне пула.

```

/* # 10 # условные реализации DAO # AbonentDao.java # OrderDao.java */

```

```

package by.epam.learn.daologic;
import by.epam.learn.entity.Abonent;
public class AbonentDao extends AbstractDao<Abonent> {
    // implementation
}
package by.epam.learn.daologic;
import by.epam.learn.entity.Order;
public class OrderDao extends AbstractDao<Order> {
    // implementation
}

```

где **Order** представляет класс-сущность в виде подкласса класса **Entity**.

При передаче объекта соединения с БД извне, реализация конкретного DAO никогда не должна закрывать соединение в методе. Соединение закрывается в той части бизнес-логики, откуда выполняются обращения к DAO. В методах DAO остаются возможности по созданию экземпляра **Statement** для выполнения запросов и его закрытию. Реализации методов DAO здесь не приводятся, так как они похожи на методы из предыдущего раздела.

Класс **EntityTransaction**, инициализирующий переданные ему объекты DAO соединением с БД, выглядит примерно так:

```
/* # 11 # инициализатор для объектов DAO # EntityTransaction.java */

package by.epam.learn.daologic;
import java.sql.Connection;
import java.sql.SQLException;
public class EntityTransaction {
    private Connection connection;
    public void initTransaction(AbstractDao dao, AbstractDao... daos) {
        if (connection == null) {
            // connection = // code -> create connection or take connection from pool
        }
        try {
            connection.setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        dao.setConnection(connection);
        for (AbstractDao daoElement : daos) {
            daoElement.setConnection(connection);
        }
    }
    public void endTransaction() {
        // connection check code of null
        try {
            // connection check code for commit
            connection.setAutoCommit(true);
        } catch (SQLException e) {
            // Log
        }
        // connection return code to the pool or closing
        connection = null;
    }
    public void commit() {
        try {
            connection.commit();
        } catch (SQLException e) {
            // Log
        }
    }
    public void rollback() {
        try {
            connection.rollback();
        } catch (SQLException e) {
            // Log
        }
    }
}
```

Соединение с базой данных инициализирует метод **initTransaction()**: он либо создает его, либо получает из пула.

Для выполнения обычных запросов, не являющихся транзакциями, в класс необходимо добавить методы инициализации и закрытия, например, вида:

```
public void init(AbstractDao dao) {
    if (connection == null) {
        // connection = // create connection or take connection from pool
    }
    dao.setConnection(connection);
}
public void end() {
    // code -> check of null connection
    // code -> return connection to pool or closing
    connection = null;
}
```

Схематически применение этого подхода можно представить в методе **doService()** класса **SomeService**.

```
/* # 12 # использование DAO на уровне логики # SomeService.java */
```

```
package by.epam.learn.service;
// imports
public class SomeService {
    public void doService(parameters_0) throws ServiceException {
// 1. init DAO
        AbonentDao abonentDao = new AbonentDao();
        OrderDao orderDao = new OrderDao();
// 2. transaction initialization for DAO objects
        EntityTransaction transaction = new EntityTransaction();
        transaction.initTransaction(abonentDao, orderDao);
// 3. query execution
        try {
            abonentDao.create(parameters_1);
            orderDao.update(parameters_2);
            orderDao.delete(parameters_3);
            transaction.commit();
        } catch (DaoException e) {
            transaction.rollback();
            throw new ServiceException(e);
        } finally {
// 4. transaction closing
            transaction.endTransaction();
        }
    }
}
```

Конфигурация стандартного пула соединений с БД достаточно проста. Для использования такого источника соединений параметры конфигурации следует поместить в **context.xml**.

```

/* # 13 # стандартный пул соединений # StandartConnectionPool.java */

package by.epam.learn.pool;
import java.sql.Connection;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
public class StandartConnectionPool {
    private static final String DATASOURCE_NAME = "jdbc/testphones";
    private static DataSource dataSource;
    static {
        try {
            Context initContext = new InitialContext();
            Context envContext = (Context) initContext.lookup("java:/comp/env");
            dataSource = (DataSource) envContext.lookup(DATASOURCE_NAME);
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
    private StandartConnectionPool() {
    }
    public static Connection getConnection() throws SQLException {
        Connection connection = dataSource.getConnection();
        return connection;
    }
    // method to return Connection to the pool
}

```

Драйвер для СУБД MySQL предоставляет собственное решение по созданию источника соединений **MysqlDataSource**. Соединение с БД из этого объекта извлекается методом **Connection getConnection()**.

Фабрика по созданию объекта **MysqlDataSource**.

```

/* # 14 # инициализация dataSource для СУБД MySQL # MySqlFactory.java */

package by.epam.learn.pool;
import com.mysql.cj.jdbc.MysqlDataSource;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.IOException;
import java.util.Properties;
public class MySqlFactory {
    public static MysqlDataSource createMysqlDataSource() {
        MysqlDataSource dataSource = null;
        Properties props = new Properties();
        try {
            InputStream inputStream = new FileInputStream("datares\\database.properties");

```

```

        props.load(inputStream);
        dataSource = new MysqlDataSource();
        dataSource.setURL(props.getProperty("db.url"));
        dataSource.setUser(props.getProperty("user"));
        dataSource.setPassword(props.getProperty("password"));
    } catch (IOException e) {
        e.printStackTrace();
    }
    return dataSource;
}
}

```

Вопросы к главе 13

1. Что такое JDBC? Перечислить основные классы и интерфейсы, входящие в состав JDBC, указать их назначение. Какие еще существуют технологии Java, работающие с БД?
2. Привести алгоритм получения соединения с базой данных, выполнения запроса и обработки результатов. Как загрузить драйвер базы данных и что он собой представляет. Какие существуют типы драйверов баз данных в JDBC? Нужно ли регистрировать драйвер БД? Если да, то как это сделать?
3. Как правильно закрыть **Connection**?
4. Какие есть типы драйверов для соединения с СУБД?
5. Чем отличается **Statement** от **PreparedStatement**? Где сохраняется запрос после первого вызова **PreparedStatement**? Будет ли тот же самый эффект, как и от **PreparedStatement**, если формировать запрос просто в строке и отправлять его в **Statement**?
6. Защищены ли **Statement** и **PreparedStatement** от *sql-injection*? Можно ли работать с несколькими объектами *statement* или *prepared statement*, полученными от одного объекта *connection* одновременно и может ли такое использование быть небезопасным?
7. Зачем нужен **CallableStatement**? Как выполняется вызов хранимых процедур из Java-программы? Что называется *batch*-командой, как выполнить *batch*-команду?
8. Чем отличаются метод **executeUpdate()** от **executeQuery()**?
9. Для чего JDBC использует объекты типа **ResultSet**?
10. Что означает прокручиваемый и непрокручиваемый, обновляемый и необновляемый **ResultSet**?
11. Как можно получить такие различные типы объектов **ResultSet**? Можно ли через объект **ResultSet** изменить значения в БД и, если да, то каким образом?
12. Как в объекте **ResultSet** вернуться в предыдущую строку? Всегда ли можно вернуться в предыдущую строку?
13. Как получить сгенерированный СУБД первичный ключ без выполнения дополнительного запроса к БД?

14. Как узнать, в какие типы Java будут конвертированы при выборке *sql*-типы данных?
15. Привести определение транзакции, *commit* и *rollback*. Как JDBC работает с транзакциями по умолчанию? Как отменить *autocommit*, как в этом случае следует работать с БД?
16. Что такое точка сохранения и как ее создать? Как откатить транзакцию до точки сохранения или до предыдущего *commit*?
17. Что такое пул соединений с БД, для чего он необходим? Каковы основные принципы создания пула соединений к БД?
18. Что означает термин «метаданные»? Какую информацию предоставляют объекты классов **DatabaseMetaData**, **ResultSetMetaData** и для чего она может быть использована?
19. Что означает термин *уровень изоляции транзакции*? Какие уровни изоляции транзакций поддерживает JDBC? Как задать уровень изоляции транзакций?

Задания к главе 13

Вариант А

В каждом из заданий необходимо выполнить следующие действия:

- организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение;
 - создать БД. Привести таблицы к одной из нормальных форм;
 - создать класс для выполнения запросов на извлечение информации из БД с использованием компилированных запросов;
 - создать класс на модификацию информации.
1. **Файловая система.** В БД хранится информация о дереве каталогов файловой системы — каталоги, подкаталоги, файлы.
Для каталогов необходимо хранить:
 - родительский каталог;
 - название.
 Для файлов необходимо хранить:
 - родительский каталог;
 - название;
 - место, занимаемое на диске.
 - Определить полный путь заданного файла (каталога).
 - Подсчитать количество файлов в заданном каталоге, включая вложенные файлы и каталоги.
 - Подсчитать место, занимаемое на диске содержимым заданного каталога.
 - Найти в базе файлы по заданной маске с выдачей полного пути.
 - Переместить файлы и подкаталоги из одного каталога в другой.
 - Удалить файлы и каталоги заданного каталога.

2. **Видеотека.** В БД хранится информация о домашней видеотеке: фильмы, актеры, режиссеры.

Для фильмов необходимо хранить:

- название;
- имена актеров;
- дату выхода;
- страну, в которой выпущен фильм.
- Для актеров и режиссеров необходимо хранить:
 - ФИО;
 - дату рождения.
- Найти все фильмы, вышедшие на экран в текущем и прошлом году.
- Вывести информацию об актерах, снимавшихся в заданном фильме.
- Вывести информацию об актерах, снимавшихся как минимум в N фильмах.
- Вывести информацию об актерах, которые были режиссерами хотя бы одного из фильмов.
- Удалить все фильмы, дата выхода которых была более заданного числа лет назад.

3. **Расписание занятий.** В БД хранится информация о преподавателях и проводимых ими занятиях.

Для предметов необходимо хранить:

- название;
- время проведения (день недели);
- аудитории, в которых проводятся занятия.

Для преподавателей необходимо хранить:

- ФИО;
- предметы, которые он ведет;
- количество пар в неделю по каждому предмету;
- количество студентов, занимающихся на каждой паре.
- Вывести информацию о преподавателях, работающих в заданный день недели в заданной аудитории.
- Вывести информацию о преподавателях, которые не ведут занятия в заданный день недели.
- Вывести дни недели, в которых проводится заданное количество занятий.
- Вывести дни недели, в которых занято заданное количество аудиторий.
- Перенести первые занятия заданных дней недели на последнее место.

4. **Письма.** В БД хранится информация о письмах и отправляющих их людях.

Для людей необходимо хранить:

- ФИО;
- дату рождения.

Для писем необходимо хранить:

- отправителя;
- получателя;

- тему письма;
 - текст письма;
 - дату отправки.
 - Найти пользователя, длина писем которого наименьшая.
 - Вывести информацию о пользователях, а также количестве полученных и отправленных ими письмах.
 - Вывести информацию о пользователях, которые получили хотя бы одно сообщение с заданной темой.
 - Вывести информацию о пользователях, которые не получали сообщения с заданной темой.
 - Направить письмо заданного человека с заданной темой всем адресатам.
5. **Сувениры.** В БД хранится информация о сувенирах и их производителях. Для сувениров необходимо хранить:
- название;
 - реквизиты производителя;
 - дату выпуска;
 - цену.
- Для производителей необходимо хранить:
- название;
 - страну.
- Вывести информацию о сувенирах заданного производителя.
 - Вывести информацию о сувенирах, произведенных в заданной стране.
 - Вывести информацию о производителях, чьи цены на сувениры меньше заданной.
 - Вывести информацию о производителях заданного сувенира, произведенного в заданном году.
 - Удалить заданного производителя и его сувениры.
6. **Заказ.** В БД хранится информация о заказах магазина и товарах в них. Для заказа необходимо хранить:
- номер заказа;
 - товары в заказе;
 - дату поступления.
- Для товаров в заказе необходимо хранить:
- товар;
 - количество.
- Для товара необходимо хранить:
- название;
 - описание;
 - цену.
- Вывести полную информацию о заданном заказе.
 - Вывести номера заказов, сумма которых не превосходит заданную, и количество различных товаров равно заданному.

- Вывести номера заказов, содержащих заданный товар.
- Вывести номера заказов, не содержащих заданный товар и поступивших в течение текущего дня.
- Сформировать новый заказ, состоящий из товаров, заказанных в текущий день.
- Удалить все заказы, в которых присутствует заданное количество заданного товара.

7. **Продукция.** В БД хранится информация о продукции компании.

Для продукции необходимо хранить:

- название;
- группу продукции (телефоны, телевизоры и др.);
- описание;
- дату выпуска;
- значения параметров.

Для групп продукции необходимо хранить:

- название;
- перечень групп параметров (размеры и др.).

Для групп параметров необходимо хранить:

- название;
- перечень параметров.

Для параметров необходимо хранить:

- название;
- единицу измерения.

- Вывести перечень параметров для заданной группы продукции.
- Вывести перечень продукции, не содержащий заданного параметра.
- Вывести информацию о продукции для заданной группы.
- Вывести информацию о продукции и всех ее параметрах со значениями.
- Удалить из базы продукцию, содержащую заданные параметры.
- Переместить группу параметров из одной группы товаров в другую.

8. **Погода.** В БД хранится информация о погоде в различных регионах.

Для погоды необходимо хранить:

- регион;
- дату;
- температуру;
- осадки.

Для регионов необходимо хранить:

- название;
- площадь;
- тип жителей.

Для типов жителей необходимо хранить:

- название;
- язык общения.

- Вывести сведения о погоде в заданном регионе.

- Вывести даты, когда в заданном регионе шел снег и температура была ниже заданной отрицательной.
 - Вывести информацию о погоде за прошедшую неделю в регионах, жители которых общаются на заданном языке.
 - Вывести среднюю температуру за прошедшую неделю в регионах с площадью больше заданной.
9. **Магазин часов.** В БД хранится информация о часах, продающихся в магазине.

Для часов необходимо хранить:

- марку;
- тип (кварцевые, механические);
- стоимость;
- количество;
- реквизиты производителя.

Для производителей необходимо хранить:

- название;
 - страна.
- Вывести марки заданного типа часов.
 - Вывести информацию о механических часах, стоимость которых не превышает заданную.
 - Вывести марки часов, изготовленных в заданной стране.
 - Вывести производителей, общая сумма часов которых в магазине не превышает заданную.

10. **Города.** В БД хранится информация о городах и их жителях.

Для городов необходимо хранить:

- название;
- год основания;
- площадь;
- количество населения для каждого типа жителей.

Для типов жителей необходимо хранить:

- город проживания;
 - название;
 - язык общения.
- Вывести информацию обо всех жителях заданного города, разговаривающих на заданном языке.
 - Вывести информацию обо всех городах, в которых проживают жители выбранного типа.
 - Вывести информацию о городе с заданным количеством населения и всех типах жителей, в нем проживающих.
 - Вывести информацию о самом древнем типе жителей.

11. **Планеты.** В БД хранится информация о планетах, их спутниках и галактиках.

Для планет необходимо хранить:

- название;
- радиус;
- температуру ядра;
- наличие атмосферы;
- наличие жизни;
- спутники.

Для спутников необходимо хранить:

- название;
- радиус;
- расстояние до планеты.

Для галактик необходимо хранить:

- название;
- планеты.

- Вывести информацию обо всех планетах, на которых присутствует жизнь, и их спутниках в заданной галактике.
 - Вывести информацию о планетах и их спутниках, имеющих наименьший радиус и наибольшее количество спутников.
 - Вывести информацию о планете, галактике, в которой она находится, и ее спутниках, имеющей максимальное количество спутников, но с наименьшим общим объемом этих спутников.
 - Найти галактику, сумма ядерных температур планет которой наибольшая.
12. **Точки.** В БД хранится некоторое конечное множество точек с их координатами.
- Вывести точку из множества, наиболее приближенную к заданной.
 - Вывести точку из множества, наиболее удаленную от заданной.
 - Вывести точки из множества, лежащие на одной прямой с заданной прямой.
13. **Треугольники.** В БД хранятся треугольники и координаты их точек на плоскости.
- Вывести треугольник, площадь которого наиболее приближена к заданной.
 - Вывести треугольники, сумма площадей которых наиболее приближена к заданной.
 - Вывести треугольники, которые помещаются в окружность заданного радиуса.
 - Вывести все равнобедренные треугольники.
 - Вывести все равносторонние треугольники.
 - Вывести все прямоугольные треугольники.
 - Вывести все тупоугольные треугольники с площадью больше заданной.
14. **Словарь.** В БД хранится англо-белорусский словарь, в котором для одного английского слова может быть указано несколько его значений и наоборот. Со стороны клиента вводятся последовательно английские (белорусские)

слова. Для каждого из них вывести на консоль все белорусские (английские) значения слова.

15. **Словари.** В двух различных базах данных хранятся два словаря: польско-белорусский и белорусско-польский. Клиент вводит слово и выбирает язык. Вывести перевод этого слова.
16. **Стихотворения.** В БД хранятся несколько стихотворений с указанием автора и года создания. Для хранения стихотворений использовать объекты типа `Clob`. Клиент выбирает автора и критерий поиска.
 - В каком из стихотворений больше всего восклицательных предложений?
 - В каком из стихотворений меньше всего повествовательных предложений?
 - Есть ли среди стихотворений сонеты и сколько их?
17. **Четырехугольники.** В БД хранятся координаты вершин выпуклых четырехугольников на плоскости.
 - Вывести координаты вершин параллелограммов.
 - Вывести координаты вершин трапеций.

СЕТЕВЫЕ ПРОГРАММЫ

*Если неправильно набрать номер,
никогда не будет гудков «занято».*

Загадка Ковака

Компьютерные сети

Компьютерная сеть — это совокупность компьютеров и устройств, соединенных с помощью каналов связи в единую систему так, что они могут связываться между собой и совместно использовать ресурсы сети. Самой простой является классификация компьютерных сетей локальных (LAN) и глобальных (WAN).

LAN (*Local Area Network*) или ЛВС (*локальные вычислительные сети*), представляют объединение компьютеров, расположенных на небольших расстояниях и имеющих замкнутую инфраструктуру до выхода в глобальную сеть.

WAN (*Wide Area Network*) — глобальные компьютерные сети, которые часто рассматривают как сети сетей, могут объединять сети на уровне континентов и государств. К глобальным сетям относится сеть интернет.

Адресация в глобальных сетях

Локальные компьютеры, объединенные в ЛС, входят в глобальные сети и интернет. Обмен информацией происходит в форме пакета данных, который передается по сети. IP-адрес компьютера-отправителя и IP-адрес компьютера-получателя указывается в заголовке пакета.

При выходе в интернет каждому компьютеру назначается внешний статический или динамический IP-адрес, например, 198.135.78.13. Функция автоматического назначения IP-адреса и прокси-сервер гарантирует уникальность IP-адреса. IP-адреса присваивается не только компьютеру, но и другим сетевым устройствам, например, маршрутизатору. Маршрутизаторы используются для соединения нескольких локальных сетей или для связи локальных сетей с глобальными, они имеют несколько IP-адресов с номерами объединяемых сетей и могут быть оснащены несколькими сетевыми адаптерами.

Адрес в сетях IPv4 состоит из адресов подсетей и номера узла. Подсети — это отдельные, самостоятельно функционирующие части сети, имеющие свой идентификатор.

Левый байт (октет) содержит номер 1..255 и указывает класс локальной интрасети, в которой находится искомый компьютер. Последний (правый) идентификатор IP-адреса обозначает номер хоста в данной локальной сети. Между правым и левым октетами в такой записи расположены номера подсетей более низкого уровня.

Значение 127 первого октета зарезервировано для служебных целей, поскольку IP-пакеты, направленные по такому адресу, не передаются в сеть, а ретранслируются обратно на этот же компьютер, как только что принятые (кольцевой хост 127.0.0.1).

При разработке протокола IPv4 не предусматривалось столь широкое его распространение, поэтому адресов перестало хватать. Для решения этой и сопутствующих проблем был разработан протокол IPv6. IPv6-адрес записывается восемь блоков по 16 бит (всего 128 бит), например, 2001:0 da8: 65b4: 05d3: 1315:7 C1F: 0461:7847. Преимуществом IPv6, помимо огромного количества IP-адресов, является возможность включать автоматическую настройку IP-адреса устройства, используя его MAC-адрес. Другие преимущества: упрощение переключения корпоративной сети между провайдерами, быстрый маршрут, шифрование.

Адреса IPv6 отображаются группами по четыре шестнадцатеричные цифры, разделенными двоеточием. Пример адреса:

2001:0db8:11a3:09d7:1f34:8a2e:07a0:765d

Если одна или более групп подряд равны 0000, они могут быть опущены и заменены на двойное двоеточие (::). Например, адрес многоадресной рассылки FF02:0000:0000:0000:0000:0000: 0002 после применения данного правила примет вид FF02::2. Еще один способ записи заключается в удалении начальных нулей в каждом 16-битном блоке, например:

1DA:D3:0:2F3B:2AA:FF:FE28:9C5A.

IPv6-адрес делится на три части: глобальный префикс (Global Routing Prefix) — присваивается провайдерам и определяется тремя первыми блоками; идентификатор подсети (Subnet ID) — представлен четвертым блоком; идентификатор интерфейса (Interface ID) — определяет уникальный адрес хоста в сети. При использовании IPv6-адреса в URL необходимо заключать адрес в квадратные скобки. Порт пишется после скобок:

http://[2001:0db8:11a3:09d7:1f34:8a2e:07a0:765d]:8080/

Доменные имена

Поскольку IP-адресация неудобна для запоминания, в интернете широко используются символьные адреса (доменные имена). Для поиска сервера в интернет проще указать его имя, например, epam.com, чем IP-адрес. DNS-имя назначается администратором сети и состоит из имени компьютера, имени ор-

ганизации, имени домена. Идентификатор не может состоять более чем из 4 слов, причем длина каждого из слов меньше 65 символов, а вся запись — не более 255 символов. К примеру, в careers.epam.by careers — имя узла (поддомен), остальные являются доменами вышестоящего уровня.

Пространство доменных имен имеет иерархическую структуру. Корневой домен располагается на самом вершине иерархии и обозначается точкой. Домены верхнего уровня объединяют компьютеры сети по географическому признаку или роду деятельности. Например, by, pl, ua определяют географическое положение (Беларусь, Польша, Украина). Ряд доменов, относящихся к США, считаются международными (com, edu, gov, org, net).

Домены второго уровня обычно относятся к названиям компаний и регистрируются владельцами доменов верхнего уровня. Домены третьего уровня обычно к подразделениям внутри компаний. Каждый компьютер, по-другому узел или хост, в интернете однозначно определяется своим полным доменным именем.

Для совместимости числового IP-адреса и буквенного имени есть служба DNS. При установке операционной системы протокол TCP/IP настраивается на сервер имен того домена, в который входит данный компьютер. Когда программе-клиенту требуется по доменному имени выяснить IP-адрес, она через протокол TCP/IP связывается с сервером DNS, передавая ему свой запрос. Сервер имен ищет домен в базе данных, находит IP-адрес и возвращает клиенту. Если запрашиваемое доменное имя не входит в его базу, он переадресует запрос вышестоящему серверу.

В случае проблем с DNS можно просто прописать в настройках в качестве первичного DNS-сервера адрес другого DNS-сервера, например, свободные серверы Google. Эти серверы имеют IP-адреса: 8.8.8.8 и 8.8.4.4. Можно их прописать в качестве первичного и вторичного серверов имен и использовать, если нет возражений своего провайдера.

URL адреса

Для доступа к ресурсу интернет указывается не просто адрес ресурса, а его URL (Uniform Resource Locator — унифицированный определитель ресурсов). Схему URL можно представить следующим образом:

```
<схема>://<логин>:<пароль>@<хост>:<порт>/<URL-путь>?<парам>
#<якорь>
https://login:pass@logging.apache.org/log4j/2.x/manual/appenders.html
#AsyncAppender
```

После знака вопроса в URL могут быть GET-параметры, а также так называемый якорь, который добавляется в конце после символа решетки «#». Якоря заранее проставляются внутри html-кода веб-страницы, а затем, добавив название этого якоря к URL-адресу страницы через символ решетки «#», осуществляется переход сразу к месту, где был проставлен якорь.

Необходимо сказать о различных кодировках, которые применяются в URL. Без перекодирования в URL можно использовать только ограниченное количество символов: [0-9],[a-z],[A-Z],[_],[-]. Употребление других символов (включая кириллицу и пробелы) в URL-адресах допустимо, но будет происходить перекодировка этих символов (URL Encoding). Каждый символ кириллицы кодируется с помощью двух байт в Unicode, записанных в шестнадцатеричном коде и разделенных знаком «%». Например:

http://your.by/%BE%D0%B3%D0%D1%82%D0%BE%20%D0

Во избежание такой кодировки можно порекомендовать использовать в URL только строчные латинские символы.

URL не может содержать такие специальные символы, как пробелы, табуляции, возврат каретки. Их можно задавать через шестнадцатеричные коды. Например, %20 обозначает пробел, также пробел может обозначаться знаком +. Другие зарезервированные символы: символ & — разделитель аргументов или параметров запроса, символ ? следует перед аргументами запроса, символ # — ссылки внутри страницы.

Кроме адреса хоста, при передаче пакета данных указывается порт. Порт имеет длину 2 байта, записывается через «:». Например, *epam.com:80*.

Комбинация IP-адреса и номера порта позволяет однозначно идентифицировать программу в сети. Такой комбинированный адрес определяет сокет (socket — разъем, соединение) — программный интерфейс для обеспечения обмена данными между распределенными приложениями типа клиент и сервер. Клиент подсоединяется к серверу, после чего чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером.

Сетевые программы

Язык Java делает сетевое программирование более простым благодаря наличию специальных средств и классов. Некоторые из этих классов находятся в пакете **java.net**. Сетевые классы имеют методы для установки сетевых соединений передачи запросов и сообщений. Многопоточность позволяет обрабатывать несколько соединений. Сетевые приложения используют интернет-приложения, к которым относятся веб-браузер, e-mail, сокеты, почта и др.

Приложения клиент/сервер используют компьютер, выполняющий специальную программу-сервер, которая обычно устанавливается на удаленном компьютере и предоставляет услуги другим программам-клиентам. Клиент — это программа, получающая услуги от сервера. Клиент устанавливает соединение с сервером и пересылает серверу запрос. Сервер осуществляет или не осуществляет прослушивание клиентов, получает и выполняет запрос после установки соединения.

Результат выполнения запроса может быть возвращен сервером клиенту. Запросы и сообщения представляют собой записи, структура которых определяется используемыми протоколами.

Определить IP-адрес в приложении можно с помощью объекта класса **java.net.InetAddress**. Можно также использовать и специфические классы **Inet4Address** и **Inet6Address**.

Класс **InetAddress** не имеет **public**-конструкторов. Создать объект класса можно с помощью статических методов. Метод **getLocalHost()** возвращает объект класса **InetAddress**, содержащий IP-адрес и имя компьютера, на котором выполняется программа. Метод **getByName(String host)** возвращает объект класса **InetAddress**, содержащий IP-адрес по имени компьютера, используя пространство имен DNS. IP-адрес может быть временным, различным для каждого соединения, однако он остается постоянным, если соединение установлено. Метод **getByAddress(byte[] addr)** создает объект класса **InetAddress**, содержащий имя компьютера, по IP-адресу, представленному в виде массива байт. Если компьютер имеет несколько IP, то получить их можно методом **getAllByName(String host)**, возвращающим массив объектов класса **InetAddress**. Если IP для данной машины один, то массив будет содержать один элемент. Метод **getByAddress(String host, byte[] addr)** создает объект класса **InetAddress** с заданным именем и IP-адресом, не проверяя существование такого компьютера. Все эти методы являются потенциальными генераторами исключительной ситуации **UnknownHostException**, и поэтому их вызов должен быть обработан с помощью **throws** для метода или блока **try-catch**. Проверить доступ к компьютеру в данный момент можно с помощью метода **boolean isReachable(int timeout)**, который возвращает **true**, если компьютер доступен, где **timeout** — время ожидания ответа от компьютера в миллисекундах.

Следующая программа демонстрирует, как получить IP-адрес текущего компьютера и IP-адрес из имени домена с помощью сервера имен доменов (DNS), к которому обращается метод **getByName()** класса **InetAddress**.

```
/* # 1 # вывод IP-адреса компьютера и интернет-ресурса # InetAddressMain.java */
```

```
package by.epam.learn.net;
import java.net.InetAddress;
import java.net.UnknownHostException;
public class InetAddressMain {
    public static void main(String[] args {
        InetAddress currentIp;
        InetAddress epamIp;
        try {
            currentIp = InetAddress.getLocalHost();
            // output IP address of local computer
            System.out.println("current IP -> " + currentIp.getHostAddress());
```

```

        epamIp = InetAddress.getByName("epam.com");
        System.out.println("EPAM IP address -> " + epamIp.getHostAddress());
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
}
}

```

В результате будет выведено, например:

current IP -> 26.227.49.230EPAM -> 3.214.134.159

Метод **getLocalHost()** класса **InetAddress** создает объект **currentIp** и возвращает IP-адрес компьютера.

```

/* # 2 # присваивание фиктивного имени реальному ресурсу, заданному через IP
# UnCheckedHost.java */

```

```

package by.epam.learn.net;
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
public class UnCheckedHostMain {
    public static void main(String[] args) {
        // setting an IP address as an array
        byte ip[] = {(byte) 217, (byte) 21, (byte) 43, (byte) 3};
        try {
            InetAddress addr = InetAddress.getByAddress("bsu.by", ip);
            System.out.println(addr.getHostAddress() + "-> connection: " + addr.isReachable(1_000));
        } catch (UnknownHostException e) {
            System.err.println("address unavailable: " + e);
        } catch (IOException e) {
            System.err.println("I/O exception: " + e);
        }
    }
}

```

В результате будет выведено в случае подключения к сети интернет:

bsu.by-> connection: true

Для доступа к ресурсам можно создать объект класса **URL**, указывающий на ресурсы в интернете. В следующем примере объект **URL** используется для доступа к HTML-файлу по указанному адресу и вывода его в окно консоли.

```
/* # 3 # чтение документа из интернета # URLInfoDocumentMain.java */
```

```
package by.epam.learn.net;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;

public class URLInfoDocumentMain {
    public static void main(String[] args) {
        String urlName = "https://logging.apache.org/log4j/2.x/download.html";
        try(BufferedReader reader = new BufferedReader(
            new InputStreamReader(new URL(urlName).openStream())) {
            URL url = new URL(urlName);
            System.out.println("protocol: " + url.getProtocol());
            System.out.println("host: " + url.getHost());
            System.out.println("port: " + url.getDefaultPort());
            System.out.println("file: " + url.getFile());
            reader.lines().forEach(System.out::println); // reading content
        } catch (MalformedURLException e) {
            e.printStackTrace(); // incorrectly set protocol, domain name or file path
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Результат работы приложения:

protocol: https

host: logging.apache.orgport: 443

file: /log4j/2.x/download.html

ниже будет выведен html-код страницы.

Для получения более полной информации о ресурсе применяются класс **URLConnection**. Для получения экземпляра следует вызвать на экземпляре класса **URL** метод **openConnection()**. Далее при необходимости можно указать значения некоторых свойств. Для установления соединения на полученном экземпляре **URLConnection** вызывается метод **connect()**. При вызове этого метода возможна серьезная блокировка основного потока, поэтому перед попыткой установления соединения следует установить таймаут на соединение и/или на чтение методами **setConnectTimeout(int timeout)** и **setReadTimeout(int timeout)**.

```
/* # 4 # информация об интернет-ресурсе # UrlMain.java */
```

```
package by.epam.learn.net;
import java.io.IOException;
import java.net.URL;
import java.net.URLConnection;
public class UrlMain {
    public static void main(String[] args) {
        String urlName = "http://www.google.com";
        int timeout = 10_000_000;
        try {
            URL url = new URL(urlName);
            URLConnection connection = url.openConnection(); connection.setConnectTimeout(timeout);//set
            timeout for connectionconnection.connect();
            System.out.println(urlName + "::content type:"+ connection.getContentType());
        } catch (IOException e) {e.printStackTrace();
        }
    }
}
```

В результате при успешной установке соединения будет выведено:
http://www.google.com :: content type: text/html; charset=ISO-8859-1

Сокетные соединения по TCP/IP

Сокеты, или сетевые разъемы, — это логическое понятие, соответствующее разъемам, к которым подключены сетевые компьютеры, и через которые осуществляется двунаправленная поточная передача данных между компьютерами. Сокет определяется номером порта и IP-адресом. При этом IP-адрес используется для идентификации компьютера, номер порта — для идентификации процесса, работающего на компьютере. Когда одно приложение знает сокет другого, можно создать сокетное протоколо-ориентированное соединение по протоколу TCP/IP. Клиент пытается соединиться с сервером, инициализируя сокетное соединение. Сервер прослушивает сообщение и ждет, пока клиент не свяжется с ним. Первое сообщение, посылаемое клиентом на сервер, содержит сокет клиента. Сервер, в свою очередь, создает сокет, который будет использоваться для связи с клиентом, и посылает его клиенту с первым сообщением. В результате устанавливается коммуникационное соединение.

Сокетное соединение с сервером создается клиентом с помощью объекта класса **java.net.Socket**. При этом указывается IP-адрес сервера и номер порта. Если указано символьное имя домена, то Java преобразует его с помощью DNS-сервера к IP-адресу. Например, если сервер установлен на этом же компьютере, соединение с сервером можно установить из приложения клиента с помощью инструкции:

```
Socket clientSocket = new Socket("Server_Name", 8030);
```

Сервер ожидает сообщения клиента и должен быть заранее запущен с указанием определенного порта. Объект класса **java.net.ServerSocket** создается с указанием конструктору номера порта и ожидает сообщения клиента с помощью метода **accept()** класса **ServerSocket**, который возвращает сокет клиента:

```
ServerSocket serverSocket = new ServerSocket(8030); Socket socket = serverSocket.accept();
```

Таким образом, для установки необходимо установить IP-адрес и номер порта сервера, IP-адрес и номер порта клиента. Обычно порт клиента и сервера устанавливаются одинаковыми. Клиент и сервер после установления сокетного соединения могут получать данные из потока ввода и записывать данные в поток вывода с помощью объектов **InputStream** и **OutputStream**, полученных из сокета вызовом методов **getInputStream()** и **getOutputStream()**.

В следующем примере для отправки клиенту строки «*java tutorial*» сервер вызывает метод **getOutputStream()** класса **Socket**. Клиент получает данные от сервера с помощью метода **getInputStream()**. Для разъединения клиента и сервера после завершения работы сокет должен быть закрыт. В данном примере сервер отправляет клиенту одну строку, после чего разрывает связь.

```
/* # 5 # передача клиенту строки # SimpleServerSocket.java */
```

```
package by.epam.learn.net;

import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class SimpleServerSocket {

    public static void main(String[] args) {
        System.out.println("server starts");
        try (ServerSocket serverSocket = new ServerSocket(2048);
            Socket socket = serverSocket.accept();
            PrintWriter writer = new PrintWriter(
                new OutputStreamWriter(socket.getOutputStream()))) {
            writer.println("java tutorial");// put string "java tutorial" into the buffer
            writer.flush();// send the contents of the buffer to the client
        } catch (IOException e) {
            System.err.println("IO error connection: " + e);
        }
        System.out.println("server is closed");
    }
}
```

```
/* # 6 # получение клиентом строки # SmallClientSocket.java */
```

```
package by.epam.learn.net;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.InetAddress;
import java.net.Socket;
public class SimpleClientSocket {
    public static void main(String[] args) {
        try (Socket socket = new Socket(InetAddress.getLocalHost(), 2048);
            BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()))){
            String message = reader.readLine();
            System.out.println("message received: " + message);
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("client is closed");
    }
}
```

Если связь с сервером была успешно установлена, то в консоль клиенту будет выведено:

message received: java tutorialclient is closed

Аналогично клиент может послать данные серверу через поток вывода, полученный с помощью метода **getOutputStream()**, а сервер может получать данные через поток ввода, полученный с помощью метода **getInputStream()**.

Если необходимо протестировать данный пример на компьютерах в сети, достаточно заменить вызов статического метода **getLocalHost()** класса **InetAddress** при инициализации клиентского сокета на IP-адрес компьютера сервера в виде строки или объекта класса **InetAddress**.

Многопоточность

Сервер должен поддерживать многопоточность, иначе он будет не в состоянии обрабатывать несколько соединений одновременно. В этом случае сервер содержит цикл, ожидающий нового клиентского соединения. Каждый раз, когда клиент просит соединения, сервер создает новый поток. В следующем примере создается класс **ServerThread**, расширяющий класс **Thread**, и используется затем для соединений с многими клиентами, каждый в своем потоке.

```
/* # 7 # сервер для множества клиентов # NetServerThreadMain.java */
```

```
package by.epam.learn.net;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
public class NetServerThreadMain {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8071);
            System.out.println(serverSocket.getInetAddress() + " server started");
            while (true) {
                Socket socket = serverSocket.accept();// waiting for a new client
                System.out.println(socket.getInetAddress().getHostName() + " connected");
                // create a separate stream for data exchange with the connected client
                ServerThread thread = new ServerThread(socket);
                thread.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ServerThread extends Thread {
    private PrintStream printStream; // send
    private BufferedReader reader; // receive

    private InetAddress address; // client address
    public ServerThread(Socket socket) {
        try {
            printStream = new PrintStream(socket.getOutputStream());
            reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        } catch (IOException e) {
            e.printStackTrace();
        }
        address = socket.getInetAddress();
    }

    public void run() {
        int counter = 0;
        String message;
        try {
            while ((message = reader.readLine()) != null) {

                if ("PING".equals(message)) {
                    printStream.println("PONG #" + ++counter);
                }
            }
        }
    }
}
```

```

        System.out.println("PING-PONG #" + counter + " from " + address.getHostName());
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    disconnect();
}
}
private void disconnect() {
    if (printStream != null) {
        printStream.close();
    }
    try {
        if (reader != null) {
            reader.close();
        }
        System.out.println(address.getHostName() + ": disconnected");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

Сервер передает сообщение клиенту. Для клиентских приложений поддержка многопоточности также необходима. Например, один поток ожидает выполнения операции ввода/вывода, а другие потоки выполняют свои функции.

```

/* # 8 # получение и передача сообщения клиентом в потоке # NetClient.java */

```

```

package by.epam.learn.net;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
public class NetClientMain {
    final static int TIMEOUT_IN_MILLIS = 1_000;
    final static int MAX_PING = 10;
    final static String ORIGINAL_MESSAGE = "PING";
    public static void main(String[] args) {
        try (Socket socket = new Socket(InetAddress.getLocalHost(), 8071);
            PrintStream printStream = new PrintStream(socket.getOutputStream());
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(socket.getInputStream()))) {
            for (int i = 0; i < MAX_PING; i++) {
                printStream.println(ORIGINAL_MESSAGE);
                System.out.println(reader.readLine());
                Thread.sleep(TIMEOUT_IN_MILLIS);
            }
        }
    }
}

```

```

    }
} catch (UnknownHostException e) {
    System.err.println("Connection refused:" + e); // not connect to the server
} catch (IOException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}
}

```

Сервер должен быть инициализирован до того, как клиент попытается осуществить сокетное соединение. При этом может быть использован IP-адрес локального компьютера.

Если с сервером соединился только один клиент, то в консоль сервера будет выведено:

```

0.0.0.0/0.0.0.0 server started NW1254.minsk.epam.com connected
PING-PONG #1 from NW1254.minsk.epam.comPING-PONG #2 from
NW1254.minsk.epam.com
...
PING-PONG #10 from NW1254.minsk.epam.comNW1254.minsk.epam.com:
disconnected

```

где **NW1254.minsk.epam.com** — адрес компьютера клиента в сети.

В итоге в консоль клиента будет выведена информация вида:

```

NW1254.minsk.epam.com connected
PING-PONG #1 from NW1254.minsk.epam.comPING-PONG #2 from
NW1254.minsk.epam.com
...
PING-PONG #10 from NW1254.minsk.epam.com
EPBYMINW1254.minsk.epam.com: disconnected

```

В местах вывода, где стоят многоточия, находятся строки с номерами от 3 до 9 включительно.

Датаграммы и протокол UDP

Протокол UDP (*User Datagram Protocol*) не устанавливает виртуального соединения и не гарантирует доставки данных. Отправитель просто посылает пакеты по указанному адресу, и если отосланная информация была повреждена В процессе пересылки или вообще не дошла, отправитель об этом даже не узнает. Однако достоинством UDP является высокая скорость передачи данных.

Данный протокол часто используется при трансляции аудио- и видеосигналов, где потеря небольшого количества данных не может привести к серьезным искажениям всей информации.

По протоколу UDP данные передаются пакетами. Пакетом UDP в этом случае является объект класса **DatagramPacket**. Объект этого класса содержит в себе передаваемые данные, представленные в виде массива байт. Конструктор класса **DatagramPacket(byte[] buf, int length)** используется в тех случаях, когда датаграмма только принимает в себя пришедшие данные, так как созданный с его помощью объект не имеет информации об адресе и порте получателя. Конструктор **DatagramPacket(byte[] buf, int length, InetAddress address, int port)** используется для подготовки отправки датаграмм. Класс **DatagramSocket** может выступать в роли клиента и сервера, т.е. он способен получать и отправлять пакеты. Отправить пакет можно с помощью метода **send(DatagramPacket pac)**, для получения пакета используется метод **receive(DatagramPacket pac)**.

Первым следует запускать приложение, которое принимает информацию, так как оно будет находиться некоторое время в режиме ожидания приема данных.

```

/* # 9 # прием файла по протоколу UDP # UDPRecipientMain.java */

package by.epam.learn.net;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketTimeoutException;
public class UDPRecipientMain {
    final static int WAIT_TIMEOUT = 60_000;
    public static void main(String[] args) {
        File file = new File("C:/tmp/Dvorak_SlavonicDance2.mp3");
        System.out.println("receiving data ...");
        acceptFile(file, 8033, 1024);
        System.out.println("data reception completed");
    }
    private static void acceptFile(File file, int port, int pacSize) {
        byte data[] = new byte[pacSize];
        DatagramPacket packet = new DatagramPacket(data, data.length);
        try (FileOutputStream outputStream = new FileOutputStream(file)) {
            DatagramSocket datagramSocket = new DatagramSocket(port);
            datagramSocket.setSoTimeout(WAIT_TIMEOUT); /* setting the timeout: if within
                60 seconds no packets were received, data reception ends */
            while (true) {
                datagramSocket.receive(packet);
                outputStream.write(data);
                outputStream.flush();
            }
        }
    }
}

```

```

    }
} catch (SocketTimeoutException e) {
    System.err.println("Timed out, data reception is finished:" + e);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
}
}

```

Вторым запускается приложение по передаче данных.

```
/* # 10 # передача данных по протоколу UDP # UDPSenderMain.java */
```

```

package by.epam.learn.net;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;
public class UDPSenderMain {
    public static void main(String[] args) {
        String fileName = "C:/tmp/Dvorak_SlavonicDance.mp3";
        try (FileInputStream inputStream = new FileInputStream(new File(fileName))) {
            byte[] data = new byte[1024];
            DatagramSocket datagramSocket = new DatagramSocket();
            InetAddress address = InetAddress.getLocalHost();

            DatagramPacket packet;
            System.out.println("sending file...");
            while (inputStream.read(data) != -1) {
                packet = new DatagramPacket(data, data.length, address, 8033);
                datagramSocket.send(packet); // data sending
            }
            System.out.println("file sent successfully.");
        } catch (UnknownHostException e) {
            e.printStackTrace(); // invalid recipient address
        } catch (SocketException e) { // errors during data transfer
            e.printStackTrace();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}

```

В результате передачи аудиофайла, переданный файл в большинстве случаев можно прослушать, но размер нового файла может отличаться и погрешности при звучании также возможны. Если попробовать передавать текстовый файл, размером в несколько мегабайт, то часть информации может быть потеряна, что хорошо видно при визуальном просмотре текста.

Электронная почта

Рассылка электронной почты, в том числе и автоматическая, является стандартным процессом при использовании консольных и веб-приложений. Собственный почтовый сервер создать легко, только необходимо указать адрес и параметры почтового сервера, который будет использован в качестве транспорта.

При построении простейшего почтового сервера требуются интерфейсы **API JavaMail**. Библиотека *JavaMail* содержит классы, позволяющие моделировать систему электронной почты. Класс **javax.mail.Session** представляет сеанс почтовой связи, класс **javax.mail.Message** — почтовое сообщение, класс **javax.mail.internet.InternetAddress** — адрес электронной почты.

Необходимую библиотеку, содержащую, в частности, архив **javax.mail.jar**, следует загрузить по адресу <https://javaee.github.io/javamail/> и добавить этот файл в каталог *jar*-файлов приложения.

В случае автономного запуска без использования промежуточного сервиса для успешной работы почтового приложения можно запустить почтовую программу *James*, являющуюся одним из проектов *apache.org*.

Процессы по отправке письма организуются следующим образом.

```
// # 11 # отправка почтового сообщения # MailMain.java
```

```
package by.epam.learn.mail;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;
public class MailMain {
    public static void main(String[] args) {
        Properties properties = new Properties();
        try {
            properties.load(new FileReader("config\\mail.properties"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(properties);
        String mailTo = "ihar_blinou@epam.com";
        String subject = "Sample Mail";
        String body = "Hello java mail";
        MailSender sender = new MailSender(mailTo, subject, body, properties);
        sender.send();
    }
}
```

Конфигурация почтового сервера размещена в файле `/config/mail.properties`.

```
##### Mail Session properties ###
mail.smtp.host=smtp.gmail.com
mail.smtp.port=465
mail.user.name=АДРЕС@gmail.com
mail.user.password=ПАРОЛЬ
mail.transport.protocol=smtp
mail.host=smtp.gmail.com
mail.smtp.auth=true
mail.smtp.socketFactory.port=465
mail.smtp.socketFactory.class=javax.net.ssl.SSLSocketFactory
mail.smtp.socketFactory.fallback=false
mail.smtp.quitwait=false
```

```
// # 12 # класс формирования и отправки почтового сообщения # MailSender.java
```

```
package by.epam.learn.mail;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
public class MailSender {
    private MimeMessage message;
    private String sendToEmail;
    private String mailSubject;
    private String mailText;
    private Properties properties;
    public MailSender(String sendToEmail, String mailSubject, String mailText, Properties props) {
        this.sendToEmail = sendToEmail;
        this.mailSubject = mailSubject;
        this.mailText = mailText;
        this.properties = props;
    }
    public void send() {
        try {
            initMessage();
            Transport.send(message); // sending mail
        } catch (AddressException e) {
            System.err.println("Invalid address: " + sendToEmail + " " + e); // in Log
        } catch (MessagingException e) {
            System.err.println("Error generating or sending message: " + e); // in Log
        }
    }
}
```

JAVA FROM EPAM

```
private void initMessage() throws MessagingException {
    Session mailSession = SessionFactory.createSession(properties); // mail session object
    mailSession.setDebug(true);
    message = new MimeMessage(mailSession); // create a mailing object
    // Loading parameters into the mail message
    objectmessage.setSubject(mailSubject);
    message.setContent(mailText, "text/html");
    message.setRecipient(Message.RecipientType.TO, new InternetAddress(sendToEmail));
}
```

```
// # 13 # создание почтовой сессии # SessionFactory.java */
```

```
package by.epam.learn.mail;
import java.util.Properties;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
public class SessionFactory {
    public static Session createSession(Properties configProperties) {
        String userName = configProperties.getProperty("mail.user.name");
        String userPassword = configProperties.getProperty("mail.user.password");
        return Session.getDefaultInstance(configProperties,
            new javax.mail.Authenticator() {
                protected PasswordAuthentication getPasswordAuthentication() {
                    return new PasswordAuthentication(userName, userPassword);
                }
            });
    }
}
```

Простой сервлет

Приложение Jakarta EE (*jakarta.ee*) запускает и выполняется в контейнере сервера приложений. Клиент общается с таким приложением посредством веб-браузера. Никаких дополнительных приложений на стороне клиента устанавливать не требуется. Для создания приложения можно воспользоваться wizard-ом создания веб-приложений *IDE IntelliJ IDEA Ultimate* или *IDE Eclipse EE*. Для запуска приложения необходимо установить веб-сервер *Apache Tomcat* в качестве обработчика страниц JSP и сервлетов. Весь java-код выполняется только на сервере. Последняя версия может быть загружена с сайта jakarta.apache.org.

Для демонстрации взаимодействия JSP-страниц и сервлета будет решена задача определения времени между загрузкой страницы в браузер и нажатием кнопки на этой же странице.

Страница **index.jsp** с последующим вызовом через сервлет другой JSP-страницы, отображающей результаты выполнения запроса.

14 # страница JSP с вызовом сервлета # index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html><head><title>JSP Timing</title></head>
<body>
<h5>Счетчик времени от запуска приложения до нажатия кнопки</h5>
<form name="Simple" action="timeaction" method="POST">
<input type="hidden" name="time" value="${System.currentTimeMillis()}" />
<input type="submit" name="button" value="Посчитать время" />
</form>
</body>
</html>
```

В результате запуска стартовой страницы проекта в браузер будет выведено:

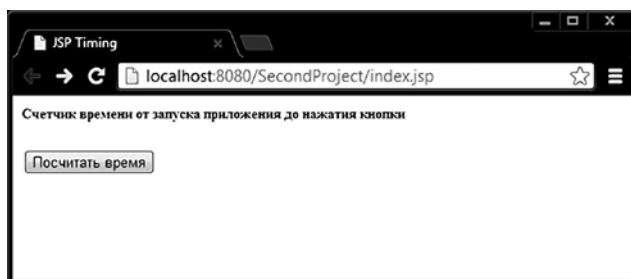


Рис. 14.1. Стартовая страница приложения

Кодировка для символов кириллицы задана с помощью директивы **pageEncoding**. Текущее время фиксируется в параметре **time** значением **System.currentTimeMillis()** и передается в сервлет. Сам сервлет **TimeServlet** вызывается из страницы тегом **form** методом **POST**, с нажатием кнопки.

// # 15 # простой контроллер # TimeServlet.java

```
package by.epam.learn.controller;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
@WebServlet(urlPatterns = "/timeaction")
public class TimeServlet extends HttpServlet {
```

```

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String mSec = request.getParameter("time");
    float delta = ((float)(System.currentTimeMillis()-Long.parseLong(mSec)))/1000;
    request.setAttribute("result", delta);

    request.getRequestDispatcher("/jsp/main.jsp").forward(request, response);
}
}

```

Обмен информацией между сервлетом и JSP осуществляется с помощью атрибутов объектов **HttpServletRequest**, **HttpSession**. Вызов **main.jsp** из сервлета в данном случае производится методом **forward()** интерфейса **RequestDispatcher**.

Для вызова JSP по относительному пути применяется метод **forward()**, для обращения к JSP по абсолютному пути используется метод **sendRedirect()**. Отличие этих методов состоит в том, что с методом **forward()** передается уже существующий объект запроса **request**, а при вызове метода **sendRedirect()** формируется абсолютно новый запрос. Информацию в последнем случае следует передавать с объектом **HttpSession**. К тому же метод **forward()** срабатывает быстрее.

Для запуска приложений версии сервлетов 3.0 и выше необходимо добавить в папку **WEB-INF** конфигурационный файл **web.xml**.

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0">
    <display-name>FirstProject</display-name>
    <session-config>

        <session-timeout>30</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>

```

16 # страница, вызванная сервлетом # main.jsp

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html><head><title>Main Page</title></head>
    <body><p>Кнопка нажата через ${result} сек </p></body>
</html>

```

После обращения к сервлету и выполнения им действий по созданию атрибута объекта **request** будет вызвана страница **main.jsp**, интерпретирующая результат в браузер в виде единственной строки:

Кнопка нажата через 18.057 сек

В коде страницы используется Expression Language (EL) для доступа к атрибуту запроса в виде **\${result}**.

Вопросы к главе 14

1. Семь уровней модели **ISO/OSI**. Для чего была разработана эта модель? Назначения каждого из уровней и основные протоколы, работающие на этих уровнях.
2. Что такое **MAC**-адрес? Кто назначает **MAC**-адреса? Какие известны топологии сетей и методы передачи данных в этих сетях?
3. Принципы адресации на сетевом уровне? Что такое **IP**-адрес, какие известны классы **IP**-адресов? Что такое маска подсети и для чего она служит? Какие зарезервированные **IP**-адреса вы знаете и для чего они служат? Почему для адресации в сети недостаточно **MAC**-адреса сетевого устройства? Для чего предназначены **ARP** и **RARP** протоколы? Как происходит выбор маршрута для передачи данных?
4. В чем отличие протокола **TCP** от **UDP**? Зачем два протокола (**TCP** и **IP**), работающие на различных уровнях, объединяют в один стек **TCP/IP**? Каково назначение протоколов **DHCP**? Что такое и зачем нужен **DNS**?
5. Что такое хост и порт? Определение сокета. Как происходит взаимодействие двух объектов типа **Socket** и **ServerSocket**? Сокеты в Java взаимодействуют по протоколу **TCP** или **UDP**?
6. Для чего предназначен класс **DatagramSocket**? Объяснить, как происходит взаимодействие между объектами **DatagramSocket**.

Задания к главе 14

Вариант А

Создать на основе сокетов клиент/серверное приложение:

1. Клиент посылает через сервер сообщение другому клиенту, выбранному из списка.
2. Чат. Клиент посылает через сервер сообщение, которое получают все клиенты. Список клиентов хранится на сервере в файле.
3. Клиент при обращении к серверу получает случайно выбранное стихотворение Максима Богдановича из файла.
4. Сервер рассылает сообщения выбранным из списка клиентам. Список клиентов хранится в файле.

5. Сервер рассылает сообщения в определенное время определенным клиентам.
6. Сервер рассылает сообщения только тем клиентам, которые в настоящий момент находятся в online.
7. Чат. Сервер рассылает всем клиентам информацию о клиентах, вошедших в чат и покинувших его.
8. Клиент выбирает изображение из списка и пересылает его другому клиенту через сервер.

Вариант В

1. Игра по сети в «Морской бой».
2. Игра по сети «Го». Крестики-нолики на безразмерном (большом) поле. Для победы необходимо выстроить пять в один ряд.
3. Написать программу, сканирующую сеть в указанном диапазоне IP-адресов на наличие активных компьютеров.
4. Прокси. Программа должна принимать сообщения от любого клиента, работающего на протоколе **TCP**, и отсылать их на соответствующий сервер. При передаче изменять сообщение.
5. Создать консольное приложение-калькулятор, используя вычислительную среду **Google Sheets**.
6. С помощью **Google Slides API** вывести статистику по заданной презентации, название презентации, локаль текста презентации, размер слайдов, количество слайдов, количество элементов на каждом слайде; задать новое название презентации, изменить размер слайдов.

Java API for XML Processing

JAXP (*Java API for XML Processing*). XML-документ как набор байт в памяти, запись в базе или текстовый файл представляет собой данные, которые еще предстоит обработать. То есть из набора строк необходимо получить данные, пригодные для использования в проекте. Поскольку XML представляет собой универсальный формат для передачи данных, существуют универсальные средства его обработки — XML-анализаторы, или парсеры.

Парсер — это библиотека, которая читает XML-документ, а затем предоставляет набор методов для обработки информации из этого документа.

Древовидная и псевдособытийная модели

- Существуют три стандартных подхода (API) к обработке XML-документов:
- **DOM** (*Document Object Model*) — платформенно-независимый программный интерфейс, позволяющий программам и скриптам управлять содержимым документов HTML и XML, а также изменять их структуру и оформление. Модель DOM не накладывает ограничений на структуру документа. Любой документ известной структуры с помощью DOM может быть представлен в виде дерева узлов, каждый узел которого содержит элемент, атрибут, текстовый, графический или любой другой объект. Узлы связаны между собой отношениями родитель-потомок.
 - **SAX** (*Simple API for XML*) базируется на модели последовательной одноразовой обработки и не создает внутренних деревьев. При прохождении по XML вызывает соответствующие методы у классов, реализующих интерфейсы, предоставляемые SAX-парсером.
 - **StAX** (*Streaming API for XML*) не создает дерево объектов в памяти, но, в отличие от SAX-парсера, за переход от одной вершины XML к другой отвечает приложение, которое запускает разбор документа.

Анализаторы, которые строят древовидную модель, — это DOM-анализаторы. Анализаторы, которые генерируют квазисобытия, — это SAX-анализаторы.

Анализаторы, которые ждут команды от приложения для перехода к следующему элементу XML, — StAX-анализаторы.

В первом случае анализатор строит в памяти объект, представляющий собой дерево из элементов, соответствующее XML-документу. Далее вся работа

ведется именно с этим объектом-деревом, который может быть очень большим, но предоставляет доступ к своим элементам в любой момент времени.

Во втором случае анализатор работает следующим образом: при чтении/анализе документа, анализатор вызывает методы, связанные с различными участками XML-файла, а программа, использующая анализатор, решает, как реагировать на тот или иной элемент XML-документа. Так, анализатор будет генерировать событие о том, что он встретил начало документа либо его конец, начало элемента либо его конец, символьную информацию внутри элемента и т.д.

Анализатор StAX работает подобно итератору, который указывает на наличие элемента с помощью метода **hasNext()** и для перехода к следующей вершине использует метод **next()**.

Когда следует использовать DOM, а когда — SAX, StAX-анализаторы?

DOM-анализаторы следует использовать тогда, когда нужно знать структуру документа и может понадобиться изменять эту структуру либо использовать информацию из XML-документа в любой момент времени работы приложения.

SAX/StAX-анализаторы используются тогда, когда нужно извлечь информацию о нескольких элементах из XML-файла, либо когда информация из документа нужна только один раз.

Валидация

XML-документ может соответствовать двум видам корректности: синтаксической (*well-formed*) — документ сформирован в соответствии с синтаксическими правилами построения — и действительной (*valid*) — документ синтаксически корректен и соответствует требованиям, заявленным в XSD.

Соответственно, есть невалидирующие и валидирующие анализаторы. И те, и другие проверяют XML-документ на соответствие синтаксическим правилам.

Но только валидирующие анализаторы знают, как проверить XML-документ на соответствие структуре, описанной в XSD.

Никакой связи между видом анализатора и видом XML-документа не существует. Валидирующий анализатор может разобрать XML-документ, для которого нет XSD, и, наоборот, невалидирующий анализатор может разобрать XML-документ, для которого есть XSD. При этом он просто не будет учитывать описание структуры документа.

Следующий пример выполняет проверку документа на валидность, то есть соответствие схеме XSD, средствами языка Java при парсинге документа.

```
/* # 1 # проверка корректности документа XML # BaseValidatorMain.java */
```

```
package by.epam.learn.xml.validator;
import java.io.*;
import javax.xml.XMLConstants;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
```

```

import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import by.epam.learn.xml.handler.StudentErrorHandler;
import org.xml.sax.SAXException;
public class BaseValidatorMain {
    public static void main(String[] args) {
        String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
        String fileName = "data_xml/students.xml";
        String schemaName = "data_xml/students.xsd";
        SchemaFactory factory = SchemaFactory.newInstance(language);
        File schemaLocation = new File(schemaName);
        try {
            // schema creation
            Schema schema = factory.newSchema(schemaLocation);
            // creating a schema-based validator
            Validator validator = schema.newValidator();
            Source source = new StreamSource(fileName);
            // document check
            validator.setErrorHandler(new StudentErrorHandler());
            validator.validate(source);
        } catch (SAXException | IOException e) {
            System.err.println(fileName + " is not correct or valid");
        }
    }
}

```

Схема XSD находится в файле **students.xsd**, код которого можно найти в примере #21 этой главы.

К валидатору можно добавить класс обработчика ошибок **StudentErrorHandler**, способный точно указать в xml-файле номер строки и позицию начала ошибки. Для обработки предупреждений и ошибок, возникающих при разборе XML-документа, применяется интерфейс **org.xml.sax.ErrorHandler**, содержащий **void**-методы: **warning()**, **error()**, **fatalError()**. У всех методов параметр типа **SAXParseException**.

Имплементация интерфейса **ErrorHandler** может выглядеть следующим образом:

```

/* # 2 # обработчик ошибок # StudentErrorHandler.java */
package by.epam.learn.xml.handler;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXParseException;
public class StudentErrorHandler implements ErrorHandler {
    private static Logger logger = LogManager.getLogger();
    public void warning(SAXParseException e) {
        logger.warn(getLineNumber(e) + "-" + e.getMessage());
    }
}

```

```

public void error(SAXParseException e) {
    logger.error(getLineColumnNumber(e) + " - " + e.getMessage());
}
public void fatalError(SAXParseException e) {
    logger.fatal(getLineColumnNumber(e) + " - " + e.getMessage());
}
private String getLineColumnNumber(SAXParseException e) {
    // determine line and position of error
    return e.getLineNumber() + " : " + e.getColumnNumber();
}
}

```

При проверке XML-документа разумно все ошибки фиксировать, в частности, с помощью логгера **Log4J2**, минимальные библиотеки которого **log4j-api-2.[version].jar** и **log4j-core-2.[version].jar** следует подключить к проекту.

Чтобы убедиться в работоспособности кода, следует внести в исходный XML-документ ошибку. Например, сделать идентичными значения атрибута **login**. Тогда в результате запуска в файл будут выведены следующие сообщения обработчика об ошибках:

ERROR - 14 : 48 - cvc-id.2: There are multiple occurrences of ID value 'MitarAlex7'.

ERROR - 14 : 48 - cvc-attribute.3: The value 'MitarAlex7' of attribute 'login' on element 'student' is not valid with respect to its type, 'Login'.

Если допустить синтаксическую ошибку в XML-документе, например, удалить закрывающую угловую скобку в элементе **telephone**, будет выведено сообщение о фатальной ошибке:

FATAL - 7 : 26 - Element type "telephone2456474" must be followed by either attribute specifications, ">" or ">".

Псевдособытийная модель

Как уже отмечалось, SAX-анализатор не строит дерево элементов по содержимому XML-файла. Вместо этого анализатор читает файл и генерирует квазисобытие при нахождении элемента, атрибута или текста. На первый взгляд, такой подход менее естествен для приложения, использующего анализатор, так как он не строит дерево, а приложение само должно догадаться, какое дерево элементов описывается в XML-документе.

Однако нужно учитывать, для каких целей используются данные из XML-файла. Очевидно, что нет смысла строить дерево объектов, содержащее десятки тысяч элементов в памяти, если все, что необходимо, — просто посчитать точное количество элементов в файле.

SAX-анализаторы

SAX2 API определяет ряд интерфейсов, используемых при разборе документа. Чаще других используется **org.xml.sax.ContentHandler** и некоторые объявленные в нем методы:

void startDocument() — вызывается на старте обработки документа;

void endDocument() — вызывается при завершении разбора документа;

void startElement(String uri, String localName, String qName, Attributes attrs) — будет вызван, когда анализатор полностью обработает содержимое открывающего тега, включая его имя и все содержащиеся атрибуты;

void endElement(String uri, String localName, String qName) — сигнализирует о завершении элемента;

void characters(char[] ch, int start, int length) — вызывается в том случае, если анализатор встретил символьную информацию внутри элемента (тело тега). Если этой информации достаточно много, то метод может быть вызван более одного раза.

В пакете **org.xml.sax** в SAX2 API содержатся также интерфейсы, необходимые для обработки интересующего события.

Для того чтобы создать простейшее приложение, обрабатывающее XML-документ, достаточно сделать следующее:

1. Создать класс или классы, которые реализуют один или несколько интерфейсов **ContentHandler**, **ErrorHandler** или **EntityResolver**, или наследует их класс-адаптер **org.xml.sax.helpers.DefaultHandler**, и реализовать методы, отвечающие за обработку интересующих частей документа или ошибок. Класс **DefaultHandler** имплементирует все вышеуказанные интерфейсы и реализует все их абстрактные методы. Его реализации методов пустые, что позволяет разработчику переопределять только необходимые для выполнения задачи методы.
2. Используя SAX2 API, поддерживаемое всеми SAX-парсерами, создать **org.xml.sax.XMLReader**, например:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
XMLReader xmlReader = parser.getXMLReader();
```

3. Передать в объект **XMLReader** объект одного или нескольких классов, созданного на первом шаге с помощью методов: **setContentHandler(ContentHandler handler)**, **setErrorHandler(ErrorHandler handler)**, **setEntityResolver(EntityResolver resolver)** и прочих.
4. Вызвать метод **parse(String filename)** или **parse(DataSource input)** класса **XMLReader**, которому в качестве параметра передать путь к анализируемому документу либо **DataSource**.

Пусть дан некоторый XML-документ с описанием студентов.

3 # описание студентов # students.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.example.com/students"
  xsi:schemaLocation="http://www.example.com/students students.xsd">
  <student login="MitarAlex7" faculty="mmf">
    <name>Mitar Alex</name>
    <telephone>2456474</telephone>
    <address>
      <country>Belarus</country>
      <city>Minsk</city>
      <street>Kalinouskaha</street>
    </address>
  </student>
  <student login="Pashkin5" faculty="csan">
    <name>Pashkin Jan</name>
    <telephone>3453789</telephone>
    <address>
      <country>Belarus</country>
      <city>Polotesk</city>
      <street>Bahdanovicha</street>
    </address>
  </student>
</students>
```

Этот файл будет использоваться в дальнейшем в этой главе для парсинга и других действий.

Следующий пример в результате парсинга выводит на консоль содержимое XML-документа.

/* # 4 # чтение и вывод XML-документа # ConsoleStudentHandler.java */

```
package by.epam.learn.xml.handler;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
public class ConsoleStudentHandler extends DefaultHandler {
  @Override
  public void startDocument() {
    System.out.println("Parsing started");
  }
  @Override
  public void startElement(String uri, String localName, String qName, Attributes attrs) {
    String tagData = qName + " ";
    for (int i = 0; i < attrs.getLength(); i++) {
      tagData += " " + attrs.getQName(i) + "=" + attrs.getValue(i);
    }
    System.out.print(tagData);
  }
  @Override
  public void characters(char[] ch, int start, int length) {
```

```

        System.out.print(new String(ch, start, length));
    }
    @Override
    public void endElement(String uri, String localName, String qName) {
        System.out.print(" " + qName);
    }
    @Override
    public void endDocument() {
        System.out.println("\nParsing ended");
    }
}

```

где **uri** — уникальное название **namespace**, **localName** — имя элемента без префикса, задаваемого именем атрибута **xmlns**, **qName** — полное имя элемента с префиксом, **attrs** — список атрибутов.

```
/* # 5 # создание и запуск простейшего парсера # SaxConsoleMain.java */
```

```

package by.epam.learn.xml.parser;
import by.epam.learn.xml.handler.ConsoleStudentHandler;
import by.epam.learn.xml.handler.StudentErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.io.IOException;
public class SaxConsoleMain {
    public static void main(String[] args) {
        try {
            // SAX parser creating & configuring
            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser parser = factory.newSAXParser();
            XMLReader reader = parser.getXMLReader();
            reader.setContentHandler(new ConsoleStudentHandler());
            reader.setErrorHandler(new StudentErrorHandler());
            reader.parse("data_xml/students.xml");
        } catch (SAXException | IOException | ParserConfigurationException e) {
            e.printStackTrace();
        }
    }
}

```

В результате в консоль будет выведено:

Parsing started

```

students xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns=http://www.example.com/students xsi:schemaLocation=http://www.
example.com/students students.xsd
  student login=MitarAlex7 faculty=mmf
    name Mitar Alex name

```

```

    telephone 2456474 telephone
    address
      country Belarus country
      city Minsk city
      street Kalinouskaha street
    address
  student
student login=Pashkin5 faculty=csan
  name Pashkin Jan name
  telephone 3453789 telephone
  address
    country Belarus country
    city Polotesk city
    street Bahdanovicha street
  address
student
students
Parsing ended

```

При XML-парсинге приложение получает важные для работы сведения, которые могут содержать информацию, необходимую для инициализации объектов или конфигурации.

```
/* # 6 # класс-сущность # Student.java */
```

```

package by.epam.learn.xml.entity;
public class Student {
    private String login;
    private String name;
    private String faculty;
    private int telephone;
    private Address address = new Address();
    public Student() {
    }
    public Student(String login, String name, String faculty, int telephone, Address address) {
        this.login = login;
        this.name = name;
        this.faculty = faculty;
        this.telephone = telephone;
        this.address = address;
    }
    public String getLogin() {
        return login;
    }
    public void setLogin(String login) {
        this.login = login;
    }
}

```

```

}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getFaculty() {
    return faculty;
}
public void setFaculty(String faculty) {
    this.faculty = faculty;
}
public int getTelephone() {
    return telephone;
}
public void setTelephone(int telephone) {
    this.telephone = telephone;
}
public Address getAddress() {
    return address;
}
public void setAddress(Address address) {
    this.address = address;
}
public String toString() {
    final StringBuilder sb = new StringBuilder("\nLogin: ");
    sb.append(login).append("\nName: ").append(name).append("\nTelephone: ").append(telephone);
    sb.append("\nFaculty: ").append(faculty).append(address);
    return sb.toString();
}
public class Address { // inner class
    private String country;
    private String city;
    private String street;
    public Address() {
    }
    public Address(String country, String city, String street) {
        this.country = country;
        this.city = city;
        this.street = street;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
    public String getCity() {

```

```

        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
    public String toString() {
        final StringBuilder sb = new StringBuilder("\nAddress:\n\tCountry:");
        sb.append(country).append("\n\tCity: ").append(city);
        sb.append("\n\tStreet: ").append(street).append('\n');
        return sb.toString();
    }
}
}

```

В следующем приложении производятся разбор документа **students.xml** и инициализация на его основе коллекции объектов класса **Student**.

```

/* # 7 # формирование коллекции объектов на основании разбора XML-документа
# StudentsSaxBuilder.java */

```

```

package by.epam.learn.xml.parser;
import by.epam.learn.xml.entity.Student;
import by.epam.learn.xml.handler.StudentErrorHandler;
import by.epam.learn.xml.handler.StudentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.io.IOException;
import java.util.Set;
public class StudentsSaxBuilder {
    private Set<Student> students;
    private StudentHandler handler = new StudentHandler();
    private XMLReader reader;
    public StudentsSaxBuilder() {
        // reader configuration
        SAXParserFactory factory = SAXParserFactory.newInstance();
        try {
            SAXParser saxParser = factory.newSAXParser();
            reader = saxParser.getXMLReader();
        } catch (ParserConfigurationException | SAXException e) {
            e.printStackTrace(); // Log
        }
    }
}

```

```

        reader.setErrorHandler(new StudentErrorHandler());
        reader.setContentHandler(handler);
    }
    public Set<Student> getStudents() {
        return students;
    }
    public void buildSetStudents(String filename) {
        try {
            reader.parse(filename);
        } catch (IOException | SAXException e) {
            e.printStackTrace(); // Log
        }
        students = handler.getStudents();
    }
}

```

Парсинг запускается также и с помощью метода **parse(String uri, DefaultHandler dh)** объекта класса **SAXParser**, но в это случае придется в реализации класса **DefaultHandler** добавлять реализацию обработчика ошибок.

```

/* # 8 # сопоставление тегов и атрибутов элементов перечисления
# StudentXmlTag.java */

```

```

package by.epam.learn.xml.handler;
public enum StudentXmlTag {
    STUDENTS("students"),
    LOGIN("login"),
    FACULTY("faculty"),
    STUDENT("student"),
    NAME("name"),
    TELEPHONE("telephone"),
    COUNTRY("country"),
    CITY("city"),
    STREET("street"),
    ADDRESS("address");
    private String value;
    StudentXmlTag(String value) {
        this.value = value;
    }
    public String getValue() {
        return value;
    }
}

```

```

/* # 9 # обработчик xml-документа # StudentHandler.java */

```

```

package by.epam.learn.xml.handler;
import by.epam.learn.xml.entity.Student;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import java.util.EnumSet;

```

```

import java.util.HashSet;
import java.util.Set;
public class StudentHandler extends DefaultHandler {
    private Set<Student> students;
    private Student current;
    private StudentXmlTag currentXmlTag;
    private EnumSet<StudentXmlTag> withText;
    private static final String ELEMENT_STUDENT = "student";
    public StudentHandler() {
        students = new HashSet<Student>();
        withText = EnumSet.range(StudentXmlTag.NAME, StudentXmlTag.STREET);
    }
    public Set<Student> getStudents() {
        return students;
    }
    public void startElement(String uri, String localName, String qName, Attributes attrs) {
        if (ELEMENT_STUDENT.equals(qName)) {
            current = new Student();
            current.setLogin(attrs.getValue(0));
            if (attrs.getLength() == 2) { // warning!!!!
                current.setFaculty(attrs.getValue(1));
            }
        } else {
            StudentXmlTag temp = StudentXmlTag.valueOf(qName.toUpperCase());
            if (withText.contains(temp)) {
                currentXmlTag = temp;
            }
        }
    }
    public void endElement(String uri, String localName, String qName) {
        if (ELEMENT_STUDENT.equals(qName)) {
            students.add(current);
        }
    }
    public void characters(char[] ch, int start, int length) {
        String data = new String(ch, start, length).strip();
        if (currentXmlTag != null) {
            switch (currentXmlTag) {
                case NAME -> current.setName(data);
                case TELEPHONE -> current.setTelephone(Integer.parseInt(data));
                case STREET -> current.getAddress().setStreet(data);
                case CITY -> current.getAddress().setCity(data);
                case COUNTRY -> current.getAddress().setCountry(data);
                default -> throw new EnumConstantNotPresentException(
                    currentXmlTag.getDeclaringClass(), currentXmlTag.name());
            }
        }
        currentXmlTag = null;
    }
}

```

```
/* # 10 # создание и запуск парсера # */
```

```
StudentsSaxBuilder saxBuilder = new StudentsSaxBuilder();
saxBuilder.buildSetStudents("data_xml/students.xml");
System.out.println(saxBuilder.getStudents());
```

В результате на консоль будет выведено следующее множество описаний объектов:

```
[
Login: MitarAlex7
Name: Mitar Alex
Telephone: 2456474
Faculty: mmf
Address:
    Country: Belarus
    City: Minsk
    Street: Kalinouskaha
,
Login: Pashkin5
Name: Pashkin Jan
Telephone: 3453789
Faculty: csan
Address:
    Country: Belarus
    City: Polotesk
    Street: Bahdanovicha
]
```

Древовидная модель

Анализатор DOM представляет собой некоторый общий интерфейс для работы со структурой документа. При разработке DOM-анализаторов различными вендорами предполагалась возможность ковариантности кода, однако при совместном использовании библиотек с аналогичными классами необходимо следить за совместимостью и корректностью взаимодействия.

DOM строит дерево, которое представляет содержимое XML-документа и определяет набор классов, представляющих каждый элемент в XML-документе: элементы, атрибуты, сущности, текст и т.д.

В пакете **org.w3c.dom** можно найти интерфейсы, представляющие указанные объекты. Разработчики приложений, которые хотят использовать DOM-анализатор, имеют готовый набор методов для манипуляции деревом объектов и не зависят от конкретной реализации используемого анализатора.

Существуют общепризнанные DOM-анализаторы: Xerces, JDOM и JAXP, входящий в JDK.

DOM

В стандартную конфигурацию Java входит набор пакетов для работы с XML. Но стандартная библиотека не всегда является самой простой в применении, поэтому часто в основе многих проектов, использующих XML, лежат библиотеки сторонних производителей. Эти библиотеки представлены проектами Xerces, Xalan, JDOM. Особенностью всех их является использование части стандартных возможностей XML-библиотек Java с добавлением собственных классов и методов, упрощающих и облегчающих обработку документов XML.

Ниже приведены интерфейсы и методы, необходимые для разбора XML-документа.

org.w3c.dom.Node

Основным объектом DOM является **Node** — некоторый базовый элемент дерева. Большинство DOM-объектов унаследовано именно от **Node**. Для представления элементов, атрибутов, сущностей разработаны свои реализации **Node**. Некоторые из них — **Element**, **Attr**, **Text**, для работы с конкретными объектами дерева.

Интерфейс **Node** определяет ряд общих для всех реализаций методов для работы с деревом:

short getNodeType() — возвращает тип объекта: элемент, атрибут, текст, CDATA и т.д.;

String getNodeValue() — возвращает значение **Node**;

Node getParentNode() — возвращает объект, являющийся родителем текущего узла **Node**;

NodeList getChildNodes() — возвращает список объектов, являющихся дочерними элементами;

NamedNodeMap getAttributes() — возвращает список атрибутов данного элемента.

org.w3c.dom.Document

Для получения информации о документе и изменения его структуры. Этот интерфейс представляет собой корневой элемент XML-документа и содержит методы доступа ко всему содержимому документа.

Element getDocumentElement() — возвращает корневой элемент документа.

org.w3c.dom.Element

Интерфейс предназначен для работы с элементом XML-документа и его содержимым. Некоторые методы:

String getTagName(String name) — возвращает имя элемента;

boolean hasAttribute() — проверяет наличие атрибутов;

String getAttribute(String name) — возвращает значение атрибута по его имени;

Attr getAttributeNode(String name) — возвращает атрибут по его имени;

NodeList getElementsByTagName(String name) — возвращает список дочерних элементов с определенным именем.

org.w3c.dom.Attr

Интерфейс служит для работы с атрибутами элемента XML-документа.

Некоторые методы интерфейса **Attr**:

String getName() — возвращает имя атрибута;

Element getOwnerElement() — возвращает элемент, который содержит этот атрибут;

String getValue() — возвращает значение атрибута;

boolean isId() — проверяет атрибут на тип ID.

Ниже приведен стандартный способ разбора документа **students.xml** с использованием DOM-анализатора и инициализация на его основе множества объектов.

```
/* # 11 # создание объектов на основе экземпляра Document
# StudentsDomBuilder.java */
```

```
package by.epam.learn.xml.parser;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import by.epam.learn.xml.entity.Student;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
public class StudentsDomBuilder {
    private Set<Student> students;
    private DocumentBuilder docBuilder;
    public StudentsDomBuilder() {
        students = new HashSet<Student>();
        // configuration
```

```

    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    try {
        docBuilder = factory.newDocumentBuilder();
    } catch (ParserConfigurationException e) {
        e.printStackTrace(); // Log
    }
}

public Set<Student> getStudents() {
    return students;
}

public void buildSetStudents(String filename) {
    Document doc;
    try {
        doc = docBuilder.parse(filename);
        Element root = doc.getDocumentElement();
        // getting a list of <student> child elements
        NodeList studentsList = root.getElementsByTagName("student");
        for (int i = 0; i < studentsList.getLength(); i++) {
            Element studentElement = (Element) studentsList.item(i);
            Student student = buildStudent(studentElement);
            students.add(student);
        }
    } catch (IOException | SAXException e) {
        e.printStackTrace(); // Log
    }
}

private Student buildStudent(Element studentElement) {
    Student student = new Student();
    // add null check
    student.setFaculty(studentElement.getAttribute("faculty"));
    student.setName(getElementTextContent(studentElement, "name"));
    Integer tel = Integer.parseInt(getElementTextContent(studentElement, "telephone"));
    student.setTelephone(tel);
    Student.Address address = student.getAddress();
    // init an address object
    Element addressElement =
        (Element) studentElement.getElementsByTagName("address").item(0);
    address.setCountry(getElementTextContent(addressElement, "country"));
    address.setCity(getElementTextContent(addressElement, "city"));
    address.setStreet(getElementTextContent(addressElement, "street"));
    student.setLogin(studentElement.getAttribute("login"));
    return student;
}

// get the text content of the tag
private static String getElementTextContent(Element element, String elementName) {
    NodeList nList = element.getElementsByTagName(elementName);
    Node node = nList.item(0);
    String text = node.getTextContent();
    return text;
}
}

```

```
/* # 12 # создание и запуск парсера # */
```

```
StudentsDomBuilder domBuilder = new StudentsDomBuilder();
domBuilder.buildSetStudents("data_xml/students.xml");
System.out.println(domBuilder.getStudents());
```

Создание XML-документа

Документы можно не только читать, но также модифицировать и создавать совершенно новые. Для этого необходимо создать объекты классов **Document**, **Element**, добавить к последнему атрибуты и текстовое содержимое, после чего присоединить их к объекту, который в дереве XML-документа будет находиться выше.

Следующий пример демонстрирует создание XML-документа и запись его в файл. Для записи XML-документа используется класс **Transformer**.

```
/* # 13 # создание и запись документа # CreateXmlDocumentMain.java */
```

```
package by.epam.learn.xml.transform;
import java.io.FileWriter;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
public class CreateXmlDocumentMain {
    public static void main(String[] args) {
        DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder documentBuilder = null;
        try {
            documentBuilder = documentBuilderFactory.newDocumentBuilder();
        } catch (ParserConfigurationException e) {
            e.printStackTrace();
        }
        // forming a document tree
        Document document = documentBuilder.newDocument();
        String root = "book";
        Element rootElement = document.createElement(root);
        document.appendChild(rootElement);
        Element elementName = document.createElement("name");
```

```
String name = "Java from EPAM";
elementName.appendChild(document.createTextNode(name));
Element elementAuthor = document.createElement("author");
String author = "Blinov";
elementAuthor.appendChild(document.createTextNode(author));
elementAuthor.setAttribute("id", "777");
rootElement.appendChild(elementName);
rootElement.appendChild(elementAuthor);
// write tree to file
TransformerFactory transformerFactory = TransformerFactory.newInstance();
try {
    Transformer transformer = transformerFactory.newTransformer();
    DOMSource source = new DOMSource(document);
    StreamResult result = new StreamResult(new FileWriter("data_xml/book.xml"));
    transformer.transform(source, result);
} catch (TransformerConfigurationException e) {
    e.printStackTrace();
} catch (TransformerException | IOException e) {
    e.printStackTrace();
}
}
```

В результате будет создан документ **book.xml** следующего содержания:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<book>
  <name>Java from EPAM</name>
  <author id="777">Blinov</author>
</book>
```

StAX

StAX (*Streaming API for XML*), который еще называют *pull*-парсером, включен в JDK, начиная с версии Java 6. Он похож на SAX отсутствием объектной модели в памяти и последовательным продвижением по XML, но в StAX не требуется реализация интерфейсов, и приложение само указывает StAX парсеру перейти к следующему элементу XML. Кроме того, в отличие от SAX, данный парсер предлагает API для создания XML-документа.

Основными классами StAX являются **XMLStreamReader** и **XMLStreamWriter**, которые, соответственно, используются для чтения и создания XML-документа и расположены в пакете **javax.xml.stream**. Для чтения XML требуется получить ссылку на экземпляр **XMLStreamReader**:

```
StringReader input = new StringReader(filename);
```

ИЛИ

```
InputStream input = new FileInputStream(new File(filename));
```

далее

```
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLStreamReader reader = inputFactory.createXMLStreamReader(input);
```

после чего по экземпляру **XMLStreamReader** можно организовать навигацию аналогично интерфейсу **Iterator**, используя методы **hasNext()** и **next()**:

boolean hasNext() — показывает, есть ли еще элементы;

int next() — переходит к следующей вершине-константе XML, извлекая тип текущей.

При попытке вызова на константе несоответствующего ей метода генерируется исключительная ситуация **IllegalStateException**.

Чаще всего данные извлекаются с применением методов:

String getLocalName() — возвращает название тега (элемента) для текущей константы;

String getAttributeValue(String namespaceURI, String localName) — возвращает значение атрибута по имени;

String getAttributeValue(int index) — возвращает значение атрибута по номеру позиции;

String getText() — возвращает текст для констант **CHARACTERS**, **CDATA**, **COMMENT** и др.

Возможные типы вершин и методы, применимые к ним (см. таблицу ниже).

Типы вершин-констант	Методы
Для всех типов констант	getProperty(), hasNext(), require(), close(), getNamespaceURI(), isStartElement(), isEndElement(), isCharacters(), isWhiteSpace(), getNamespaceContext(), getEventType(), getLocation(), hasText(), hasName()
START_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getAttributeXXX(), isAttributeSpecified(), getNamespaceXXX(), getElementText(), nextTag()
ATTRIBUTE	next(), nextTag() getAttributeXXX(), isAttributeSpecified(),
NAMESPACE	next(), nextTag() getNamespaceXXX()
END_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getNamespaceXXX(), nextTag()
CHARACTERS, CDATA, COMMENT, SPACE	next(), getTextXXX(), nextTag()
START_DOCUMENT	next(), getEncoding(), getVersion(), isStandalone(), standaloneSet(), getCharacterEncodingScheme(), nextTag()
END_DOCUMENT	close()
PROCESSING_INSTRUCTION	next(), getPITarget(), getPIData(), nextTag()
ENTITY_REFERENCE	next(), getLocalName(), getText(), nextTag()

Организация процесса разбора документа XML и создания множества объектов класса **Student** с помощью Cursor API StAX приведена в следующем примере:

```
/* # 14 # реализация разбора XML-документа на основе Stream StAX
# StudentsStaxBuilder.java */
```

```
package by.epam.learn.xml.parser;
import by.epam.learn.xml.entity.Student;
import by.epam.learn.xml.handler.StudentXmlTag;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;
public class StudentsStaxBuilder {
    private Set<Student> students;
    private XMLInputFactory inputFactory;
    public StudentsStaxBuilder() {
        inputFactory = XMLInputFactory.newInstance();
        students = new HashSet<Student>();
    }
    public Set<Student> getStudents() {
        return students;
    }
    public void buildSetStudents(String filename) {
        XMLStreamReader reader;
        String name;
        try(FileInputStream inputStream = new FileInputStream(new File(filename))) {
            reader = inputFactory.createXMLStreamReader(inputStream);
            // StAX parsing
            while (reader.hasNext()) {
                int type = reader.next();
                if (type == XMLStreamConstants.START_ELEMENT) {
                    name = reader.getLocalName();
                    if (name.equals(StudentXmlTag.STUDENT.getValue())) {
                        Student student = buildStudent(reader);
                        students.add(student);
                    }
                }
            }
        } catch (XMLStreamException | FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

}
private Student buildStudent(XMLStreamReader reader) throws XMLStreamException {
    Student student = new Student();
    student.setLogin(reader.getAttributeValue(null, StudentXmlTag.LOGIN.getValue()));
    // null check
    student.setFaculty(reader.getAttributeValue(null, StudentXmlTag.FACULTY.getValue()));
    String name;
    while (reader.hasNext()) {
        int type = reader.next();
        switch (type) {
            case XMLStreamConstants.START_ELEMENT:
                name = reader.getLocalName();
                switch (StudentXmlTag.valueOf(name.toUpperCase())) {
                    case NAME -> student.setName(getXMLText(reader));
                    case TELEPHONE -> student.setTelephone(Integer.parseInt(getXMLText(reader)));
                    case ADDRESS -> student.setAddress(getXMLAddress(reader));
                }
                break;
            case XMLStreamConstants.END_ELEMENT:
                name = reader.getLocalName();
                if (StudentXmlTag.valueOf(name.toUpperCase()) == StudentXmlTag.STUDENT) {
                    return student;
                }
        }
    }
    throw new XMLStreamException("Unknown element in tag <student>");
}
private Student.Address getXMLAddress(XMLStreamReader reader)
    throws XMLStreamException {
    Student.Address address = new Student().new Address();
    int type;
    String name;
    while (reader.hasNext()) {
        type = reader.next();
        switch (type) {
            case XMLStreamConstants.START_ELEMENT:
                name = reader.getLocalName();
                switch (StudentXmlTag.valueOf(name.toUpperCase())) {
                    case COUNTRY -> address.setCountry(getXMLText(reader));
                    case CITY -> address.setCity(getXMLText(reader));
                    case STREET -> address.setStreet(getXMLText(reader));
                }
                break;
            case XMLStreamConstants.END_ELEMENT:
                name = reader.getLocalName();
                if (StudentXmlTag.valueOf(name.toUpperCase()) == StudentXmlTag.ADDRESS) {
                    return address;
                }
        }
    }
}

```

```

    }
  }
}
throw new XMLStreamException("Unknown element in tag <address>");
}
private String getXMLText(XMLStreamReader reader) throws XMLStreamException {
    String text = null;
    if (reader.hasNext()) {
        reader.next();
        text = reader.getText();
    }
    return text;
}
}
}

```

Для запуска приложения разбора документа с помощью StAX ниже приведен достаточно простой код, аналогичный коду запуска SAX и DOM-парсеров.

```
/* # 15 # создание и запуск StAX-парсера # */
```

```

StudentsStaxBuilder staxBuilder = new StudentsStaxBuilder();
staxBuilder.buildSetStudents("data_xml/students.xml");
System.out.println(staxBuilder.getStudents());

```

Iterator StAX API, базирующийся на итераторах, позволяет приложению обрабатывать XML как серию объектов событий, каждый из которых содержит определенную часть структуры XML-документа. Приложению требуется определить тип анализируемого события и использовать его методы для получения информации, связанной с ним.

Основными интерфейсами этого парсера StAX являются **XMLStreamReader** и **XMLStreamWriter**, которые, соответственно, используются для чтения и создания XML-документа и расположены в пакете **javax.xml.stream**. Для чтения XML требуется создать экземпляр **XMLStreamReader**:

```
InputStream input = new FileInputStream(new File(filename));
```

далее

```

XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLStreamReader reader = inputFactory.createXMLStreamReader(input);

```

после чего по экземпляру **XMLStreamReader** можно организовать навигацию аналогично интерфейсу **Iterator**, используя методы **hasNext()** и **nextEvent()**:

boolean hasNext() — показывает, есть ли еще элементы;

XMLStreamEvent nextEvent() — переходит к следующему событию XML, извлекая текущее.

У интерфейса **XMLStreamEvent** используются методы **isStartElement()** и **isEndElement()** для определения является ли тег начальным или конечным. Метод **asCharacters()** извлекает информацию из тела элемента.

Методы **getName()** и **getAttributeByName(QName qName)** интерфейса **StartElement** позволяют получить имя элемента и значение его атрибутов.

Метод **buildSetStudents()** для StAX-парсера в случае использования событийной модели будет записан следующим образом:

```
/* # 16 # реализация разбора XML-документа на основе Event StAX # */
```

```
public void buildSetStudents(String fileName) {
    Student student = null;
    XMLInputFactory inputFactory = XMLInputFactory.newInstance();
    try {
        XMLEventReader reader = inputFactory.createXMLEventReader(new FileInputStream(fileName));
        while (reader.hasNext()) {
            XMLEvent event = reader.nextEvent();
            if (event.isStartElement()) {
                StartElement startElement = event.asStartElement();
                if (startElement.getName().getLocalPart().equals("student")) {
                    student = new Student();
                    Attribute login = startElement.getAttributeByName(new QName("login"));
                    student.setLogin(login.getValue());
                    Attribute faculty = startElement.getAttributeByName(new QName("faculty"));
                    if (faculty != null) {
                        student.setFaculty(faculty.getValue());
                    }
                } else if (startElement.getName().getLocalPart().equals("telephone")) {
                    event = reader.nextEvent();
                    student.setTelephone(Integer.parseInt(event.asCharacters().getData()));
                } else if (startElement.getName().getLocalPart().equals("name")) {
                    event = reader.nextEvent();
                    student.setName(event.asCharacters().getData());
                } else if (event.isStartElement()) {
                    StartElement startElementAddr = event.asStartElement();
                    if (startElementAddr.getName().getLocalPart().equals("country")){
                        event = reader.nextEvent();
                        student.getAddress().setCountry(event.asCharacters().getData());
                    } else if (startElementAddr.getName().getLocalPart().equals("city")){
                        event = reader.nextEvent();
                        student.getAddress().setCity(event.asCharacters().getData());
                    } else if (startElementAddr.getName().getLocalPart().equals("street")){
                        event = reader.nextEvent();
                        student.getAddress().setStreet(event.asCharacters().getData());
                    }
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    if (event.isEndElement()) {
        EndElement endElement = event.asEndElement();
        if (endElement.getName().getLocalPart().equals("student")) {
            students.add(student);
        }
    }
}
```

```

    }
  }
}
} catch (FileNotFoundException | XMLStreamException e) {
    e.printStackTrace();
}
}
}

```

Обработку документов с помощью всех трех парсеров можно объединить в одно приложение с использованием шаблонов проектирования **Factory Method** и **Builder**. Для этого будет построена иерархия классов *builder*-ов во главе с абстрактным классом **AbstractStudentsBuilder** в виде:

```

/* # 17 # вершина иерархии builder-ов # AbstractStudentsBuilder.java */

public abstract class AbstractStudentsBuilder {
    // protected - it is often accessed from a subclass
    protected Set<Student> students;
    public AbstractStudentsBuilder() {
        students = new HashSet<Student>();
    }
    public AbstractStudentsBuilder(Set<Student> students) {
        this.students = students;
    }
    public Set<Student> getStudents() {
        return students;
    }
    public abstract void buildSetStudents(String filename);
}

```

Приведенные выше классы разбора XML-документов с помощью SAX, DOM, StAX следует адаптировать вследствие определения для них абстрактного класса и передачи ему атрибута **Set<Student> students** и метода **getStudents()**.

```

/* # 18 # реализации конкретных bulder-ов # StudentsSaxBuilder.java
# StudentsDomBuilder.java # StudentsStaxBuilder.java */

public class StudentsSaxBuilder extends AbstractStudentsBuilder {
    private StudentHandler handler;
    private XMLReader reader;
    public StudentsSaxBuilder() {
    }
    public StudentsSaxBuilder(Set<Student> students) {
        super(students);
    }
    @Override
    public void buildSetStudents(String fileName) {
    }
}

```

```

}
public class StudentsDomBuilder extends AbstractStudentsBuilder {
    private DocumentBuilder docBuilder;
    public StudentsDomBuilder() {
    }
    public StudentsDomBuilder(Set<Student> students) {
        super(students);
    }
    @Override
    public void buildSetStudents(String fileName) {
    }
}
public class StudentsStaxBuilder extends AbstractStudentsBuilder {
    private XMLInputFactory inputFactory;
    public StudentsStaxBuilder() {
    }
    public StudentsStaxBuilder(Set<Student> students) {
        super(students);
    }
    @Override
    public void buildSetStudents(String fileName) {
    }
}

```

Шаблон **Factory Method** предназначен для создания объектов, находящихся в иерархической зависимости. В данной ситуации класс, реализующий шаблон, будет производить экземпляры подклассов абстрактного класса **AbstractStudentsBuilder**. Принятие решения о конкретном типе будет производиться на основании передаваемого в *factory*-метод строкового значения с именем желаемого парсера.

```
/* # 19 # фабрика для создания конкретных builder-ов # StudentBuilderFactory.java */
```

```

package by.epam.learn.xml.parser;
public class StudentBuilderFactory {
    private enum TypeParser {
        SAX, STAX, DOM
    }
    private StudentBuilderFactory() {
    }
}

```

```
public static AbstractStudentsBuilder createStudentBuilder(String typeParser) {
    // insert parser name validation
    TypeParser type = TypeParser.valueOf(typeParser.toUpperCase());
    switch (type) {
        case DOM -> {
            return new StudentsDomBuilder();
        }
        case STAX -> {
            return new StudentsStaxBuilder();
        }
        case SAX -> {
            return new StudentsSaxBuilder();
        }
        default -> throw new EnumConstantNotPresentException(type.getDeclaringClass(), type.name());
    }
}
}
```

Запускать приложение из метода **main()** стало возможным с помощью кода:

```
/* # 20 # запуск приложения с выбором парсеров */

String type = "stax";
AbstractStudentsBuilder builder = StudentBuilderFactory.createStudentBuilder(type);
builder.buildSetStudents("data_xml/students.xml");
System.out.println(builder.getStudents());
```

Схема XSD

Схема XSD представляет собой руководство по созданию и валидации XML-документа. XSD-схема является XML-документом, и поэтому она отличается гибкостью при использовании в приложениях, при задании правил документа, а также для дальнейшего расширения новой функциональностью. Схема содержит 44 базовых типа и имеет поддержку пространств имен (*namespace*). С помощью схемы XSD можно также проверить документ на валидность.

Схема XSD первой строкой содержит XML-декларацию. Любая схема своим корневым элементом должна содержать элемент **schema**.

В схеме нужно описать все элементы: их тип, количество повторений, дочерние элементы. Сам элемент создается элементом **element**, который может включать следующие атрибуты: **name** — определяет имя элемента; **type** — указывает тип элемента; **ref** — ссылается на определение элемента, находящегося в другом месте; **minOccurs** и **maxOccurs** — количество повторений этого элемента (по умолчанию принимает значение 1), чтобы указать, что количество элементов не ограничено, в атрибуте **maxOccurs** необходимо задать **unbounded**.

Если стандартных типов не хватает для полноты описания элемента, то можно создать свой собственный тип элемента. Типы элементов делятся на

простые и сложные. Различия заключаются в том, что сложные типы могут содержать другие элементы, а простые — нет.

Для списка студентов XSD-схема **students.xsd** может выглядеть следующим образом:

21 # xsd-схема для документа students.xml # students.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.com/students"
  xmlns:tns="http://www.example.com/students" elementFormDefault="qualified">
  <element name="students">
    <complexType>
      <sequence>
        <element name="student" type="tns:Student" minOccurs="1" maxOccurs="100"/>
      </sequence>
    </complexType>
  </element>
  <complexType name="Student">
    <sequence>
      <element name="name" type="string"/>
      <element name="telephone" type="positiveInteger"/>
      <element name="address" type="tns:Address"/>
    </sequence>
    <attribute name="login" type="tns:Login" use="required"/>
    <attribute name="faculty" use="optional" default="mmf">
      <simpleType>
        <restriction base="string">
          <enumeration value="mmf"></enumeration>
          <enumeration value="famcs"></enumeration>
          <enumeration value="csan"></enumeration>
        </restriction>
      </simpleType>
    </attribute>
  </complexType>
  <simpleType name="Login">
    <restriction base="ID">
      <pattern value="([a-zA-Z])[a-zA-Z0-9]{7,19}"/>
    </restriction>
  </simpleType>
  <complexType name="Address">
    <sequence>
      <element name="country" type="string"/>
      <element name="city" type="string"/>
      <element name="street" type="string"/>
    </sequence>
  </complexType>
</schema>
```

Вопросы к главе 15

1. Какие существуют xml-парсеры в Java? Их классификация. Что такое DOM-модель?
2. Принципы работы парсера SAX. Пример получения объекта этого парсера? Как создать обработчик события для этого парсера, как его зарегистрировать?
3. Зачем SAX-парсеру обработчик ошибок?
4. Объяснить параметры, передаваемые методам `startElement()` и `endElement()`.
5. Объяснить принципы работы StAX-парсера. В чем различия между Cursor API и Iterator API парсера StAX? Привести пример получения парсера StAX.
6. Как происходит переход от одного элемента xml к другому в StAX-парсере? Как в StAX-парсере определяется, какой элемент xml-документа сейчас парсится?
7. Объяснить принцип работы DOM-парсера. Чем DOM принципиально отличается от SAX? Какие существуют реализации DOM-парсера? Привести пример получения экземпляра DOM-парсера. Перечислить основные DOM-интерфейсы в Java и основные принципы работы с ними.
8. Что такое JAXP, какие возможности дает применение JAXP. Перечислить классы, входящие в состав JAXP. Привести примеры получения SAX и DOM-парсеров с помощью JAXP. Что такое замена анализатора?
9. Что такое JAXB? Какие возможности дает использование JAXP? Как можно конвертировать объекты в/из xml-файла? В чем назначение утилиты `xjc`?

Задания к главе 15

1. Создать файл XML и соответствующую ему схему XSD.
2. При разработке XSD использовать простые и комплексные типы, перечисления, шаблоны и предельные значения.
3. Сгенерировать класс, соответствующий данному описанию.
4. Создать приложение для разбора XML-документа и инициализации коллекции объектов информацией из XML-файла. Для разбора использовать SAX, DOM и StAX-парсеры. Для сортировки объектов использовать интерфейс `Comparator`.
5. Произвести проверку XML-документа с привлечением XSD.
6. Определить метод, производящий преобразование разработанного XML-документа в документ, указанный в каждом задании.
1. **Оранжевая.**
Растения, содержащиеся в оранжевое, имеют следующие характеристики:
— Name — название растения;
— Soil — почва для посадки, которая может быть следующих типов: подзолистая, грунтовая, дерново-подзолистая;

- Origin — место происхождения растения;
 - Visual parameters (должно быть несколько) — внешние параметры: цвет стебля, цвет листьев, средний размер растения;
 - Growing tips (должно быть несколько) — предпочтительные условия произрастания: температура (в градусах), освещение (светолюбиво либо нет), полив (мл в неделю);
 - Multiplying — размножение: листьями, черенками либо семенами.
- Корневой элемент назвать Flower.

С помощью XSL преобразовать XML-файл в формат HTML, где отобразить растения по предпочитаемой температуре (по возрастанью).

2. Алмазный фонд.

Драгоценные и полудрагоценные камни, содержащиеся в павильоне, имеют следующие характеристики:

- Name — название камня;
- Preciousness — может быть драгоценным либо полудрагоценным;
- Origin — место добывания;
- Visual parameters (должно быть несколько) — могут быть: цвет (зеленый, красный, желтый и т.д.), прозрачность (измеряется в процентах 0–100%), способы огранки (количество граней 4–15);
- Value — вес камня (измеряется в каратах).

Корневой элемент назвать Gem.

С помощью XSL преобразовать XML-файл в формат XML, где корневым элементом будет место происхождения.

3. Тарифы мобильных компаний.

Тарифы мобильных компаний могут иметь следующую структуру:

- Name — название тарифа;
- Operator name — название сотового оператора, которому принадлежит тариф;
- Payroll — абонентская плата в месяц (0–n рублей);
- Call prices (должно быть несколько) — цены на звонки: внутри сети (0–n рублей в минуту), вне сети (0–n рублей в минуту), на стационарные телефоны (0–n рублей в минуту);
- SMS price — цена за смс (0–n рублей);
- Parameters (должно быть несколько) — наличие любимого номера (0–n), тарификация (12-секундная, поминутная), плата за подключение к тарифу (0–n рублей).

Корневой элемент назвать Tariff.

С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать тарифы по абонентской плате.

4. Лекарственные препараты.

Лекарственные препараты имеют следующие характеристики:

- Name — наименование препарата;

- Pharm — фирма-производитель;
- Group — группа препаратов, к которым относится лекарство (антибиотики, болеутоляющие, витамины и т.п.);
- Analogs (может быть несколько) — содержит наименование аналога;
- Versions — варианты исполнения (консистенция/вид: таблетки, капсулы, порошок, капли и т.п.). Для каждого варианта исполнения может быть несколько производителей лекарственных препаратов со следующими характеристиками:

Certificate — свидетельство о регистрации препарата (номер, даты выдачи/истечения действия, регистрирующая организация);

Package — упаковка (тип упаковки, количество в упаковке, цена за упаковку);

Dosage — дозировка препарата, периодичность приема;

Корневой элемент назвать Medicine.

С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать лекарства по цене.

5. Компьютеры.

Компьютерные комплектующие имеют следующие характеристики:

- Name — название комплектующего;
- Origin — страна производства;
- Price — цена (0–n рублей);
- Type (должно быть несколько) — периферийное либо нет, энергопотребление (ватт), наличие кулера (есть либо нет), группа комплектующих (устройства ввода-вывода, мультимедийные), порты (COM, USB, LPT);
- Critical — критично ли наличие комплектующего для работы компьютера.

Корневой элемент назвать Device.

С помощью XSL преобразовать XML-файл в формат XML, при выводе корневым элементом сделать Critical.

6. Электроинструменты.

Электроинструменты можно структурировать по следующей схеме:

- Model — название модели;
- Handy — одно- или двухручное;
- Origin — страна производства;
- TC (должно быть несколько) — технические характеристики: энергопотребление (низкое, среднее, высокое), производительность (в единицах в час), возможность автономного функционирования и т.д.;
- Material — материал изготовления.

Корневой элемент назвать PowerTools или Power.

С помощью XSL преобразовать XML-файл в формат XML, при выводе корневым элементом сделать страну производства.

7. Столовые приборы.

Столовые приборы можно структурировать по следующей схеме:

- Type — тип (нож, вилка, ложка и т.д.);
 - Origin — страна производства;
 - Visual (должно быть несколько) — визуальные характеристики: лезвие, зубец (длина лезвия, зубца [10–n см], ширина лезвия [10–n мм]), материал (лезвие [сталь, чугун, медь и т.д.]), рукоять (деревянная [если да, то указать тип дерева], пластик, металл);
 - Value — коллекционный либо нет.
- Корневой элемент назвать FlatWare.

С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать по длине лезвия, зубца, объему.

8. Самолеты.

Самолеты можно описать по следующей схеме:

- Model — название модели;
 - Origin — страна производства;
 - Chars (должно быть несколько) — характеристики, могут быть следующими: тип (пассажирский, грузовой, почтовый, пожарный, сельскохозяйственный), количество мест для экипажа, характеристики (грузоподъемность, число пассажиров), наличие радара;
 - Parameters — длина (в метрах), ширина (в метрах), высота (в метрах);
 - Price — цена (в талерах).
- Корневой элемент назвать Plane.

С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать по стоимости.

9. Конфеты.

- Name — название конфеты;
 - Energy — калорийность (ккал);
 - Type (должно быть несколько) — тип конфеты (карамель, ирис, шоколадная [с начинкой либо нет]);
 - Ingredients (должно быть несколько) — ингредиенты: вода, сахар (в мг), фруктоза (в мг), тип шоколада (для шоколадных), ванилин (в мг);
 - Value — пищевая ценность: белки (в г), жиры (в г) и углеводы (в г);
 - Production — предприятие-изготовитель.
- Корневой элемент назвать Candy.

С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать по месту изготовления.

10. Периодические издания.

- Title — название;
- Type — тип (газета, журнал, буклет);
- Monthly — периодичность выхода;
- Chars (должно быть несколько) — характеристики: цветность (да либо нет), объем (n страниц), глянецовое (да [только для журналов и буклетов] либо нет [для газет]), подписной индекс (только для газет и журналов).

Корневой элемент назвать Paper.

С помощью XSL преобразовать XML-файл в формат plain text, при выводе организовать подачу информации в удобном для прочтения виде.

11. Туристические путевки.

Туристические путевки, предлагаемые агентством, имеют следующие характеристики:

- Type — тип (выходного дня, экскурсионная, отдых, паломничество и т.д.);
- Country — страна, выбранная для путешествия;
- Number days/nights — количество дней и ночей;
- Transport — вид перевозки туристов (авиа, ж/д, авто, лайнер);
- Hotel characteristic (должно быть несколько) — количество звезд, включено ли питание и какое (НВ, ВВ, А1), какой номер (1-, 2-, 3-местные), есть ли телевизор, кондиционер и т.д.;
- Cost — стоимость путевки (сколько и что включено).

Корневой элемент назвать Tourist voucher.

С помощью XSL преобразовать XML-файл в формат HTML, с выводом информации в табличном виде.

12. Старые открытки.

- Thema — тема изображения (городской пейзаж, природа, люди, религия, спорт, архитектура...);
- Type — тип (поздравительная, рекламная, обычная). Была ли отправлена;
- Country — страна производства;
- Year — год издания;
- Author — имя автора/ов (если известен);
- Valuable — историческая, коллекционная или тематическая ценность.

Корневой элемент назвать Old Card.

С помощью XSL преобразовать XML-файл в формат PDF с выводом информации в отдельную страницу для каждой открытки.

13. Банковские вклады.

- Name — название банка;
- Country — страна регистрации;
- Type — тип вклада (до востребования, срочный, расчетный, накопительный, сберегательный, металлический);
- Depositor — имя вкладчика;
- Account id — номер счета;
- Amount on deposit — сумма вклада;
- Profitability — годовой процент;
- Time constraints — срок вклада.

Корневой элемент назвать Bank.

С помощью XSL преобразовать XML-файл в формат PDF с выводом информации в табличном виде.

ОТВЕТЫ НА ТЕСТОВЫЕ ВОПРОСЫ

Глава 1

1.1. — a, b; 1.2. — d; 1.3. — c; 1.4. — b, e; 1.5. — d; 1.6. — b.

Глава 2

2.1. — a; 2.2. — a; 2.3. — d; 2.4. — a; 2.5. — a; 2.6. — a, c; 2.7. — a, b, c; 2.8. — b.

Глава 3

3.1. — d; 3.2. — d; 3.3. — a, b, e; 3.4. — c, e; 3.5. — c; 3.6. — b; 3.7. — b;
3.8. — c; 3.9. — a, c, d, e.

Глава 4

4.1. — d; 4.2. — c; 4.3. — c; 4.4. — c; 4.5. — b, c, e; 4.6. — d.

Глава 5

5.1. — b; 5.2. — c; 5.3. — b; 5.4. — c; 5.5. — d.

Глава 6

6.1. — d; 6.2. — c; 6.3. — a, d; 6.4. — a, c, d.

Глава 7

7.1. — b; 7.2. — b; 7.3. — b, d; 7.4. — a; 7.5. — b.

Глава 8

8.1. — c; 8.2. — b; 8.3. — c, d; 8.4. — a; 8.5. — a.

Глава 9

9.1. — d; 9.2. — c; 9.3. — a, e; 9.4. — c, d, f.

Глава 10

10.1. — a; 10.2. — e; 10.3. — a; 10.4. — e; 10.5. — c.

Глава 11

11.1. — a; 11.2. — b; 11.3. — d; 11.4. — c; 11.5. — b; 11.6. — c.

Глава 12

12.1. — c; 12.2. — e; 12.3. — b; 12.4. — d; 12.5. — a.

Log4J2

Процесс разработки, отладки и эксплуатации программ сопровождается ведением журнала сообщений и ошибок, в котором отслеживаются различные системные и информационные события. Ведение такого журнала называется логированием. Логирование — это запись в «лог» данных о работе программы. В большинстве случаев в качестве места хранения «логов» используется текстовый файл. Для уменьшения объема вывода логирование в Java можно настроить так, чтобы в журнал или лог записывались только данные об ошибках или только о критических ошибках.

Потребность ведения логов возникла у программистов раньше, чем этот функционал был добавлен в язык Java. Популярной стала библиотека Log4J. Apache Log4J изначально обладает качественной архитектурой, вследствие чего быстро занял доминирующие позиции и применяется в большинстве промышленных приложений. В новой версии Log4J2 стал модульным и серьезно расширил возможности, в том числе и за счет усовершенствований, доступных в Logback, устраняя при этом некоторые из его недостатков.

К тому времени, как в Java появилась собственная библиотека для логирования, программисты уже пользовались Log4J2. Для решения проблемы совместимости версий был создан абстрагирующий фреймворк SLF4J (*Service Logging Facade For Java*). Абстрагирующим он называется потому, что классы SLF4J работают как все предыдущие фреймворки логирования: Log4J2, стандартный **java.util.logging** и другие. Если Log4J2 больше не нужен, надо перенастроить «обертку» SLF4J на использование другой библиотеки.

Любой регистратор событий состоит из трех элементов:

- собственно регистрирующего — **logger**;
- направляющего вывод — **appender**;
- форматирующего вывод — **layout**.

В итоге **logger** регистрирует и направляет вывод события в пункт назначения, определяемый направляющим элементом **appender**, в формате, заданном форматирующим элементом **layout**.

Log4J2

В современном практическом программировании представляет основной инструмент журналирования событий. Формирует журнал сообщений: отладочных,

информационных, системных, security сообщений об ошибках. Версия Log4J2 сохраняет все возможности предыдущей версии и предоставляет дополнения: формирование сообщений произвольной структуры, асинхронное логирование, то есть выполнение операций ввода-вывода в отдельном потоке, что серьезно увеличивает пропускную способность в многопоточной среде, поддержка лямбда-выражений, расширенная фильтрация. Конфигурирование логгера теперь можно выполнять с помощью файлов JSON, YAML, в дополнение к XML и файлам *properties*. Позволяет изменять конфигурацию во время исполнения приложения.

Log4J2 можно загрузить по адресу: <http://logging.apache.org/log4j/2.x/>. Перед использованием необходимо зарегистрировать для приложения минимальный набор библиотек:

log4j-api-2.[version].jar и **log4j-core-2.[version].jar**.

Logger

Основным элементом API регистрации событий и ошибок является регистратор **org.apache.logging.log4j.Logger**, который управляет регистрацией сообщений. Вывод регистратора может быть направлен на консоль, в файл, базу данных или любой поток вывода. Это компонент приложения, принимающий и выполняющий запросы на запись в регистрационный журнал. Apache Log4J2 поддерживает несколько способов конфигурации.

Каждый класс приложения может иметь свой собственный логгер или быть прикреплен к общему для всего приложения. Регистраторы образуют иерархию, как и пакеты Java. Каждый логгер имеет имя, описывающее иерархию, к которой он принадлежит. Разделителем является точка. Принцип полностью аналогичен формированию имени пакета в Java.

Регистратор может быть создан или получен с помощью одного из статических методов класса **org.apache.logging.log4j.LogManager** **getLogger()**, **getLogger(String name)** или **getLogger(Class name)**, где **name** — имя пакета или класса. На вершине иерархии находится корневой регистратор. У него нет имени. Корневой (Root) логгер обязателен в любой конфигурации и присутствует по умолчанию, даже если его конфигурация не задана вообще. Для него логирование настроено на вывод в консоль на уровне **ERROR**. Для доступа к корневому логгеру достаточно вызвать **LogManager.getRootLogger()**.

В итоге получить доступ к конфигурированным логгерам или к объявленным по умолчанию можно объявлением:

```
static Logger logger = LogManager.getLogger();
static Logger logger = LogManager.getLogger(NameOfClass.class);
static Logger rootLogger = LogManager.getRootLogger();
```

Приложения, использующие Log4j2, запрашивают **Logger** с определенным именем из **LogManager**. **LogManager** ищет соответствующий **LoggerContext**

и затем получает от него **Logger**. Если **Logger** должен быть создан, он будет связан с **LoggerConfig**, который содержит один из элементов:

- а) то же имя, что и **Logger**;
- б) имя родительского пакета;
- с) корневой **LoggerConfig**.

Объекты **LoggerConfig** создаются из деклараций **Logger**-ов в конфигурации. **LoggerConfig** связан с **Appenders**, которые фактически поставляют **LogEvents**.

Файл с конфигурацией, поиск которого происходит в *classpath*, должен иметь название **log4j2.xml** или **log4j2-test.xml** и располагаться в пакете по умолчанию.

Теперь логгер умеет принимать помимо **String**, **Object** и **Throwable** еще новые типы — **Marker**, **Message**, **MessageSupplier**, **Supplier** и **MapMessage**.

Чтобы вывести информацию о возникшем исключении, в перечисленные методы следует передать объект класса, производного от **Throwable**.

У каждого **LoggerConfig** есть уровень сообщения **LogLevel** по возрастанию (**ALL**, **TRACE**, **DEBUG**, **INFO**, **WARN**, **ERROR**, **FATAL**, **OFF**), который управляет выводом сообщений. Для вывода сообщений конкретного уровня используются методы **log()**, **trace()**, **debug()**, **info()**, **warn()**, **error()**, **fatal()**.

Для вывода сообщения необходимо, чтобы уровень выводимого сообщения был не ниже, чем уровень **LoggerConfig** (**ALL** < **TRACE** < **DEBUG** < **INFO** < **WARN** < **ERROR** < **FATAL** < **OFF**), т.е. если уровень регистратора **INFO**, то вызов **logger.debug("message")** не даст никакого эффекта, т.к. **DEBUG** < **INFO**. Уровень регистратора можно указать с помощью метода **setLevel(Level level)**, который принимает объект класса перечисления **Level**, содержащий одноименные константы для каждого уровня. Если уровень **LoggerConfig** не указывается, то применяется уровень, унаследованный от его родителя. И в этом заключается основное преимущество: можно вставлять в программный код вывод информации на различных уровнях (об ошибках — на уровне **ERROR**, о нормальном ходе выполнения — на уровне **INFO**, отладочную и трассировочную — на уровне **DEBUG** или **TRACE**), и далее в конфигурационном файле гибко регулировать, что именно будет выводиться.

Также вместо именованных методов логирования можно использовать только один метод **log(Level level, String s)**.

Appender

Вывод регистратора может быть направлен в различные места назначения: файл, консоль и т.д. Каждому из них соответствует класс, реализующий интерфейс **org.apache.logging.log4j.core.Appender**.

Если логгер — это та точка, откуда уходят сообщения в коде, то аппендер — это та точка, куда они приходят в конечном итоге. Например, файл или консоль. Список основных аппендеров, поддерживаемых Log4J2:

- консоль (**ConsoleAppender**);
- файлы (**FileAppender**, **RollingFileAppender**);
- база данных (**JdbcAppender**, **JpaAppender**, **NoSqlAppender**);
- темы (topics) JMS (**JmsAppender**);
- SMTP (**SmtAppender**);
- Сокет (**SocketAppender**);
- Syslog (**SyslogAppender**) поддерживает как TCP, так и UDP;
- асинхронный (**AsyncAppender**);
- любой **java.io.Writer** или **java.io.OutputStream**, соответственно **WriterAppender** или **OutputStreamAppender**.

Также объявлены аппендеры, способные оборачивать несколько других аппендеров. Существует возможность написать собственный класс аппендера и использовать его.

Логгеры связываются с аппендерами в соотношении «многие ко многим» — у одного логгера может быть несколько аппендеров, а к одному аппендеру может быть привязано несколько логгеров.

Уровень логирования наследуется (или устанавливается) независимо от аппендера. Иначе, если на корневом Root-логгере сконфигурирован вывод в консоль с уровнем **ERROR**, а на дочернем логгере — в файл с уровнем **INFO**, то вывод в дочерний логгер с уровнем **INFO** попадет и в файл, и в консоль.

Appender наследуется аддитивно из иерархии **LoggerConfig**. Например, если консольное приложение добавляется в корневой журнал, то все включенные запросы на логирование будут выводиться в консоль. Пусть файловый аппендер добавляется в **LoggerConfig**, тогда включенные запросы на логирование для заданного в **LoggerConfig** и всех его потомков записи будут вестись и в файле, и в консоли. Чтобы отказаться от наследования аппендеров, логгеру надо выставить свойство **additivity** в **false**, по умолчанию оно выставлено в **true**. Любой вывод, сделанный в регистраторе, будет направлен всем его предкам.

Layout

Вывод осуществляется в различных форматах. Каждый формат представлен классом, имплементирующим интерфейс **Layout**.

В пакете **org.apache.logging.log4j.core.layout** для конфигурирования формата вывода созданы следующие основные реализации:

- **JsonLayout** — формирует сообщения в виде JSON.
- **YamlLayout** — формирует сообщения в виде YAML в виде строк, сериализованных как байты.
- **SerializedLayout** — сериализует событие в байтовый массив.
- **HtmlLayout** — форматирует сообщения в виде HTML-страницы.
- **XmlLayout** — формирует сообщения в виде XML.
- **CSV Layout** — макет создает записи, разделенные запятыми (CSV).

- **GELF Layout** — выделяет события в расширенном формате журнала Graylog (GELF) 1.1.
- **SyslogLayout** — форматирует **LogEvent** как записи BSD Syslog, соответствующие формату, используемому в Log4j 1.2.
- **PatternLayout** — использует шаблонную строку для форматирования выводимого сообщения. Например:

```
<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{5} - %msg%n"/>
```

или

```
<Properties>
  <property name="pattern_1">%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{5} - %msg%n
  </property>
</Properties>
...
<PatternLayout pattern="${pattern_1}"/>
```

Данный форматтер принимает шаблон вывода лога со следующими атрибутами:

%d{DATE} — выводит дату-время. В скобках можно указать собственный произвольный формат вывода. Также применимы именованные шаблоны, а именно **DATE**, **ISO8601** и **ABSOLUTE**. Последний содержит формат **HH:mm:ss.SSS**, подходящий для логов с кратким сроком хранения. По умолчанию дата будет выведена в формате **ISO8601**;

%r — число миллисекунд, прошедших с начала работы приложения;

%t — выводит имя потока, выводящего сообщение, для однопоточного приложения будет выводить **main**;

%5level — выводит уровень лога (**ERROR**, **DEBUG**, **INFO** и пр.), где цифра указывает число выводимых символов, если символов меньше, то сообщение будет дополнено пробелами;

%logger{5} — категория с числом выдаваемых уровней. Категорией в общем случае будет имя класса с пакетом. Обычно это строка, где уровни разделены точками. По умолчанию без **{}** будет выводить полный путь к корню проекта. Верхний уровень при значении **1** будет выводить только имя класса;

%M — имя метода, в котором произошел вызов записи в лог;

%L — номер строки, в которой произошел вызов записи в лог;

%msg — собственно сообщение, передаваемое в лог;

%n — перевод строки.

Конфигурация

Перед использованием Log4J2 его необходимо сконфигурировать. Конфигурирование может быть выполнено в теле программы, создав реализацию

ConfigurationFactory и **Configuration**. Но на практике конфигурирование осуществляется с привлечением конфигурационных файлов одного из четырех видов: через XML, JSON, YAML, а начиная с версии 2.4 возвращена конфигурация через файл *properties*. Преимущество следует отдать способам, использующим конфигурационные файлы, как динамичным и легко изменяемым даже в процессе выполнения приложения. Если конфигурация отсутствует или некорректна, то Log4J2 будет работать только для вывода сообщений об ошибках.

Более удобным для понимания считается XML-конфигурирование и наиболее близкое к нему JSON. Конфигурация состоит из описания логгеров, аппендеров и фильтров.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Properties>
    <property name="name1">value1</property>
    <property name="name2" value="value2"/>
  </Properties>
  <filter ... />
  <Appenders>
    <appender ... >
      <filter ... />
    </appender>
    ...
  </Appenders>
  <Loggers>
    <Logger name="name1">
      <filter ... />
    </Logger>
    ...
    <Root level="level">
      <AppenderRef ref="name"/>
    </Root>
  </Loggers>
</Configuration>
```

Простейший конфигурационный файл **log4j2.xml** следует разместить в корне проекта в виде:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status = "ERROR">
  <Properties>
    <property name="pattern_file">
      %d{yyyy-MM-dd HH:mm:ss} [%t] %-5level %logger{5}- %msg%n
    </property>
  </Properties>
  <Appenders>
    <File name="TxtFile" fileName="logs/log.txt" bufferedIO="true">
      <PatternLayout pattern="${pattern_file}"/>
    </File>
  </Appenders>
```

```
<Loggers>
  <Root level = "debug">
    <AppenderRef ref = "TxtFile"/>
  </Root>
</Loggers>
</Configuration>
```

В итоге создан файловый аппендер с именем **TxtFile** для записи в файл **logs/log.txt** с поддержкой кодировки UTF-8 и упрощенным компоновщиком. Сконфигурирован корневой логгер уровня **debug**. Аппендер **TxtFile** добавляет данные в файл до бесконечности, поэтому файл может быть очень большого размера, что значительно усложняет его чтение при превышении разумных размеров. Но особые проблемы несет запись в большой текстовый файл, что реально замедляет работу системы. В чистом виде практически не используется. Файловый аппендер является основой для других, предлагая ключевые способы взаимодействия с файлами. Поддерживает следующие свойства: **append** — по умолчанию **true**, дописывать в существующий файл или каждый раз создавать новый, **bufferedIO** — по умолчанию **false**, буферизовать ли вывод в файл, **fileName** — имя файла, **bufferSize** — по умолчанию 8192, размер буфера, если включена буферизация.

Непрерывно работающее приложение иногда требует внесения изменений в настройки, что требует остановки и перезапуска приложения после их корректировки. В Log4J2 предусмотрена опция обновления конфигурации без перезапуска приложения. Автоматическую реконфигурацию можно включить атрибутом **monitorInterval**, что будет означать обновление конфигурации логгера через интервал времени в секундах.

```
<Configuration monitorInterval="86400"/>
```

В приведенном ниже примере производятся регистрация и вывод как обычных информационных сообщений о выполненных действиях, так и сообщений о возникающих ошибках при попытке вычисления факториала отрицательного числа.

```
/* # 1 # регистратор ошибок # DemoLogMain.java */

package by.epam.learn.log4j2.base;
import org.apache.logging.log4j.Level;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
public class DemoLogMain {
    static Logger logger = LogManager.getLogger();
    public static void main(final String... args) {
        try {
            factorial(9);
            factorial(-3);
        } catch (IllegalArgumentException e) {
            logger.log(Level.ERROR, "negative argument: ", e);
        }
    }
}
```

```

public static int factorial(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("argument " + n + " less than zero");
    }
    logger.log(Level.DEBUG, "Argument n is " + n);
    int result = 1;
    for (int i = n; i >= 1; i--) {
        result *= i;
    }
    logger.log(Level.INFO, "Result is " + result);
    return result;
}
}

```

Порции выводимых данных называются сообщениями. После запуска программы можно увидеть список сообщений в файле **log.txt**, как показано ниже.

```

2020-09-07 04:33:47 [main] DEBUG by.epam.learn.log4j2.base.
DemoLogRunner - Argument n is 9
2020-09-07 04:33:47 [main] INFO   by.epam.learn.log4j2.base.
DemoLogRunner - Result is 362880
2020-09-07 04:33:47 [main] ERROR by.epam.learn.log4j2.base.
DemoLogRunner - negative argument:
java.lang.IllegalArgumentException: argument -3 less than zero
    at by.epam.learn.log4j2.base.DemoLogRunner.factorial(DemoLogRunner.
java:19) ~[Learn1/:?]
    at by.epam.learn.log4j2.base.DemoLogRunner.main(DemoLogRunner.
java:11) [Learn1/:?]

```

Для вывода одновременно в текстовый файл и в консоль в более простом виде необходимо добавить информацию о новом аппендере в раздел **Appenders**:

```

<Console name="Console" target="SYSTEM_OUT">
    <PatternLayout pattern="%d{ABSOLUTE} %-5level %logger{1} - %msg%n"/>
</Console>

```

и добавить строку о появлении еще одного корневого логгера в **Root**:

```

<AppenderRef ref = "Console"/>

```

Таким образом, логгеров может быть несколько и использовать их можно не только для дублирования информации в разные точки сохранения, но и независимо друг от друга.

В консоль будет выведено следующее:

```

14:44:17,222 DEBUG DemoLogRunner - Argument n is 9
14:44:17,223 INFO   DemoLogRunner - Result is 362880
14:44:17,224 ERROR DemoLogRunner - negative argument:

```

```
java.lang.IllegalArgumentException: argument -3 less than zero
    at by.epam.learn.log4j2.base.DemoLogRunner.
factorial(DemoLogRunner.java:19) ~[Learn1/:?]
    at by.epam.learn.log4j2.base.DemoLogRunner.main(DemoLogRunner.
java:11) [Learn1/:?]
```

Можно создавать не только корневые логгеры. Такой логгер может обслуживать один класс или пакет.

```
<Loggers>
  <Logger name="by.epam.learn.log4j2.base.Action" level="debug" additivity="true">
    <AppenderRef ref = "Console"/>
  </Logger>
</Loggers>
```

Класс **org.apache.logging.log4j.message.MapMessage** позволяет формировать составные сообщения для передачи методу логирования, хранящие информацию по принципу карт отображения в виде пары ключ-значение.

```
/* # 2 # регистратор ошибок в виде Map # LogMapMessageMain.java */
```

```
package by.epam.learn.log4j2;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.message.MapMessage;
public class LogMapMessageMain {
    static Logger logger = LogManager.getLogger();
    public static void main(final String... args) {
        MapMessage mapMessage = new MapMessage().with("\nId", "LogMapMessageMain");
        try {
            mapMessage.with("\nattempt1", 7);
            factorial(7);
            mapMessage.with("\nattempt2", -3);
            factorial(-3);
        } catch (IllegalArgumentException e) {
            mapMessage.with("\nException", e);
        }
        logger.info(mapMessage);
    }
    public static int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("argument " + n + " less than zero");
        }
        int result = 1;
        for (int i = n; i >= 1; i--) {
            result *= i;
        }
        return result;
    }
}
```

Запись логов будет выглядеть:

```
2020-04-21 10:45:11 [main] INFO by.epam.learn.log4j2.LogMapMessageMain -
Exception="java.lang.IllegalArgumentException: argument -3 less than zero"
Id="LogMapMessageMain"
attempt1="7"
attempt2="-3"
```

Реализация интерфейса **org.apache.logging.log4j.message.Message** позволяет формировать сообщения для логгера в произвольном виде, задавая созданному классу необходимые собственные поля любого типа.

Логгеры могут оказать помощь в диагностике проблем, в разработке без необходимости отладки. Объявлена целая группа методов для этих целей: **traceEntry()**, **traceExit()**, **throwing()**, **catching()**. Метод **traceEntry()** должен быть помещен в начале методов, **traceExit()**, соответственно, в конце.

Метод **throwing()** применяется в точке, когда он генерирует исключение, которое, скорее всего, не будет обработано, например, **RuntimeException** или любой его подкласс. Это обеспечит надлежащую диагностику при необходимости. Сгенерированное событие регистрации будет иметь уровень **ERROR** и связанный **Marker** с именем **Throwing**.

Метод **catching()** может использоваться приложением в блоке **catch**, когда перехватывается исключение, которое не будет перебрасываться явно в вызывающий метод. Сгенерированное событие регистрации будет иметь уровень **ERROR** и связанный **Marker** с именем **Catching**. Ни один из этих методов сам исключение не генерирует, но сообщение, которое он формирует, выглядит практически как сообщение об исключении.

В методе **doAction()** класса **Action** производится преобразование строки в число, которое может привести к генерации исключения **NumberFormatException**, но в самом методе эта ситуация никак не обрабатывается. Для трассировки выставлен метод **throwing()**.

В методе **main()** явно генерируется **RuntimeException**. В **catch** вызывается метод **catching()** для трассировки этого события, также пишется лог уровня **ERROR** для визуального сравнения его сообщения с сообщением метода **catching()**.

```
/* # 3 # методы throwing(), catching() # Action.java # LogMain.java */
```

```
package by.epam.learn.log4j2;
import org.apache.logging.log4j.Level;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
class Action {
    static Logger logger = LogManager.getLogger(Action.class.getName());
    void doAction(String number) {
        logger.traceEntry("<<yes - entry>> " + number);
        logger.throwing(new NumberFormatException());
        Integer.parseInt(number);
    }
}
```

```

        logger.traceExit(">>ok - exit<<");
    }
}

public class LogMain {
    static Logger logger = LogManager.getLogger();
    public static void main(String[] args) {
        Action action = new Action();
        action.doAction("77");
        try {
            throw new RuntimeException("Exception runtime");
        } catch (RuntimeException e) {
            logger.log(Level.ERROR, "first runtime", e);
            logger.catching(e);
        }
    }
}

```

Конфигурационный файл для этого примера:

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration status = "info">
    <Properties>
        <property name= "LOG_DIR">logs</property>
        <property nam = "pattern_file">%d{yyyy-MM-dd HH:mm:ss} [%t] %-5level
            %logger{5} - %msg%n</property>
        <property name = "pattern_console">%relative %-5level %logger{1} - %msg%n</property>
    </Properties>
    <ThresholdFilter level="trace"/>
    <Appenders>
        <Console name = "Console" target="SYSTEM_OUT">
            <PatternLayout pattern = "${pattern_console}"/>
        </Console>
        <RollingFile name = "FileRolling" fileName="logs/log_roll.txt" append = "true"
            filePattern = "logs/${date:yyyy-MM-dd}/${d{yyyy-MM-dd_HH-mm}}_i.txt">
            <PatternLayout pattern = "${pattern_file}"/>
            <Policies>
                <SizeBasedTriggeringPolicy size = "15 KB"/>
            </Policies>
            <DefaultRolloverStrategy max = "10"/>
        </RollingFile>
    </Appenders>
    <Loggers>
        <Root level = "trace">
            <AppenderRef ref="FileRolling" />
            <AppenderRef ref="Console" />
        </Root>
    </Loggers>
</Configuration>

```

В консоль будет выведено:

```

1425 TRACE Action - Enter <<yes - entry>> 77
1428 ERROR Action - Throwing
java.lang.NumberFormatException: null
    at by.epam.learn.log4j2.Action.doAction(LogMain.java:13) [Learn1/?:?]
    at by.epam.learn.log4j2.LogMain.main(LogMain.java:24) [Learn1/?:?]
1437 TRACE Action - Exit with(>>ok - exit<<)
1446 ERROR LogMain - first runtime
java.lang.RuntimeException: Exception runtime
    at by.epam.learn.log4j2.LogMain.main(LogMain.java:28) [Learn1/?:?]
1446 ERROR LogMain - Catching
java.lang.RuntimeException: Exception runtime
    at by.epam.learn.log4j2.LogMain.main(LogMain.java:28) [Learn1/?:?]

```

RollingFileAppender

Аппендер **RollingFileAppender** позволяет создавать новый файл по достижении заданного предельного размера. «Создавать» — означает изменить имя текущего файла путем добавления к нему «_1» и открыть следующий. По достижении им максимального размера к имени текущего файла добавляется «_2», открывает-ся следующий и т.д. Файловая структура будет выглядеть примерно так:

```

log_2.txt
log_1.txt
log.txt

```

Одновременно с этим поддерживается обязательная замена файла логов по прохождении определенного временного промежутка: год, месяц, день, час и т.д.

```

<RollingFile name="File" fileName="logs/log.txt" append="true"
    filePattern = "logs/${date:yyyy-MM-dd}/${yyyy-MM-dd_HH-mm}_%i.txt">
  <PatternLayout pattern="%{pattern_0}"/>
  <Policies>
    <TimeBasedTriggeringPolicy/>
    <SizeBasedTriggeringPolicy size="25 KB"/>
  </Policies>
  <DefaultRolloverStrategy max="10"/>
</RollingFile>

```

В атрибут **filename** помещается имя актуального файла логов, в атрибуте **filePattern** задается шаблон записи архивных файлов:

```
filePattern = "logs/${date:yyyy-MM-dd}/${yyyy-MM-dd_HH-mm}_%i.txt"
```

Это означает, что архивные файлы будут размещаться в директории **logs**, в которой, в свою очередь, будут размещаться директории с указанием года месяца и дня **/\${date:yyyy-MM-dd}** записи логов, например:

2020-04-09

2020-11-20

2020-11-28

Имена файлов в этих директориях будут состояться из года, месяца, дня, часа и минуты создания `%d{yyyy-MM-dd_HH-mm}_%i.txt`, а также `%i` порядкового номера файла на данную минуту, например:

2020-09-26_17-08_1.txt

2020-09-26_17-08_2.txt

2020-09-26_17-08_3.txt

2020-09-26_17-23_1.txt

2020-09-26_17-42_1.txt

Определять имена директорий для файла логов и\или их архива можно с помощью **property**:

```
<property name="LOG_DIR">logs</property>
<property name="ARCHIVE_LOG_DIR">${LOG_DIR}/archive</property>
```

Такое использование повышает визуальное восприятие конфигурации:

```
<RollingFile name="File" fileName="${LOG_DIR}/archlog.txt" immediateFlush="false"
  append="false" filePattern="${ARCHIVE_LOG_DIR}/%d{dd-MM-yyyy}_%i.txt.zip">
```

Если у шаблона имени файла указано расширение **zip** или **gz**, то файл архивных логов будет автоматически архивирован.

Принципы создания архивных файлов для **RollingFileAppender** по времени и количеству можно определить достаточно гибкой.

Пусть задано следующее:

```
<TimeBasedTriggeringPolicy interval="5" modulate="true"/>
```

И определено:

```
filePattern="logs/${date:yyyy-MM}/app-%d{yyyy-MM-dd-HH}-%i.txt.gz"
```

В итоге может быть создано до 6 архивов в тот же день (1-6), которые хранятся в каталоге на основе текущего года и месяца, каждый архив упаковывается с помощью **zip** и будет производиться с интервалом в 5 часов.

Максимальный размер файла, устанавливаемый сохраняемым предыдущим файлом, задается в **Policies** свойством

```
<SizeBasedTriggeringPolicy size="25 KB"/>
```

а максимальный индекс архивного файла свойством

```
<DefaultRolloverStrategy max="10"/>
```

Если индекс 10 превышен — новый файл не появляется, а удаляется самый ранний с циклическим переименованием файлов и уменьшением номера на

единицу. Таким образом, всегда будет в наличии не больше определенного количества файлов, каждый из которых не больше заданного объема.

AsyncAppender

Аппендер **AsyncAppender** использует другие аппендеры для записи логов в отдельном потоке. Данный аппендер применяется в случаях, когда запись логов в файл критична по скорости или когда логи необходимо отправлять на какой-либо сторонний сервис. Для записи логов в консоль обычно не используется.

Конфигурация **AsyncAppender** может выглядеть следующим образом:

```
<Appenders>
  <File name="TxtFile" fileName="logs/log.txt" bufferedIO="true">
    <PatternLayout pattern="${pattern_file}"/>
  </File>
  <Async name="Async_File" bufferSize="100" blocking="true" ignoreExceptions="false">
    <AppenderRef ref="TxtFile"/>
    <LinkedTransferQueue/>
  </Async>
</Appenders>
<Loggers>
  <Root level="trace">
    <AppenderRef ref="Async_File"/>
  </Root>
</Loggers>
```

где **AppenderRef** — ссылка на аппендер, который будет вызван асинхронно; **name** — имя аппендера; **blocking** — по умолчанию **true**, в зависимости от значения **true** или **false** будет ждать или добавит сообщение в **error** аппендер, если внутренняя очередь полностью занята; **bufferSize** — задает максимальное число сообщений, которые могут быть помещены в очередь; **ignoreExceptions** — по умолчанию **true**, при значении **false** ошибки будут возвращаться пользователю.

Пусть в качестве примера в цикле 9999 раз вычисляется гипотенуза. Цикл повторится 4 раза. Засекается время начала и окончания вычислений и вычисляется продолжительность работы внутреннего цикла в миллисекундах для асинхронного логгера.

```
/* # 4 # асинхронный регистратор # AsyncMain.java */
```

```
package by.epam.learn.log4j2;
import org.apache.logging.log4j.*;
public class AsyncMain {
    static Logger logger = LogManager.getLogger();
    public static void main(String[] args) {
        for (int i = 0; i < 4; i++) {
            long start = System.currentTimeMillis();
```

```

    for (int num = 0; num < 9_999; num++) {
        logger.log(Level.INFO, "Hypot= " + Math.hypot(num, num));
    }
    long end = System.currentTimeMillis();
    System.out.println("loop " + i + ", time = " + (end - start) + " ms");
}
}
}

```

В консоль будет выведено примерно следующее:

```

loop 0, time = 147 ms
loop 1, time = 42 ms
loop 2, time = 40 ms
loop 3, time = 45 ms

```

Если заменить асинхронный логгер синхронным

```
<AppenderRef ref="File"/>
```

То вывод может быть следующим:

```

loop 0, time = 381 ms
loop 1, time = 145 ms
loop 2, time = 134 ms
loop 3, time = 123 ms

```

Фильтры

Необходимость записывать не все сообщения, а только удовлетворяющее заданным условиям, реализуется с использованием фильтров. В Log4j2 добавлены новые фильтры и модернизированы существующие. Некоторые из них:

BurstFilter предоставляет механизм для контроля скорости, с которой событие **LogEvent** обрабатывается, игнорируя события по достижении ее верхнего предела. Основные параметры: **rate** — среднее число разрешенных событий в секунду, **maxBurst** — максимальное число событий до начала работы фильтра, **level** — все сообщения ниже заданного уровня будут отфильтрованы после того, как **maxBurst** будет превышен, **onMatch**, **onMismatch** — действие, которое произойдет в случае совпадения\несовпадения фильтра.

```
<BurstFilter level="INFO" rate="2" maxBurst="10"/>
```

Чтобы проверить работоспособность, достаточно вставить эту строку с конфигурацией **BurstFilter** в конфигурацию **File** для асинхронного аппендера в предыдущем примере. В итоге **BurstFilter** будет игнорировать каждое событие с уровнем **INFO** и выше, если их более двух, но максимум 10 произошедших событий. В итоге без фильтра строк в файле **log.txt** будет 3996, а с **BurstFilter**

всего 10. Этот фильтр позволяет отсеять повторяющиеся или близкие по смыслу сообщения.

DynamicThresholdFilter позволяет фильтровать по уровню журнала на основе определенных параметров. Например, только для объектов, записанных в карту **ThreadContext**, можно включить ведение журнала отладки. Основные параметры: **keyValuePair** — пары ключ-значение, определяющие, какое значение уровня логирования сопоставить переданному ключу; **defaultThreshold** — определяет уровень фильтруемых сообщений, применяется, если **ThreadContext** содержит переданный ключ и его значение не совпадает ни с одним ключом из **keyValuePair**; **key** — название параметра в объекте **ThreadContext**.

```
<DynamicThresholdFilter key="customValueKey" defaultThreshold="ERROR"
                        onMatch="ACCEPT" onMismatch="DENY">
  <KeyValuePair key="customKey1" value="DEBUG"/>
</DynamicThresholdFilter>
```

ContextMapFilter фильтрует элементы данных, находящихся в текущем объекте **ThreadContext**.

TimeFilter используется для ограничения фильтра на определенную часть дня. Параметрами при конфигурировании фильтра являются **start** — начальное время, **end** — конечное время в формате **HH:mm:ss**, **timezone** — временная зона.

```
<TimeFilter start="00:01:00" end="07:00:00" onMatch="ACCEPT" onMismatch="DENY"/>
```

RegexFilter позволяет сравнивать форматированное или неформатированное сообщение с регулярным выражением **regex** для фильтрации.

```
<RegexFilter regex=".*card.*" onMatch="ACCEPT" onMismatch="DENY"/>
```

CompositeFilter позволяет создавать композиции из нескольких фильтров.

Можно также создать пользовательский фильтр. Для этого требуется создать подкласс абстрактного класса **AbstractFilter** из пакета **org.apache.logging.log4j.core.filter**.

Параметры **onMatch**, **onMismatch** присутствуют во всех фильтрах. Каждый фильтр возвращает одно из трех значений перечисления **Filter.Result**: **ACCEPT**, **DENY** или **NEUTRAL**.

ACCEPT — никакие другие фильтры не будут вызваны, и событие должно продолжаться.

DENY — событие будет проигнорировано, и управление возвращено вызывающему методу или классу.

NEUTRAL — событие передается другим фильтрам. Если других фильтров нет, оно будет обработано текущим.

Хотя событие может быть принято фильтром, событие по-прежнему может не регистрироваться в ситуации, когда событие принимается фильтром, но затем отклоняется всеми аппендерами.

Класс **org.apache.logging.log4j.ThreadContext** — ассоциируется с отдельным потоком и служит для хранения информации в **Map** методом **put(String key, String value)** и **Stack** методом **push(String message)**. Информация выводится логгером в зависимости от текущего наполнения объекта **ThreadContext**.

```
/* # 5 # динамический регистратор # ContextLogMain.java */

package by.epam.learn.log4j2;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.ThreadContext;
public class ContextLogMain {
    static Logger logger = LogManager.getLogger();
    public static void main(String[] args) {
        ThreadContext.put("key1", "value1");// add context map information
        logger.debug("debug message");
        ThreadContext.put("key2", "value2");
        ThreadContext.push("stackValue");// add context stack information
        logger.info("info message");
        ThreadContext.clearMap();// clear the map
        logger.debug("ThreadContext Map cleaned");
        ThreadContext.clearStack();
        logger.debug("ThreadContext Stack cleaned");
    }
}
```

Методы **clearMap()** и **clearStack()** очищают текущий **Map** и **Stack** сообщений. В результате будет выведено:

```
1752 [main] DEBUG {key1=value1} [] ContextLogMain - debug message
1758 [main] INFO {key1=value1, key2=value2} [stackValue]
ContextLogMain - info message
1758 [main] DEBUG {} [stackValue] ContextLogMain - ThreadContext Map
cleaned
1759 [main] DEBUG {} [] ContextLogMain - ThreadContext Stack cleaned
```

Чтобы сообщения типа **Map** выводились в **PatternLayout**, нужно добавить **%X**, а для сообщений **Stack** добавить **%x**. Конфигурационный файл для этого примера следующий:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="info">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern=
                "%relative [%t] %-5level %X %x %logger{1} - %msg%n"/>
        <Filters>
            <DynamicThresholdFilter key="customValueKey"
                defaultThreshold="ERROR" onMatch="ACCEPT" onMismatch="DENY">
```

```

        <KeyValuePair key="key1" value="INFO"/>
    </DynamicThresholdFilter>
</Filters>
</Console>
</Appenders>
<Loggers>
    <Root level="debug">
        <AppenderRef ref="Console"/>
    </Root>
</Loggers>
</Configuration>

```

В Log4J2 разрешено создавать пользовательские уровни логирования. Новый уровень может быть помещен в иерархию существующих уровней:

```

<CustomLevels>
    <CustomLevel name="CUSTOM_LEVEL" intLevel="350" />
</CustomLevels>

```

Тогда в коде можно к нему обратиться вызовом:

```
logger.log(Level.getLevel("CUSTOM_LEVEL"), "custom level message");
```

Созданный уровень логирования расположится между **WARN(300)** и **INFO(400)**, т.е. сообщения с уровнем **WARN** будут логироваться, а с уровнем **INFO** нет.

Вопросы к приложению 1

1. Зачем нужны логгеры?
2. Из каких частей состоит логгер?
3. Какие существуют уровни сообщений в логгере? Какова их иерархия?
4. Что такое **Appender**? Какие они бывают?
5. Что такое **Layout**? Какие они бывают?

TestNG

Оболочки модульного тестирования — это программные средства для разработки тестов, включающие: построение, выполнение тестов и создание отчетов. Первая оболочка модульного тестирования SUnit была создана Кентом Бекем в 1999 году для языка Smalltalk. Позднее Эрик Гамма создал JUnit. Сейчас существует множество оболочек для модульного тестирования, которые известны как программные средства семейства XUnit. TestNG — имеет много общего с NUnit и JUnit. Это достаточно широко используемая и расширенная версия оболочек модульного тестирования. TestNG создан на языке Java и используется для тестирования Java кода.

Модульное тестирование, или unit testing, — процесс проверки на корректность функционирования отдельных частей исходного кода программы путем запуска тестов в искусственной среде. Под частью кода в Java следует понимать исполняемый компонент. С помощью модульного тестирования обычно тестируют такие низкоуровневые элементы кода, как методы. TestNG позволяет вне исследуемого класса создавать тесты, при выполнении которых произойдет корректное завершение программы. Кроме основного положительного сценария, может выполняться проверка работоспособности системы в альтернативных сценариях, например, при генерации методом исключения как реакция на ошибочные исходные данные.

Оценивая каждую часть кода изолированно и подтверждая корректность ее работы, точно установить проблему значительно проще, чем если бы элемент был частью системы.

Технология позволяет и предлагает сделать более тесной связь между разработкой кода и его тестированием, а также предоставляет возможность проверить корректность работы класса, не прибегая к пробному выводу при отладке кода.

TestNG полностью построен на аннотациях. Для использования технологии необходимо загрузить библиотеку TestNG с сервера testng.org и включить архив **testng-[version].jar** в список библиотек приложения. При включении модульного тестирования в проект:

- тесты разрабатываются для нетривиальных методов системы;
- ошибки выявляются в процессе проектирования метода или класса;
- в первую очередь разрабатываются тесты на основной положительный сценарий;
- разработчику приходится больше уделять внимания альтернативным сценариям поведения, так как они являются источником ошибок, выявляемых на поздних стадиях разработки;

- разработчику приходится создавать более сфокусированные на своих обязанностях методы и классы, так как сложный код тестировать значительно труднее;
- снижается число новых ошибок при добавлении новой функциональности;
- устаревшие тесты можно игнорировать;
- тест отражает элементы технического задания, то есть некорректное завершение теста сообщает о нарушении технических требований заказчика;
- каждому техническому требованию соответствует тест;
- и как результат, получение работоспособного кода с наименьшими затратами.

Аннотация @Test

Аннотация **@Test** из пакета **org.testng.annotations** помечает метод как тестовый, что позволяет использовать возможности класса **org.testng.Assert** запускать его в режиме тестирования. Тестовый метод должен всегда объявляться как **public void**. Аннотация может использовать параметры:

- description** — содержит описание тестового метода;
- enabled** — определяет, включена аннотация либо игнорируется;
- expectedExceptions** — определяет список ожидаемых классов исключений;
- timeOut** — определяет время, превышение которого делает тест ошибочным;
- alwaysRun** — при значении **true** метод будет запускаться всегда вне зависимости от принадлежности к группе;
- priority** — задает приоритет теста;
- groups** — включение тестового метода в группу тестов;
- dataProvider** — имя для доступа к данным для теста;
- invocationCount** — число вызовов метода;
- dependsOnMethods, dependsOnGroups** — определяет зависимость теста от выполнения предыдущего метода или группы. Применение параметров будет рассмотрено ниже.

Пусть разработан некоторый класс по пересчету температуры из Цельсия в значение по Фаренгейту.

```
/* # 1 # преобразование температуры из Цельсия в Фаренгейт # Converter.java */
```

```
package by.epam.learn.action;
public class Converter {
    private static final int ABSOLUTE_ZERO = -273;
    public double convertCelsiusToFahrenheit(double celsius) {
        if (celsius < ABSOLUTE_ZERO) {
            throw new IllegalArgumentException("error data");
        }
        double fahrenheit = celsius * 1.8 + 32;
        return fahrenheit;
    }
    public boolean checkCelsius(double celsius) {
```

```
    return celsius >= ABSOLUTE_ZERO;
}
}
```

Необходимо протестировать методы этого класса. Метод, предназначенный для функционирования в качестве теста, маркируется аннотацией **@Test**. Пометить все методы класса как тестовые можно, применив данную аннотацию к классу. Простейший тест на метод будет выглядеть следующим образом.

```
/* # 2 # тест с аннотацией @Test # ConverterTest.java */
```

```
package test.epam.learn.action;
import by.epam.learn.action.Converter;
import org.testng.annotations.Test;
import static org.testng.Assert.*;
public class ConverterTest {
    Converter converter = new Converter();
    @Test()
    public void testConvertCelsiusToFahrenheit() {
        double actual = converter.convertCelsiusToFahrenheit(10.0);
        double expected = 50.;
        assertEquals(actual, expected, 0.001, " Test failed as...");
    }
}
```

Статический метод **assertEquals()** проверяет на равенство значений **expected** и **actual** с возможной погрешностью **delta**. При выполнении заданных условий сообщает об успешном завершении, в противном случае — об аварийном завершении теста. При аварийном завершении генерируется ошибка **java.lang.AssertionError**. Все методы класса **Assert** в качестве возвращаемого значения имеют тип **void**. Среди них можно выделить:

assertTrue(boolean condition)/assertFalse(boolean condition) — проверяет на истину/ложь значение **condition**;

assertSame(Object actual, Object expected) — проверяет, ссылаются ли ссылки на один и тот же объект;

assertNotSame(Object actual, Object unexpected) — проверяет, ссылаются ли ссылки на различные объекты;

assertNull(Object object)/assertNotNull(Object object) — проверяет, имеет или не имеет ссылка значение **null**;

assertThrows(params) и **expectThrows(params)** — тестируют генерацию исключений;

fail() — вызывает ошибку, используется для проверки, достигнута ли определенная часть кода или для заглушки, сообщающей, что тестовый метод пока не реализован.

Все перечисленные методы имеют перегруженную версию с параметром типа **String** в первой позиции, в который можно передавать сообщение, выводимое при аварийном завершении теста.

В тестовом методе не следует размещать более одного **assert**-метода.

Некоторые примеры использования:

```
@Test
public void checkSameTest() {
    String actual = "java" + 17;
    String expected = "ja" + "va" + 17;
    Assert.assertSame(actual, expected);
}

@Test
public void testCheckCelsiusFalse() {
    Converter converter = new Converter();
    boolean condition = converter.checkCelsius(-300);
    Assert.assertFalse(condition, "test false failed as...");
}
```

Фикстуры

Фикстура (*Fixture*) — состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Может быть представлено набором каких-либо объектов, состоянием базы данных, наличием определенных файлов, соединений и проч.

В TestNG аннотации позволяют исполнять одну и ту же фикстуру для каждого теста или всего один раз для всего класса, или не исполнять ее совсем. Предусмотрено восемь аннотаций фикстур — по две для фикстур уровня теста, группы, класса и метода.

- **@BeforeSuite** и **@AfterSuite** — запускается только один раз перед и после запуска всех тестов. Например, конфигурирование пула соединений, создание и заполнение тестовой БД.
- **@BeforeGroups** и **@AfterGroups** — запускается до вызова первого и после вызова последнего метода каждой группы.
- **@BeforeClass** и **@AfterClass** — запускается только один раз до и после запуска всех тестов класса.
- **@BeforeMethod** и **@AfterMethod** — запускается до и после каждого тестового метода.

Использование фикстур позволяет выделить этапы и точно определить моменты, например, создания/удаления объекта, инициализации необходимых ресурсов, очистки памяти и проч.

Фикстурой **@BeforeClass** задан момент создания объектов класса перед каждым тестом, что позволит всем тестам класса пользоваться одними и теми же объектами. По окончании работы всех методов объект может быть уничтожен.

```
/* # 3 # тест с фикстурами @Before и@After # ConverterTest.java */
```

```
package test.epam.learn.action;
import by.epam.learn.action.Converter;
import org.testng.annotations.AfterClass;
```

```
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;
import static org.testng.Assert.*;
public class ConverterTest {
    Converter converter;
    @BeforeClass
    public void setUp() {
        converter = new Converter();
    }
    @Test
    public void testConvertCelsiusToFahrenheit() {
        double actual = converter.convertCelsiusToFahrenheit(10.0);
        double expected = 50.;
        assertEquals(actual, expected, 0.001, " Test failed as...");
    }
    @Test()
    public void testCheckCelsius() {
        boolean condition = converter.checkCelsius(-45);
        assertTrue(condition);
    }
    @AfterClass
    public void tierDown(){
        converter = null;
    }
}
```

Тестирование исключительных ситуаций

При тестировании альтернативных сценариев работы метода часто требуется точно определить тип генерируемого методом исключения на основе переданных некорректных параметров. Если тест выдает исключение, то инфраструктура тестирования сообщает о корректном результате его исполнения.

Аннотацию **@Test** при необходимости тестирования генерации конкретно-го исключения следует использовать с параметром **expectedExceptions**. Параметр предназначен для задания типа исключения, которое данный тест должен генерировать в процессе своего выполнения.

```
/* # 4 # метод с атрибутом expectedExceptions # */
```

```
@Test(enabled = true, expectedExceptions = IllegalArgumentException.class,
        expectedExceptionsMessageRegExp = "error data")
public void testConvertCelsiusToFahrenheitException() {
    converter.convertCelsiusToFahrenheit(-300);
}
```

Метод **convertCelsiusToFahrenheit(double celsius)** генерирует исключение **IllegalArgumentException**, если число, переданное в метод, выходит за границы минимального отрицательного значения.

Тест завершится успешно лишь в том случае, если будет сгенерирована исключительная ситуация. Исключение **IllegalArgumentException** не нужно указывать в секции **throws** тестового метода, так как оно относится к непроверяемым. Если же метод может выбрасывать проверяемое исключение, то, по правилам работы с проверяемыми исключениями, его следует указывать в секции **throws** тестового метода.

Может появиться необходимость проверить не только возникновение исключительной ситуации, но и текст сообщения в экземпляре исключения. В этом случае лучше прибегнуть к обычному подходу без параметра **expected**.

```
/* # 5 # метод тестирования сообщения в исключении после его генерации # */
@Test(expectedExceptionsMessageRegExp = "error data")
public void testConvertCelsiusToFahrenheitExceptionMessage() {
    double celsius = -300;
    try {
        converter.convertCelsiusToFahrenheit(celsius);
        fail("Test for celsius " + Celsius + " should have thrown a IllegalArgumentException");
    } catch (IllegalArgumentException e) {
        assertEquals("error data", e.getMessage());
    }
}
```

В блоке **try**, если исключение не возникло, вызывается метод **fail()**, сигнализирующий о провале теста. В блоке **catch**, если исключительная ситуация произошла, проверяется на эквивалентность текстов сообщения об ошибке.

Ограничение по времени

В качестве параметра тестового сценария аннотации **@Test** может быть использовано значение лимита времени **timeOut**. Параметр **timeOut** определяет максимальный временной промежуток в миллисекундах, отводимый на исполнение теста. Если выделенное время истекло, а тест продолжает выполняться, то тест завершается неудачей.

```
/* # 6 # тест с ограничением по времени # */
@Test(timeOut = 1_000)
public void testTime() {
    IntStream.range(-273, 100_000_000)
        .boxed()
        .forEach(t -> converter.convertCelsiusToFahrenheit(t));
}
```

Примерно через 500 миллисекунд после запуска тест провалится, так как на выполнение метода уходит несколько больше времени. Если увеличить время до одной секунды, то тест пройдет успешно. Если заменить реализацию тела

теста со Stream API на обычный цикл, то времени на выполнение теста понадобится намного меньше:

```
@Test(timeout = 250)
public void testTime() {
    for (int t = -273; t < 100_000_000; t++) {
        converter.convertCelsiusToFahrenheit(t);
    }
}
```

Процесс формирования *stream* и выполнение им действий происходит более медленно, чем стандартные операции.

Игнорирование тестов

При контроле корректности функционирования бизнес-логики приложений игнорирование нереализованных, незавершенных или устаревших тестов может представлять определенную проблему. Параметр **enabled=false** заставляет инфраструктуру тестирования проигнорировать данный тестовый метод. Параметр **description** в этом случае предусматривает наличие комментария о причине игнорирования теста, полезного при следующем к нему обращении, например, в виде:

```
@Test(description = "test is ignored because ...", enabled = false)
```

Параметризованные тесты

TestNG позволяет создавать тест, который может работать с различными наборами значений параметров, что дает возможность разработать единый тестовый сценарий и запускать его несколько раз — по числу наборов параметров. Создание параметризованного теста с набором данных требует создания метода для формирования и поставки данных, помеченного аннотацией **@DataProvider**. Аннотации следует задать имя, по которому к нему будут обращаться методы-тесты. Тестовый метод для использования данных, предоставляемых **DataProvider**, должен в аннотации теста указать имя провайдера в параметре **dataProvider**.

```
/* # 7 # параметризованный тест # */
```

```
@DataProvider(name = "celsius_3")
public Object[][] createData() {
    return new Object[][]{{10, 50}, {0, 32}, {40, 104}};
}

@Test(dataProvider = "celsius_3")
public void testParamsConvert(double celsius, double expectedFahrenheit) {
    double actual = converter.convertCelsiusToFahrenheit(celsius);
    assertEquals(actual, expectedFahrenheit, 0.001);
}
```

Тест будет запущен по числу наборов данных, в данном случае — три раза, что и будет отражено в консоли.

Совершенно необязательно хранить данные в коде тестового класса. Параметры можно передавать через файл конфигурации **testing.xml**. К методу добавляется аннотация **@Parameters** со списком имен параметров.

```
@Test
@Parameters({"celsius", "expectedFahrenheit"})
public void testConvertWithParam(double celsius, double expectedFahrenheit) {
    double actual = converter.convertCelsiusToFahrenheit(celsius);
    assertEquals(actual, expectedFahrenheit, 0.001);
}
```

Эти же параметры становятся параметрами метода и получают свои значения из файла **testing.xml**.

```
<suite name="SuiteParam" verbose="1">
  <parameter name="celsius" value="10"/>
  <parameter name="expectedFahrenheit" value="50"/>
  <test name="TestParam">
    <classes>
      <class name="test.epam.learn.action.ConverterTest">
        <methods>
          <include name="testConvertWithParam"/>
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

Параметры будут передаваться всем классам, объявленным после тега **parameter**. Методы могут исключаться из теста класса тегом **<exclude>**.

Еще два способа хранения параметров для последующей передачи в метод представлены в виде класса со статическими методами, возвращающими массив массивов и итератор.

```
/* # 8 # параметризованный тест # # StaticDataProvider.java */
```

```
package test.epam.learn.action;
import org.testng.annotations.DataProvider;
import java.util.Iterator;
import java.util.List;
public class StaticDataProvider {
    @DataProvider(name = "dataProviderArray")
    public static Object[][] createData() {
        return new Object[][]{{0.0, 32.0}, {10, 50.0}, {20, 68}};
    }
    @DataProvider(name = "dataProviderIterator")
    public static Iterator<Object[]> createIterator() {
        Object[][] ob= new Object[][]{{5, 41}, {15, 59}, {30, 86}} ;
    }
}
```

```

        List<Object[]> list = List.of(ob);
        return list.iterator();
    }
}

```

Методы получают данные, указывая имя необходимого ему **dataProvider** и класс, где он расположен.

```

@Test(dataProvider = "dataProviderArray", dataProviderClass = StaticDataProvider.class)
public void testConvertCelsiusArray(double in, double expected){
    double actual = converter.convertCelsiusToFahrenheit(in);
    assertEquals(actual, expected);
}
@Test(dataProvider = "dataProviderIterator", dataProviderClass = StaticDataProvider.class)
public void testConvertCelsiusIterator(double in, double expected){
    double actual = converter.convertCelsiusToFahrenheit(in);
    assertEquals(actual, expected);
}

```

Группы тестов

Тесты можно объединять в группы и отправлять на выполнение группами. В группе могут находиться часть методов одного и часть методов нескольких классов. Один метод может находиться в нескольких группах. Группы тестов могут тестировать какую-либо часть приложения, либо относиться к какому-либо отдельному методу. Группы, в свою очередь, также могут быть частью других групп. Группы можно запускать одну за другой либо параллельно. Гибкость организации процесса тестирования повышается.

Тег **<groups>**, определяющий группу, может располагаться в тегах **<suite>** или **<test>**. Группа объявляется со своим именем, на которые и ссылается метод, включенный в эту группу.

```
/* # 9 # группы тестов # ConverterTestGroup.java*/
```

```

package test.epam.learn.action;
import by.epam.learn.action.Converter;
import org.testng.annotations.*;
import static org.testng.Assert.*;
public class ConverterTestGroup {
    Converter converter;
    @BeforeGroups(groups = {"calc", "check"})
    public void setUp() {
        converter = new Converter();
    }
    @Test(groups = {"calc"})
    @Parameters({"celsius", "expectedFahrenheit"})
    public void testConvertWithParam(double celsius, double expectedFahrenheit) {

```

```

    double actual = converter.convertCelsiusToFahrenheit(celsius);
    assertEquals(actual, expectedFahrenheit, 0.001);
}
@Test(groups = {"check", "calc"}, timeOut = 1000)
public void testTime() {
    for (int t = -273; t < 100_000_000; t++) {
        converter.convertCelsiusToFahrenheit(t);
    }
}
@Test(groups = {"calc"})
public void testConvertCelsiusToFahrenheit() {
    double actual = converter.convertCelsiusToFahrenheit(10.0);
    double expected = 551.;// failed value
    assertEquals(actual, expected, 0.001, " Test failed as...");
}
@Test(groups = {"check"}, expectedExceptions = IllegalArgumentException.class)
public void testConvertCelsiusToFahrenheitException() {
    converter.convertCelsiusToFahrenheit(-300);
}
@Test(groups = {"check"})
public void testCheckCelsius() {
    boolean condition = converter.checkCelsius(-45);
    assertTrue(condition);
}
}

```

тогда **testng.xml** выглядит следующим образом:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Suite1" verbose="1">
    <test name="CheckGroup_2">
        <groups>
            <run>
                <include name="check"/>
            </run>
        </groups>
        <classes>
            <class name="test.epam.learn.action.ConverterTestGroup"/>
        </classes>
    </test>
</suite>

```

Добавление еще одной группы на выполнение, это размещение тега **<include name="calc"/>** в теге **<run>**. Тестовые методы, включенные в обе группы, вызовутся только один раз.

Если выполнение метода зависит от создания объекта в методе **setUp()** класса, то этот метод следует помечать аннотацией **@BeforeGroups(groups = {"calc", "check"})**.

Приоритеты и зависимости

Когда тестируется некоторый последовательный рабочий процесс приложения: чтение данных, парсинг строк, создание объектов, выполнение бизнес-логики и пр., то необходимо организовать работу теста, чтобы методы вызывались в определенном порядке. Также можно сделать запуск методов зависимым одного от другого, то есть если не сработал некоторый метод в тесте, то последующий уже вызывать нет смысла по логике тестирования.

Пусть разработано приложение для конвертации данных из шкалы по Цельсию в шкалу по Фаренгейту, которое включает чтение данных из файла в виде строки:

```
/* # 10 # чтение строки из файла # DataReader.java*/

package by.epam.learn.action;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
public class DataReader {
    public String read(String filename) {
        String data = null;
        Path path = Paths.get(filename);
        if(Files.exists(path) && !Files.isDirectory(path) && Files.isReadable(path)) {
            try {
                StringBuilder builder = new StringBuilder();
                data = Files.lines(path)
                    .reduce((s1, s2) -> s1 + " " + s2)
                    .orElse("empty");
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return data;
    }
}
```

где файл **celsius.txt** представлен в виде:

```
0 10.1 20 a7b 36.6
45 -25 -300
7 5n.9 -888.1
-1 777
```

Разбор данных из строки, преобразование их к числовому виду и фильтрация некорректных данных, не являющихся числом:

```
/* # 11 # разбор строки # DataParser.java*/

package by.epam.learn.action;
import java.util.ArrayList;
import java.util.Arrays;
```

```
import java.util.List;
public class DataParser {
    public List<Double> parseData(String data) {
        List<Double> doubles = new ArrayList<>();
        Arrays.stream(data.split("\\s+"))
            .filter(s -> s.matches("-?\\d{1,6}\\\\.?\\d{0,2}?"))
            .forEach(d -> doubles.add(Double.valueOf(d)));
        return doubles;
    }
}
```

Класс **Converter** конвертации и проверки на граничные значения был объявлен выше в примере # 1.

Без применения TestNG самотестирование приложения могло быть проведено в классе:

```
/* # 12 # запуск приложения # CelsiusMain.java*/
```

```
package by.epam.learn.action;
import java.util.List;
public class CelsiusMain {
    public static void main(String[] args) {
        String data = new DataReader().read("data\\celsius.txt");
        System.out.println(data);
        List<Double> doubleList = new DataParser().parseData(data);
        System.out.println(doubleList);
        Converter converter = new Converter();
        doubleList.stream()
            .filter(c -> converter.checkCelsius(c))
            .forEach(c -> converter.convertCelsiusToFahrenheit(c));
    }
}
```

Чтобы организовать вывод результатов конвертации, достаточно после вызова метода **filter()** вставить вызов метода **peek()**

```
.peek(c -> System.out.println(c + " " + converter.convertCelsiusToFahrenheit(c)))
```

Разработанные тесты на основной положительный сценарий приложения объединены в группу **base_flow**, для которой файл **testing.xml** содержит:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="SuiteDependenPrior" verbose="1">
    <test name="Test Group_Main">
        <groups>
            <run>
                <include name="base_flow"/>
            </run>
        </groups>
        <classes>
```

```

        <class name="test.epam.learn.action.ConverterPriorTest"/>
        <class name="test.epam.learn.action.DataParserTest"/>
        <class name="test.epam.learn.action.DataReaderTest"/>
    </classes>
</test>
</suite>

```

Классы тестов:

```
/* # 13 # тест на чтение из файла # DataReaderTest.java*/
```

```

package test.epam.learn.action;
import by.epam.learn.action.DataReader;
import org.testng.annotations.Test;
import static org.testng.Assert.*;
public class DataReaderTest {
    static final String TEST_FILE = "data\\celsius.txt";
    @Test(groups = "base_flow")
    public void testRead() {
        DataReader reader = new DataReader();
        String actual = reader.read(TEST_FILE);
        String expected = "0 10.1 20 a7b 36.6 45 -25 -300 7 5n.9 -888.1 -1 777";
        assertEquals(actual, expected);
    }
}

```

```
/* # 14 # тест парсинга # DataParserTest.java*/
```

```

package test.epam.learn.action;
import by.epam.learn.action.DataParser;
import org.testng.Assert;
import org.testng.annotations.Test;
import java.util.List;
public class DataParserTest {
    @Test(groups = "base_flow")
    public void testParseData() {
        DataParser parser = new DataParser();
        List<Double> actual = parser.parseData("0 10.1 20 a7b 36.6 45 -25 -300 7 5n.9 -888.1 -1 777");
        List<Double> expected = List.of(0d, 10.1d, 20d, 36.6d, 45d, -25d, -300d, 7d, -888.1, -1d, 777d);
        Assert.assertEquals(actual, expected);
    }
}

```

```
/* # 15 # параметризованный и другие тесты # ConverterPriorTest.java*/
```

```

package test.epam.learn.action;
import by.epam.learn.action.Converter;
import org.testng.annotations.*;
import static org.testng.Assert.*;

```

```

public class ConverterPriorTest {
    Converter converter;
    @BeforeGroups(groups = "base_flow")
    public void setUp() {
        converter = new Converter();
    }
    @Test(dataProvider = "celsius_9", groups = "base_flow")
    public void testParamsConvert(double celsius, double expectedFahrenheit) {
        double actual = converter.convertCelsiusToFahrenheit(celsius);
        assertEquals(actual, expectedFahrenheit, 0.001);
    }
    @Test(timeOut = 250, groups = "base_flow")
    public void testTime() {
        for (int t = -273; t < 100_000_000; t++) {
            converter.convertCelsiusToFahrenheit(t);
        }
    }
    @Test(dataProvider = "celsius_check_10", groups = "base_flow")
    public void testCheckCelsius(double celsius, boolean flag) {
        boolean condition = converter.checkCelsius(celsius);
        assertTrue(condition == flag);
    }
    @DataProvider(name = "celsius_9")
    public Object[][] createData() {
        return new Object[][]{
            {0.0, 32.0},
            {10.1, 50.18},
            {20.0, 68.0},
            {36.6, 97.88},
            {45.0, 113.0},
            {-25.0, -13.0},
            {7.0, 44.6},
            {-1.0, 30.2},
            {777.0, 1430.6}};
    }
    @DataProvider(name = "celsius_check_10")
    public Object[][] createDataCheck() {
        return new Object[][]{
            {0.0, true},
            {10.1, true},
            {20.0, true},
            {36.6, true},
            {45.0, true},
            {-25.0, true},
            {7.0, true},
            {-888.1, false},
            {-1.0, true},
            {777.0, true}};
    }
}

```

Если запустить группу тестов **base_flow**, то методы не будут выполняться в порядке функционирования бизнес-логики приложения. Чтобы организовать необходимую последовательность вызова методов, следует расставить приоритеты, за которые отвечает параметр **priority** аннотации **@Test**. Самый высокий приоритет дает значение **0**, оно же и значение по умолчанию.

Чтобы поставить в зависимость выполнение следующего теста от результатов предыдущего, к аннотации **@Test** зависимого метода следует добавить параметр **dependsOnMethods** с перечислением имен методов, от которых зависит выполнение текущего теста.

В приведенном примере выполнение парсинга строки зависит от успешного чтения этой строки из файла. Чтобы определить порядок вызова и задать зависимость, следует объявить эти методы в виде:

```
@Test(priority = 3, timeOut = 250, groups = "base_flow")
public void testTime() { ...

@Test(priority = 4, dependsOnMethods = {"testTime"},
      dataProvider = "celsius_check_10", groups = "base_flow")
public void testCheckCelsius(double celsius, boolean flag) { ...
```

Аналогично можно расставить приоритеты и зависимости для всех остальных методов класса. Зависимости между методами разных классов можно прописать с помощью параметра **dependsOnGroups**, предварительно объединив необходимые методы каждого класса в отдельную группу. После чего и предоставляются зависимости между группами.

Само выполнение тестового метода не должно зависеть от данных, которые были получены предыдущим или каким-либо другим тестом. Каждый тест должен быть самодостаточным и способным выполняться без вызова других методов.

Все файлы и прочие источники данных не должны использоваться в рабочем приложении.

События

При выполнении больших или сложных тестов бывает недостаточно информации о процессе их выполнения. Возникает необходимость документирования результатов работы или выполнения определенных программных действий. Для реализации таких возможностей применяется модель прослушивания событий.

Чтобы реализовать прослушивание событий, следует имплементировать все методы интерфейса **org.testng.ITestListener** или создать подкласс класса **org.testng.TestListenerAdapter**. Класс, содержащий тесты и заинтересованный в обработке событий, необходимо пометить аннотацией **@Listeners** с указанием класса регистратора событий.

Если логирование в тестовых методах практически не используется, то при обработке событий это одно из лучших мест записи логов.

Класс **TestListenerAdapter** позволяет переопределять следующие **void**-методы:
onStart(ITestContext iTestContext) — вызывается при старте всего теста.
 В параметр **iTestContext** помещена информация о тесте, условиях и параметрах его старта;

onFinish(ITestContext iTestContext) — вызывается после завершения всего теста;

onTestStart(ITestResult testResult) — вызывается перед запуском каждого тестового метода. Параметр **testResult** содержит имя метода и класса, запустившего тест, статус, параметры теста, время старта и др.;

onTestSuccess(ITestResult testResult) — вызывается после успешного завершения тестового метода;

onTestSkipped(ITestResult testResult) — вызывается, если тестовый метод не был запущен;

onTestFailure(ITestResult testResult) — вызывается при неудачном завершении тестового метода.

Примерная реализация класса обработчика событий:

```
/* # 16 # реализация обработки событий #ConverterTestListener.java */

package test.epam.learn.action;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.testng.ITestContext;
import org.testng.ITestResult;
import org.testng.TestListenerAdapter;
public class ConvertTestListener extends TestListenerAdapter {
    static Logger logger = LogManager.getLogger();
    @Override
    public void onStart(ITestContext iTestContext) {
        logger.info("Test Started..." + iTestContext.getName() + " " + iTestContext.getStartDate());
    }
    @Override
    public void onFinish(ITestContext iTestContext) {
        logger.info("Test finished..." + iTestContext.getEndDate());
    }
    @Override
    public void onTestStart(ITestResult testResult) {
        logger.info("test " + testResult.getName() + " start");
        logger.info("Priority of this method is " + testResult.getMethod().getPriority());
    }
    @Override
    public void onTestSuccess(ITestResult testResult) {
        logger.info("test [" + testResult.getName() + "] Success");
        logger.info(testResult.getTestClass().getName());
    }
}
```

```

@Override
public void onTestFailure(ITestResult testResult) {
    logger.error(testResult.getStatus() + "Test [" + testResult.getName() + "] failed");
    logger.error(testResult.getThrowable());
    logger.error("Priority of this method is " + testResult.getMethod().getPriority());
}
@Override
public void onTestSkipped(ITestResult testResult) {
    logger.warn(testResult.getStatus() + "test [" + testResult.getName() + "] skipped");
}
}

```

Для демонстрации документирования обработки событий следует проаннотировать класс с тестами:

```

@Listeners(ConvertTestListener.class)
public class ConverterPriorTest {...

```

с несколько измененными параметрами так, чтобы были задействованы predefined методы класса **TestListenerAdapter**, а именно: успешный тест, неудачный тест. В итоге в консоль может быть выведено:

```

14:09:52 INFO - Test Started...Learn1 Thu May 14 14:09:52 MSK 2020
14:09:52 INFO - test testTime start
14:09:52 INFO - Priority of this method is 3

14:09:52 INFO - test [testTime] Success
14:09:52 INFO - test.epam.learn.action.ConverterPriorTest

14:09:52 INFO - test testCheckCelsius start
14:09:52 INFO - Priority of this method is 4

14:09:52 INFO - test [testCheckCelsius] Success
14:09:52 INFO - test.epam.learn.action.ConverterPriorTest

14:09:52 INFO - test testParamsConvert start
14:09:52 INFO - Priority of this method is 5

14:09:52 ERROR - Status 2: Test [testParamsConvert] failed
14:09:52 ERROR - java.lang.AssertionError: expected [31.2] but found [30.2]
14:09:52 ERROR - Priority of this method is 5

java.lang.AssertionError: expected [31.2] but found [30.2]
Expected :31.2
Actual   :30.2

14:09:52 INFO - test testParamsConvert start
14:09:52 INFO - Priority of this method is 5

```

14:09:52 INFO - test [testParamsConvert] Success

14:09:52 INFO - test.epam.learn.action.ConverterPriorTest

14:09:52 INFO - Test finished... Thu May 14 14:09:52 MSK 2020

Рекомендации по созданию тестов

Разработка тестов в современном программировании представляет собой часть технического задания на проект, потому не менее важна, чем основной код приложения.

Некоторые рекомендации по разработке тестов:

- В первую очередь тесты пишутся на методы системы, определяющие ее основную функциональность.
- Тестируются в основном **public**-методы. Методы **private** не тестируются никогда по двум причинам: его невозможно вызвать из теста, но такой метод вызывается **public**-методом, поэтому все же тестируется опосредованно через него.
- Простые **public**-методы, например, **equals()**, **hashCode()**, геттеры сеттеры, как правило, не тестируются.
- В коде тестового метода не следует использовать циклы и операторы условного перехода.
- Желательно в тестовом методе использовать только один **assert**-метод. Если на одном **assert**-методе тест провалится, то стоящие ниже просто не выполняются.
- Кроме теста на основной положительный сценарий работы метода, должны быть разработаны отдельные тесты на каждый из отрицательных сценариев метода.
- Тестовый метод должен пользоваться только своими исходными данными. Если в процессе прохождения теста данные были изменены, то к окончанию работы теста данные должны быть приведены в исходное состояние либо уничтожены. Тесты должны быть независимыми друг от друга. Результатом их работы не должны пользоваться другие методы. Иначе в другой конфигурации последовательного вызова тестов перед таким методом будет вызван другой метод, а текущий метод будет работать с другими исходными данными. Следствием чего станет искажение работы метода или провал теста.
- Хорошей практикой повышения качества тестов считается проектирование и создание теста до начала создания кода самого тестируемого метода. Такая технология называется TDD (*Test Driven Development*). Тест — это такой же код, как и *production*-код.
- Не использовать **Reflection API**.
- При работе с файлами использовать только относительные пути. Абсолютный путь при тестировании на другой платформе или просто на другом компьютере может проваливать тесты, даже если все файлы находятся на своих местах.

- Файлы с исходными данными и файлы результатов должны быть тестовыми и находиться в директориях, отличных от директорий, используемых проектом. Файлы результатов после отработки тестов желательно уничтожать.
- Для случая провала теста следует писать в **assert**-методы сообщения, ясно указывающие на причину ошибки.
- Если в процессе отработки теста может быть сгенерировано исключение, и это не предусмотрено положительным течением теста, то не следует гасить его пустым блоком **catch**. В этом случае тест должен быть немедленно провален размещением в теле **catch** метода **fail()**.
- Тестовые классы не могут находиться в одной директории с тестируемыми. Но имена пакетов, классов и методов должны четко указывать на тестируемый класс или метод. Например, классу **by.epam.learn.service.Action** в тестах может соответствовать **test.epam.learn.service.ActionTest**.
- Для обслуживания тестовых классов можно создавать обычные вспомогательные классы, например, для хранения констант, фабрики объектов и т.д.
- Хорошим тоном написания тестов представляется использование *mock*-объектов, настраиваемых на поведение реального объекта. Решение эффективно только в случае, если поведение реального объекта определено явным образом. Использование *mock*-объектов повышает взаимную изолированность сложных компонентов системы и, как следствие, увеличивает возможности системы для тестирования. Для Java применяются библиотеки *Mockito*, *EasyMock* и некоторые другие.

Вопросы к приложению 2

1. Что такое модульное тестирование?
2. Какие методы приложения не тестируются?
3. Что такое фикстура? Какие они бывают?
4. Как тестируется метод на генерацию исключения?
5. Как называется тест для нескольких наборов данных по очереди передаваемых в метод?
6. Что такое группы тестов?
7. Какие бывают зависимости между тестовыми методами?
8. Какие бывают зависимости между группами тестов?
9. Как создать тест на скорость работы метода?
10. Можно ли в одном тесте вызывать несколько *assert*-методов?



СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Гослинг, Дж. Язык программирования Java SE 8 / Дж. Гослинг, Б. Джой, Г. Стил, Г. Брача, А. Бакли. — 5-е изд. — М. : Вильямс, 2015. — 672 с.
2. Эккель, Б. Философия Java / Б. Эккель. — 4-е изд. — СПб. : Питер, 2019. — 1168 с.
3. Блох, Д. Java. Эффективное программирование / Д. Блох. — М. : ДМК, 2013. — 294 с.
4. Макконнелл, С. Совершенный код / С. Макконнелл. — 2-е изд. — СПб. : Русская редакция, 2019. — 896 с.
5. Хорстманн, К. С. Java. Библиотека профессионала. Т. 1. Основы / К. С. Хорстманн, Г. Корнелл. — 11-е изд. — Киев : Диалектика, 2020. — 864 с.
6. Хорстманн, К. С. Java. Библиотека профессионала. Т. 2. Расширенные средства программирования / К. С. Хорстманн, Г. О. Корнелл. — 11-е изд. — Киев : Диалектика, 2020. — 976 с.
7. Блинов, И. Н. Java 2. Практическое руководство / И. Н. Блинов, В. С. Романчик. — Минск : Универсал Пресс, 2005. — 400 с.
8. Блинов, И. Н. Java. Промышленное программирование / И. Н. Блинов, В. С. Романчик. — Минск : Универсал Пресс, 2007. — 704 с.
9. Блинов, И. Н. Java. Методы программирования / И. Н. Блинов, В. С. Романчик. — Минск : Четыре четверти, 2013. — 896 с.
10. Goetz, B. Java Concurrency in Practice / B. Goetz. — Addison Wesley, 2006. — 384 p.
11. Тейт, Б. Горький вкус Java / Б. Тейт. — СПб. : Питер, 2003. — 334 с.

Учебное издание

БЛИНОВ Игорь Николаевич
РОМАНЧИК Валерий Станиславович

Java

from EPAM
2-е издание



Лицензия и распространение

Данная книга распространяется под лицензией
«Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International»
[<https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>]

Подписано в печать 19.08.2021г
Формат 70х100/16. Усл. печ. л. 45,5. Уч.-изд. л. 44,38.
Тираж 500 экз. Заказ 21050917

Отпечатано в типографии ООО «Типография Лесник»
ул. Сабировская, 37, 197183, г.Санкт-Петербург
Тел: (+812) 649-73-09, www.l-print.spb.ru

ISBN 978-985-581-391-1



9 789855 813911