JAVA. СОСТОЯНИЕ ЯЗЫКА И ЕГО ПЕРСПЕКТИВЫ

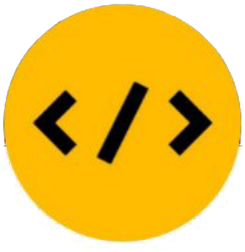
РАЗВИТИЕ ЯЗЫКА И ЕГО ВЕРСИЙ

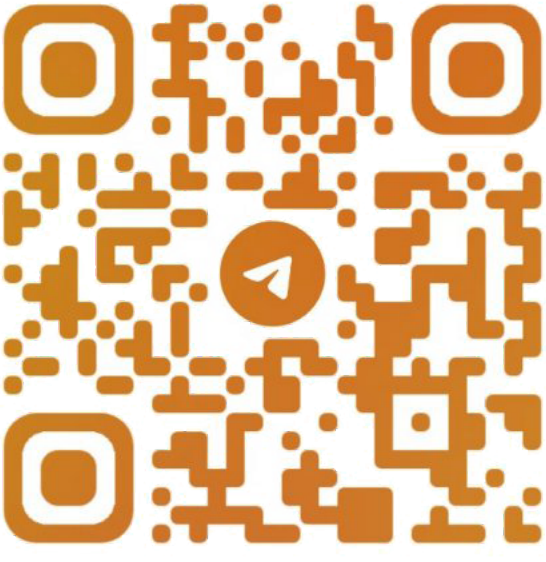
Федор Урванов

Федор Урванов

JAVA. СОСТОЯНИЕ ЯЗЫКА И ЕГО ПЕРСПЕКТИВЫ

2023





@CODELIBRARY_IT

Оглавление

ГЛАВА 1

Введение

1.1. Для кого эта книга

Когда-то давно я уже писал свой онлайн-учебник по Java 8. Его можно найти на сайте автора **https://urvanov.ru**. Однако с тех пор произошло довольно много кардинальных изменений, да и новые версии выпускаются теперь каждые полгода. Этот учебник представляет собой переработанный и дополненный вариант, описывающий новые возможности языка Java вплоть до версии Java 18 включительно. В нем рассказывается о самом языке Java и изменениях, которые произошли с ним за последние годы: какие новые возможности и в каких версиях появились, для чего они были внесены и как с ними взаимодействует старый код.

Предполагается, что читатель уже знаком с программированием на каком-нибудь другом языке, имеет общее представление об устройстве процессора, работе операционных систем, структурах данных и алгоритмах.

1.2. Что понадобится

Можно использовать компьютер с Windows, MacOS или любым дистрибутивом Linux.

Вам нужно установить JDK 17 или выше для разработки. Его необходимо скачать с официального сайта Oracle **https://java.oracle.com**.

В книге затронуты изменения и Java 18, но на момент написания книги IDE ее еще не поддерживали, поэтому примеры используют только версию 17. К счастью, в 18-й версии не так много изменений. Для набора листингов подойдет любой редактор текстовых файлов. Например, для Windows это может быть встроенный блокнот. Для Linux — kate, vim, nano или любой другой. Или можно напрямую набирать примеры в утилите для интерактивного выполнения выражений JShell, которая появилась в Java 9.

Но лучше сразу учиться использовать какую-нибудь IDE: Eclipse, IntelliJ IDEA или NetBeans. Сложно рекомендовать какую-то определенную IDE, но сейчас в большинстве организаций используется IntelliJ IDEA, поэтому имеет смысл сразу учиться использовать ее. Для изучения вполне подойдет бесплатная версия Community Edition, которую можно скачать с официального сайта разработчиков **https://jetbrains.com**, а на работе вам уже, скорее всего, выдадут лицензию.

Все примеры кода из этого учебника вместе с инструкциями по запуску в различных IDE можно скачать с сайта автора **https://urvanov.ru**, набрав название книги в строке поиска по сайту. Там же приведена ссылка на GitHub с теми же примерами (**https://github.com/urvanov-ru/java-in-dynamics-2022**).

1.3. Первая программа

Откройте свой текстовый редактор или выбранную IDE, создайте там файл HelloWorld.java и наберите следующий текст:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Это несложная программа, которая просто выводит в консоль "Hello, World!"

Нам нужно создать из нее платформонезависимый байт-код и запустить его. Для этого требуется запустить компилятор javac, который расположен в подкаталоге bin каталога установки Java. Если вы пользуетесь Linux и устанавливали компилятор Java через менеджер пакетов, то компилятор, скорее всего, уже добавлен в ваши пути поиска исполняемых файлов, поэтому вы можете сразу запустить:

```
javac HelloWorld.java
```

Эта команда создаст файл HelloWorld.class, который мы можем запустить на исполнение командой (обратите внимание, что мы НЕ указываем расширение файла):

```
java HelloWorld
```

В случае с Windows вам придется найти каталог, в который была установлена JDK 17, и запускать javac и java оттуда. Обычно это C:\Program Files\Java\jdk-17\bin\java.

Файл HelloWorld.class можно скопировать на другой компьютер с другой операционной системой. При этом его можно будет запустить без перекомпиляции. Нужно только наличие Java той же версии или выше. В этом и заключается платформонезависимость на уровне байт-кода (подробнее расписано в статье **https://urvanov.ru/2021/12/30/платформонезависимость-java/**).

На самом деле отдельно запускать javac и java вовсе не обязательно. Начиная с Java 11, программы, состоящие из одного файла, можно запускать одной командой (обратите внимание, что мы указываем расширение файла):

```
java HelloWorld.java
```

Или для Windows:

```
>"C:\Program Files\Java\jdk-17\bin\java" HelloWorld.java
Hello, World!
```

В самом коде программы нет ничего сложного. Мы просто объявляем класс с именем `HelloWorld`, а в нем — публичный статический метод `main`, принимающий в ка-

честве аргументов массив строк. Значения в этот массив можно передать, указав их при запуске после имени класса:

```
java HelloWorld arg0 arg1
```

Начиная с Java 9, можно использовать утилиту JShell, которая позволяет выполнять куски кода Java интерактивно. Она находится в подкаталоге bin каталога, где была установлена Java:

```
$ jshell
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro

jshell> System.out.println("Hello, World!");
Hello, World!
```

В этом варианте код выглядит гораздо короче, т. к. мы не объявляли ни класса, ни метода (хотя и могли это сделать). JShell как раз и предназначен для запуска таких небольших кусков кода, а не полноценных программ.

Как уже говорилось выше, обычно разработчики используют IntelliJ IDEA для написания кода. Для того чтобы вам легче было начать с ним работать, далее приведена пошаговая инструкция по созданию HelloWorld.

Запустите IntelliJ IDEA и в окне приветствия кликните на **New Project** (рис. 1.1) В списке типов проекта выберите **Java**, а в Project SDK укажите 17-ю версию. Если у вас ее нет, то ее нужно скачать с **https://java.oracle.com**.

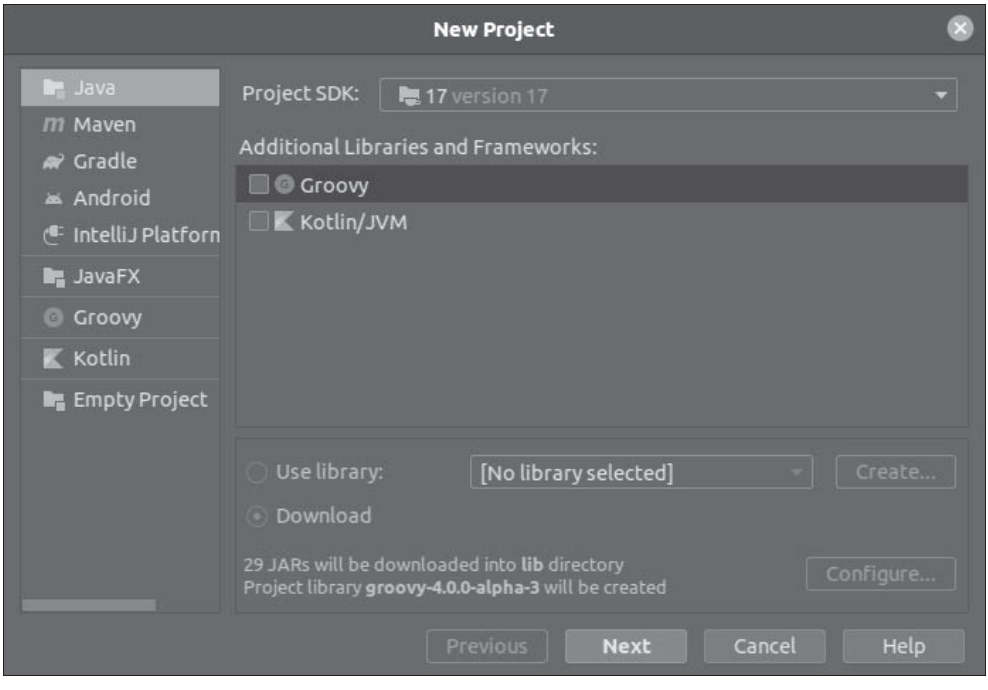


Рис. 1.1. Создание нового проекта в IntelliJ IDEA

Затем кликните **Next** несколько раз, укажите "HelloWorld" в качестве названия проекта и каталог, где он будет размещаться. В результате у вас получится пустой проект. В этом проекте нужно кликнуть на узле **src** правой кнопкой мышки и создать новый класс `HelloWorld`, как показано на рис. 1.2.

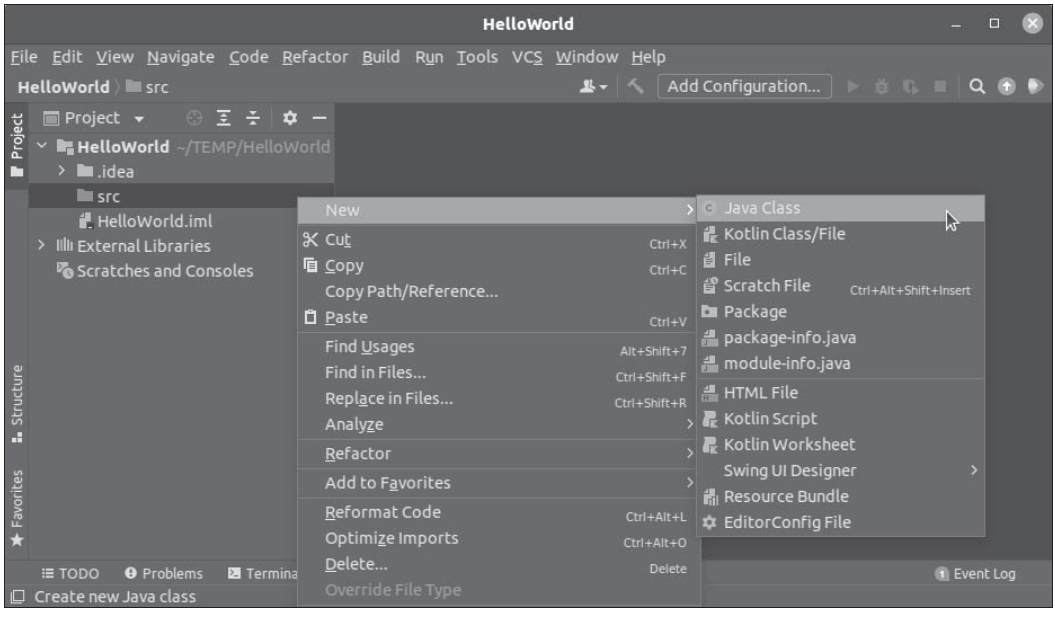


Рис. 1.2. Создание нового класса

В созданном классе наберите код, приведенный выше, а затем запустите его, кликнув на коде правой кнопкой мышки и выбрав пункт **Run 'HelloWorld.main()'** (рис. 1.3)

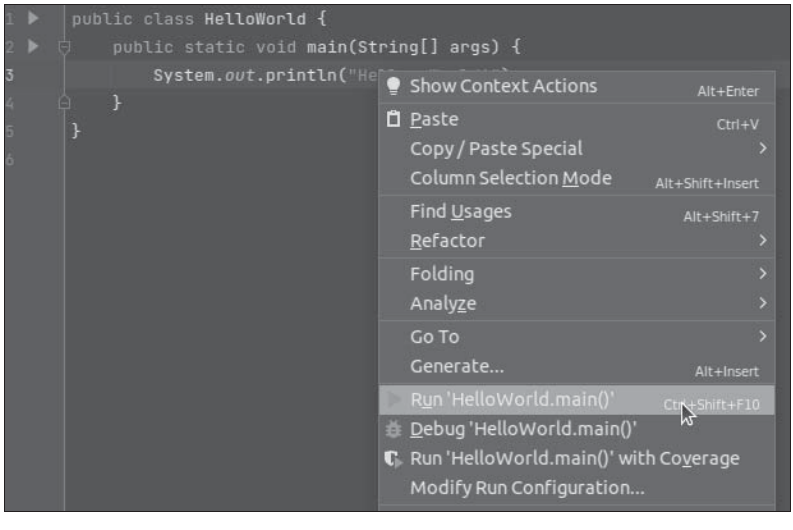


Рис. 1.3. Запуск программы HelloWorld из IntelliJ IDEA

IDEA сама скомпилирует класс и запустит его. В консоли выведется "Hello, World!", как и в предыдущих вариантах.

1.4. Задания

1. Напишите программу HelloWorld, используя только текстовый редактор и компилятор javac.
2. Напишите программу HelloWorld во всех трех IDE: Eclipse, NetBeans, IntelliJ IDEA.
3. Попробуйте изменить программу HelloWorld из этой статьи так, чтобы она выводила первый переданный параметр, например при вызове java HelloWorld Cat Dog она вывела в консоль "Cat".

ГЛАВА 2

Переменные

2.1. Типы переменных

Переменные в Java объявляются либо с указанием типа, либо, начиная с Java 10, можно использовать var:

```
int weight = 80;
double monsterHealth;
var ammoCount = 100;
```

Язык программирования Java является строго типизированным. Каждая переменная должна быть объявлена перед использованием либо с прямым указанием своего типа, либо с var. Если объявлять переменную с var, то ему обязательно нужно присвоить определенное значение, фактический тип вычисляется на основе типа выражения справа.

```
var monstersCount = 10 * 200 / 2;
var ammo = 250;
```

Сами переменные бывают четырех типов:

- Переменные экземпляров (нестатические свойства/поля) или Instance Variables (Non-Static Fields). Это те переменные, которые объявлены внутри класса без ключевого слова `static`. Их значения отличны для каждого экземпляра класса.

- Переменные класса (статические свойства/поля) или Class Variables (Static Fields). Это любое свойство класса, объявленное с ключевым словом `static`. Это свойство относится к самому классу, а не к его экземплярам. Оно существует всегда в единственном экземпляре.
- Локальные переменные или Local Variables. Локальные переменные — это переменные, объявленные внутри метода (внутри фигурных скобок метода). Они доступны только внутри метода, в котором они объявлены (внутри фигурных скобок). В них методы хранят информацию, необходимую только им, например промежуточные значения своих вычислений.
- Параметры или Parameters. Параметры — это переменные, которые принимают значение переданных аргументов метода. Мы уже видели пример параметров в описании метода `public static void main(String[] args)`. В этом описании метода `args` — параметр метода.

В последующих главах будет расписано подробнее про классы, методы, переменные экземпляров и переменные классов. Пока же вы можете посмотреть пример объявления каждого из перечисленных выше типов в примере `VariableType`, который находится внутри проекта с примерами кода для книги. Найти этот класс легче всего с помощью главного меню IntelliJ IDEA, выбрав пункты **Navigate ➢ Class...**

В появившемся окне достаточно ввести две заглавные буквы из составного имени класса, как показано на рис. 2.1.

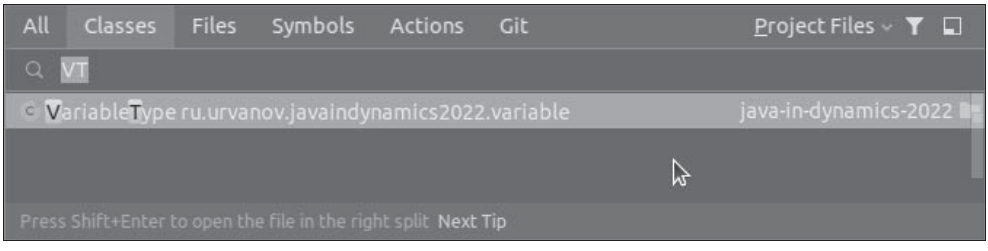


Рис. 2.1. Поиск класса по первым буквам слов его названия

VariableType.java

```
package ru.urvanov.javaindynamics2022.ch1;

public class VariableType {

    // переменные класса ( статические свойства / поля )
    static int variableTypesCounter;
    static String globalName;

    // Переменные экземпляров ( нестатические свойства / поля )
    // Объявлены без ключевого поля static
    // их значения различны для каждого из экземпляров
```

```
    private double health;
    private String instanceName;

    // args -  это параметры
    public static void main(String[] args) {

        // Объявление локальных переменных.
        // Доступны только внутри метода
        float value1;
        String str1;
        double sum;
    }
}
```

2.2. Соглашение об именовании переменных

В Java существуют общепринятые правила именования переменных, которые появились еще тогда, когда она принадлежала Sun, а не Oracle:

- Имена переменных чувствительны к регистру. Переменные `var1` и `VAR1` — это две разные переменные. Длина имени переменной не ограничена, оно может содержать любое количество символов Юникода и цифр (теоретически можно использовать русские буквы и иероглифы, и оно будет работать), может начинаться с буквы, знака доллара "$" или символа подчеркивания "_". Однако по соглашению об именовании Java имена переменных всегда должны начинаться со строчной буквы английского алфавита, но не с символа "$" или "_". Также по этому соглашению знак доллара не используется совсем. Некоторые утилиты могут генерировать имена переменных с символом доллара, но вы не должны его использовать.
- Последующие символы могут быть буквами, цифрами, знаком доллара и подчеркивания. Рекомендуется использовать полные английские слова при именовании переменных, а не сокращения.
- Если имя переменной содержит несколько слов, то первые буквы второго и последующего слова делаются прописными: `monsterBag`, `transferMoney`, `abstractProxyFactorySingletonBean`.
- Нельзя использовать зарезервированные (или ключевые) слова в качестве имен переменных.
- Если значение переменной никогда не меняется, например `static final int BUFFER_SIZE=1024;`, то по соглашению об именовании нужно каждую букву делать ЗАГЛАВНОЙ, а между словами использовать символ подчеркивания "_".

2.3. Типы данных

В Java используются восемь примитивных типов данных:

- `byte` — 8-битное знаковое число. Может хранить значения от –128 до +127 (включительно). Используется, например, в массивах, когда нужно сэкономить память.

- `short` — 16-битное знаковое число. Может хранить значения от –32 768 до +32 767 (включительно). Используется вместо `int`, когда необходимо сэкономить память.
- `int` — 32-битное знаковое число. Может хранить значения от -2^{31} до $2^{31}-1$. В Java можно использовать этот тип для хранения беззнакового целого числа от 0 до $2^{32}-1$, используя методы `compareUnsigned`, `divideUnsigned` и другие из класса `java.lang.Integer`.
- `long` — 64-битное знаковое число. Может хранить значения от -2^{63} до $2^{63}-1$. В Java можно использовать этот тип для хранения беззнакового целого числа от 0 до $2^{64}-1$, используя методы `compareUnsigned`, `divideUnsigned` и другие из класса java.lang.Long.
- `float` — 32-битное число с плавающей точкой одинарной точности согласно IEEE 754. Используется вместо `double`, когда нужно сэкономить память. Нельзя использовать float для хранения и обработки денежных значений. Для денежных значений нужно использовать `java.math.BigDecimal`.
- `double` — 64-битное число с плавающей точкой двойной точности согласно IEEE 754. Используется при необходимости хранить дробные значения. Нельзя использовать `double` для хранения и обработки денежных значений. Для денежных значений нужно использовать `java.math.BigDecimal`.
- `boolean` — логическое значение. Имеет только два возможных значения: `true` и `false`. Используется для флагов. Его размер точно не определен, несмотря на то, что он несет 1 бит полезной информации.
- `char` — 16-битный символ Юникода. Его минимальное значение '\u0000' или 0, а максимальное — '\uffff' (65 535 включительно)

2.4. Значения по умолчанию

Если переменная объявлена внутри метода (локальные переменные), то перед ее использованием ей обязательно нужно присвоить значение. При попытке использовать переменную без присваивания значения выйдет ошибка компиляции.

Для полей классов и полей экземпляров классов присваивать начальные значения необязательно, при этом для каждого из них будет присвоено значение по умолчанию в соответствии с таблицей (табл. 2.1).

Таблица 2.1. *Значения по умолчанию для различных типов*

Тип данных	Значение по умолчанию для полей
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`

Таблица 2.1 (окончание)

Тип данных	Значение по умолчанию для полей
`float`	0.0f
`double`	0.0d
`char`	'\u0000'
`String` (или любой объект)	`null`
`boolean`	`false`

Таблицу значений по умолчанию можно посмотреть в работающем примере в классе `DefaultValue`.

2.5. Литералы

Литерал — это запись в исходном коде, представляющая собой фиксированное значение.

Можно присвоить литерал переменной примитивного типа (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`):

Literal.java

```
boolean b = true;
char ch1 = 'f';
int i1 = 100000;
byte b1 = 100;
short sh1 = 10000;
```

Запустить пример можно в классе `Literal` прилагаемого проекта с примерами.

2.6. Целочисленные литералы

Целочисленный литерал имеет тип `int` в обычном случае и тип `long`, если заканчивается на символ "`l`" (строчная буква "эл" английского алфавита) или "`L`" (прописная буква "эл" английского алфавита). Рекомендуется использовать "`L`", т. к. "`l`" трудно отличить от единицы.

Значения типов `byte`, `short`, `int` и `long` могут быть созданы из литералов типа `int`. Значения типа `long`, превышающие `int`, могут быть созданы из литералов типа `long`.

Десятичные литеры могут быть записаны в четырех системах счисления:

- Десятичной. По основанию 10. Цифры от 0 до 9. Эту систему счисления мы используем в повседневной жизни.
- Шестнадцатеричной. По основанию 16. Цифры от 0 до 9 и буквы от A до F. Префикс 0x.

- Восьмеричной. По основанию 8. Цифры от 0 до 7. Префикс цифра 0.
- Двоичной. По основанию 2. Цифры от 0 до 1. Начиная с Java SE 7. Префикс 0b.

Пример:

IntegerLiteral.java

```
// Число 41 в десятичной системе счисления
int v1 = 41;
//  Число 41 в шестнадцатеричной системе счисления
int v2 = 0x29;
// Число 41 в восьмеричной системе счисления
int v3 = 051;
// Число 41 в двоичной системе счисления
int v4 = 0b101001;
```

Можно использовать символ подчеркивания между любыми двумя цифрами любой записи целочисленного литерала. Это может пригодиться, например, для отделения групп цифр:

IntegerLiteral.java

```
int v5 = 1_000_000; //1000000
int v6 = 0x3f_3f; //16191
int v7 = 0_5_1; //41
int v8 = 0b100_101_001;//297
```

Но целочисленный литерал не может заканчиваться или начинаться символом подчеркивания, и символ подчеркивания не может стоять сразу после 0x или 0b при использовании шестнадцатеричной и двоичной систем счисления.

2.7. Литералы с плавающей точкой

Литералы с плавающей точкой имеют тип float, если они оканчиваются на "`f`" или "`F`", и тип `double`, если они не имеют окончания или оканчиваются на "`d`" или "`D`".

Можно также использовать научную запись числа (**https://urvanov.ru/2021-12-08/научная-запись-числа/**) с помощью "`e`" или "`E`", что означает экспоненту (умножить на 10 в степени указанного числа).

FloatingPointLiteral.java

```
double d1 = 123.4;
// То же значение, что и у d1 (1.234 умножить 10 во второй степени)
double d2 = 1.234e2;
float f1  = 123.4f;
```

Существует также двоично-десятичная запись:

FloatingPointLiteral.java

```
// 1 (в шестнадцатеричной системе) умножить на 2 в степени 3.
// То есть 8.0
double d3 = 0x1p3;
// 0xF (в шестнадцатеричной системе) умножить на 2 в степени 3
// То есть 120.0
double d4 = 0xFp3;
```

Можно использовать символ подчеркивания для отделения групп в целой части, дробной части и экспоненте (после "`e`" или "`E`", "`p`" или "`P`"). Символ подчеркивания может стоять только между двумя цифрами!

Примеры:

FloatingPointLiteral.java

```
double d5 = 1_000.000_001;// 1000.000001
```

2.8. Символьные и строковые литералы

Строковые и символьные литералы могут содержать любой символ Юникода (UTF-16). Если ваш редактор текста не поддерживает эти символы, то вы можете вставлять их коды вида '\u00A9' (знак копирайта), где после \u стоит код символа Юникод в шестнадцатеричной системе счисления. Строки всегда заключаются в двойные кавычки, а символы — в одинарные.

Можно также использовать коды `\b` (backspace), `\t` (табуляция), `\n` (подача строки), `\f` (конец страницы, такое сейчас почти не используется), `\r` (возврат каретки), `\"` (двойная кавычка), `\'` (одинарная кавычка) и `\\` (косая черта).

SymbolAndStringLiteral.java

```
char ch1 = '\t'; // Символ табуляции
char ch2 = 'f'; // Буква f
char ch3 = '\u00A9'; // Знак копирайта
char ch4 = '©'; // В коде можно использовать все символы Юникода.
                // Так тоже сработает
System.out.println("символы: " + ch1 + ch2 + ch3 + ch4);
String myString1 = "Просто пример строки\nС переводами строк\n";
System.out.println("myString1 = " + myString1);
```

Для многострочных строковых литералов можно использовать блоки текста, которые окончательно вошли в Java 15 (**https://urvanov.ru/2021/02/15/что-нового-**

в-java-15/#textblocks). Для блоков текста используют три двойные машинописные кавычки:

SymbolAndStringLiteral.java

```
String myTextBlock1 = """
        Взгляни на милую, когда свое чело
        Она пред зеркалом цветами окружает,
        Играет локоном — и верное стекло
        Улыбку, хитрый взор и гордость отражает.
        """;
System.out.println("myTextBlock1 = ");
System.out.println(myTextBlock1);
```

С помощью блоков текста можно также создавать длинные строки, не помещающиеся в ширину экрана. Для этого используется обратная косая черта. В следующем куске кода создается переменная `textBlockWithOneLine`, содержащая одну строку без символов перевода строки:

SymbolAndStringLiteral.java

```
String textBlockWithOneLine = """
        Lorem ipsum dolor sit amet, consectetur adipiscing \
        elit, sed do eiusmod tempor incididunt ut labore \
        et dolore magna aliqua.\
        """;
```

Если в многострочном блоке текста нужно каждую строку дополнить пробелами, то после нужного количества пробелов указывается `\s`:

SymbolAndStringLiterla.java

```
String myTextBlockWithAdjust = """
        Взгляни на милую, когда свое чело          \s
        Она пред зеркалом цветами окружает,        \s
        Играет локоном — и верное стекло           \s
        Улыбку, хитрый взор и гордость отражает.   \s
        """;
```

Рекомендуется посмотреть примеры строковых литералов в классе `SymbolAndStringLiteral` прилагаемого к книге проекта с примерами.

2.9. Другие литералы

Существует еще литерал `null`, который можно присвоить переменным ссылочных типов данных.

Плюс еще есть литерал класса. Он формируется с помощью добавления `class` к имени типа, например `String.class`. Этот литерал ссылается на объект `java.lang.Class`, который представляет собой тип.

2.10. Массивы

Массив — это объект-контейнер, хранящий фиксированное количество элементов одинакового типа. Длина массива фиксируется после создания и не меняется на всем протяжении существования этого массива.

Нумерация элементов массива начинается с нуля. Первый элемент имеет индекс 0, второй — 1 и т. д. Последний элемент имеет индекс, на единицу меньший длины массива.

При объявлении массива сначала указывается тип данных, затем идут квадратные скобки и имя переменной. Квадратные скобки можно ставить не только после типа данных, но и после имени переменной, однако рекомендуется первый вариант.

Пример:

```
int[] arrayOfInt;
int arrayOfInt2[];
long[] arrayOfLong;
String[] arrayOfString;
double[] arrayOfDouble;
```

Для инициализации массива используется оператор `new` и имя типа данных с указанием размерности в квадратных скобках:

```
int a[] = new int[10]; // массив из 10 элементов типа int.
double[] arrayOfDouble = new double[12]; // массив из 12 элементов
                                         // типа double
```

После инициализации можно присваивать значения элементам массива. Доступ к элементам происходит по индексу, указанному в квадратных скобках после имени переменной:

```
a[1] = 12;
a[3] = 14;
arrayOfDouble[2] = 3.3;
System.out.println("The second element of a: " + a[1]); // 12
```

Для первоначальной инициализации массива можно использовать вот такую конструкцию:

```
// Инициализируем массив длиной 4 с элементами 10, 3, -4 и 67.
int[] myArray = {10, 3, -4, 67};

// или так
// Инициализируем массив длиной 4 с элементами 10, 3, -4 и 67.
int[] myArray = new int[] {10, 3, -4, 67};
```

Общий пример работы с массивами:

ArrayExample.java

```
public class ArrayExample {
    public static void main(String[] args) {
        int[] myArray = {10, 3, -4, 67};
        double [] arrayOfDouble = new double[3];
        myArray[0] = 12;
        arrayOfDouble[2] = 3.4;
        System.out.println("The first element of myArray is "
                + myArray[0]); // 12
        System.out.println("The second element of myArray is "
                + myArray[1]);  // 3
        System.out.println("The third element of arrayOfDouble is "
                + arrayOfDouble[2]); // 3.4
    }
}
```

Класс `java.util.Arrays` содержит множество полезных методов для работы с массивами: с помощью этих методов реализуются копирование, поиск, заполнение, сортировка. Имеет смысл ознакомиться с этими методами. Также может быть полезен метод `static void java.lang.System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`, который позволяет копировать элементы одного массива в другой.

2.11. Задания

1. Представьте, что вы работаете в организации, занимающейся разработкой приложения для службы доставки. Напишите простую программу, которая объявляет переменные всех примитивных типов: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`. Предположите, что переменные используются для хранения информации об одной посылке: адрес, вес, стоимость, размеры и т. д. Дайте переменным соответствующие названия.
2. Создайте простую программу, в которой объявите переменные для хранения информации об одной покупке в магазине: цена, количество, вес, калорийность и т. д. Постарайтесь использовать как можно больше типов данных.
3. Объявите по одной переменной каждого из примитивных типов: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`. Не присваивайте им начальных значений, пусть будут значения по умолчанию. Выведите их в консоль с помощью `System.out.println`.

ГЛАВА 3

Операции

Операция (*англ.* operator) — конструкция в языках программирования, аналогичная математическим операциям. Операции бывают арифметические, логические, битовые, строковые.

Иногда его путают с термином "оператор" из-за схожего написания с английским вариантом operator. На самом деле правильно переводится так:

операция — operator;

инструкция или оператор — statement.

Так получилось по историческим причинам.

Инструкция или оператор (*англ.* statement) — это одна команда языка программирования, наименьшая законченная часть.

3.1. Операция присваивания

Примеры операций присваивания находятся в классе `Assignment` прилагаемого к книге проекта с примерами.

Операция "=" позволяет присвоить значение переменной:

Assignment.java

```
int x1 = 3;
long l1 = 10_100_200_300L;
float f1 = 2.4f;
double weight = 81.34;
byte b1  = 100;
short sh1 = -10000;
char ch1 = 60000;
```

КОНСТАНТНЫЕ значения до `int` можно присвоить без приведения типа к переменным меньшего размера (например, `short` в `byte`), если значение помещается в эту переменную.

Вы можете присвоить переменной, имеющей больший тип, значение меньшего типа, например переменной типа `double` можно присвоить значение `int`, но не наоборот (но можно использовать приведение типа, если очень нужно).

Примеры:

Assignment.java

```
double d1 = 2; // Это можно
// int x2 = 2.3; // так нельзя. Будет ошибка компиляции.

byte b2 = 100; //Это можно, т. к. литерал 100 гарантированно
               // поместится в byte.
// byte b3 = 10000; //Нельзя. Ошибка компиляции.
int n = 100;
// byte b4 = n; // А вот так тоже нельзя, т. к.
                // переменная n имеет тип int.
```

Операция присваивания возвращает значение, которое присвоила, поэтому можно присваивать значение сразу нескольким переменным по цепочке:

Assignment.java

```
int x3;
int y1;
int z1 = x3 = y1 = 10; // y1, x3 и z1 будет присвоено 10.
```

3.2. Преобразование примитивных типов

Есть два типа преобразования примитивных типов:

- расширяющее преобразование (Widening Primitive Conversion);
- сужающее преобразование (Narrowing Primitive Conversion).

Их мы подробно рассмотрим в следующих разделах.

3.3. Расширяющее преобразование примитивов

Следующие преобразования называются расширяющими преобразованиями примитивов:

- `byte` ➢ `short, int, long, float, double`
- `short` ➢ `int, long, float, double`
- `char` ➢ `int, long, float, double`
- `int` ➢ `long, float, double`
- `long` ➢ `float, double`
- `float` ➢ `double`

Расширяющее преобразование не приводит к потере информации в следующих случаях:

- из целого типа в другой целый тип;
- из `byte`, `short`, `char` в тип с плавающей точкой;
- из `int` в `double`;
- из `float` в `double` (начиная с Java 17 либо со `strictfp` до Java 17).

До Java версии 17 расширяющее преобразование из `float` в `double` в обычном режиме (без `strictfp`) могло привести к потере точности, но теперь все операции с вещественными числами всегда происходят в режиме `strictfp`, как это было до введения `strictfp` в Java 1.2, поэтому преобразования из `float` в `double` происходят без потери точности, если вы используете Java 17.

Расширяющее преобразование `int` во `float`, или из `long` во `float`, или из `long` в `double` может привести к потере точности, т. е. результат может потерять несколько наименее значимых бит информации, что приведет к получению округленного значения.

Примеры расширяющего преобразования примитивов:

WideningPrimitiveConversion.java

```
byte b1 = 100;
short sh1 = b1;  // Расширяющее преобразование byte->short
char ch1 = 100;
int i1 = sh1;    // Расширяющее преобразование short->int
int i2 = ch1;    // Расширяющее преобразование char->int
long l1 = i1;    // Расширяющее преобразование int->long
float f1 = l1;   // Расширяющее преобразование long->float
double d1 = f1;  // Расширяющее преобразование float->double
```

Несмотря на возможность потери точности, расширяющее преобразование примитивов никогда не приводит к ошибкам во время выполнения.

3.4. Сужающее преобразование примитивов

Следующие преобразования называются сужающими преобразованиями примитивов:

- `short` ➢ `byte, char`
- `char` ➢ `byte, short`
- `int` ➢ `byte, short, char`
- `long` ➢ `byte, short, char, int`
- `float` ➢ `byte, short, char, int, long`
- `double` ➢ `byte, short, char, int, long, float`

Сужающее преобразование примитивов может привести к потере точности и даже к получению совсем другого числа из-за выхода за границу размерности типа.

Преобразование `double` во `float` может привести к потере точности и получению значения `-0.0f` или `+0.0f` вместо очень маленького значения `double`, а также к `-Infinity` и `+Infinity` вместо очень большого значения `double`. `NaN` из `double` преобразуется в `NaN` из `float`. `Infinity` в `double` преобразуется в `Infinity` в `float` того же знака.

При целочисленном сужающем преобразовании примитивов значения старших бит просто теряются, поэтому результат операции может запросто оказаться даже другого знака, в отличие от исходного числа.

Несмотря на возможность потери точности и даже получения совсем другого числа сужающее преобразование никогда не приводит к ошибке во время выполнения.

При сужающем преобразовании нужно явно приводить к необходимому типу, указав его в скобках, иначе возникнет ошибка компиляции:

NarrowingPrimitiveConversion.java

```
double d1 = 2.3;
double dPositiveInfinity = Double.POSITIVE_INFINITY;
double dNegativeInfinity = Double.NEGATIVE_INFINITY;
double dPlusZero = +0.0;
double dMinusZero = -0.0;
double dNaN = Double.NaN;

float fPositiveInfinity = (float) dPositiveInfinity; // +Infinity
                                  // сужающее преобразование
float fNegativeInfinity = (float) dNegativeInfinity; // -Infinity
                                  // сужающее преобразование
float fPlusZero = (float) dPlusZero; //0.0f сужающее преобразование
float fMinusZero = (float) dMinusZero; //-0.0f сужающее преобразование
float fNaN = (float) dNaN; // NaN сужающее преобразование
float f1 = (float) d1;             // сужающее преобразование
int n = (int) d1; // 2             // сужающее преобразование.
                                  // Дробная часть отбрасывается

System.out.println(fPositiveInfinity);
System.out.println(fNegativeInfinity);
System.out.println(fPlusZero);
System.out.println(fMinusZero);
System.out.println(fNaN);
System.out.println(f1);
System.out.println(n);
```

3.5. Арифметические операции

Арифметические операции позволяют выполнять сложение (операция "+"), вычитание (операция "-"), умножение (операция "*"), деление (операция "/") и взятие остатка (операция "%"). Эти операции имеют такие же приоритеты, что и в обыч-

ной математике, которую изучают в школе, т. е. умножение и деление выполняются перед сложением и вычитанием.

Arithmetic.java

```
double x1 = 1.1 + 2.3;   // 3.4
double x2 = 1.1 - 0.1;   // 1.0
double x3 = 1.1 * 2 + 1; // 3.2
double x4 = 6 / 2.0;   // 3.0
int x5 = 12 + 3; // 15
int x6 = 13 % 5; // 3
```

При выполнении арифметических операций операнды всегда преобразуются как минимум в `int` (например, при умножении двух переменных типа `byte` оба значения сначала преобразуются в `int`, и результат выражения будет `int`).

При выполнении арифметической операции над операндами разных типов результат операции будет иметь наибольший тип, что можно описать следующими правилами:

1. Если один из операндов имеет тип `double`, то результат выражения имеет тип `double`, иначе см. пункт 2.
2. Если один из операндов имеет тип `float`, то результат выражения имеет тип `float`, иначе см. пункт 3.
3. Если один из операндов имеет тип `long`, то результат выражения имеет тип `long`, иначе результат выражения имеет тип `int`.

(например, при сложении `int` и `long` результат будет иметь тип `long`, а при сложении `long` и `float` результат будет иметь тип `float`, а при сложении `float` и `double` результат будет иметь тип `double`).

Если результат операции с целочисленными данными выходит за диапазон, то старшие биты отбрасываются и результирующее значение будет совершенно неверным. При попытке деления на 0 возникает исключение `java.lang.ArithmeticException / zero`.

При выполнении операций с плавающей точкой при выходе за верхнюю или нижнюю границу диапазона получается `+Infinity` (`Double.POSITIVE_INFINITY` и `Float.POSITIVE_INFINITY`) и `-Infinity` (`Double.NEGATIVE_INFINITY` и `Float.NEGATIVE_INFINITY`) соответственно, а при получении слишком маленького числа, которое не может быть нормально сохранено в этом типе данных, получается -0.0 или +0.0.

При выполнении операций с плавающей точкой результат `NaN` (`Double.NaN` или `Float.NaN`) получается в следующих случаях:

- Когда один из операндов `NaN`.
- В неопределенных результатах:
  - Деления 0/0, ∞/∞, ∞/−∞, −∞/∞, −∞/−∞;
  - Умножения 0×∞ и 0×−∞;

- Степень 1^{∞};
- Сложения $\infty + (-\infty)$, $(-\infty) + \infty$ и эквивалентные вычитания.

❒ Операции с комплексными результатами:

- Квадратный корень из отрицательного числа;
- Логарифм отрицательного числа;
- Тангенс 90 градусов и ему подобных (или $\pi/2$ радиан);
- Обратный синус и косинус от числа меньше −1 и больше +1.

3.6. Унарные операции

Унарными называются операции, которые включают только один операнд. Унарные операции бывают префиксные и постфиксные.

Постфиксные унарные операции ставятся после операнда:

❒ Инкремент (увеличение на 1) ++.

❒ Декремент (уменьшение на 1) --.

Примеры:

Unary.java

```
int x = 3;
short y = 100;
x++; // после выполнения x становится равным 4.
y--; // после выполнения y становится равным 99.
```

Префиксные унарные операции ставятся перед операндом:

❒ Унарный плюс (обозначает положительные числа, хотя числа положительными будут и без него) "+".

❒ Унарный минус (обозначает отрицательные числа) "-".

❒ Логическое НЕ (инвертирует значение логического типа, превращая ИСТИНА в ЛОЖЬ и наоборот) "!".

❒ Префиксный инкремент (увеличивает значение на 1) "++".

❒ Префиксный декремент (уменьшает значение на 1) "--".

Примеры:

Unary.java

```
int x1 = +10; // положительная десятка
int x2 = -x1; // -10

boolean b1 = true;
boolean b2 = !b1; // false
```

```
++x1; // теперь x1 равен 11.
--x2; // теперь x2 равен -11
```

3.7. Отличие постфиксного и префиксного инкремента и декремента

На первый взгляд может показаться, что префиксный и постфиксный инкремент и декремент одинаковы, но это не так. Их отличие в том, что префиксный инкремент и декремент возвращают значение, которое получилось после операции увеличения и уменьшения соответственно, а постфиксный инкремент и декремент возвращают исходное значение, которое было до увеличения или уменьшения.

Пример:

PostfixPrefixDifference.java

```
class PostfixPrefixDifference {
    public static void main(String[] args) {
        int x1 = 100;
        int x2 = 145;

        int y1 = ++x1;
        int y2 = --x2;

        // Вывод для префиксных операций
        System.out.println("\nPrefix ++, -- test");
        System.out.println("x1=" + x1 + "; y1=" + y1);
        System.out.println("x2=" + x2 + "; y2=" + y2);

        // Возвращаем исходные значения
        x1 = 100;
        x2 = 145;

        int z1 = x1--;
        int z2 = x2++;

        // Вывод для постфиксных операций
        System.out.println("\nPostfix ++, -- test");
        System.out.println("x1=" + x1 + "; z1=" + z1);
        System.out.println("x2=" + x2 + "; z2=" + z2);
    }
}
```

Две косые черты `//` означают комментарий. Компилятор игнорирует любой текст, находящийся правее `//`, что позволяет записать какое-нибудь пояснение для будущего читателя программы. Строки `System.out.println` выводят текст в консоль.

Этот пример выводит в консоль следующее:

```
Prefix ++, -- test
x1=101; y1=101
x2=144; y2=144

Postfix ++, -- test
x1=99; z1=100
x2=146; z2=145
```

Как видно из примера, `y1` и `y2` стали равны значениям `x1` и `x2`, которые получились после осуществления операций инкремента и декремента соответственно, а `z1` и `z2` стали равны значениям `x1` и `x2`, которые были до операций инкремента и декремента.

3.8. Операции сравнения

Операции сравнения позволяют проверить, больше ли один операнд другого, либо убедиться, что один операнд равен другому и т. д.

Вот список операций сравнения в Java:

- `==` равенство (обратите внимание, что нужно использовать два символа "равно" для сравнения, а не один);
- `!=` неравенство;
- `>` больше;
- `>=` больше или равно;
- `<` меньше;
- `<=` меньше или равно.

Все операции сравнения возвращают логическое значение `boolean`, что означает: результат операции сравнения можно присвоить переменной этого типа и использовать в любом месте, где требуется значение типа `boolean`.

Пример:

Conditional.java

```
public class Conditional {
    public static void main(String[] args) {
        int x = 3;
        double d = 3.1;
        System.out.println(x == d); // false
        System.out.println(x > d);  // false
        System.out.println(x < d);  // true
    }
}
```

При сравнении используются следующие правила:

- Если один из операндов `NaN` ("не число"), то результат `false`.
- `-Infinity` меньше `+Infinity`.
- –0.0 с плавающей точкой равен +0.0 с плавающей точкой.
- При сравнении примитивов разных типов значение меньшего типа приводится к большему типу.

3.9. Логические И и ИЛИ

Логическое И `&&` и логическое ИЛИ `||` ведут себя вполне ожидаемо для логического И или логического ИЛИ:

Logical.java

```
boolean b1 = true && true; //true
boolean b2 = true && false; //false
boolean b3 = true || false; // true
boolean b4 = false || false; //false

System.out.println(b1);
System.out.println(b2);
System.out.println(b3);
System.out.println(b4);
```

Логическое И (обозначается: `&&`) вычисляет свой правый операнд только в том случае, если левый равен `true`. Если левый операнд равен `false`, то сразу возвращается `false`. Логическое ИЛИ (обозначается: `||`) вычисляет правый операнд только в том случае, если левый равен `false`. Если левый операнд равен `true`, то сразу возвращается `true`. Эти два правила сокращения вычислений позволяют сразу откинуть последующие вычисления, если результат всего выражения уже известен. Это можно использовать для проверки на `null` перед проверкой результата какого-либо метода объекта (будет описано в дальнейшем):

Logical.java

```
if (obj != null && obj.method1()) { // obj.method1()
                                    // будет вызываться, только если
                                    // проверка obj!= null вернула true.
}
```

3.10. Операция instanceof

Классы и интерфейсы я пока здесь не описал, но эту операцию невозможно объяснить без них. Операция `instanceof` проверяет, является ли объект экземпляром

класса, или экземпляром дочернего класса, или экземпляром класса, реализующего интерфейс.

```
obj1 instanceof A
```

Возвращается `true`, если `obj1` не `null` и является экземпляром класса `A`, или экземпляром дочернего класса `A`, или экземпляром класса, реализующего интерфейс `A`.

InstanceOf.java

```
Object obj1 = new String("test1");
if (obj1 instanceof String) {
    System.out.println("YES");
}
```

Если левый операнд равен `null`, то результатом будет `false`. Код ниже выведет `"NO"`:

InstanceOf.java

```
Object obj2 = null;
if (obj2 instanceof String) {
    System.out.println("YES");
} else {
    System.out.println("NO");
}
```

Операцию `instanceof` используют перед объявлением переменной нужного типа и перед присвоением ей значения из старой с явным указанием преобразования типа:

InstanceOf.java

```
// Переменная obj3 типа Object, но фактически содержит LocalDate
Object obj3 = LocalDate.of(2021, 9, 28);
if (obj3 instanceof LocalDate) {
    LocalDate myDate2 = (LocalDate) obj3;
    System.out.println("year=" + myDate2.getYear());
}
```

В Java 16 появилась возможность упростить такой код с помощью `pattern matching instanceof`:

InstanceOf.java

```
// pattern matching instance of
if (obj3 instanceof LocalDate myDate2) {
    System.out.println("year=" + myDate2.getYear());
}
```

```
// Можно даже так
if ((obj3 instanceof LocalDate myDate2) && (myDate2.getYear() > 1000)) {
    System.out.println("year=" + myDate2.getYear());
}
```

3.11. Тернарная операция

Операция `?:` называется тернарной, потому что принимает три операнда, что более двух, тогда как унарная операция принимает один операнд, а бинарная — два операнда.

```
<выражение_boolean> ? <выражение1> : <выражение2>
```

Тернарная операция вычисляет `<выражение_boolean>`, если оно равно `true`, то вычисляет и возвращает `<выражение1>`, а если `false`, то `<выражение2>`.

Ternary.java

```
public class Ternary {
    public static void main(String[] args) {
        int x = 255 > 34 ? 10 : -22; // 10
        String str1 = 44 == 66 ? "YES": "NO"; //"NO"
        System.out.println(x);
        System.out.println(str1);
    }
}
```

3.12. Битовые операции

Битовые операции в Java используются редко, но знать их нужно. Работают они так же, как и в Javascript и во многих других языках.

Битовые операции в Java:

- Битовый сдвиг влево `<<`.
- Битовый знаковый сдвиг вправо `>>`.
- Беззнаковый битовый сдвиг вправо `>>>`. Он отличается от `>>` тем, что ставит 0 в самую левую позицию, а `>>` ставит то, что было в знаковом бите.
- Инвертация бит ~ меняет 0 на 1 и 1 на 0 во всех битах.
- Битовый `&` применяет побитовую операцию И.
- Битовый `|` применяет побитовую операцию ИЛИ.
- Битовый `^` применяет операцию `XOR` (исключающее или).

Эти операнды работают так же, как и их аналоги в других языках программирования. Вряд ли имеет смысл их особенно рассматривать.

Bitwise.java

```
int n1 = 4; // 100 в двоичной системе
System.out.println("n1 >> 1 = " + (n1 >> 1)); //2 или 10 в двоичной системе.

System.out.println("n1 << 1 = " + (n1 << 1)); ;// 8 или 100 в двоичной системе.

System.out.println("0b101 & 0b100 = " + (0b101 & 0b100)); // 4  (0b100)
System.out.println("0b001 | 0b100 = " + (0b001 | 0b100)); // 5  (0b101)
System.out.println("0b1110 ^ 0b1011 = " + (0b1110 ^ 0b1011));// 5(0b101);

System.out.println("-2 >> 1 = " + (-2 >> 1)); // -1 (единица со знака
                                // сдвинется вправо, так что знак не поменяется)

System.out.println("-2 >>> 1= " + (-2 >>> 1)); // 2147483647 (сменит
                          // знак, т. к. левый бит заполнится нулем).

System.out.println("~1 = " + ~1) ;  // -2 (0b000...001
                                    // превратится в 0b1111..10)
```

3.13. Присвоение с выполнением другой операции

Операции += (сложение с присвоением), -= (вычитание с присвоением), *= (умножение с присвоением), "/=" (деление с присвоением), "%=" (взятие остатка с присвоением), "&=" (битовый И с присвоением), "^=" (битовое исключающее ИЛИ с присвоением), "|=" (битовое ИЛИ с присвоением), "<<=" (сдвиг влево с присвоением), ">>=" (знаковый сдвиг вправо с присвоением), ">>>=" (беззнаковый сдвиг вправо с присвоением) позволяют сразу выполнить операции и присвоить результат другой переменной.

Они работают так:

```
E1 compop E2
```

эквивалентно

```
E1 = (T) E1 op E2;
```

где `T` — это тип переменной `E1`.

То есть `int x1 += x2` эквивалентно `int x1 = (int) x1 + x2`.

AssignmentWithOther.java

```
int x1 = 100;
byte x2 = 100;
int x3 = 100;

x1 += 300; // эквивалентно x1 = (int) x1 + 300;
x2 += 300; // эквивалентно x2 = (byte) x2 + 300;
x3 += 300.1; // эквивалентно x3 = (int) x3 + 300.1;
```

```
System.out.println("x1=" + x1);  // 400
System.out.println("x2=" + x2);  // -112
System.out.println("x3=" + x3);  // 400
```

3.14. Приоритеты операций

Все операции вычисляются слева направо (сначала вычисляется левый операнд, затем правый, а потом сама операция, кроме операции присваивания. Операция присваивания вычисляется справа налево.

Вычисления производятся в соответствии с таблицей приоритетов операций (табл. 3.1).

Таблица 3.1. *Приоритеты операций*

Группа операций	Приоритет
Группировка	*(...)*
Доступ к члену	*... . ...*
Постфиксные	*expr*++ *expr*--
Унарные	++*expr* --*expr* +*expr* -*expr* ~ !
Мультипликативные	* / %
Аддитивные	+ -
Сдвиги	<< >> >>>
Сравнения	< > <= >= instanceof
Равенства	== !=
Бинарный И	&
Бинарный исключающее ИЛИ	^
Бинарный ИЛИ	|
Логический И	&&
Логический ИЛИ	||
Тернарный	? :
Лямбда	->
Присваивания	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Операция, находящаяся выше в таблице, имеет более высокий приоритет, чем операция, находящаяся ниже, и вычисляется раньше. Операции, находящиеся на одной строке, имеют одинаковый приоритет. Если в одном выражении находится несколько разных операций, то сначала вычисляется результат операции с наивысшим приоритетом. Можно использовать скобки для указания того, что сначала нужно вычислить эту часть выражения.

Пример 1:

```
int z = 200 * (3 + 4);
```

Последовательность вычисления такая:

```
3+4 = 7
200 * 7 = 1 400
z = 1 400
```

Пример 2:

```
int x;
int y;
int z = x = y = 10000 + 20000 >> 1 + 3 * 2;
```

Последовательность вычисления такая:

- 10 000 + 20 000 = 30 000 (присвоение вычисляется справа налево, поэтому сначала смотрится y = 10 000 + 20 000 >> 1 + 3 * 2, и вычисляется правая часть. В правой части (10 000 + 20 000 >> 1 + 3 * 2) вычисление идет слева направо, и берется 10 000 + 20 000 (выбирается среди 10 000 + 20 000, 20 000 >> 1, 1 + 3 и 3 * 2), которое вычисляется перед сдвигом, т. к. у сложения приоритет выше).
- 3 * 2 = 6 (в выражении 30 000 >> 1 + 3 * 2 вычисление идет слева направо, и выбирается умножение (среди 30 000 >> 1, 1 + 3 и 3 * 2), т. к. у него приоритет выше, что означает, что умножение будет выполнено раньше).
- 1 + 6 = 7 (в выражении 30 000 >> 1 + 6 вычисление идет слева направо, и сложение вычисляется раньше сдвига, т. к. приоритет у сложения выше).
- 30 000 >> 7 = 234 (0b00...111010100110000 сдвигаем на 7 бит вправо и получаем 0b00...0011101010).
- y = 234.
- x = 234.
- z = 234.

3.15. Задания

1. Составьте программу вычисления площади прямоугольного садового участка по ширине и высоте.
2. Составьте программу для подсчета платы за электроэнергию по значению расхода электроэнергии и тарифу (кВт × ч).
3. Напишите программу вычисления длины ткани, которую нужно отрезать, чтобы ее хватило для полного оборачивания бочки. Считается, что ширина ткани и бочки одинакова. Вычисления производить по известному радиусу бочки.
4. Напишите программу для определения индекса массы тела по формуле $I = m \div h^2$, где m — масса тела в килограммах, h — рост в метрах. Программа должна выводить результат по таблице:

Индекс массы тела	Результат
16 и менее	Выраженный дефицит массы тела
16–18,5	Дефицит массы тела
18,5–25	Норма
25–30	Избыточная масса тела (предожирение)
30–35	Ожирение 1-й степени
35–40	Ожирение 2-й степени
40 и более	Ожирение 3-й степени

ГЛАВА 4

Выражения, инструкции и блоки

Выражения — это конструкции, состоящие из переменных, операций и вызовов методов.

Тип данных, возвращаемый выражением, зависит от используемых элементов. Например, выражение 9 * 3.0 возвращает тип `double`.

В составных выражениях с различными операторами рекомендуется расставлять скобки, а не надеяться на приоритет операций:

```
(a == b) && (d == c)
```

и

```
a + (b * c)
```

Инструкции или операторы — это наименьшая законченная часть языка программирования.

Следующие виды выражений могут быть преобразованы в инструкции с помощью завершающего символа `;`:

- Выражение с присвоением.
- Любое использование `++` или `--`.
- Вызов методов.
- Выражения создания объектов.

Примеры:

Instruction.java

```
// Присваивание
f1 = 34.5f;

// Инкремент
y++;

// Вызов метода
System.out.println("Hello, World!");

// Создание объекта
java.math.BigDecimal sum1 = new java.math.BigDecimal("100.0");
```

В дополнение к вышеперечисленным инструкциям есть еще инструкции объявления и инструкции управления порядком выполнения. Инструкции управления порядком выполнения будут рассмотрены позже, а инструкции объявления выглядят так:

```
double myVal = 3.459;
```

Блок инструкций или операторов — это группа из нуля или нескольких инструкций, заключенная в фигурных скобках. Блок может быть использован в любом месте, где употребительна одиночная инструкция.

Пример:

Block.java

```
{
    System.out.println("In block operator 1");
    System.out.println("In block operator 2");
}
```

Чаще всего блок операторов используется с операторами `if-then-else`, `while`, `do`, которые будут рассмотрены в следующей главе.

4.1. Операторы управления порядком выполнения

Инструкции (операторы) в программе выполняются сверху вниз по исходному файлу. Операторы управления порядком выполнения могут прервать обычный ход выполнения, позволив выполнить один кусок кода несколько раз, выполнить кусок кода только при выполнении определенного условия.

4.2. Операторы if-then и if-then-else

Оператор `if-then / if-then-else` позволяет выполнить один оператор или блок операторов только при выполнении определенного условия. Он имеет две формы. Форма `if-then` позволяет выполнить кусок кода при выполнении определенного условия, а форма `if-then-else` в дополнение к этому позволяет еще указать кусок кода, который будет выполняться при НЕвыполнении этого условия.

Синтаксис `if-then` для блока операторов:

```
if (<выражение_boolean>) {
    <оператор1>;
    <оператор2>;
    ...
    <операторN>;
}
```

Оператор `if-then` выполняется так:

Вычисляется результат выражения `<выражение_boolean>`, которое должно обязательно вернуть тип `boolean`.

Если выражение `<выражение_boolean>` вернуло `true`, то выполняется `<оператор1>` для случая с одним оператором и `<оператор1>`, `<оператор2>`, ... `<операторN>` для случая блока операторов.

Примеры:

IfThenElse.java

```
if (isCracked)
    System.out.println("Cracked");

if (isCracked) {
    System.out.println("In block");
    System.out.println("Cracked");
}
```

Согласно официальному соглашению по оформлению кода рекомендуется всегда использовать блок операторов, даже если оператор только один. Пример:

```
if (isCracked) {
    System.out.println("Cracked");
}
```

Синтаксис if-then-else:

```
if (<выражение_bolean>) {
    <оператор_then_1>;
    <оператор_then_3>;
    ...
    <оператор_then_N>;
```

```
} else
    <оператор_else_1>;

if (<выражение_bolean>)
    <оператор_then_1>;
else {
    <оператор_else_1>;
    <оператор_else_2>;
    ...
    <оператор_else_N>;
}

if (<выражение_bolean>) {
    <оператор_then_1>;
    <оператор_then_3>;
    ...
    <оператор_then_N>;
} else {
    <оператор_else_1>;
    <оператор_else_2>;
    ...
    <оператор_else_N>;
}
```

Работает он так:

- Если `<выражение_boolean>` вернуло `true`, то выполняется оператор или блок операторов внутри фигурных скобок, как и для `if-then`.
- Если `<выражение_boolean>` вернуло `false`, то выполняется оператор или блок операторов в `else`.

Примеры:

IfThenElse.java

```
if (isCracked) {
    System.out.println("Cracked");
} else {
    System.out.println("not cracked");
}

if (isCracked)
    System.out.println("Cracked");
else
    System.out.println("not cracked");
```

Можно в блоке else проверить другое условие, тогда получится конструкция вида:

IfThenElse.java

```
if (x < 0) {
    System.out.println("< 0");
} else if (x == 3) {
    System.out.println(" == 3");
} else if (x > 10) {
    System.out.println(" > 10");
} else {
    System.out.println("else (x >= 0 and x <> 3 and x <= 10)");
}
```

В конструкциях такого вида будет сначала проверяться условие в первом `if` (в данном случае x < 0), если оно вернет `true`, то выполнится его блок, если оно вернет `false`, то будет проверяться следующее условие (в данном случае x == 3), если оно вернет `true`, то выполнится его блок. Таким образом, будут проверяться подряд все условия до такого условия, которое вернет `true`. Если же ни одно из условий не вернет `true`, то выполнится блок `else` (если он есть, но его может не быть, тогда управление перейдет дальше к следующему оператору за такой конструкцией).

Обратите внимание, что `<выражение_boolean>` может быть абсолютно любым выражением, возвращающим `boolean`, можно использовать даже операцию присваивания переменной логического типа, т. к. операция присваивания тоже возвращает значение.

IfThenElse.java

```
boolean b;
int x = 3;

if (b = x > 0) {
    // Теперь b присвоено true как результат метода
    // obj1.someMethodReturnsBoolean()
} else {
    // А вот здесь b присвоено false.
}
```

Однако соглашение по оформлению кода в Java гласит, что использовать подобную возможность не стоит.

4.3. Оператор switch

Рассмотрим следующий кусок кода:

```
if (mode == 0) {
    // Инструкции для mode 0
```

```
} else if (mode == 1) {
    // Инструкции для mode 1
} else if (mode == 2) {
    // Инструкции для mode 2
} else {
    // Инструкции для остальных mode.
}
```

В куске кода, приведенном выше, проверяется значение переменной mode. Для значений 0, 1 и 2 предусмотрены отдельные блоки кода, и еще один блок кода предусмотрен для всех остальных значений. Оператор `switch` делает то же самое, но код становится более наглядным:

Switch1.java

```
switch (mode) {
case 0:
    // Инструкции для mode 0
    break;
case 1:
    // Инструкции для mode 1
    break;
case 2:
    // Инструкции для mode 2
    break;
default:
    // Инструкции для других mode
    break;
}
```

Оператор `switch` работает в следующем порядке:

1. Вычисляется выражение в скобках (в данном примере оно состоит просто из переменной `mode`).
2. Полученное значение проверяется подряд со значениями в блоках `case`, и выполняется тот блок операторов, который относится к `case` со значением, совпадающим со значением выражения.
3. Если ни одно из значений не совпало, то выполняется блок `default`.
4. По ключевому слову break выполнение блока внутри `case` или `default` завершается, и управление передается на инструкцию, следующую за блоком `switch`.

С помощью `if-then` и `if-then-else` можно проверять любые условия, но с помощью `switch` можно проверять только значения выражений типа `byte`, `short`, `char`, `int`, `enum` (перечисления будут описаны позднее), `String` (начиная с Java SE 7), а также классы `java.lang.Byte`, `java.lang.Short`, `java.langCharacter`, `java.lang.Integer`. Проверяемые значения в `case` обязательно должны быть константными литералами. Если значение выражения в `switch` равно `null`, то возникает исключение

java.lang.NullPointerException. Нагляднее всего switch выглядит именно с перечислениями.

Ключевое слово break не обязательно. В случае его отсутствия по завершении выполнения блока операторов внутри одного case выполняются операторы следующего за ним case. Это позволяет использовать один блок операторов для нескольких значений case:

Switch2.java

```
switch (mode) {
case -1:
    System.out.println("mode -1");
    break;
case 0:
    System.out.println("mode 0");
case 1:
case 2:
    System.out.println("mode 0 or 1 or 2");
    break;
case 3:
    System.out.println("mode 2");
    break;
default:
    System.out.println("mode default");
    break;
}
```

Если mode равно 0, то код выше выведет в консоль:

```
mode 0
mode 0 or 1 or 2
```

Если mode равно 1, то код выше выведет в консоль:

```
mode 0 or 1 or 2
```

Если mode равно 2, то код выше выведет в консоль:

```
mode 0 or 1 or 2
```

Блок default не обязательно указывать в конце блока switch. Он может стоять и в начале, и в середине; но рекомендуется всегда писать его последним, так получается гораздо нагляднее, потому что он выполняется в том случае, если ни один из case-ов не подошел:

Switch3.java

```
switch (mode) {
case 0:
    System.out.println("mode 0");
    break;
```

```
default:
    System.out.println("mode default");
    break;
case 1:
    System.out.println("mode 1");
    break;
case 2:
    System.out.println("mode 2");
    break;
}
```

Можно даже вообще не указывать блок `default`:

Switch4.java

```
switch (mode) {
case 0:
    System.out.println("mode 0");
    break;
case 1:
    System.out.println("mode 1");
    break;
case 2:
    System.out.println("mode 2");
    break;
}
```

В Java 14 появились switch expressions, позволяющие записывать `switch` более кратким способом, а также возвращать значения из `switch`.

SwitchExpression1.java

```
int moneyType = 3;
String moneyDescription;

// исходный switch:
switch (moneyType) {
    case 1:
    case 2:
        moneyDescription = "Gold";
        break;
    case 3:
        moneyDescription = "Aden";
        break;
    case 4:
    case 5:
        moneyDescription = "Dollar";
        break;
```

```
    default:
        moneyDescription = "Septim";
}
System.out.println("moneyDescription = " + moneyDescription);

// Идентичное ему switch expression:
switch (moneyType) {
    case 1, 2 -> moneyDescription = "Gold";
    case 3 -> moneyDescription = "Aden";
    case 4, 5 -> moneyDescription = "Dollar";
    default -> moneyDescription = "Septim";
};
System.out.println("moneyDescription = " + moneyDescription);
```

Обратите внимание: со switch expressions код стал гораздо короче, потому что теперь мы можем перечислять через запятую значения, для которых нужно выполнить одну и ту же ветку кода, а не писать отдельный case на новой строке. Также нам не нужно писать `break`, который по невнимательности можно было бы легко перепутать с обычным `switch`.

Кроме более компактного кода switch expressions позволяют возвращать значения из блока `switch`. Для этого используется ключевое слово `yield`:

SwitchExpression2.java

```
int moneyType = 3;
String moneyDescription = switch (moneyType) {
    case 1, 2:
        System.out.println("Some text");
        yield "Gold";
    case 3:
        yield "Aden";
    case 4, 5:
        yield "Dollar";
    default:
        yield "Septim";
};
```

4.4. Оператор while

Оператор `while` позволяет выполнить инструкцию или блок инструкций несколько раз.

Синтаксис оператора `while`:

```
while (<условие>)
    <оператор1>;
```

И с блоком операторов/инструкций (в соответствии с соглашением по оформлению кода Java рекомендуется использовать вариант с блоком даже в случае одной инструкции):

```
while (<условие>) {
    <оператор1>;
    <оператор2>;
    ...
    <оператор3>;
}
```

Оператор `while` вычисляет выражение `<условие>` и выполняет оператор или блок операторов, если результат выражения `<условие>` равен `true`. Затем он еще раз вычисляет и проверяет `<условие>` и, если оно вернуло `true`, то снова выполняет оператор или блок операторов. Так происходит до тех пор, пока `<условие>` не вернет `false`.

Следующий код выведет числа от 0 до 10:

While.java

```
int n = 0;
while (n <= 10) {
    System.out.println(n);
    n++;
}
```

Выражение `<условие>` в `while` может быть любым, но оно обязательно должно возвращать тип `boolean`. Следующий код тоже корректен:

WhileEndless.java

```
// Бесконечный цикл
while (true) {
    // операторы
}
```

Это тоже работает:

While.java

```
// Это тоже корректный код, но obj1 должен иметь
// метод someMethodReturnsBoolean(), возвращающий boolean.
boolean b;
while (b = obj1.someMethodReturnsBoolean()) {
    // операторы
}
System.out.println("moneyDescription = " + moneyDescription);
```

4.5. Оператор do-while

Оператор `do-while` похож на оператор `while`, но он сначала выполняет операторы, а лишь потом проверяет условие, таким образом, каждый оператор или блок операторов выполнятся хотя бы один раз.

Примеры:

DoWhile.java

```
// вывод чисел от 0 до 10
int x = 0;
do {
    System.out.println(x);
    x++;
} while (x <= 10);

// Условие ложно изначально, но выведется 0, т. к. сначала
// будет выполняться блок операторов, а лишь затем —
// проверяться условие
x = 0;
do {
    System.out.println(x);
    x++;
} while (x < 0);
```

4.6. Оператор for

Оператор `for` представляет собой компактную форму перебора диапазона чисел. Его синтаксис:

```
for (<инициализация>; <условие>; <инкремент>)
    <оператор1>;
```

Или для блока операторов (в соответствии с соглашением по оформлению кода в Java рекомендуется использовать этот вариант даже для случая одного оператора):

```
for (<инициализация>; <условие>; <инкремент>) {
    <оператор1>;
    <оператор2>;
    ...
    <операторN>;
}
```

Выражение `<инициализация>` выполняется только один раз перед началом итерации. Переменные, объявленные в `<инициализация>`, действуют только внутри цикла `for`, включая `<инициализация>`, `<условие>` и `<инкремент>`.

Выражение <условие> выполняется перед каждым циклом итерации. Блок операторов или одиночный оператор выполняются только в тех случаях, когда <условие> вернуло true. Если <условие> возвращает false, то выполнение цикла for завершается.

Выражение <инкремент> выполняется после каждого цикла перед проверкой выражения <условие>. Его обычно используют для увеличения или уменьшения значения.

Любое из выражений <инициализация>, <условие>, <инкремент> можно опустить. Можно даже сделать цикл for вообще без инициализации, условия, инкремента и оператора/блока операторов:

ForEndless.java

```
// Бесконечный цикл
for (;;);
```

Пример использования цикла for для вывода значений от 0 до 10:

For.java

```
for (int n = 0; n <= 10; n++) {
    System.out.println(n);
}
```

В <инициализация> и <инкремент> можно указывать несколько выражений инициализации и несколько инкрементов. В этом случае они указываются через запятую и вычисляются слева направо:

For.java

```
for (int n = 0, m = 3; n <= 10; n++, m--) {
    System.out.println("n=" + n + "; m=" + m);
}
```

Существует еще специальная форма for для обхода по массивам и коллекциям:

For.java

```
int[] myarray = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (int n : myarray) {
    System.out.println(n);
}
```

Выведет в консоль следующее:

```
0
1
2
3
4
5
6
7
8
9
10
```

4.7. Оператор break

Оператор `break` позволяет прервать текущее выполнение `switch` или цикла `for`, `while` и `do-while` и перейти к следующему оператору после него. Примеры:

Break.java

```
int n1 = 0;
while (true) {
    System.out.println(n1);
    if (n1 == 10) break;
    n1++;
}

int n2 = 0;
do {
    System.out.println(n2);
    if (n2 == 10) break;
    n2++;
} while (true);

for (int n3 = 0; ;) {
    System.out.println(n3);
    if (n3 == 10) break;
    n3++;
}
```

Все приведенные выше примеры выводят в консоль числа от 0 до 10.

В случае вложенных циклов оператор `break` прерывает выполнение самого глубокого из текущих вложенных циклов. Можно прервать внешний цикл, если указать для него метку:

BreakLabel.java

```
// метка для внешнего цикла.
outer_for:
for (int n = 0; n < 10; n++) {
    //...
    // метка для внутреннего цикла
inner_for:
    for (int m = 0; m < 10; m++) {

        // прерываем внешний цикл для n == 2 и m ==4
        if ((n == 9) && (m == 4)) break outer_for;

        // прерываем внутренний цикл для n == 7 и m == 2
        if ((n == 7) && (m == 2)) break;

        // можно прервать внутренний цикл и по метке.
        if ((n == 9) && (m % 2 == 1)) break inner_for;

        System.out.println("n=" + n + "; m=" + m);
    }
}
```

Примечание

В Java не принято использовать метки, а значит, и оператор `break` в варианте с меткой тоже использовать не принято.

4.8. Оператор continue

Оператор `continue` позволяет пропустить текущую итерацию и перейти сразу к следующей итерации цикла `for`, `while` или `do-while`. Он тоже бывает с меткой и без нее.

Пример без метки:

Continue.java

```
for (int n = 0; n <= 10; n++) {
    if (n % 2 == 0) continue;
    System.out.println(n);
}
```

Выведет в консоль:

```
1
3
5
```

```
7
9
```

Пример с меткой:

ContinueLabel.java

```
// ...
outer_label:
    for (int n = 4; n <= 9; n++) {
        int m = 3;
        while (m <= 5) {
            // Если остаток от деления n на m равен 0,
            // то переходим к следующей итерации
            // цикла outer_label
            if (n % m == 0) continue outer_label;
            System.out.println("n=" + n + "; m=" + m);
            m++;
        }
    }
```

Это выведет в консоль:

```
n=4; m=3
n=5; m=3
n=5; m=4
n=7; m=3
n=7; m=4
n=7; m=5
n=8; m=3
```

> ***Примечание***
>
> В Java не принято использовать метки, а соответственно, и оператор `continue` с метками.

4.9. Оператор return

Оператор `return` возвращает управление из текущего метода в тот, который вызвал текущий метод. Он имеет две формы: с возвращаемым значением и без.

С возвращаемым значением:

```
return <выражение>;
```

Пример:

```
return m + 3;
```

И без возвращаемого значения (для метода с `void`):

```
return;
```

Оператор `return` используется как в статических методах, так и методах экземпляров, которые будут изучены в последующих главах.

4.10. Задания

1. Счастливым считается билет, в шестизначном номере которого сумма первых трех цифр совпадает с суммой трех последних. Пусть номер билета хранится в переменной целого типа. Напишите программу, которая выводит строку "билет счастливый, я его съел", если билет счастливый, а в противном случае выводит в консоль "обычный билет".
2. Создайте программу для определения, находится ли точка (x;y) внутри круга с радиусом r.

ГЛАВА 5

Классы и объекты

5.1. Классы

Классы в Java объявляются с помощью ключевого слова `class`. Пример самого простого объявления класса:

Goblin.java

```
package ru.urvanov.javaindynamics2022.classes;

public class Goblin {
}
```

Здесь мы объявляем новый класс с именем `Goblin`.

Внутри фигурных скобок объявляются все поля, конструкторы и методы класса.

Перед ключевым словом `class` может стоять модификатор `public`, который делает класс доступным из всех пакетов. Если модификатора `public` нет, как в нашем случае, то класс доступен только в том пакете, в котором он объявлен.

5.2. Поля

Пример объявления полей:

GoblinFields.java

```
package ru.urvanov.javaindynamics2022.classes;

public class GoblinFields {
    private int money;
    double health;
    protected int diamonds = 10;
    public String name;
}
```

В этом примере мы объявили четыре поля:

- поле `money` с типом `int`;
- поле `health` с типом `double`;
- поле `diamonds` с типом `int`;
- поле `name` с типом `String`.

Каждый экземпляр класса `GoblinFields` будет иметь свое значение полей money, `health`, `diamonds` и `name`.

В самом начале объявления поля либо указывается модификатор доступа к полю (`private`, `protected` или `public`), либо не указывается, и тогда используется доступ по умолчанию `package-private`. Затем при необходимости указывается ключевое слово `static` (будет объяснено позже), а также, если нужно, ключевое слово `final` (будет объяснено позже). После них — тип поля и имя. Затем поле может сразу инициализироваться начальным значением, например, как поле `diamonds` инициализируется числом 10 в нашем примере.

Модификаторы доступа, `static` и `final`, могут располагаться в любом порядке, но в соглашении по оформлению кода принят именно такой порядок, какой описан в этом разделе.

Поле `money` объявлено с модификатором доступа `private`, и оно будет доступно только внутри этого класса.

Поле `health` объявлено без модификаторов доступа, и для него будет использоваться уровень доступа `package-private` (поле будет доступно только внутри своего пакета).

Поле `diamonds` объявлено с модификатором доступа `protected`, и оно будет доступно в этом пакете, этом классе и классах-наследниках от этого класса (как объявлять наследники, будет объяснено позже).

Поле `name` объявлено с модификатором доступа `public`, и оно будет доступно во всех классах всех пакетов (табл. 5.1).

Таблица 5.1. *Уровни доступа*

Модификатор	Класс	Пакет	Дочерний класс	Все классы
Public	Есть	Есть	Есть	Есть
Protected	Есть	Есть	Есть	Нет
Без модификатора	Есть	Есть	Нет	Нет
Private	Есть	Нет	Нет	Нет

Имя поля следует давать в соответствии с соглашениями по оформлению кода (**https://www.oracle.com/java/technologies/javase/codeconventions-contents.html**).

Обращаться к полю внутри класса, которому оно принадлежит, можно просто по имени поля:

GoblinFire.java

```
package ru.urvanov.javaindynamics2022.classes;

class GoblinFire {
    int ammo = 10;
    //... другие поля класса

    // метод стрельбы
    public void fire() {
        // уменьшаем количество пуль.
        // Обратите внимание, что к полю класса
        // обращаемся просто по имени
        ammo --;

        // ... остальной код
    }

    // ... другие методы
}
```

Из других классов обращение к полю класса происходит через точку, например:

AccessGoblinField.java

```
package ru.urvanov.javaindynamics2022.classes;

public class AccessGoblinField {
    public static void main(String[] args) {
        // Создаем объект GoblinFire
        GoblinFire goblinObj = new GoblinFire();
```

```
        // Обращение к полю ammo
        goblinObj.ammo ++;
    }
}
```

Имейте в виду, что прямое обращение к полю другого класса является примером плохого стиля, поскольку нарушает принципы ООП. Все обращения должны происходить к методам, которые уже сами меняют значения полей в соответствии с заложенной в них логикой.

Также рекомендуется давать всем полям класса минимальный из всех возможных уровней доступа. Это означает, что большинство полей класса должны иметь уровень доступа `private`. Остальные уровни доступа должны даваться отдельным переменным только в том случае, если это действительно нужно.

При объявлении полей можно в одной инструкции объявить несколько полей с одинаковым типом и одинаковыми модификаторами, но соглашение по оформлению кода так делать не рекомендует:

```
// Так делать НЕ рекомендуется
int x, y, z; // Объявляем три переменные: x, y, z
```

5.3. Объявление методов

Мы уже видели примеры объявления методов в *главе 1 "Введение"* и в примере доступа к полям в данном разделе.

Пример объявления метода:

GoblinMethods.java

```
package ru.urvanov.javaindynamics2022.classes;

public class GoblinMethods {
    private int hitPoints;
    public int fire(boolean withAim, double windDirection,
                    double windPower) {
        // ... инструкции...
        return hitPoints;
    }
}
```

Объявление метода состоит из следующих частей:

1. Модификатор доступа: `private`, без модификатора (`package-private`), `protected`, `public`.
2. Ключевое слово `static`, если нужно (будет описано позже).
3. Ключевое слово `final`, если нужно (будет описано позже).

4. Тип возвращаемого значения (в данном примере `int`) или `void`, если метод не возвращает значение.
5. Имя метода (в этом примере `fire`).
6. Список параметров (в нашем примере три параметра: `withAim`, `windDirection`, `windPower`).
7. Список исключений (будет описан в последующих разделах).
8. Тело метода в фигурных скобках.

Модификаторы доступа, `static` и `final`, могут располагаться в любом порядке, но по соглашению о написании кода принят именно такой порядок, который описан здесь.

Имя метода может содержать символы подчеркивания, знак доллара, цифры и многие другие символы Юникода (даже русские буквы), но не может начинаться с цифры, так же как и имя переменной. Однако по соглашению об оформлении кода на именование методов распространяются почти такие же правила, что и на именование переменных, с тем отличием, что имя метода должно быть глаголом: `fire`, `buildObject`, `connect`, `compareTo`.

Если в объявлении метода не используется ключевое слово `void`, то метод должен явно вернуть значение с помощью оператора `return`. Пример:

GoblinMethods.java

```
public int returnSix() {
    return 6;
}
```

В операторе `return` можно указать выражение, тогда оно будет вызвано, и результатом вызова метода будет результат этого выражения:

GoblinMethods.java

```
public double sum(double x1, double x2) {
    return x1 + x2;
}
```

Тип возвращаемого значения должен совпадать с тем, что указан в объявлении, иначе будет ошибка компиляции. Если в объявлении присутствует ключевое слово `void`, то использование `return` необязательно, но его можно указать для досрочного завершения выполнения метода:

GoblinMethods.java

```
public void myVoidMethod1() {
    // ...
```

```
    if (hitPoints > 3) {
        return;
    }
    //...
}
```

В качестве типа возвращаемого значения может использоваться имя интерфейса или класса.

Если в качестве возвращаемого значения указан интерфейс, то метод должен вернуть экземпляр любого класса, реализующего этот интерфейс, или `null`.

Если в качестве возвращаемого значения указан класс, то метод должен вернуть экземпляр этого класса, экземпляр его класса-потомка или `null`.

Сигнатура метода — это имя метода вместе со списком его параметров.

Java различает методы между собой по их сигнатуре. В одном классе не может быть двух и более методов с одинаковой сигнатурой, но, поскольку список параметров входит в сигнатуру методов, их можно перегружать.

Перегрузка методов — создание нескольких методов с одинаковым именем, но разным списком параметров.

Перегрузка методов позволяет создавать, например, для каждого типа свою реализацию метода с тем же именем, но другим типом параметра, либо создать отдельный метод, принимающий другое количество параметров.

Пример перегрузки методов:

GoblinMethods.java

```
class GoblinMethods {
    public void hit(Axe axe) {
        // ... инструкции...
    }

    public void hit(Flail flail) {
        // ... инструкции...
    }

    public void hit(Scimitar scimitar) {
        // ... инструкции...
    }

    public void hit(Torch torch) {
        // ... инструкции...
    }

    public void hit(Sword sword) {
        // ... инструкции...
    }
```

```
    public void hit(Sword sword, int comboCount) {
        // ... инструкции...
    }
}
```

Все методы в примере выше — это разные методы. Компилятор различает их по списку параметров. Возвращаемое значение не входит в сигнатуру метода, а значит, вы не можете создать несколько методов с одинаковым именем и параметрами, но разными возвращаемыми значениями.

Примеры вызова методов `hit`:

GoblinMethods.java

```
Scimitar scimitar = new Scimitar();
goblin.hit(scimitar); // будет вызван public void hit(Scimitar scimitar)

Sword sword = new Sword();
goblin.hit(sword); // будет вызван public void hit(Sword sword)
```

Не стоит злоупотреблять перегрузкой методов. Используйте ее только там, где это действительно нужно, иначе это может усложнить понимание вашего кода другими разработчиками.

5.4. Конструкторы

Конструкторы вызываются для создания объектов. Они похожи на методы, но не имеют возвращаемого значения (даже `void`), и они носят то же самое имя, что и сам класс.

Пример конструктора для класса `GoblinConstructor`:

GoblinConstructor.java

```
public GoblinConstrutor(int initialMoney, double initialHealth) {
    money = initialMoney;
    health = initialHealth;
}
```

Теперь, чтобы создать экземпляр класса `GoblinConstructor`, нужно вызвать конструктор с ключевым словом new:

```
GoblinConstructor myGoblin = new GoblinConstructor (8, 100.0);
```

Код выше создаст экземпляр класса `GoblinConstructor` с помощью нашего конструктора и присвоит ссылку на этот класс переменной `myGoblin`.

С помощью перегрузки можно создать несколько конструкторов, но они должны иметь различное количество или тип параметров:

GoblinConstructor.java

```
package ru.urvanov.javaindynamics2022.classes;

public class GoblinConstructor {
    private int money;
    double health;
    protected int diamonds = 10;
    public String name;

    // Конструктор без параметров.
    public GoblinConstructor() {
    }

    // Конструктор с двумя параметрами
    public GoblinConstructor(int initialMoney, double initialHealth) {
        money = initialMoney;
        health = initialHealth;
    }

    // Конструктор с одним параметром.
    public GoblinConstructor(String goblinName) {
        name = goblinName;
    }

    // Приватный конструктор. Его можно будет вызвать
    // только внутри этого класса.
    private GoblinConstructor(int initialDiamonds) {
        diamonds = initialDiamonds;
    }

    //... еще конструкторы и методы
}
```

Теперь мы можем создавать экземпляры класса `GoblinConstructor`, используя любой из этих конструкторов, но приватный можно вызывать только внутри самого класса `GoblinConstructor` (например, в одном из его методов или полей инициализации).

GoblinConstructor.java

```
GoblinConstructor goblin0 = new GoblinConstructor();
GoblinConstructor goblin1 = new GoblinConstructor("Vasya");
GoblinConstructor goblin2 = new GoblinConstructor(3, 45.0);
```

Если мы не объявим ни одного конструктора в описании класса, то компилятор добавит один конструктор по умолчанию без параметров и с модификатором доступа `public`. Если же мы объявим хотя бы один конструктор, даже приватный, то конст-

руктор без параметров добавляться не будет, но мы можем объявить его сами, если нужно.

Хитрость

Операция `new` возвращает ссылку на объект. Можно сразу же вызвать какой-нибудь метод этого объекта или обратиться к свойству, не присваивая эту ссылку переменной:

```
new GoblinConstructor(myParam1).someMethod1(myParam2);
```

Если метод тоже возвращает ссылку на объект, то можно сразу вызвать метод этого объекта:

GoblinConstructor.java

```
new GoblinConstructor(myParam1)
        .someMethod1(myParam2).someMethod2(); //... и т. д.
```

Ключевые слова `static`, `final` и `abstract` будут описаны позднее, но если вы перечитываете учебник второй раз, то:

Запомните

Конструктор НЕ может быть `static`, `final` или `abstract`.

5.5. Передача параметров

Метод может иметь любое число параметров, но каждый параметр должен иметь уникальное имя в пределах описания этого метода. Нельзя объявить метод, у которого два параметра называются одинаково, даже если они имеют разный тип. Имя параметра не может совпадать с именем локальной переменной, объявленной в методе. Однако имя параметра может совпадать с именем поля класса, в этом случае параметр затеняет (shadows) поле, поскольку при прямом обращении к этому имени мы будем обращаться к параметру, а не к полю класса.

Параметры, описанные в объявлении метода, называются формальными. Значения, передаваемые в формальные параметры при вызове метода или конструктора, называются аргументами или фактическими параметрами.

Вы можете использовать любой примитивный тип для параметров, объект или массив.

Примитивные типы передаются по значению — изменения внутри метода или конструктора не отражаются на значении переменной, которую передали:

PrimitiveTypeParameters

```
package ru.urvanov.javaindynamics2022.classes;

/**
 * Пример передачи параметров примитивных типов.
 * Они передаются по значению.
 */
```

```
public class PrimitiveTypeParameters {
    public void tryChangeParameterValue(int val1) {
        // Мы можем менять значение val1,
        // но val1 содержит копию переданного значения.
        // Все изменения val1 видны только внутри этого метода
        System.out.println("Inside method 2: " + val1);  // 100
        val1++;
        System.out.println("Inside method 2: " + val1);  // 101
    }

    public static void main(String[] args) {
        PrimitiveTypeParameters primitiveTypeParameters =
                new PrimitiveTypeParameters();
        int parameter1 = 100;
        System.out.println("parameter1 = " + parameter1); // 100
        primitiveTypeParameters.tryChangeParameterValue(parameter1);
        System.out.println("parameter1 = " + parameter1); // 100
        // изменения внутри метода происходили
        // с копией переменной, а не с нашим
        // parameter1.
    }
}
```

Объекты и массивы передаются по ссылке: изменения внутри метода или конструктора меняют объект, который нам передали. Однако, если внутри метода присвоить значению параметра `null` или ссылку на другой объект/массив, такое изменение коснется только параметра метода, а исходный объект или массив останется неизменным.

Пример:

GoblinWithMoney.java

```
package ru.urvanov.javaindynamics2022.classes;

/**
 * Передача параметров по ссылке.
 */
public class GoblinWithMoney {
    private int money;

    /**
     * @param goblinMoney передается по ссылке.
     *      Меняя его поля, мы меняем исходный объект.
     * @param arr1 тоже передается по ссылке.
     *             Меняя значения в нем, мы меняем исходный массив.
     */
```

```
    public static void tryChangeParameterValue(
            GoblinWithMoney goblinMoney,
            int[] arr1) {

        // Эти изменения будут видны снаружи метода.
        goblinMoney.money++;
        arr1[0] = 200;

        // Эти изменения затрагивают только наш параметр ссылочного типа
        // Объекты снаружи метода не будут изменены.
        goblinMoney = null;
        arr1 = null;
        goblinMoney = new GoblinWithMoney();
        goblinMoney.money = -400;
        arr1 = new int[100];
        arr1[2] = 3;
    }

    public static void main(String[] args) {
        GoblinWithMoney goblin = new GoblinWithMoney();
        goblin.money = 45;
        int[] arr1 = {3, 4, 7};
        tryChangeParameterValue(goblin, arr1);
        System.out.println(goblin.money); // 46
        System.out.println(arr1[0]); // 200;
        System.out.println(arr1[2]); // 7
    }
}
```

Выведет в консоль:

```
46
200
7
```

Иногда бывает полезно иметь метод с произвольным числом параметров. В таком случае можно передавать ему массив в качестве параметра. Однако в Java есть возможность создать метод, принимающий произвольное число параметров. Для этого после типа последнего параметра ставится многоточие, а внутри метода этот параметр обрабатывается как обычный массив. Пример:

Varargs.java

```
package ru.urvanov.javaindynamics2022.classes;

/**
 * Пример метода с переменным числом параметров
 */
```

```
public class Varargs {
    public void sum(
            int par1,
            double par2,
            String par3,
            int... lastParameter) {
        System.out.println("lastParameter[0] = "
                + lastParameter[0]); // 3
        System.out.println("lastParameter[1] = "
                + lastParameter[1]); // 5
        //...

    }

    public static void main(String[] args) {
        Varargs main = new Varargs();
        main.sum(100, 3.4, "par3", 3, 5, 6, 8, 9);
    }
}
```

Многоточие является токеном само по себе, и технически корректно ставить пробел между типом и многоточием, но, согласно принятому соглашению по оформлению кода в Java, так делать не рекомендуется.

При вызове метода или конструктора можно передавать выражение, и тогда в метод или конструктор поступит вычисленное значение этого выражения. Параметры методов и конструкторов вычисляются слева направо: сначала вычисляется первый параметр, затем второй и т. д. Вызов метода происходит только после того, как все параметры будут вычислены.

5.6. Сборка мусора

В некоторых языках программирования нужно вручную освобождать память, выделенную под объекты. В Java такой необходимости нет. Виртуальная машина Java сама освобождает память от объектов, которые больше не используются. Объект считается более не использующимся, если на него больше нет ссылок. Ссылки на объект обычно исчезают после того, как объект выходит из своей области видимости. Вы можете самостоятельно убрать ссылку на объект, присвоив переменной значение `null`.

Сборщик мусора автоматически освобождает память от объектов, которые больше не используются, когда сочтет нужным.

5.7. Ключевое слово this

Если переменная, объявленная в методе, или параметр метода имеет то же самое имя, что и свойство класса, то эта переменная затеняет свойство класса. Обращаясь

по имени переменной, в этом случае мы будем обращаться к переменной метода или к параметру метода, а не к свойству класса. Чтобы обратиться к затененному свойству класса, нужно использовать ключевое слово `this`, которое означает этот класс.

Orc.java

```
package ru.urvanov.javaindynamics2022.classes;

class Orc {
    double health = 100.0;

    public void someMethod1() {
        double health = 200.0;
        System.out.println(health); // 200.0
        System.out.println(this.health); // 100.0
    }

    public void setHealth(double health) {
        // присваиваем свойству класса
        // переданное значение
        this.health = health;
    }

    public static void main(String[] args) {
        Orc orc = new Orc();
        orc.setHealth(999.9);
        orc.someMethod1();;
    }
}
```

Ключевое слово `this` может также использоваться для вызова из конструктора класса другого конструктора этого класса. Вызов другого конструктора должен быть обязательно первым оператором/инструкцией в конструкторе:

LegendaryBeast.java

```
package ru.urvanov.javaindynamics2022.classes;

class LegendaryBeast {
    private double health;
    private int ammo;
    private int gold;

    public LegendaryBeast() {
        this(100.0, 0, 0);
        // ... остальная инициализация.
    }
```

```
    public LegendaryBeast(double health) {
        this(health, 0, 0);
        // ... остальная инициализация.
    }

    public LegendaryBeast(double health, int ammo) {
        this(health, ammo, 0);
        // ... остальная инициализация.
    }

    public LegendaryBeast(double health, int ammo, int gold) {
        this.health = health;
        this.ammo = ammo;
        this.gold = gold;
    }
}
```

В этом примере класс `LegendaryBeast` имеет несколько конструкторов с разным числом параметров (перегруженные конструкторы). Конструкторы с меньшим числом параметров вызывают конструктор с самым большим количеством параметров. Компилятор Java различает эти конструкторы по параметрам (типу, количеству и порядку).

5.8. Ключевое слово static

При создании каждый объект получает свой отдельный набор переменных экземпляров. Если же нужно сделать какую-то переменную разделяемой для всех экземпляров, то используется ключевое слово `static`.

Пример:

GoblinWithCounter.java

```
package ru.urvanov.javaindynamics2022.classes;

class GoblinWithCounter {
    static int idCounter = 0;
    int id;

    GoblinWithCounter() {
        idCounter++;
        id = idCounter;
    }
}
```

В таком классе `GoblinWithCounter` переменная `idCounter` одна, общая для всех экземпляров. Для всех экземпляров этого класса значение этой переменной будет

всегда одно и то же, благодаря чему каждый экземпляр класса будет получать в поле `id` уникальное значение, большее, чем значение поля `id` предыдущего экземпляра. Переменная `idCounter` называется статическим свойством/полем или переменной класса и относится к классу, а не к его экземплярам.

Обратиться к статическому свойству можно либо через имя класса, либо через имя экземпляра, однако рекомендуется всегда обращаться к статическим свойствам через имя класса, чтобы подчеркнуть, что оно относится именно к классу:

```
System.out.println("idCounter="
        + GoblinWithCounter.idCounter); // предпочтительно
GoblinWithCounter goblin = new GoblinWithCounter ();
System.out.println("idCounter=" + goblin.idCounter); // нежелательно
```

Статическое свойство также может иметь модификаторы `private`, `protected` или `public`.

Модификатор `static` можно применить к методу, тогда он будет статическим, и его можно будет вызывать через имя класса:

```
static int getIdCounter() {
    return idCounter;
}
```

Пример вызова:

```
int x = Goblin.getIdCounter();
```

Статические методы можно вызывать и через имя экземпляра, но рекомендуется всегда вызывать их через имя класса, т. к. они относятся именно к нему.

К статическим методам и свойствам можно обратиться даже тогда, когда еще нет ни одного экземпляра класса.

Запомните:

- Методы экземпляров могут обращаться к переменным экземпляров (нестатическим свойствам/полям) и методам экземпляров напрямую.
- Методы экземпляров могут обращаться к переменным класса (статическим полям) и методам класса (статическим методам) напрямую.
- Методы классов могут обращаться к методам класса (статическим методам) и переменным класса (статическим свойствам/полям) напрямую.
- Методы классов не могут напрямую обращаться к переменным экземпляров (нестатическим свойствам/полям) и методам экземпляров, и они не могут использовать ключевое слово `this`, т. к. для них нет экземпляра класса. Они должны использовать ссылку на какой-нибудь экземпляр.

5.9. Ключевое слово final

Это ключевое слово больше относится к наследованию, которое будет рассмотрено в дальнейших разделах. Здесь я опишу `final` лишь в общих чертах.

Ключевое слово `final` может быть применено к локальной переменной, переменной экземпляра или параметру. Оно означает, что значение переменной не будет меняться после инициализации. Если попытаться изменить значение такой переменной, то возникнет ошибка компиляции.

GoblinFinal.java

```
class GoblinFinal {
    final String name;

    public GoblinFinal (String name) {
        this.name = name; // Инициализируем переменную final.
    }

    public void someMethod1(final String secondName) {
        final String thirdName = "Third";

        // Переменные с final менять после инициализации нельзя!
        // this.name = secondName; // Нельзя! Ошибка компиляции.
        // secondName = "3"; // Нельзя! Ошибка компиляции.
        // thirdName = "4";  // Нельзя! Ошибка компиляции.
    }
    // . . .
}
```

Переменные `final` не инициализируются значением по умолчанию. Им обязательно должно быть присвоено какое-нибудь значение, иначе возникнет ошибка компиляции.

Ключевое слово `final` может применяться к методу, тогда этот метод нельзя переопределять в классах-потомках для методов экземпляров и нельзя скрывать (hide) в классах потомках для случая статических методов (наследование и переопределение метода будут описаны в разделе про наследование):

GoblinFinal.java

```
class GoblinFinal {
    // . . .
    // Этот метод нельзя переопределять в потомках.
    public final void myFinalMethod1() {
    }

    // Этот метод нельзя скрывать в потомках.
    public static final void myFinalMethod2() {
    }
    // . . .
}
```

Можно применить `final` ко всему классу, это означает, что у класса не может быть потомков, т. е. нельзя будет наследоваться от этого класса.

GoblinFinalClass.java

```
final class GoblinFinalClass {
}
```

Модификатор `static`, примененный совместно с `final` к свойствам класса, используется для объявления констант. Такое свойство не может быть изменено после инициализации, и оно обязательно должно быть проинициализировано.

Компилятор подставляет реальное значение констант во все места программы, где они используются. Если значение константы пришлось впоследствии поменять, то нужно перекомпилировать все классы, которые ее используют, т. к. иначе там останется старое значение константы.

Пример объявления константы:

GoblinFinal.java

```
static final double PI = 3.141592653589793;
```

По соглашению об именовании имена констант записываются прописными буквами, а между словами ставится символ подчеркивания.

5.10. Инициализация полей

Полям можно присвоить начальное значение сразу при объявлении:

GoblinInitializeFields.java

```
public class GoblinInitializeFields {
    int x = 300;

    int y = x * 3; // Будет вычислено значение и присвоено 900.

    public static void main(String[] args) {
        GoblinInitializeFields spider = new GoblinInitializeFields();
        System.out.println(spider.x);
        System.out.println(spider.y);
    }
    // . . . Остальная часть класса
}
```

Инициализация происходит сверху вниз в порядке объявления в исходном коде. Сначала `x` присвоится 300, а затем `y` присвоится 300 * 3.

Если выражение инициализации не помещается в одну строку или требуется обработка ошибок, использование циклов и прочее, то можно задействовать блоки инициализации:

GoblinInitializeFields.java

```
class GoblinInitializeFields {
    // ... начало класса

    static int idCounter;

    int money;

    static {
        // Инициализация статических полей
        idCounter = 3;
    }

    {
        // Инициализации переменных экземпляров.
        money = 300;
    }
}
```

Блоки статической инициализации выполняются однократно при инициализации класса. Может быть несколько блоков инициализации, и в таком случае они будут выполняться в том порядке, в котором фигурируют в исходном файле сверху вниз.

Блоки инициализации экземпляров выполняются при создании экземпляров объекта. Может быть несколько блоков инициализации экземпляров, в таком случае они выполняются в порядке появления в исходном файле.

Не стоит слишком сильно перемешивать блоки инициализации, конструкторы и инициализацию при объявлении, иначе код может получиться запутанным и сложным для понимания. Однако при наличии всех этих видов инициализации в одном классе инициализация экземпляра происходит так:

1. Вычисляются аргументы конструктора. Если конструктор начинается с вызова другого конструктора этого класса, то вычисляются и его аргументы и т. д.
2. Если конструктор не начинается с вызова другого конструктора, то он начинается с явного или неявного вызова конструктора базового класса (будет описано в разделе про наследование), и выполняется этот конструктор базового класса.
3. Выполняются все выражения инициализации экземпляров и блоки инициализации экземпляров в том порядке, в котором они объявлены в исходном файле, словно они идут одним блоком.
4. Выполняется остаток тела конструктора.

Пример:

SpiderInitializeFields.java

```
package ru.urvanov.javaindynamics2022.classes;

public class SpiderInitializeFields {
    int x1;

    {
        x1 = 100;
    }

    int x2 = x1 + 1;

    SpiderInitializeFields() {
        x1 = 200;
    }

    int x3 = x1 + 2;

    {
        x1 = -100;
    }

    public static void main(String[] args) {
        SpiderInitializeFields s = new SpiderInitializeFields();
        System.out.println("x1=" + s.x1);
        System.out.println("x2=" + s.x2);
        System.out.println("x3=" + s.x3);
    }
}
```

Выведет в консоль:

```
x1=200
x2=101
x3=102
```

5.11. Задания

1. Создайте класс, описывающий содержимое ящика с сокровищами. В ящике может лежать, например, определенное количество золота, серебра, редких вещей, простых вещей, мусора и легендарных вещей. Опишите методы открытия ящика, взятия всех сокровищ и генерации нового содержимого.
2. Создайте класс для хранения информации о банковском счете клиента. Класс должен содержать дату открытия, сумму денег, процентную ставку, информа-

цию о клиенте, валюту счета и т. д. Добавьте методы пополнения счета, перевода на другой счет и снятия наличных.

3. Создайте класс для хранения и обработки транзакций домашней бухгалтерии. Придумайте и создайте классы для различных счетов доходов и расходов. Основной класс, хранящий транзакции, должен использовать классы счетов доходов и расходов.
4. Создайте класс или классы, который(ые) бы мог(ли) представлять из себя один пост в социальной сети. Пост может содержать текст, изображения (можете просто представлять их как массив байт `byte[]`), лайки и дизлайки, комментарии. Используйте разные типы методов и разные типы конструкторов.
5. Создайте класс, который бы представлял собой кандидата на выборах: Ф.И.О, дата рождения, биография, предвыборный план, процент поддержки населением и т. д.
6. Создайте класс для работы с таблицей рекордов видеоигры. Таблица должна хранить 10 лучших игроков по набранным очкам. В каждой строке таблицы должны храниться имя игрока и набранное количество очков. Добавьте метод для обработки нового результата игрока, который будет проверять, попал ли игрок в таблицу рекордов, и вставлять строчку в нужное место.
7. Создайте класс калькулятора с методами для сложения, вычитания, умножения, сохранения значения, смены знака и т. д. Постарайтесь использовать как можно большее количество типов и уровней доступа.

ГЛАВА 6

Аннотации

6.1. Объявление аннотаций

Аннотации представляют собой некую метаинформацию. Они не выполняют какого-либо действия сами по себе, но могут предоставлять дополнительную информацию, которая может быть использована компилятором, различными утилитами сборки и генерации кода. Также аннотации могут обрабатываться во время выполнения программы.

Аннотации предваряются знаком собачки. Пример часто используемой аннотации — `@Override`, которая указывает компилятору, что этот метод переопределяет базовый:

StandardAnnotation.java

```
@Override
public void someMethod1() {
    // ...
}
```

Аннотации могут иметь элементы:

StandardAnnotation.java

```
@SuppressWarnings(value = "unchecked")
public void method1() {
}
```

Если элементов много, то они разделяются запятой, если элемент только один, и его имя `value`, то его название можно не указывать:

StandardAnnotation.java

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

Предположим, что вы по традиции при объявлении каждого нового класса монстра записываете в комментариях информацию об авторах в таком виде:

IceElemental.java

```
package ru.urvanov.javaindynamics2022.annotation;

// Пример к разделу про аннотации
class IceElemental {
    // автор : Петров Иван
    // художник : Ишова Джамиля
    // звуки : Спиридонов Михаил
    // программирование : Пушкин Александр
    // дата создания : 30 марта 2016
    // описание : Ледяной элементаль - это сильное существо из льда
    // комментарии :

    // ... код
}
```

Вы можете записывать эту информацию с помощью аннотаций. Для этого вам сперва нужно ее объявить:

Monster.java

```
package ru.urvanov.javaindynamics2022.annotation;

@interface Monster {
    String author();
    String sprites();
    String sound();
    String code();
    String createdAt() default "0000-00-00";
    String description();

    // Пример использования массива
    String[] comments();
}
```

Теперь можно применить созданную аннотацию к нашему классу `IceElemental`:

IceElementalAnnotated.java

```
package ru.urvanov.javaindynamics2022.annotation;

@Monster(
        author="John Clark",
        sprites = "Izabella Simpson",
        sound = "Michael Lermontov",
        code = "Pushkin A.",
        createdAt = "2016-03-30",
        description = "Ice elemental is a powerful creature from ice",
        comments = {"lol", "gg", "Аффтар жжет"}
)
class IceElementalAnnotated {
}
```

Обратите внимание на запись элементов аннотации (author, sprites, sound) и на запись массива элементов с использованием фигурных скобок (comments).

Можно использовать предопределенную аннотацию `@Documented`, чтобы наша аннотация попадала в документацию, сгенерированную утилитой JavaDoc:

DocumentedMonster.java

```
package ru.urvanov.javaindynamics2022.annotation;

import java.lang.annotation.Documented;

@Documented
@interface DocumentedMonster {
    String author();
...
```

6.2. Предопределенные аннотации

`@Deprecated` указывает, что элемент устарел и не должен использоваться. Компилятор Java генерирует предупреждение, если вы используете класс, метод или поле, помеченные аннотацией `@Deprecated`. Устаревший элемент должен быть также помечен тегом JavaDoc `@deprecated`:

PredefinedAnnotations.java

```
/**
 * @deprecated
 * Объяснение, почему устарело,
 * что использовать вместо устаревшего метода
 */
 @Deprecated
 static void deprecatedMethod() { }
```

`@Override` указывает, что метод переопределяет метод базового класса. Эту аннотацию использовать необязательно, но рекомендуется.

PredefinedAnnotations.java

```
@Override
public int myMethod1(double x) {
    // ...
    Return 10;
}
```

Если метод с `@Override` не может корректно переопределить метод базового класса, то компилятор генерирует ошибку.

PredefinedAnnotations.java

```
// Указываем компилятору не генерировать предупреждение об использовании
// устаревшего метода.
@SuppressWarnings("deprecation")
void useDeprecatedMethod() {
    // Используем устаревший метод.
    objectOne.deprecatedMethod();
}
```

Каждое предупреждение принадлежит какой-либо категории. В спецификации Java описаны две категории. В `@SupressWarnings` можно указывать несколько категорий:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

Различные реализации компиляторов и различные IDE могут добавлять свои категории предупреждений. Неподдерживаемые названия категорий пропускаются при использовании `@SuppressWarnings`.

`@SafeVarargs` применяется к методу или конструктору и указывает, что код не осуществляет потенциально опасных операций со своим varargs-параметром (параметр, принимающий произвольное число параметров). Подробнее о нем рассказано в *разделе 18.31. "Подавление предупреждений для методов с произвольным количеством параметров с нематериализуемыми формальными параметрами" главы 18* .

`@FunctionalInterface` указывает, что это объявление типа будет функциональным интерфейсом. Функциональные интерфейсы появились в Java 8, про них рассказывается в *разделе 7.9. "Лямбда-выражения"*.

6.3. Мета-аннотации

Аннотации, применяемые к другим аннотациям, называются мета-аннотациями. В пакете `java.lang.annotation` есть несколько мета-аннотаций:

`@`Retention определяет, как аннотация будет сохранена:

- `RetentionPolicy.SOURCE` — аннотация будет только в исходном коде, и она будет игнорироваться компилятором.
- `RetentionPolicy.CLASS` — аннотация будет доступна компилятору, но будет игнорироваться виртуальной машиной Java.
- `RetentionPolicy.RUNTIME` — аннотация будет сохраняться JVM и будет доступна во время выполнения.

`@Documented` — указывает, что элементы, помеченные этой аннотацией, должны документироваться в JavaDoc. По умолчанию аннотации не включаются в документацию.

`@Target` — указывает, какие элементы можно помечать этой аннотацией:

- `ElementType.ANNOTATION_TYPE` — данная аннотация может быть применена к другой аннотации.
- `ElementType.CONSTRUCTOR` — может быть применена к конструктору.
- `ElementType.FIELD` — может быть применена к полю.
- `ElementType.LOCAL_VARIABLE` — может быть применена к локальной переменной.
- `ElementType.METHOD` — может быть применена к методу.
- `ElementType.MODULE` — может быть применена к модулю. Подробнее про модули рассказано в *главе 17 "Модули"*.
- `ElementType.PACKAGE` — может быть применена к пакету. Подробнее про пакеты рассказано в *главе 10 "Пакеты"*.
- `ElementType.PARAMETER` — может быть применена к параметрам метода.
- `ElementType.RECORD_COMPONENT` — может быть применена к записям. Подробнее про записи рассказано в *главе 12 "Записи"*.

- `ElementType.TYPE` — может быть применена к классу, интерфейсу, аннотации или перечислению.
- `ElementType.TYPE_PARAMETER` — может быть применена к аргументам типа. Подробнее про аргументы типа рассказано в *главе 18 "Обобщения"*.
- `ElementType.TYPE_USE` — может быть применена к использованию типа.

`@Inherited` — аннотация может быть унаследована от базового класса (по умолчанию не наследуются). Когда запрашивается аннотация класса, а у класса нет такой аннотации, то запрашивается аннотация базового класса. Она может быть применена только к классам.

`@Repeatable` — аннотация может быть применена несколько раз.

Допустим, мы хотим применить аннотацию `@Author` несколько раз для указания нескольких авторов:

DevilBird.java

```
@Author("Petya")
@Author("Vasya")
@Author("Suslik")
class DevilBird {
}
```

Тогда мы должны объявить такую аннотацию вот так:

Author.java

```
import java.lang.annotation.Repeatable;
@Repeatable(Authors.class)
@interface Author {
    String value();
}
```

Обратите внимание, что добавлена аннотация `@Repeatable` с указанием `Authors.class`, которую мы должны объявить как аннотацию с массивом — аннотация `Author`:

Authors.java

```
@interface Authors{
    Author[] value();
}
```

Теперь мы можем указывать аннотацию `@Author` столько раз, сколько захотим, для любого класса.

6.4. Задания

1. Создайте аннотации, с помощью которых можно бы было помечать объекты, принадлежащие игровому движку: объекты физического движка, объекты игрового мира, объекты механики игры и т. д. Постарайтесь использовать как можно большее количество типов аннотаций.
2. Создайте аннотации для ссылок на различные страницы в Интернете, относящиеся к классу: URL на GitHub, URL справочной информации и т. д. Сделайте так, чтобы они были доступны только в исходном коде.
3. Представьте, что вы разрабатываете банкомат. Создайте аннотации, с помощью которых можно бы было разделять классы на те, которые работают с внешними устройствами, и те, которые работают с внутренней логикой.

ГЛАВА 7

Вложенные классы и лямбда-выражения

7.1. Что такое вложенные классы

Язык программирования Java позволяет объявлять классы внутри другого класса. Такой класс называется вложенным классом:

OuterClass.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Вложенные классы бывают статическими и нестатическими. Вложенные классы, объявленные с ключевым словом `static`, называются статическими вложенными

классами (static nested classes). Вложенные классы, объявленные БЕЗ ключевого слова `static`, называются внутренними классами (inner classes).

OuterClass.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

class OuterClass {
    ...
    // Статический вложенный класс
    static class StaticNestedClass {
        ...
    }

    // Внутренний класс
    class InnerClass {
        ...
    }
}
```

Вложенный класс является членом того класса, в который он вложен. Внутренние классы имеют доступ к членам класса, в который они вложены, даже если эти члены объявлены с модификатором `private`. Для этого компилятор создает специальные методы доступа к этим полям, так что сама виртуальная машина принципов ООП не нарушает.

Как члены класса вложенные классы могут быть объявлены с ключевым словом `private`, `protected`, `public` или без модификатора доступа (`package-private`).

Внешний класс (`OuterClass`) может иметь только модификаторы `public` или `package-private`!

7.2. Для чего использовать вложенные классы

Причины для использования вложенных классов в Java:

- Логическая группировка классов, которые используются только в одном месте. Если класс используется только одним другим классом, то есть смысл вложить его в этот класс, чтобы обозначить их связь.
- Увеличение инкапсуляции. Если класс `B` должен обращаться к членам класса `A`, которые в противном случае были бы объявлены `private`, то имеет смысл вложить класс `B` в класс `A`, тогда эти члены можно будет объявить `private`, но `B` сможет к ним обращаться. В дополнение `B` можно будет скрыть от внешнего мира.
- Облегчение чтения и сопровождения кода. Маленькие классы можно вложить во внешние классы, ближе к месту использования.

7.3. Статические вложенные классы

Статические вложенные классы связаны со своим внешним классом так же, как статические методы и переменные.

И так же, как и статические методы, они не могут напрямую обращаться к переменным экземпляров и методам экземпляров внешнего класса, в который они вложены, они могут обращаться к переменным экземпляров и методам экземпляров внешнего класса только через ссылку на объект. Через ссылку на объект они могут обращаться к членам экземпляров внешнего класса независимо от их модификатора доступа.

Статические вложенные классы могут обращаться к `static` членам класса, в который они вложены, с любым модификатором доступа.

Пример:

OuterClassWithStaticNested.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

class OuterClassWithStaticNested {
    private static int a1;
    protected static int a2;
    static int a3;
    public static int a4;

    private int x1;
    protected int x2;
    int x3;
    public int x4;

    private static void privateStaticOuterMethod1() {}

    static void packagePrivateStaticOuterMethod1() {}

    protected static void protectedStaticOuterMethod1() {}

    public static void publicStaticOuterMethod1() {}

    private void privateInstanceOuterMethod1() {}

    void packagePrivateInstanceOuterMethod1() {}

    protected void protectedInstanceOuterMethod1() {}

    public void publicInstanceOuterMethod1() {}
```

```
    static class StaticNestedClass {
        public void method1() {
            // можно обращаться к приватным статическим членам.
            int y1 = a1;
            int y2 = a2;
            int y3 = a3;
            int y4 = a4;

            privateStaticOuterMethod1();
            packagePrivateStaticOuterMethod1();
            protectedStaticOuterMethod1();
            publicStaticOuterMethod1();

            //int x4 = x2;  НЕЛЬЗЯ! Только через ссылку на объект
        }

        public void method2(OuterClassWithStaticNested oc) {
            // К членам экземпляров только через ссылку.
            int z1 = oc.x1;
            int z2 = oc.x2;
            int z3 = oc.x3;
            int z4 = oc.x4;

            oc.privateInstanceOuterMethod1();
            oc.packagePrivateInstanceOuterMethod1();
            oc.protectedInstanceOuterMethod1();
            oc.publicInstanceOuterMethod1();
        }
    }
}
```

К статическим вложенным классам обращаются через имя их внешнего класса:

OuterClassWithStaticNestedClass.java#main

```
OuterClassWithStaticNested.StaticNestedClass obj1
        = new OuterClassWithStaticNested.StaticNestedClass();
obj1.method1();
```

Либо можно импортировать статический вложенный класс и обращаться к нему по имени:

OuterForImportStaticNested.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

public class OuterForImportStaticNested {
    public static final int X1 = 4;
```

```
    static class NestedClass {

    }
}
```

ImportStaticNested.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

import static
ru.urvanov.javaindynamics2022.nestedclass.OuterForImportStaticNested.X1;
import static
ru.urvanov.javaindynamics2022.nestedclass.OuterForImportStaticNested.NestedClass;

class ImportStaticNested {
    public static void main(String[] args) {
        System.out.println("X1=" + X1);
        NestedClass obj1 = new NestedClass();
    }
}
```

7.4. Внутренние классы

Внутренние классы связаны с экземпляром класса, в который они вложены, и они имеют доступ ко всем методам и полям внешнего класса независимо от модификатора доступа этих методов и полей.

Внутренние классы не могут объявлять внутри себя `static` члены (кроме констант), т. к. они связаны с экземпляром внешнего класса.

Пример:

OuterClassWithInnerClass.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

// Класс с внутренним классом
class OuterClassWithInnerClass {

    private int x1;

    private void method1() {}

    private static void method2() {}

    // Внутренний класс
    class InnerClass {

        // Внутренние классы имеют доступ ко всем членам внешнего класса
        // независимо от модификатора доступа.
        int y1 = x1;
```

```
        {
            method1();
            method2();
        }
        // ... и т. д.

        // Можно объявлять константы
        public static final int MY_CONSTANT = 34;

        // Никаких других статических членов объявлять
        // во внутренних классах нельзя!
        // Будет ошибка компиляции.

        // НЕЛЬЗЯ!
        //static {
        //
        //}
        //

        // НЕЛЬЗЯ
        //static void method1() {};
    }
}
```

Внутренние классы бывают:

- Нестатическими членами класса.
- Локальными классами.
- Анонимными классами.

В примере выше класс `InnerClass` будет нестатическим членом класса `OuterClassWithInnerClass`.

Хотя сериализация (запись в поток байт стандартными механизмами Java) конструкций с внутренними классами возможна, но на практике строго НЕ рекомендуется. Для работы с внутренними классами компилятор создает синтетические конструкции, которые могут сильно отличаться в различных реализациях компиляторов.

7.5. Внутренний класс, являющийся нестатическим членом класса

Внутренний класс будет нестатическим членом внешнего класса, если он объявлен прямо внутри тела внешнего класса без ключевого слова static, как уже показывалось в примере `OuterClassWithInnerClass`.

Такие внутренние классы обычно логически связаны со своим внешним классом. Они имеют доступ ко всем полям этого внешнего класса. Экземпляры этих классов

могут создаваться внутри внешнего класса. При достаточном уровне доступа они могут также создаваться другими классами из других пакетов.

Нестатические вложенные классы, являющиеся членами класса, могут быть объявлены с любым из модификаторов `private`, `protected`, `public` или без модификатора (`package-private`).

7.6. Локальные классы

Локальными классами называются классы, которые не являются членами какого-либо другого класса и имеют имя. Они объявляются внутри блока инструкций (нескольких инструкций, сгруппированных внутри одного блока с фигурными скобками). Их можно объявлять, например, в теле методов.

Локальные классы не могут иметь никаких модификаторов доступа: ни `private`, ни `protected`, ни `public`.

OuterClassWithLocalClass.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

// Пример класса с локальным классом
class OuterClassWithLocalClass {
    class Cyclic {}

    private int someField1;

    void foo1() {
        // Нельзя. Циклическое наследование.
        // Область видимости Cyclic для локального класса
        // включает само объявление класса.
        // class Cyclic extends Cyclic {}

        // Нельзя. Здесь еще LocalClass не объявлено.
        // LocalClass lc = new LocalClass();

        final int x1 = 100;
        int x2 = 200;
        int x3 = 300;

        // А вот так можно. LocalClass будет локальным классом,
        // доступным внутри метода, в котором объявлен
        class LocalClass {
            private void method1() {
                // Переменная x1  final.
                // Можно обращаться из локального класса
                System.out.println(x1);
```

```
                // Переменная x2 не меняется фактически, хотя и
                // не объявлена как final. Можно обращаться.
                System.out.println(x2);

                //System.out.println(x3); НЕЛЬЗЯ!.

                // Внутренние классы имеют доступ ко всем членам
                // своего внешнего класса.
                System.out.println(someField1++);
            }
        }

        // Еще один локальный класс
        class LocalClassB {
            void method2(LocalClass lc) {
                lc.method1(); // Можно. Так как они внутренние классы
                // одного и того же внешнего класса
            }
        }
        x3++;
        LocalClass lc = new LocalClass();
        lc.method1();
    }

    public static void main(String[] args) {
        OuterClassWithLocalClass oc = new OuterClassWithLocalClass();
        oc.foo1();
    }
}
```

7.7. Анонимные классы

Анонимные классы объявляются внутри выражения с ключевым словом `new`. Пример:

OuterClassWithAnonymousClass.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

// Пример класса с анонимным классом
class OuterClassWithAnonymousClass {

    interface MyInterface {
        void someMethod1();
    }

    public static void main(String[] args) {

        final int x1 = 100;
```

```
        int x2 = 200;
        int x3 = 300;

        // Объявляем новый класс, который реализует
        // интерфейс MyInterface, и не имеет своего имени.
        // Присваиваем переменной obj1 экземпляр этого класса
        MyInterface obj1 = new MyInterface() {
            private int x1;
            // ... еще поля.

            public void someMethod1() {
                // ... выполение действий.
                // x1 доступна, т. к. она объявлена как final
                System.out.println(x1);
                // x2 не объявлена как final, но она никогда
                // не меняет своего значения
                System.out.println(x2);
                // System.out.println(x3); НЕЛЬЗЯ, т. к.
                // x3 меняет значение
            }

            //... еще методы.
        };
        obj1.someMethod1();

        x3++; // x3 НЕ final
    }
}
```

Выражение анонимного класса состоит из:

- Операции `new`.
- Имени интерфейса для реализации или родительского класса. В данном примере используется интерфейс `MyInterface`.
- Скобки с аргументами для конструктора родительского класса. Анонимный класс не может объявить в своем теле новых конструкторов, т. к. у него нет имени.
- Тела класса.

Анонимный класс никогда не может быть `abstract` (абстрактные классы будут рассмотрены позже).

Анонимный класс всегда неявно `final` (финальный).

Анонимные классы могут обращаться к переменным метода, в котором они объявлены, если эти переменные объявлены как `final` (переменная x1 в примере), или они `final` по действию, т. е. фактически не меняются (переменная x2 в примере).

7.8. Затенение переменных

Если имя переменной в какой-либо области имеет такое же имя, что и переменная во внешней области, то она затеняет переменную из внешней области. Вы не можете обратиться к переменной из внешней области просто по имени. Пример ниже показывает, как нужно обращаться к затененной переменной:

ShadowClass.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

class ShadowClass {
    int x = -1;

    class FirstInnerClass {
        int x = 1;

        class SecondInnerClass {
            int x = 2;

            void method1(int x) {
                // Параметр метода method1
                System.out.println("x=" + x);

                // Член класса SecondInnerClass
                System.out.println("this.x=" + this.x);

                // Член класса FirstInnerClass
                System.out.println("FirstInnerClass.this.x="
                        + FirstInnerClass.this.x);

                // Член класса ShadowClass
                System.out.println("ShadowClass.this.x="
                        + ShadowClass.this.x);
            }
        }
    }

    public static void main(String[] args) {
        ShadowClass sc = new ShadowClass();
        ShadowClass.FirstInnerClass fic = sc.new FirstInnerClass();
        ShadowClass.FirstInnerClass.SecondInnerClass sic
                = fic.new SecondInnerClass();
        sic.method1(3);
    }
}
```

В этом примере параметр x метода method1 затеняет член класса SecondInnerClass, а x из SecondInnerClass затеняет x из FirstInnerClass, а x из FirstInnerClass закрывает x из ShadowClass соответственно.

Обратите внимание на обращение к x из различных уровней вложенности классов (x, this.x, SecondInnerClass.this.x).

7.9. Лямбда-выражения

Зачастую анонимный класс реализует интерфейс, который содержит только один абстрактный метод. В этом случае код можно написать еще более коротко и понятно, если использовать лямбда-выражения.

Интерфейс, который содержит только один абстрактный метод, называется функциональным интерфейсом. Функциональный интерфейс может также содержать любое количество статических методов и default-методов. Более подробно интерфейсы будут разобраны в соответствующем разделе. Пока достаточно знать, что интерфейсы содержат описания тех методов, которые должны быть реализованы в классах; речь идет о классах, которые реализуют этот интерфейс.

Существует также специальная аннотация @java.lang.FunctionalInterface, которую можно добавить к интерфейсу, чтобы показать, что он является функциональным интерфейсом, но это необязательно.

Лямбда-выражение состоит из:

1. Списка формальных параметров, разделенных запятой и заключенных в скобки. Если формальный параметр только один, то скобки можно опустить. Если формальных параметров нет, то используются просто пустые скобки. Тип формальных операторов указывать можно, но необязательно.
2. Токен стрелки "->".
3. Тело, состоящее из одного оператора/инструкции или из блока операторов/инструкций. В случае блока операторов и результата метода, отличного от void, для возврата значения используется ключевое слово return. Если тело состоит из одного выражения, то оператор return не нужен, а результатом будет результат этого выражения. Блок операторов может быть пустым.

Примеры лямбда-выражений:

```
() -> {}

x -> x * 2 - 3

x->{
    System.out.println(x);
    int z = x * 2 - 3;
    System.out.println(x);
    return z;
}
```

```
(x,y) -> x + y

x -> System.out.println(x)
```

Простой пример использования лямбда-выражений вместе с определением функциональных интерфейсов:

Lambda.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

class Lambda {
    interface Operation {
        int operation(int x, int y);
    }

    interface SimpleOperation {
        void simpleOperation(int x);
    }

    static int[] arrayOperation(
            int[] x,
            int[] y,
            Operation operation) {
        int[] result = new int[x.length];
        for (int n = 0; n < x.length; n++) {
            result[n] = operation.operation(x[n], y[n]);
        }
        return result;
    }

    static void arraySimpleOperation(
            int[] x,
            SimpleOperation simpleOperation) {
        for (int n = 0; n < x.length; n++) {
            simpleOperation.simpleOperation(x[n]);
        }
    }

    public static void main(String[] args) {
        // Пример сложения элементов массива:
        int[] resultSum = arrayOperation(
                new int[] {1, 0, 3},
                new int[] {2, 1, 0},
                (int x, int y) -> x + y);
        // Пример вычитания элементов массива:
        int[] resultMinus = arrayOperation(
                new int[] {1, 2, 3, 4},
```

```
                new int[] {2, 2, 3, 1},
                (x, y) -> x - y);

        // Вывод в консоль
        SimpleOperation writelnOperation
                = x -> System.out.println(x);
        System.out.println("Sum result:");
        arraySimpleOperation(resultSum, writelnOperation);
        System.out.println("Minus result:");
        arraySimpleOperation(resultMinus, writelnOperation);
    }
}
```

Так же как и локальные, и анонимные классы, лямбда-выражения могут обращаться к локальным переменным своей области видимости, если эти переменные объявлены `final` или являются `final` по действию. Лямбда-выражения в Java не порождают новую область видимости переменных, они используют ту же область видимости, что и метод.

LambdaScopeTest.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

class LambdaScopeTest {
    int x = 23;

    interface A {
        void method2(int y);
    };

    class InnerClass {
        int x = 10;

        public void method1(int x) {

            A a = z -> {
                System.out.println("z=" + z);

                // можем обратиться к параметру method1,
                // т. к. он final по действию,
                // т. е. его значение не меняется.
                System.out.println("x=" + x);

                System.out.println("this.x=" + this.x);
                System.out.println("LambdaScopeTest.this.x="
                        + LambdaScopeTest.this.x);
            };
```

```
            a.method2(x);
        }
    }

    public static void main(String[] args) {
        LambdaScopeTest lsc = new LambdaScopeTest();
        LambdaScopeTest.InnerClass ic = lsc.new InnerClass();
        ic.method1(44);
    }
}
```

Результатом работы этого класса будет:

```
z=44
x=44
this.x=10
LambdaScopeTest.this.x=23
```

Так как лямбда-выражения используют ту же область видимости переменных, что и метод, в котором они объявлены, они не могут вызывать затенения переменных. Если в коде выше мы объявим лямбда-выражение так:

```
A a = x -> {
...
```

то произойдет ошибка компиляции, т. к. переменная `x` в этой области видимости уже объявлена.

Тип результата у лямбда-выражения будет такой, какой ожидается в этом месте, поэтому лямбда-выражения можно использовать только там, где компилятор Java может определить его тип:

- ❒ Объявления переменных.
- ❒ Операции присваивания.
- ❒ Операторы `return`.
- ❒ Инициализации массивов.
- ❒ Аргументы конструкторов или методов.
- ❒ Тела лямбда-выражений.
- ❒ Условные операторы "?: ".
- ❒ Выражения приведения типа.

Лямбда-выражение можно сериализовать, если его аргументы и результат сериализуемые, однако так делать строго НЕ рекомендуется.

Пакет `java.util.function` содержит большое количество стандартных интерфейсов, которые специально предназначены для использования с лямбда-выражениями. Хотя в примерах выше мы всегда создавали свой интерфейс, но в реальных приложениях рекомендуется использовать подходящий стандартный интерфейс, который можно поискать в документации.

Некоторые из функциональных интерфейсов пакета `java.util.function`:

- `Consumer<T>` — содержит один метод с одним объектом в качестве параметра без результата метода.
- `Function<T, R>` — содержит один метод с одним объектом в качестве параметра, возвращающий другой объект в качестве результата.
- `Predicate<T>` — содержит один метод с объектом в качестве параметра, возвращающий результат `boolean`.
- `Supplier<T>` — содержит один метод без параметров, возвращающий объект.

7.10. Ссылки на методы

Если лямбда-выражение выполняет только вызов определенного метода или конструктора, то вместо этого выражения можно использовать ссылку на метод.

Посмотрите код:

MethodReference.java

```
class MethodReference {

    interface Operation {
        double method1(double x, double y);
    }

    static class OperationProvider {
        static double staticSum(double x, double y) {
            return x + y;
        }

        double instanceMinus(double x, double y) {
            return x - y;
        }
    }

    static double[] massOperation(
            double[] a, double[] b, Operation operation) {
        double[] result = new double[a.length];
        for (int n = 0; n < a.length; n++ ) {
            result[n] = operation.method1(a[n], b[n]);
        }
        return result;
    }

    public static void main (String[] args) {
        double[] a = {1.0, 2.2, 3.1};
```

```
        double[] b = {3.2, 4.1, 9.3};
        final OperationProvider myOperationProvider
                = new OperationProvider();

        massOperation(a, b, (x, y) -> OperationProvider.staticSum(x, y));
        massOperation(a, b, (x, y)
                -> myOperationProvider.instanceMinus(x, y));
    }
}
```

Здесь лямбда-выражения просто вызывают один метод. В таком случае можно использовать ссылки на методы:

MethodReference.java

```
// Ссылка на статический метод
massOperation(a, b, OperationProvider::staticSum);
// Ссылка на метод экземпляра
massOperation(a, b, myOperationProvider::instanceMinus);
```

Всего существует четыре вида ссылок на методы:

1. Ссылка на статический метод (`ContainingClass::staticMethodName`).
2. Ссылка на метод экземпляра определенного объекта (`containingObject::instanceMethodName`).
3. Ссылка на метод экземпляра произвольного объекта (`ContainingType::methodName`).
4. Ссылка на конструктор (`ClassName::new`).

Пример ссылки на статический метод:

MethodReference.java

```
massOperation(a, b, OperationProvider::staticSum);
```

Пример ссылки на метод определенного экземпляра:

MethodReferene.java

```
OperationProvider myOperationProvider = new OperationProvider();
massOperation(a, b, myOperationProvider::instanceMinus);
```

Пример ссылки на метод экземпляра произвольного объекта:

MethodReference.java

```
String[] stringArray = { "Джо", "Александр", "Марфа", "Святослав" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

Эквивалентное лямбда-выражение для `String::compareToIgnoreCase` будет иметь список формальных параметров (`String a, String b`), где `a` и `b` — произвольные имена. Эта ссылка на метод будет вызывать `a.compareToIgnoreCase(b)`.

Пример ссылки на конструктор:

ConstructorReference.java

```
package ru.urvanov.javaindynamics2022.nestedclass;

import java.util.function.Supplier;

class ConstructorReference {

    static ConstructorReference[] createArray(
            int count,
            Supplier<ConstructorReference> supplier) {
        ConstructorReference[] result = new ConstructorReference[count];
        for (int n = 0; n < count; n++) {
            result[n] = supplier.get();
        }
        return result;
    }

    public static void main(String[] args) {
        // Лямбда-выражение.
        ConstructorReference[] a1 = createArray(
                10,
                () -> new ConstructorReference());

        // То же самое, но с ссылкой на конструктор.
        ConstructorReference[] a2 = createArray(
                10,
                ConstructorReference::new);
    }
}
```

7.11. Когда использовать вложенные классы, локальные классы, анонимные классы и лямбда-выражения

Используйте локальные классы, если вы собираетесь создавать больше одного экземпляра класса, применять конструктор или планируете вводить новый именованный тип.

Используйте анонимные классы, если вам нужно добавлять поля или дополнительные методы.

Используйте лямбда-выражения, если вам нужен один экземпляр функционального интерфейса или если собираетесь передавать одно действие в другой метод, например обработчик события.

Используйте вложенный класс, если у вас такие же требования, как и для локального класса, но вы хотите сделать его более широкодоступным. Если вам нужен доступ к переменным внешнего класса, то используйте нестатический вложенный класс (внутренний класс), в противном случае используйте статический вложенный класс.

7.12. Задания

В большинстве случаев использовать вложенные классы не имеет особого смысла, кроме анонимных классов и лямбда-выражений. Однако технически можно написать всю иерархию классов внутри одного огромного класса и одного файла, но так делать не рекомендуется, поскольку поддерживать такое будет потом очень сложно.

1. Создайте класс "стол", а внутри него опишите класс "ящик стола" и класс "ножки стола". Напишите, например, методы открытия ящика и методы закрытия ящика. В качестве полей классов используйте ширину, длину, материал и т. д.

2. Создайте один класс `Engine`, который мог бы быть основным классом игрового движка. Внутри него создайте классы: игрок, преграда, дверь, монстр, ключ, сундук, золото. Постарайтесь использовать и локальные, и анонимные, и внутренние классы.

3. Представьте, что вы пишете калькулятор. Создайте класс `Calculator` с методом для выполнения математической операции над двумя числами. Создайте функциональный интерфейс, который будет параметром этого метода. Создайте как минимум четыре реализации этого класса. Используйте лямбда-выражения, ссылки на методы и локальные классы.

ГЛАВА 8

Интерфейсы

8.1. Теория

Интерфейс в Java — это некоторый контракт, описание методов, которые обязательно должны присутствовать в классе, реализующем этот интерфейс. Интерфейсы позволяют иметь несколько различных реализаций одного и того же действия, но выполняемого различными способами или с различными видами данных. Когда вы пишете какую-либо библиотеку, имеет смысл давать пользователям работать только с публичными интерфейсами. Тогда пользователи смогут заменить одну реализацию этих интерфейсов на другую без переписывания большей части кода, также вы сможете менять внутреннюю архитектуру библиотеки без необходимости переписывания зависящего клиентского кода.

Интерфейсы в Java являются ссылочными типами, как классы, но они могут содержать в себе только константы, сигнатуры методов, `default` методы (методы по умолчанию), `static` методы (статические методы) и вложенные типы. Тела могут быть только у статических методов и методов по умолчанию.

Нельзя создать экземпляр интерфейса. Интерфейс может быть только реализован каким-либо классом либо наследоваться другим интерфейсом.

Пример интерфейса, описывающий общие методы для всех монстров:

Monster.java

```
package ru.urvanov.javaindynamics2022.interfaces;

public interface Monster {

    // Объявляем константу
    int MONSTER_OBSTACLE_ID = 23;

    // методы

    boolean isSensitiveToSilver();

    void logic(VisibleWorld visibleWorld);

    void setPosition(double x, double y);

    boolean isAggressive();
}
```

Ключевое слово `public` означает, что интерфейс будет доступен из всех пакетов. Можно не указывать модификатор доступа, и тогда интерфейс будет `package-private`.

Любой интерфейс является абстрактным (`abstract`), нет никакого смысла писать дополнительно это слово при объявлении интерфейса, хотя компилятор и проглотит фразу `public abstract interface Monster`.

Любое объявление поля в интерфейсе является `public static final`. Нет никакой нужды дополнительно писать любое из этих ключевых слов. Обратите внимание на объявление `MONSTER_OBSTACLE_ID` в примере.

Любой нестатический и не `default` метод в интерфейсе является `public` и `abstract`. Нет никакого смысла писать любое из этих ключевых слов.

Любой метод в интерфейсе является `public`, нет нужды указывать этот модификатор.

Ключевое слово `abstract` для метода означает, что у него нет реализации, а ключевое слово `abstract` у всего интерфейса означает, что все методы экземпляров не имеют реализации (кроме статических методов и методов по умолчанию). Для классов ключевое слово `abstract` имеет примерно такое же действие, принцип его работы будет объяснен в *главе 9* про наследование.

Чтобы использовать интерфейс, нужно объявить класс, реализующий этот интерфейс, с помощью ключевого слова `implements`:

Leshy.java

```
package ru.urvanov.javaindynamics2022.interfaces;

public class Leshy implements Monster {
    private double x;
    private double y;

    @Override
    public boolean isSensitiveToSilver() {
        return false;
    }

    @Override
    public void logic(VisibleWorld visibleWorld) {
        // некая логика.
    }

    @Override
    public void setPosition(double x, double y) {
        this.x = x;
        this.y = y;
    }
```

```
    @Override
    public boolean isAggressive() {
        return false;
    }

}
```

Мы объявили класс `public`, чтобы он был доступен из всех пакетов, но можно объявить и без модификатора, если нужно.

В нашем классе `Leshy` мы объявили все методы из интерфейса и указали для них аннотацию `@Override`, чтобы показать компилятору, что мы действительно хотим реализовать методы из интерфейса. Аннотация `@Override` не обязательна, но она может помочь найти ошибку в случае неправильного объявления сигнатуры. Поскольку методы из интерфейса неявно имеют модификатор доступа `public`, то мы тоже должны объявить эти методы как `public`, иначе произойдет ошибка компиляции.

Класс, реализующий интерфейс, должен реализовать все методы из этого интерфейса либо быть `abstract` (будет описано в *главе 9* про наследование).

Интерфейс может расширять другие интерфейсы, наследуясь от них с помощью ключевого слова `extends`:

Elemental.java

```
public interface Elemental extends Monstr, Obstacle, Ghost, Enemy {
    // Константы и методы...
}
```

В этом примере интерфейс `Elemental` расширяет (наследуется от) интерфейсы `Monster`, `Obstacle`, `Ghost` и `Enemy`. Класс, реализующий интерфейс `Elemental`, должен будет реализовать все методы из `Elemental`, `Monster`, `Obstacle`, `Ghost` и `Enemy`.

Интерфейс может содержать в себе вложенные типы:

Elemental.java

```
public interface Elemental extends Monstr, Obstacle, Ghost, Enemy {
    // Константы и методы...

    class ElementalForce {
        private double x;
        private double y;

        public void someMethod1() {
        }
    }
}
```

Вложенные типы неявно характеризуются как `public` и `static`, нет смысла использовать эти модификаторы, хотя компилятор и допускает это.

Вложенный тип не может иметь модификатор доступа `private` или `protected`, иначе произойдет ошибка компиляции.

Интерфейс можно использовать как обычный тип: его можно указывать в качестве типа объявляемой переменной или в качестве параметра метода, и можно приводить переменную к типу интерфейса.

InterfaceAsParameter.java

```
package ru.urvanov.javaindynamics2022.interfaces;

public class InterfaceAsParameter {
    void someMethod(Obstacle obstacle, Enemy enemy) {
        if (obstacle instanceof Enemy) {
            // Приводим к интерфейсу Enemy
            Enemy obstacleEnemy = (Enemy) obstacle;
            // остальные действия.
        }
        // Объявляем переменную с типом интерфейса Monstr.
        Monster monstr = null;

        //...
    }
}
```

Предположим, что у вас есть интерфейс:

GreekMonster.java

```
package ru.urvanov.javaindynamics2022.interfaces;

public interface GreekMonster {
    boolean isSensitiveToSilver();
    void logic(VisibleWorld visibleWorld);
}
```

И класс, реализующий этот интерфейс:

Pan.java

```
package ru.urvanov.javaindynamics2022.interfaces;

public class Pan implements GreekMonster {
    @Override
```

```
    public boolean isSensitiveToSilver() {
        return false;
    }

    @Override
    public void logic(VisibleWorld visibleWorld) {
        // некая логика.
    }
}
```

Спустя какое-то время вам понадобилось добавить новый метод в интерфейс `GreekMonster`:

```
public interface GreekMonster {
    boolean isSensitiveToSilver();
    void logic(VisibleWorld visibleWorld);

    // Новый метод
    void doSomething();
}
```

Теперь класс `Pan` не может скомпилироваться, т. к. он уже не реализует полностью интерфейс `GreekMonster`. Если класс и интерфейс находятся в одном проекте, разрабатываемом вами, то новый метод легко можно туда добавить. Но мы не можем добавить новые методы в реализации этого интерфейса в других проектах, к которым у вас нет доступа.

Если вам понадобилось добавить новый метод в интерфейс, то вам и другим людям, использующим его, придется добавить реализацию этого метода во все классы, которые реализуют этот интерфейс, поэтому старайтесь тщательно продумывать варианты использования вашего интерфейса с самого начала и сразу описывать его полностью.

Если вам понадобилось добавить новый метод в интерфейс, то вы можете создать новый интерфейс, расширяющий старый и добавляющий этот метод:

ExtendedMonster.java

```
package ru.urvanov.javaindynamics2022.interfaces;

public interface ExtendedGreekMonster extends GreekMonster {
    void doSomething();
}
```

Теперь пользователи смогут выбрать, остаться ли им на старом интерфейсе либо перейти на новый и получить дополнительные возможности.

Или вы можете использовать методы по умолчанию (default methods):

GreekMonster.java

```
public interface GreekMonster {
    boolean isSensitiveToSilver();
    void logic(VisibleWorld visibleWorld);

    // Новый метод
    default void doSomething() {
        // Некий код
    }
}
```

Для методов по умолчанию нужно обязательно указывать реализацию. Эта реализация может вызывать другие методы из этого интерфейса и интерфейсов, от которых он наследуется.

Теперь классы, реализующие интерфейс `GreekMonster`, и интерфейсы, расширяющие его, получат метод `doSomething()`, и им не нужно будет изменять либо перекомпилировать себя.

Когда вы расширяете своим интерфейсом другой интерфейс, который содержит `default` метод, то вы можете:

- Не упоминать этот метод, и тогда ваш интерфейс унаследует его.
- Переобъявить `default`-метод, что сделает его `abstract`.
- Объявить свой `default`-метод с теми же параметрами и именем, что переопределит его.

Интерфейс может содержать статические методы, как и класс. Статические методы относятся к самому типу и вызываются через него.

Пример:

CelticMonster.java

```
package ru.urvanov.javaindynamics2022.interfaces;

public interface CelticMonster {
    boolean isSensitiveToSilver();
    void logic(VisibleWorld visibleWorld);

    static void logicForSensitiveToSilver(CelticMonster[] celticMonsters,
                                          VisibleWorld visibleWorld) {
        for (CelticMonster celticMonster : celticMonsters) {
            if (celticMonster.isSensitiveToSilver()) {
                celticMonster.logic(visibleWorld);
            }
        }
    }
}
```

8.2. Задания

1. Придумайте интерфейс, который содержал бы спецификацию приложения, управляющего шлагбаумом перед въездом на территорию многоквартирного дома: открытие шлагбаума по ключу, добавление нового ключа, удаление ключа.
2. Придумайте интерфейсы приложения, которое могло бы описывать панель управления температурой в помещении: датчики температуры, выставление нужной температуры, отключение, включение, установка границ температуры.

ГЛАВА 9

Наследование

9.1. Введение

В предыдущих разделах я уже несколько раз упоминал наследование. Настало время подробно разобрать эту вещь (рис. 9.1).

В Java класс может наследоваться от другого класса, получая его методы и поля, а этот родительский класс, в свою очередь, может наследоваться от еще одного класса и т. д. В Java нет множественного наследования классов. Один класс может наследоваться напрямую только от одного другого класса.

Класс, который наследуется от другого класса, называется подклассом (subclass), дочерним классом (child class), потомком или расширенным классом (extended class).

Класс, от которого наследуется дочерний класс, называется родительским классом (parent class), предком, суперклассом (superclass) или базовым классом (base class).

В самой вершине иерархии наследования находится класс `Object`, от которого наследуются все классы, для которых явно не указан предок. Таким образом, все классы (кроме самого `Object`) напрямую или через какое-либо количество уровней наследования происходят от класса `Object`.

Идея наследования классов состоит в том, что, когда вы хотите создать новый класс, например `Goblin`, и уже существует какой-нибудь класс, который реализует часть функциональности, необходимой нашему классу, например `Monster`, вы можете указать этот класс в качестве родительского, унаследовав таким образом все

его члены (поля, вложенные классы и методы экземпляров). Конструкторы не наследуются и не являются членами классов, но можно вызвать конструктор базового класса из конструктора дочернего.

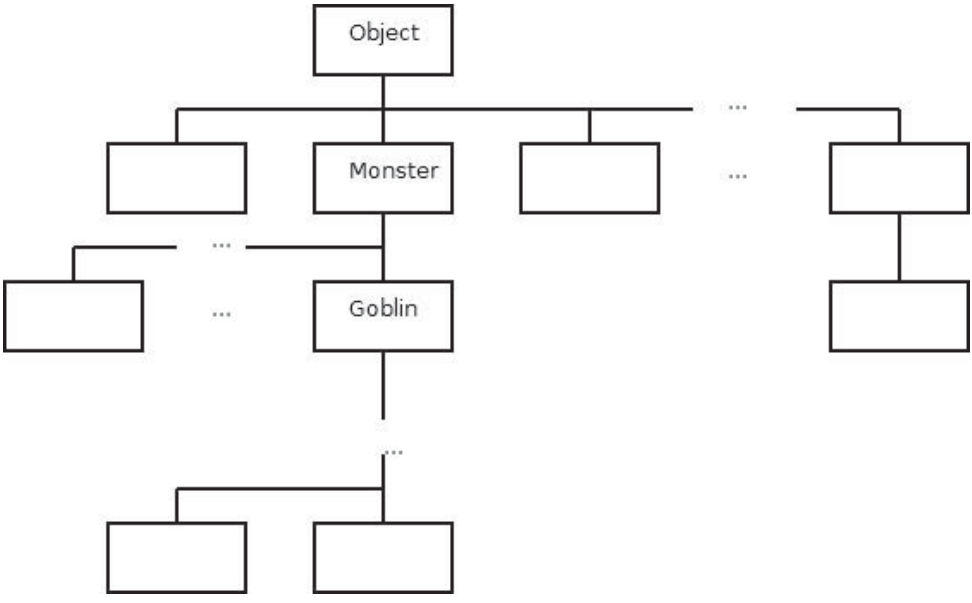


Рис. 9.1. Пример иерархии классов в Java

Дочерний класс наследует все `public` и `protected` члены своего родителя независимо от пакета, в котором расположен родительский класс. Если дочерний и родительский классы находятся в одном пакете, то дочерний наследует также `package-private` члены своего родителя.

- Унаследованные поля можно использовать напрямую, как и все другие поля.
- Можно объявить в дочернем классе поле с таким же именем, как и поле в родительском классе, тогда это поле скроет (hide) поле родительского класса (НЕ рекомендуется так делать).
- В дочернем классе можно объявлять поля, которых нет в родительском классе.
- Унаследованные методы можно использовать напрямую.
- Можно объявить метод экземпляров в дочернем классе с точно такой же сигнатурой, что и метод экземпляров в родительском классе, тогда этот метод переопределит (override) метод суперкласса.
- Можно объявить в дочернем классе статический метод с точно такой же сигнатурой, что и статический метод в родительском классе, тогда этот метод скроет (hide) метод родительского класса.
- В дочернем классе можно объявлять новые методы, которых нет в родительском классе.

- В дочернем классе можно объявить конструктор, который будет явно (с помощью ключевого слова `super`) или неявно вызывать конструктор базового класса.

Дочерний класс не наследует `private` члены родительского класса, однако, если в родительском классе есть `protected`, `public` или `package-private` (для случая нахождения дочернего и родительского класса в одном пакете) методы для доступа к `private` полям, они могут использоваться дочерним классом.

9.2. Приведение типов

Посмотрите на создание экземпляра объекта `Goblin`:

TypeConversion.java

```
Goblin obj = new Goblin();
```

Мы знаем, что `Goblin` наследуется от `Monster`, который, в свою очередь, наследуется от `Object`. Таким образом, `Goblin` является и `Monster`, и `Object`. Экземпляр класса `Goblin` можно использовать в любом месте, где ожидается экземпляр класса `Monster` или `Object`.

Но `Monster` необязательно должен являться `Goblin`. Экземпляр класса `Monster` МОЖЕТ быть экземпляром класса `Goblin`, а может быть экземпляром самого `Monster` или любого другого дочернего класса.

Неявное приведение типов используется, когда мы приводим дочерний класс к классу выше по наследованию:

TypeConversion.java

```
Object obj1 = new Goblin();
Monster obj2 = new Goblin();
```

При обратном приведении типов нужно указывать явное приведение типов. Таким образом мы обещаем компилятору, что в этом объекте действительно будет содержаться экземпляр нашего дочернего класса:

TypeConversion.java

```
Goblin goblin = (Goblin) obj;
```

Компилятор вставит проверку на соответствие типа в эту операцию, которая будет проверять, что `obj` действительно ссылается на экземпляр класса `Goblin`. Если `obj` ссылается на объект, НЕ являющийся экземпляром класса `Goblin` или его потомков, то возникнет исключение `java.lang.ClassCastException`.

9.3. Переопределение (overriding) и скрытие (hiding) методов

Если метод экземпляров в дочернем классе с той же сигнатурой и с тем же типом результата, что и метод в родительском классе, то он переопределяет (override) этот метод. Эта возможность позволяет указать в качестве родительского класса наиболее подходящий, а затем изменить его поведение, переопределив некоторые методы и добавив новые.

Можно поставить аннотацию `@Override` у метода дочернего класса, чтобы указать компилятору, что вы хотите переопределить метод базового класса. В этом случае компилятор будет генерировать ошибку, если не найдет подобного метода в родительском классе. Более подробно можете прочесть про эту аннотацию в соответствующем разделе.

Переопределяющий метод может также вернуть тип, являющийся потомком типа, который возвращается переопределяемым методом. Этот дочерний тип называется ковариантным возвращаемым типом (covariant return type).

Если дочерний класс определяет статический метод с той же сигнатурой, что и метод в родительском классе, то этот метод скрывает (hide) метод родительского класса.

Скрытие статических методов и переопределение методов экземпляров — это не одно и то же:

- При вызове переопределенного метода экземпляра будет вызван метод дочернего класса.
- Версия скрытого статического метода зависит от того, откуда будет производиться вызов: из суперкласса или дочернего класса.

Monster.java

```
package ru.urvanov.javaindynamics2022.inheritance;

class Monster {

    void instanceMethod() {
        System.out.println("Monster instance method");
    }

    static void staticMethod() {
        System.out.println("Monster static method");
    }
}
```

Goblin.java

```
package ru.urvanov.javaindynamics2022.inheritance;

public class Goblin extends Monster {
```

```
    @Override
    void instanceMethod() {
        System.out.println("Goblin instance method");
    }

    static void staticMethod() {
        System.out.println("Goblin static method");
    }
}
```

OverridingHiding.java

```
package ru.urvanov.javaindynamics2022.inheritance;

public class OverridingHiding {
    public static void main(String[] args) {
        System.out.println("Begin test:");

        Monster monster = new Monster();
        Monster goblin = new Goblin();

        monster.instanceMethod();
        goblin.instanceMethod();

        Monster.staticMethod();
        Goblin.staticMethod();
    }
}
```

Этот код выведет в консоль:

```
Begin test:
Monster instance method
Goblin instance method
Monster static method
Goblin static method
```

Как видно из результата, при вызове метода экземпляра вызывается переопределивший метод дочернего класса. Обратите внимание, что переменная `goblin` — типа `Monster`, но фактически ссылается на экземпляр его дочернего класса `Goblin`, чей переопределивший метод `instanceMethod()` и вызывается. Статический метод вызывается у того класса, у которого он вызван.

При вызове метода объекта Java вызывает подходящий метод экземпляра класса, на который ссылается переменная, но не метод типа, с которым переменная объявлена. Это называется полиморфизмом в Java.

Методы по умолчанию (`default`-методы) и абстрактные методы интерфейсов наследуются так же, как и методы экземпляров.

Если несколько супертипов класса или интерфейса объявляют методы с одинаковыми сигнатурами, то компилятор Java использует следующие правила для разрешения конфликтов:

Правило 1

Методы экземпляров имеют преимущество перед методами по умолчанию (`default` методами). Это вполне логично, т. к. методы по умолчанию появились только в Java 8, и они не должны ломать старый код, где уже могут быть объявлены методы с такими же сигнатурами.

Правило 2

Методы, которые уже были переопределены другими кандидатами, игнорируются. Это может произойти в том случае, если несколько интерфейсов наследуются от одного и того же родительского интерфейса, и класс реализует оба эти интерфейса.

Второе правило лучше всего рассмотреть на примере:

Leshy.java

```
package ru.urvanov.javaindynamics2022.inheritance;

interface Obstacle {
    default void writeName() {
        System.out.println("Obstacle");
    }
}

interface ForestDeity extends Obstacle {
    default void writeName() {
        System.out.println("Forest deity");
    }
}

interface SlavicBeast extends Obstacle {}

public class Leshy implements SlavicBeast, ForestDeity {
    public static void main(String[] args) {
        Leshy leshy = new Leshy();
        leshy.writeName();
    }

}
```

Класс `Leshy` получает метод `writeName` из `SlavicBeast`, который, в свою очередь, наследует этот метод от `Obstacle`, и из `ForestDeity`, который переопределяет метод, унаследованный от `Obstacle`. Пример выше выведет "Forest deity", т. к. `writeName` из `Obstacle`, который имеет `SlavicBeast`, уже переопределен в `ForestDeity`, который тоже наследует этот метод от `Obstacle`.

Правило 3

Если конфликт происходит между двумя независимо объявленными методами по умолчанию, или независимо объявленный метод по умолчанию конфликтует с другим независимо объявленным абстрактным методом, то компилятор генерирует ошибку компиляции. В таком случае нужно явно переопределить этот метод:

Domovoy.java

```
package ru.urvanov.javaindynamics2022.inheritance;

interface HouseSpirit {
    default void writeName() {
        System.out.println("House spirit");
    }
}

public class Domovoy implements Obstacle, HouseSpirit {

    // Возникает конфликт методов по умолчанию
    // из двух интерфейсов. Поэтому мы должны
    // обязательно переопределить метод
    @Override
    public void writeName() {
        System.out.println("Domovoy");
    }

    public static void main(String[] args) {
        Domovoy dryad = new Domovoy();
        dryad.writeName();
    }
}
```

Правило 4

Унаследованные методы экземпляров могут переопределять абстрактные методы интерфейсов:

EuropeanDragon.java

```
package ru.urvanov.javaindynamics2022.inheritance;

interface EuropeFolklore {
    void doIt();
}

class Dragon {
    public void doIt() {
```

```
        System.out.println("doIt");
    }
}

public class EuropeanDragon extends Dragon implements EuropeFolklore{
    // Метод doIt из интерфейса EuropeFolklore
    // уже переопределен методом из класса Dragon
    public static void main(String[] args) {
        EuropeanDragon europeanDragon = new EuropeanDragon();
        europeanDragon.doIt();
    }
}
```

Статические методы никогда НЕ наследуются.

Чтобы подвести итог переопределению (overriding) и скрытию (hiding), приведу здесь табл. 9.1.

Таблица 9.1. *Переопределение методов*

	Метод экземпляров родителя	**Статический метод родителя**
Метод экземпляров дочернего класса	Переопределяет (override)	Ошибка компиляции
Статический метод дочернего класса	Ошибка компиляции	Скрывает (hide)

9.4. Использование ключевого слова super

Если ваш метод переопределяет метод базового класса, то вы не можете вызвать метод базового класса напрямую по имени. Вам нужно использовать ключевое слово `super`.

Если ваш класс определяет поле с тем же именем, что и поле в родительском классе, пусть даже с другим типом, то оно скрывает (hide) поле родительского класса. В этом случае вы не можете напрямую обратиться к полю родительского класса по имени. Если все же нужно обратиться к этому полю родительского класса, то нужно использовать ключевое слово `super`.

Пример:

Monster.java

```
package ru.urvanov.javaindynamics2022.inheritance;

class Monster {

    double gold = 10.0;
    int ammo = 40;
    double health = 100.0;
```

```
    void walk() {
        System.out.println("Monster walk");
        System.out.println("Monster gold = " + gold);
        System.out.println("Monster ammo = " + ammo);
        System.out.println("Monster health = " + health);
    }

    // ...
}
```

Goblin.java

```
package ru.urvanov.javaindynamics2022.inheritance;

public class Goblin extends Monster {

    double gold = 20.0;
    int trunks = 2;

    void walk() {
        System.out.println("Goblin walk");
        System.out.println("Goblin gold = " + gold);
        // Мы можем обратиться к скрытому полю родительского класса:
        System.out.println("Monster gold = " + super.gold);
        System.out.println("Goblin trunks = " + trunks);
        super.walk();
    }
}
```

Super.java

```
package ru.urvanov.javaindynamics2022.inheritance;

/**
* Пример с ключевым словом super
*/
public class Super {
 public static void main(String[] args) {
   Goblin goblin = new Goblin();
   goblin.walk();
 }
}
```

Результат в консоли:

```
Goblin walk
Goblin gold = 20.0
```

```
Monster gold = 10.0
Goblin trunks = 2
Monster walk
Monster gold = 10.0
Monster ammo = 40
Monster health = 100.0
```

С помощью ключевого слова `super` можно вызвать конструктор родительского класса в классе-потомке:

Monster.java

```
class Monster {

    double gold = 10.0;
    int ammo = 40;
    double health = 100.0;

    Monster() {

    }

    Monster(double gold, int ammo, double health) {
        this.gold = gold;
        this.ammo = ammo;
        this.health = health;
    }
}
```

Goblin.java

```
package ru.urvanov.javaindynamics2022.inheritance;

public class Goblin extends Monster {

    double gold = 20.0;
    int trunks = 2;

    Goblin() {

    }

    Goblin(
            double goblinGold,
            double monsterGold,
            int ammo,
            double health,
            int trunks) {
```

```
        super(monsterGold, ammo, health);
        this.gold = goblinGold;
        this.trunks = trunks;
    }
}
```

Вызов конструктора суперкласса должен быть первой инструкцией в конструкторе дочернего класса. Можно вызвать конструктор суперкласса без параметров (конструктор по умолчанию):

```
super()
```

Если вы не вставили ни одного явного вызова конструктора родительского класса, то компилятор Java автоматически добавит вызов конструктора родительского класса без параметров (конструктора по умолчанию). Если он недоступен из-за модификатора доступа или конструктора без параметров нет в родительском классе, то возникнет ошибка компиляции.

При создании экземпляра любого объекта происходит цепочка вызовов от конструктора создаваемого объекта до конструктора класса `Object`. Это называется цепочкой вызова конструкторов (constructor chaining).

9.5. Общий предок Object и его методы

Класс `java.lang.Object` является прямо или через череду других классов суперклассом для всех других классов в языке Java.

Он имеет некоторое количество методов экземпляров, которые от него наследуют все его потомки.

Метод `clone()`:

```
protected Object clone()
                throws CloneNotSupportedException
```

Метод `clone()` создает копию объекта, если класс объекта реализует интерфейс `java.lang.Cloneable`, а в противном случае генерирует исключение `CloneNotSupportedException`. Сам класс `Object` не реализует интерфейс `Cloneable`, поэтому, если вы хотите воспользоваться методом `clone()`, вы должны реализовать интерфейс `Cloneable` в своем классе. Сам интерфейс `Cloneable` не содержит никаких методов, а просто является меткой того, что объект, который его реализует, поддерживает метод `clone()`.

По соглашению объект, который возвращается методом `clone()`, должен быть того же типа, что и объект, у которого он вызван, и этот возвращенный объект должен быть копией исходного, а не исходным (`obj.clone() != obj`), но метод `equals` должен возвращать `true (obj.equals(obj.clone()) == true)`.

По соглашению классы, которые реализуют интерфейс `Cloneable` и для объектов которых планируется использовать метод `clone()`, должны переопределить этот метод с `protected` на `public`:

WillOrTheWisp.java

```
package ru.urvanov.javaindynamics2022.inheritance;

// Класс реализует метод Cloneable
// В противном случае метод clone() будет генерировать
// исключение CloneNotSupportedException.
public class WillOrTheWisp implements Cloneable {

    private int x;
    private double y;

    // Переопределяем метод clone()
    // с protected на public.
    // И возвращаемый методом тип делаем более специфичным
    public WillOrTheWisp clone()
            throws CloneNotSupportedException {
        // Мы должны вызвать метод базового класса,
        // чтобы гарантировать, что возвращаемое значение
        // будет именно того типа, у экземпляра которого
        // вызван метод clone().
        WillOrTheWisp result = (WillOrTheWisp) super.clone();

        // Метод clone() в классе Object копирует только значения полей.
        // Если у нас есть ссылочные типы, то мы должны создать копии
        // для них самостоятельно
        // ...

        return result;
    }

    public static void main(String[] args)
            throws CloneNotSupportedException {
        WillOrTheWisp willOrTheWisp = new WillOrTheWisp();
        willOrTheWisp.x = 100;
        willOrTheWisp.y = 140.33;
        WillOrTheWisp clonedObject = willOrTheWisp.clone();
        System.out.println("x = " + clonedObject.x);
        System.out.println("y = " + clonedObject.y);
    }
}
```

Реализация метода `clone()` в `Object` создает копию объекта, просто копируя поля, что вполне нормально для полей примитивных типов. Для полей ссылочных типов и различных сложных структур (деревьев, массивов и прочих) вы должны сами создать копии при переопределении метода `clone()`.

Метод `equals()`:

```
public boolean equals(Object obj)
```

Этот метод проверяет равенство двух объектов. Реализация в классе `Object` возвращает `true` только тогда, когда две ссылки относятся к одному и тому же экземпляру объекта. Если нужно сравнение по содержимому полей, то нужно переопределить этот метод в своем классе. При переопределении метода `equals()` нужно обязательно переопределить и метод `hashCode()`, который должен возвращать одинаковое значение для любых объектов, для которых `equals()` вернул `true`.

Метод `finalize()`:

```
protected void finalize()
                throws Throwable
```

Метод `finalize()` может быть вызван, когда сборщик мусора решит удалить ваш объект. Реализация `finalize()` в `Object` не делает ничего. Вам не стоит полагаться на вызов этого метода для очищения ресурсов, т. к. он может никогда не вызваться. Я, честно говоря, смутно представляю, где вообще может понадобиться использование этого метода. Начиная с Java 9, метод объявлен устаревшим. С Java 18 объявлен как метод для удаления в последующих релизах.

Метод `getClass()`:

```
public final Class<?> getClass()
```

Возвращает объект `java.lang.Class`, который содержит информацию о классе, для которого вызван метод.

Настоящий результат этого метода `Class<? extends |X|>` (это будет описано в последующих разделах), где `|X|` — выведенный тип для выражения, на котором вызван метод getClass. Например, для следующего кода не нужно приведения типа:

```
Number n = 0;
Class<? extends Number> c = n.getClass();
```

Вы не можете переопределить этот метод.

Метод `hashCode()`

```
public int hashCode()
```

Возвращает хеш-код для объекта. Этот хеш-код используется в хеш-таблицах вроде HashMap. По соглашению о методах `equals` и `hashCode`, если метод `equals()` для двух объектов возвращает `true`, `hashCode()` для них должен возвращать одинаковое значение. Вызовы `hashCode()` должны возвращать одно и то же значение в течение всего времени выполнения программы.

Метод `toString()`:

```
public String toString()
```

Возвращает человекочитаемое строковое представление объекта. Старайтесь всегда переопределять этот метод в своих классах, т. к. он очень полезен во время отладки. Реализация в `Object` возвращает следующее значение:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Методы `notify()`, `notifyAll()`, `wait()`, `wait(long timeout)`, `wait(long timeout, int nanos)` класса `Object` представляют собой простую реализацию семафоров. Полезны при доступе нескольких потоков к одному ресурсу. Более подробно эти методы будут рассмотрены в *разделе 23.12 "Защищенные блокировки (guarded blocks)"*.

9.6. Ключевое слово final и неизменяемые классы

Если при определении метода указать его как `final`, то этот метод нельзя будет переопределять в классах-потомках. Это может быть полезно для методов, используемых конструкторами, т. к. переопределение метода, который используется конструктором, может привести к нежелательным последствиям. Методы, используемые в конструкторе объекта, старайтесь делать с модификатором `final`.

Можно сделать весь класс `final`, что запретит указывать его в качестве родительского класса. Это может быть полезно для создания неизменяемых классов (вроде `String`). Неизменяемые классы (immutable) могут безопасно использоваться при многопоточном программировании.

```
final class Goblin {
}

// ОШИБКА! Нельзя наследоваться от класса final
//class HugeGoblin extends Goblin {
//}
```

9.7. Абстрактные методы и классы

Если класс объявлен с ключевым словом `abstract`, то он называется абстрактным классом. Он может иметь абстрактные методы, а может и не иметь их.

BaseMonster.java

```
package ru.urvanov.javaindynamics2022.inheritance;

// Пример абстрактного класса
abstract class BaseMonster {

    abstract void myAbstractMethod(int myParam1, double myParam2);
}
```

Абстрактным называется метод, объявленный с ключевым словом `abstract` и не имеющий тела метода.

BaseMonster.java

```
abstract void myAbstractMethod(int myParam1, double myParam2);
```

Если в классе есть абстрактные методы, то он ДОЛЖЕН быть объявлен абстрактным.

Нельзя создать экземпляр абстрактного класса, но можно указать абстрактный класс в качестве базового.

Класс, который наследуется напрямую от абстрактного, должен либо дать реализацию всем его абстрактным методам, либо сам быть абстрактным классом.

Абстрактные классы очень похожи на интерфейсы, но абстрактные классы могут иметь поля экземпляров и методы с модификатором доступа, отличным от `public`. Также вы можете наследовать свой класс только от одного другого класса, но реализовать возможно любое количество интерфейсов.

Выбрать между абстрактным классом и интерфейсом бывает довольно сложно. Старайтесь руководствоваться правилами, описанными ниже.

Используйте абстрактные классы, если:

- Вы хотите использовать общий код в нескольких близкородственных классах.
- Вы ожидаете, что классы, которые будут наследоваться от вашего абстрактного класса, имеют большое количество общих полей или требуют использования модификаторов доступа, отличных от `public`.
- Вам нужно объявить поля экземпляров или класса, а не только константы.

Используйте интерфейсы в следующих ситуациях:

- Вы ожидаете, что интерфейсами будут реализовываться не связанные друг с другом классы. Интерфейсы `Comparable` и `Cloneable`, например, реализуют очень большое количество совершенно разных классов.
- Вы хотите указать поведение определенного типа, но вам абсолютно не важно, кто будет реализовывать это поведение.
- Вам нужно множественное наследование типов.

Для примера абстрактного класса представьте ситуацию, что вам нужно реализовать несколько различных видов монстров: `Goblin`, `Hobgoblin`, `Orc`, `Gremlin` и `Genie`. Каждый из этих монстров имеет свои различные особенности, которые будут реализовываться в соответствующем классе, но все эти монстры умеют ходить, у них есть координаты в пространстве, и у каждого из них есть уровень здоровья. В этом случае можно заложить умение ходить, координаты и уровень здоровья в базовом классе `Monster`, который делается абстрактным, и в этом классе объявить абстрактные методы для управления повадками и прочими характеристиками, реализации которых будут в соответствующих дочерних классах.

9.8. Задания

1. Придумайте иерархию классов для библиотеки пользовательского интерфейса: панели, кнопки, переключатели, галочки, надписи, ползунки, полосы прогресса, рамки. Иерархия классов должна начинаться с самого общего абстрактного компонента.

2. Придумайте иерархию классов для объектов видеоигры: различные враги, препятствия, объекты физического движка, абстрактные объекты и т. д. У вас может получиться не одно дерево иерархии, а несколько.
3. Придумайте иерархию классов (хотя бы из двух-трех) автомобилей. Например: легковой автомобиль ➢ грузовой автомобиль ➢ автопоезд, легковой автомобиль ➢ амфибия.

ГЛАВА 10

Пакеты

10.1. Теория

Пакеты позволяют организовать связанные между собой типы в отдельные пространства имен. Это позволяет типам (классам, интерфейсам и перечислениям) в разных пакетах иметь одинаковые имена, но при этом не порождать конфликтов и неопределенностей.

Чтобы указать пакет, к которому относится класс, нужно использовать ключевое слово `package` в самом верху файла с классом. Пример:

Main.java

```
package ru.urvanov.javaindynamics2022.packageexamples;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Большинство реализаций Java рассчитаны на иерархическую файловую систему при организации файлов-исходников и файлов .class, хотя спецификация Java НЕ требует этого.

Чтобы успешно запустить скомпилированный `Main.class` из вышеприведенного примера, нужно создать структуру каталогов, повторяющую структуру пакета:

```
.
-> ru
    -> urvanov
        -> javaindynamics2022
            -> packageexamples
                -> Main.java
                -> Main.class
```

Запускать класс нужно из каталога, в котором находится каталог ru (в случае Linux нужно использовать знак "/" вместо "\"):

```
javac ru\urvanov\javaindynamics2022\packageexamples\Main.java
java ru.urvanov.javaindynamics2022.packageexamples.Main
```

Обратите внимание на используемое имя пакета: "`ru.urvanov.javaindynamics2022.packageexamples`". По соглашению о кодировании имена пакетам даются в соответствии с правилами:

- Пакеты записываются строчными буквами, чтобы избежать конфликтов с именами классов и интерфейсов.
- Используется обратная запись домена организации, например для urvanov.ru получится ru.urvanov.
- После обратной записи домена добавляется имя пакета или имя проекта с именем пакета внутри проекта. В примере выше имя проекта — `javaindynamics2022`, а имя пакета внутри проекта — `packageexamples`.
- Чтобы избежать конфликтов имен внутри организации, можно добавлять между именем домена и именем проекта имя региона или отдела. Например:

  `ru.urvanov.moscow.javaindynamics2022.packageexamples.`
- Пакеты внутри Java начинаются с java или javax.
- Дефис в имени домена заменяется на символ подчеркивания.
- Если имя домена содержит ключевое слово, то к имени пакета добавляется подчеркивание в конце.
- Если имя домена начинается с цифры, то добавляется подчеркивание в начало имени пакета.

Примеры — в табл. 10.1.

Таблица 10.1. *Именование пакета*

Имя домена	Префикс имени пакета
hyphenated-name.example.org	org.example.hyphenated_name
example.int	int_.example
123name.example.com	com.example._123name

Если же пакет не указывать, то используется пакет по умолчанию. Пакет по умолчанию можно использовать только для очень простых проектов.

Обратиться к членам пакета внутри самого пакета можно по имени самого члена; имя пакета указывать необязательно:

```
Main main = new Main();
```

Обратиться к членам пакета из другого пакета можно по полному имени, включающему имя пакета. Например, для класса `Main` выше:

```
ru.urvanov.javaindynamics2022.packageexamples.Main
```

Создать экземпляр класса Main из другого пакета можно так:

```
ru.urvanov.javaindynamics2022.packageexamples.Main main
        = new ru.urvanov.javaindynamics2022.packageexamples.Main();
```

Если пакет используется часто, то постоянное написание полного имени пакета усложнит понимание кода. В таких случаях можно использовать операторы `import`, которые добавляются между `package` и объявлениями классов:

OtherClass.java

```
package ru.urvanov.javaindynamics2022.otherpackage;

import ru.urvanov.javaindynamics2022.packageexamples.Main;
// другие операторы import

class OtherClass {
 public void method1() {
   Main main = new Main();
 }
}
```

С помощью оператора `import` можно также импортировать все содержимое пакета с помощью символа "*", а не только один класс:

OtherClass.java

```
package ru.urvanov.javaindynamics2022.otherpackage;

import ru.urvanov.javaindynamics2022.packageexamples.*;
// другие операторы import

class OtherClass {
 public void method1() {
   Main main = new Main();
 }
}
```

При указании звездочки можно импортировать только содержимое пакета целиком, но нельзя импортировать лишь какую-то его часть.

По умолчанию в Java в каждый файл уже импортированы два пакета: `java.lang` и пакет текущего файла.

Несмотря на то что пакеты кажутся иерархическими, они такими не являются. Пакет `ru.urvanov.javaindynamics2022` не включает в себя пакет `ru.urvanov.javaindynamics2022.packageexamples`. Это два совершенно разных пакета. Код

```
import ru.urvanov.javaindynamics2022.*;
```

импортирует только члены пакета `ru.urvanov.javaindynamics2022`, но не члены пакета `ru.urvanov.javaindynamics2022.packageexamples`.

Если два пакета содержат член (класс, интерфейс или перечисления) с одинаковым именем и оба этих пакета импортированы, то для доступа к этим членам нужно использовать полное имя с указанием пакета:

```
ru.urvanov.javaindynamics2022.packageexamples.Main main;
```

Если вы часто используете константы из какого-либо класса, то постоянное указание его имени только запутает код. В таких случаях можно использовать оператор `import static`. Класс java.lang.Math, например, содержит константу `PI`. Для более простого доступа к ней можно импортировать эту константу:

```
import static java.lang.Math.PI;
```

> ***Важно***
>
> Слова `import` и `static` менять местами нельзя! Нельзя писать `static import`, только `import static`!

Теперь можно обращаться к `PI` только по имени:

```
double x = PI * 2;
```

Можно импортировать все статические члены класса с помощью звездочки:

```
import static java.lang.Math.*;
```

Не стоит лишний раз использовать `import` со звездочкой и `import static`. Лучше включать в файл только нужное содержимое. Современные IDE позволяют буквально парой кликов организовать операторы `import` в файле.

Причины использования пакетов:

- Можно легко понять, что эти типы в одном пакете связаны между собой.
- Легче искать классы с определенным функционалом, т. к. они находятся в одном пакете.
- Имена типов не будут конфликтовать с именами типов, созданными другими разработчиками.
- Можно дать классам внутри пакета полный доступ к членам друг друга, но при этом запретить доступ к некоторым из них снаружи (объяснено в главе про классы).

10.2. Задания

1. Создайте какой-нибудь пакет и поместите туда несколько классов. Для этого можете взять классы, созданные в процессе выполнения заданий из других глав.
2. Создайте второй пакет и поместите туда класс, который использует классы из первого задания.
3. Добавьте метод `main` в класс из второго задания и попробуйте запустить его из консоли.

ГЛАВА 11

Перечисления

11.1. Теория

Тип `enum` (перечисление) — это специальный тип, который позволяет переменной иметь одно из предопределенных константных значений. Объявление перечислений похоже на объявления классов: модификатор доступа, ключевое слово `enum` (НЕ `class`!), имя типа, тело перечисления. Тело перечисления содержит имена предопределенных константных значений.

Direction.java

```
public enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST
}
```

Затем можно объявить переменную нашего созданного типа перечисления:

DirectionTest.java

```
Direction direction = Direction.NORTH;
```

Используйте перечисления всегда, когда вам нужен набор предопределенных связанных между собой констант.

Пример использования:

Coach.java

```
package ru.urvanov.javaindynamics2022.enums;

public class Coach {
    private int x;
    private int y;

    public void move(Direction direction) {
        switch (direction) {
            case NORTH:
                y--;
                break;
            case EAST:
                x++;
                break;
            case SOUTH:
                y++;
                break;
            case WEST:
                x--;
                break;
        }
    }
}
```

Перечисления в Java являются классами.

Все перечисления неявно наследуются от класса `java.lang.Enum`. Поскольку в Java нет множественного наследования, перечисление не может наследоваться от какого-либо другого класса дополнительно, но может реализовывать сколько угодно интерфейсов.

Из класса `java.lang.Enum` каждое перечисление наследует пару полезных методов:

```
public final int ordinal()
```

Возвращает порядковый номер константы в перечислении. Нумерация начинается с нуля.

```
public final String name()
```

Метод `name()` возвращает имя константы так, как оно было объявлено в перечислении. Например, NORTH.

Также все перечисления неявно становятся `final`, если у них нет ни одной константы с телом класса (описано ниже). Нельзя указывать перечисление в качестве рас-

ширяемого класса. Нельзя указывать модификаторы `final` и `abstract` для перечислений.

Для вложенных перечислений всегда подразумевается модификатор `static`. Нельзя явно указывать этот модификатор, иначе будет ошибка компиляции.

Так как вложенные перечисления являются статическими, отсюда вытекает, что анонимные классы, лямбда-выражения, внутренние классы, являющиеся нестатическими членами класса, и локальные классы НЕ могут содержать перечислений.

К каждому перечислению компилятор добавляет статический метод `values()`, который возвращает массив возможных значений для перечисления в том порядке, в котором они объявлены, и статический метод `valueOf(String name)`, который возвращает ссылку на константу перечисления по ее имени.

DirectionTest.java

```
// Перебор всех значений перечисления
for (Direction dir : Direction.values()) {
    System.out.println("toString(): " + dir.toString());
    System.out.println("ordinal(): " + dir.ordinal());
    System.out.println("name() :" + dir.name());
}
```

Выведет:

```
toString(): NORTH
ordinal(): 0
name() :NORTH
toString(): EAST
ordinal(): 1
name() :EAST
toString(): SOUTH
ordinal(): 2
name() :SOUTH
toString(): WEST
ordinal(): 3
name() :WEST
```

Константные значения перечислений (`Direction.NORTH`, `Direction`, `EAST` и т. д.) являются экземплярами этого класса перечисления (`Direction`), но их можно сравнивать через ==, т. к. для каждой из этих констант всегда будет существовать только один объект. Нельзя пытаться создать экземпляр перечисления вручную любым способом: будет ошибка компиляции. Java Reflection API, метод `clone()` и механизм сериализации/десериализации обрабатывают перечисления таким образом, что повторного создания экземпляров перечислений не происходит.

Для каждого константного значения в перечислении создается поле с модификаторами `public static final` с тем же именем, что и константное значение. Это поле

имеет выражение инициализации этой константы и имеет те же аннотации, что и константное значение.

Объявление констант в перечислении позволяет указать параметры для конструктора перечисления:

MagicCreature.java

```
package ru.urvanov.javaindynamics2022.enums;

enum MagicCreature {
    GOBLIN(100, 50),
    HOBGOBLIN(110, 30),
    GREMLIN(200, 10);

    private int health;
    private int magic;

    MagicCreature(int health, int magic) {
        this.health = health;
        this.magic = magic;
    }

    public int getHealth() {
        return this.health;
    }

    public int getMagic() {
        return this.magic;
    }

    public static void main(String[] args) {
        System.out.println("Goblin. Health: "
                + MagicCreature.GOBLIN.getHealth()
                + " magic: " + MagicCreature.GOBLIN.getMagic());
    }
}
```

Перечисление может иметь несколько конструкторов, тогда выбирается тот, который обладает наиболее подходящими параметрами. Перечисление не может содержать конструкторов с модификаторами доступа `public` или `protected`. Конструктор без модификатора доступа в перечислении становится приватным (`private`).

Если в перечислении нет ни одного объявления конструктора, то автоматически добавляется конструктор по умолчанию без параметров, с модификатором доступа `private` и без списка возможных исключений.

Если конструктор перечисления содержит ключевое слово `super`, то возникает ошибка компиляции.

Нельзя обращаться к статическим полям перечисления, если они не являются константами (например, `public static int CONST1 = 200;`), из конструкторов, блоков инициализации экземпляров, выражений инициализации.

Константы перечисления могут содержать тело класса, тогда эти классы автоматически являются анонимными и наследуются от класса текущего перечисления. Перечисление может содержать абстрактные методы, но тогда все константы должны иметь тело класса, в котором обязательно должны реализовать все абстрактные методы:

MagicBeast.java

```
package ru.urvanov.javaindynamics2022.enums;

enum MagicBeast {
    GOBLIN {
        void doSomething() {
            System.out.println("Do something.");
        }

        public void someEnumMethod() {
            // реализация 1.
        }
    },
    HOBGOBLIN(110, 30) {
        void doOtherThing() {
            System.out.println("Do other thing");
        }

        public void someEnumMethod() {
            // реализация 2.
        }
    },
    GREMLIN(200, 10) {
        public void someEnumMethod() {
            // реализация.
        }
    };

    private int health;
    private int magic;

    MagicBeast() {
    }

    MagicBeast(int health, int magic) {
        this.health = health;
```

```
        this.magic = magic;
    }

    // Абстрактный метод
    public abstract void someEnumMethod();

}
```

Перечисление не может содержать метод-финализатор `finalize()` (который все равно помечен как метод для удаления в Java 18), иначе возникнет ошибка компиляции.

11.2. Задания

1. Создайте перечисление, которое могло бы описывать операции в банкомате: выдача наличных, прием наличных, оплата услуг, перевод денег, проверка баланса и т. д. Подумайте, какие методы и конструкторы можно было бы добавить.
2. Создайте перечисление, которое описывало бы состояние игры в игровом движке: просмотр меню, игры, проигрыш, победа, проигрывание сюжетной анимации. Постарайтесь добавить в него как можно больше разных методов и конструкторов.
3. Создайте перечисление, которое описывало бы степень родства членов семьи: папа, мама, брат, сестра, бабушка, дедушка, дядя, тетя и т. д. Выведите в консоль все возможные значения из перечисления с помощью метода `values()`.

ГЛАВА 12

Записи

12.1. Теория

Records или записи появились в Java 16. Очень часто приходится создавать классы для хранения состояния какого-либо объекта предметной области. При этом в каждом классе нужно описывать достаточно большое количество однотипных методов: конструктор, методы получения значений полей, `hashCode`, `equals` и `toString`. В большинстве случаев такие методы генерируются с помощью IDE. Рассмотрим следующий класс:

PointClass.java

```
package ru.urvanov.javaindynamics2022.records;

import java.util.Objects;

public final class PointClass {

    private final int x;
    private final int y;

    public PointClass(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PointClass point = (PointClass) o;
        return x == point.x && y == point.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    @Override
    public String toString() {
        return "Point{" +
                "x=" + x +
                ", y=" + y +
                '}';
    }

    public static void main(String[] args) {
        PointClass pointClass = new PointClass(100, 200);
```

```
        System.out.println("pointClass = " + pointClass);
    }
}
```

Конструктор, методы получения значений, `hashCode`, `equals` и `toString` занимают большую часть описания класса, но все они просто сгенерированы с помощью IDEA. Для генерации используется пункт **Code ➢ Generate** из горизонтального меню (рис. 12.1).

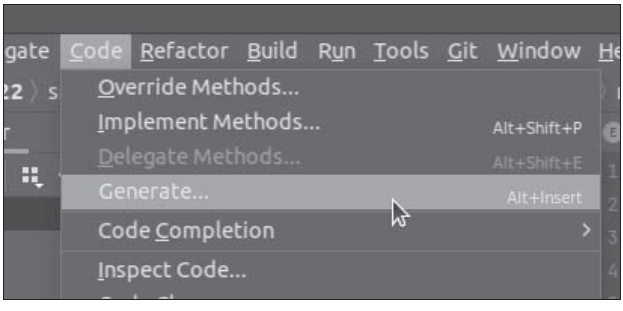


Рис. 12.1. Генерация методов get, set, hashCode, equals в IntelliJ IDEA

С помощью записей, которые появились в Java 16, код можно упростить до нескольких строк:

PointReord.java

```
package ru.urvanov.javaindynamics2022.records;

public record PointRecord(int x, int y) {

    public static void main(String[] args) {
        PointRecord pointRecord = new PointRecord(100, 100);
        System.out.println("record = " + pointRecord);
    }
}
```

Записи автоматически получают приватные переменные для хранения своего состояния, методы `hashCode`, `equals`, `toString`, работающие с переменными состояния, а также методы для получения значения каждой из переменных состояния.

Сами записи неявно являются final, их переменные состояния — тоже `final`.

12.2. Задания

1. Создайте запись для хранения информации на ценнике в магазине: название, цена, вес/поштучно, масса нетто, скидка и т. д.
2. Создайте запись для хранения информации о должности: название, зарплата, обязательность наличия диплома.

ГЛАВА 13

Числа

13.1. Введение

Для каждого примитивного типа существует класс, позволяющий представить этот тип в объекте. Все такие классы, соответствующие числам, наследуются от абстрактного класса `java.lang.Number` и находятся в пакете `java.lang`.

Классы для числовых примитивных типов:

- Класс `Byte` соответствует примитивному типу `byte`.
- Класс `Short` соответствует примитивному типу `short`.
- Класс `Integer` соответствует примитивному типу `int`.
- Класс `Long` соответствует примитивному типу `long`.
- Класс `Float` соответствует примитивному типу `float`.
- Класс `Double` соответствует примитивному типу `double`.

Все эти классы неизменяемые, т. е. не могут менять своего состояния после создания, что позволяет безопасно использовать их в нескольких потоках.

В большинстве случаев можно использовать примитивный тип там, где ожидается объект, и объект там, где ожидается примитивный тип. В таких случаях компилятор Java автоматически вставляет преобразование из примитива в объект и обратно. Это преобразование называется автоупаковкой (autoboxing) и распаковкой (unboxing).

Numbers.java

```
Integer i1 = 3334; // Упаковка int в объект Integer
int i2 = i1;  // Распаковка Integer в примитивный тип int
```

Каждый из этих классов содержит константы `MAX_VALUE` и `MIN_VALUE`, которые позволяют узнать верхнюю и нижнюю границы диапазона значений для соответствующего примитивного типа:

Numbers.java

```
System.out.println("Byte.MIN_VALUE=" + Byte.MIN_VALUE);
System.out.println("Byte.MAX_VALUE=" + Byte.MAX_VALUE);
```

```
System.out.println("Integer.MIN_VALUE=" + Integer.MIN_VALUE);
System.out.println("Integer.MAX_VALUE=" + Integer.MAX_VALUE);
```

Классы `java.lang.Double` и `java.lang.Float` дополнительно содержат константы `NaN` (не число), `NEGATIVE_INFINITY` (минус бесконечность), `POSITIVE_INFINITY` (плюс бесконечность), `MIN_EXPONENT` (минимальная экспонента) и `MAX_EXPONENT` (максимальная экспонента), `MIN_NORMAL` (наименьшее положительное число, которое может хранить этот тип).

Абстрактный класс `Number` содержит несколько полезных методов, которые реализуются всеми его дочерними классами.

Методы для преобразования объекта `Number` в соответствующее число примитивного типа:

```
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
```

Методы сравнения с примитивными типами:

```
int compareTo(Byte anotherByte)
int compareTo(Double anotherDouble)
int compareTo(Float anotherFloat)
int compareTo(Integer anotherInteger)
int compareTo(Long anotherLong)
int compareTo(Short anotherShort)
```

Метод проверки на равенство:

```
boolean equals(Object obj)
```

Метод проверки на равенство возвращает `true` в случае, когда `obj` не равен `null`, и тип `obj` совпадает с типом объекта, у которого вызван метод, и они содержат одинаковое значение.

Каждый из классов `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` также содержит методы для преобразования чисел в строку и строки в числа. Пример для класса `Integer` (остальные классы содержат аналогичные методы для своих типов):

```
static Integer decode(String s)
```

Преобразует строку в число. Поддерживает десятичную, шестнадцатеричную и восьмеричную системы счисления.

```
static int parseInt(String s)
```

Возвращает `int`. Только для десятичной системы счисления.

```
static int parseInt(String s, int radix)
```

Возвращает `int`. Система счисления задается параметром `radix` (2, 8, 10 или 16).

`String toString()`

Возвращает строковое представление.

`static String toString(int i)`

Возвращает строковое представление для i.

`static Integer valueOf(int i)`

Возвращает объект `Integer`, содержащий значение `i`. Рекомендуется для получения объектов из соответствующих им примитивов всегда использовать этот метод и аналогичные в других классах, поскольку они кешируют часто используемые значения. Метод `public static Byte valueOf(byte b)` кеширует ВСЕ значения. Методы `public static Short valueOf(short s)` и `public static Integer valueOf(int i)` всегда кешируют значения от –128 до +127 включительно, а также часто используемые. Методы `public static Long valueOf(long l)`, `public static Float valueOf(float f)` и `public static Double valueOf(double d)` кешируют часто используемые значения.

`static Integer valueOf(String s)`

Возвращает объект `Integer`, содержащий значение из строки.

`static Integer valueOf(String s, int radix)`

Возвращает объект `Integer`, содержащий значение из строки `s` в соответствии с системой счисления `radix`.

13.2. BigInteger

Класс `java.math.BigInteger` позволяет хранить целые числа любой величины. Объекты этого класса являются неизменямыми, т. е. не меняют своего состояния после создания, но могут порождать новые объекты при сложении, вычитании и т. д., что означает, что их можно безбоязненно использовать в нескольких потоках.

Класс `java.math.BigInteger` является потомком класса `java.lang.Number`.

Пример создания:

```
BigInteger bigNumber1 = new BigInteger("10000000000000000000000000");
```

Здесь был использован конструктор, принимающий строку. У `BigInteger` есть еще другие конструкторы, но этот используется наиболее часто.

Логика операций с `BigInteger` схожа с логикой целочисленных операций в Java. Например, деление на 0 приводит к `ArithmeticException`.

Переполнение, как в целочисленных операциях примитивов, в `BigInteger` невозможно, т. к. он автоматически занимает достаточно места, чтобы поместить результат операции.

Операции побитовых сдвигов для `BigInteger` расширены: они могут принимать отрицательное расстояние. Сдвиг вправо с отрицательным расстоянием приводит

к сдвигу влево и т. д. Беззнакового сдвига вправо для `BigInteger` нет, т. к. он не имеет смысла при неограниченном размере числа.

В Java нет перегрузки операций, поэтому для выполнения арифметических операций с `BigInteger` нужно использовать методы.

Список методов:

```
public BigInteger abs()
```

Возвращает `BigInteger` с положительным знаком и тем же числом.

```
public BigInteger negate()
```

Возвращает `BigInteger` с отрицательным знаком и тем же значением.

```
public int signum()
```

Возвращает –1 для отрицательных чисел, 0 для нуля и 1 для положительных.

```
public BigInteger shiftLeft(int n)
```

Возвращает результат побитового сдвига влево на `n` позиций.

```
public BigInteger shiftRight(int n)
```

Возвращает результат побитового сдвига вправо на `n` позиций.

```
public int compareTo(BigInteger val)
```

Сравнивает значение в `BigInteger` со значением в `val`. Возвращает –1, если текущее значение меньше `val`, 0, если они равны, и 1, если текущее значение больше `val`.

```
public BigInteger add(BigInteger val)
```

Возвращает результат сложения текущего `BigInteger` с `val`.

```
public BigInteger subtract(BigInteger val)
```

Возвращает разность текущего значения и `val`.

```
public BigInteger multiply(BigInteger val)
```

Возвращает результат умножения текущего значения на `val`.

```
public BigInteger divide(BigInteger val) throws ArithmeticException
```

Возвращает результат целочисленного деления текущего значения на `val`. Если `val` равен нулю, то возникает `ArithmeticException`.

```
public BigInteger remainder(BigInteger val) throws ArithmeticException
```

Возвращает остаток от деления текущего значения на `val`. Если `val` равен нулю, то возникает `ArithmeticException`.

И другие методы, включая методы из `Number`.

Пример использования:

BitIntegerExample.java

```
BigInteger bigNumber1 = new BigInteger("10000000000000000000000000000");
BigInteger result = bigNumber1.add(new BigInteger("123"))
        .subtract(new BigInteger("333"))
        .multiply(new BigInteger("2"));
```

13.3. BigDecimal

Неизменяемое десятичное число произвольной точности. `BigDecimal` состоит из целого числа произвольного размера и 32-битного коэффициента масштабирования (scale), который показывает, на сколько позиций нужно сдвинуть десятичную точку.

Фактическое значение, хранящееся в `BigDecimal`, получается по формуле unscaled value $\times 10^{-scale}$, где unscaled value — это само значение, а scale — коэффициент масштабирования, т. е. положительное значение scale определяет количество цифр, на которое нужно сдвинуть десятичную запятую влево, а отрицательное scale определяет количество нулей, которые нужно добавить к целому unscaled value, чтобы получить конечное значение.

Класс `java.math.BigDecimal` является потомком класса `java.lang.Number`.

Для хранения и обработки финансовых значений нужно всегда использовать `BigDecimal`, т. к. он позволяет избежать ошибок округления, которые возникают из-за невозможности представить некоторые десятичные дроби без потери точности в числах с плавающей точкой `float` и числах с плавающей точкой двойной точности `double`.

```
BigDecimal value1 = new BigDecimal("100.01");
```

Для многих методов в `BigDecimal` нужно указать экземпляр класса `java.math.MathContext`, который определяет точность результирующего значения и способ округления или `scale` и способ округления. У некоторых методов есть варианты без `MathContext` и с ним, а также со `scale` и без него, в этом случае нужно всегда использовать вариант либо с `MathContext`, либо со scale, т. к. методы без этого параметра будут генерировать `ArithmeticException`, если их результат невозможно представить конечной десятичной дробью.

```
// 4 значащие цифры
// Стандартное округление, которому учат в школе.
MathContext mathContext = new MathContext(4, RoundingMode.HALF_UP);

BigDecimal result1 = new BigDecimal("100").divide(
    new BigDecimal("3"), mathContext);

BigDecimal result2 = new BigDecimal("100").divide(
    new BigDecimal("3"), 4, RoundingMode.HALF_UP);
```

```
System.out.println("result1 = " + result1); // 33.33 (4 значащие цифры)

System.out.println("result2 = " + result2); // 33.3333 (4 цифры
                                            // после запятой)
```

В Java нет перегрузки операций, поэтому для арифметических операций с `BigDecimal` нужно использовать соответствующие методы.

Список полезных методов:

```
public BigDecimal abs()
```

Возвращает положительный `BigDecimal` со значением, равным текущему (без знака), т. е. по модулю равным текущему.

```
public BigDecimal negate()
```

Возвращает значение с противоположным знаком.

```
public int signum()
```

Возвращает знак числа (–1, 0 или 1).

```
public int scale()
```

Возвращает коэффициент масштабирования `scale` согласно формуле:

текущее значение = целое_число * 10^{-scale}

```
public int compareTo(BigDecimal val)
```

Сравнивает текущее значение с `val`. Возвращает –1, если текущее значение меньше `val`, 0 — текущее значение равно `val`, 1 — текущее значение больше `val`. Всегда сравнивайте значения `BigDecimal` через compareTo, а не через `equals`, т. к. метод `equals` сравнивает еще и коэффициент масштабирования, поэтому для `equals` выражение `new BigDecimal("100.0").equals(new BigDecimal("100.00"))` вернет `false`, поскольку у `new BigDecimal("100.0")` количество цифр после запятой (`scale`) равно 1, а у `new BigDecimal("100.00")` оно равно 2.

```
public BigDecimal add(BigDecimal augend) throws ArithmeticException
```

Возвращает результат сложения текущего значения и `augend`. Коэффициент масштабирования `scale` результата равен `max(this.scale(), augend.scale())`.

```
public BigDecimal subtract(BigDecimal subtrahend) throws ArithmeticException
```

Возвращает разность текущего значения и `subtrahend`. Коэффициент масштабирования `scale` результата равен `max(this.scale(), subtrahend.scale())`.

```
public BigDecimal multiply(BigDecimal multiplicand) throws ArithmeticException
```

Возвращает результат умножения текущего значения на `multiplicand`. Коэффициент масштабирования результата равен `this.scale + multiplicand.scale()`.

```
public BigDecimal divide(BigDecimal divisor,
                         MathContext mc) throws ArithmeticException
```

Возвращает результат деления текущего значения на `divisor`. Количество значащих цифр в результате берется из `mc`.

```
public BigDecimal divide(BigDecimal divisor,
                int scale,
                RoundingMode roundingMode) throws ArithmeticException
```

Возвращает результат деления текущего значения на `divisor`. Коэффициент масштабирования (`scale` или количество цифр после десятичной запятой) результата равен переданному `scale`.

```
public BigDecimal remainder(BigDecimal divisor,
                            MathContext mc) throws ArithmeticException
```

Возвращает остаток от деления текущего значения на `divisor`.

И другие методы, включая методы из `java.lang.Number`.

Пример использования:

BigDecimalExample.java

```
BigDecimal value1 = new BigDecimal("100.01");

// 4 значащие цифры
// Стандартное округление, которому учат в школе.
MathContext mathContext = new MathContext(4, RoundingMode.HALF_UP);

BigDecimal result1 = value1.add(new BigDecimal("2"), mathContext)
        .subtract(new BigDecimal("0.001"), mathContext);

System.out.println(result1); // 102.0 (4 значащие цифры, как в mathContext.

BigDecimal result2 = value1.add(new BigDecimal("2"))
        .subtract(new BigDecimal("0.001"));

System.out.println(result2); // 102.009 (scale=3, т. к. это максимальный
                             // который задался в операции subtract
```

13.4. Math

Класс `java.lang.Math` содержит некоторые предопределенные константы и методы, позволяющие вычислять синус, косинус, возводить в степень и т. д.

Константы:

```
public static final double E = 2.7182818284590452354;
```

Основание натурального логарифма.

```
public static final double PI = 3.14159265358979323846;
```

Число π. Отношение длины окружности к длине ее диаметра.

Методы:

```
public static double sin(double a)
```

Возвращает синус угла a. Угол задается в радианах.

```
public static double cos(double a)
```

Возвращает косинус угла a. Угол задается в радианах.

```
public static double tan(double a)
```

Возвращает тангенс угла a. Угол задается в радианах.

```
public static double asin(double a)
```

Возвращает арксинус угла a. Угол задается в радианах.

```
public static double acos(double a)
```

Возвращает арккосинус угла a. Угол задается в радианах.

```
public static double atan(double a)
```

Возвращает арктангенс угла a. Угол задается в радианах.

```
public static double log(double a)
```

Возвращает натуральный логарифм угла a. Угол задается в радианах.

```
public static double sqrt(double a)
```

Возвращает квадратный корень от a.

```
public static double pow(double a,
                         double b)
```

Возвращает a, возведенное в степень b.

```
public static long round(double a)
```

Возвращает округленное значение a.

И другие методы.

13.5. Задания

1. Представьте, что у вас имеется лестница с известной длиной, которая приставлена к забору под углом в 25°. Напишите программу определения высоты забора.

2. Рассчитайте прибыль в рублях от размещения суммы денег на банковском счете со ставкой в 5 % годовых, если вклад размещается не на один год, а на задаваемое количество месяцев. Имейте в виду, что вы работаете с денежными суммами и должны использовать соответствующие классы.

3. Дифференцированный платеж по кредиту — это способ выплаты, когда размер платежа убывает со временем. Очередная выплата рассчитывается по формуле DP = (S ÷ n) + OD × (i ÷ k), где DP — дифференцированный платеж, S — первоначальная сумма кредита, OD — остаток долга по кредиту, i — годовая процентная ставка в виде 0,05 для 5 %, k — количество процентных периодов в году, n — количество процентных периодов во всем сроке кредита, (S ÷ n) — это сумма, которая идет в погашение основного долга по кредиту. Напишите программу для вычисления очередной суммы платежа, ее процентной части и части, которая идет на выплату основного долга. Имейте в виду, что вы работаете с денежными суммами и должны использовать соответствующие классы.

ГЛАВА 14

Строки

14.1. Класс String

В Java строки являются объектами класса `java.lang.String`, который является обычным классом, но со специальной поддержкой компилятором.

Первоначально стандарт Юникода 1991 года предполагал 16-битные символы фиксированной длины, однако впоследствии, в 1996 году, он был расширен для возможности представления символов, которым необходимо больше 16 бит. Создатели Java хотели сделать строки максимально простыми и понятными, поэтому класс String первоначально представлял строки в виде набора шестнадцатибитных символов, которые хранились в массиве `char`-ов.

Внутреннее представление в виде массива `char`-ов, разумеется, облегчало хранение и обработку строк, однако после появления символов, которые не умещаются в 16 бит, класс `String` был усложнен. Он стал представлять собой строку в виде набора символов UTF-16, где дополнительные символы представляются в виде суррогатных пар: другими словами, один символ Юникода мог состоять из двух `char`

`code unit`. Индекс в строке адресует не на char code unit, а на символ. Дополнительные символы используют две позиции в строке.

Еще один недостаток такого хранения строк заключался в том, что, если строка состоит только из символов, которым достаточно восьми бит, первоначальный байт всегда будет равен нулю. В Java 9 было изменено внутреннее представление строк. Теперь строки хранятся в виде массива байт плюс флаг кодировки. Если строка содержит только символы, которым нужно 8 бит, то используется LATIN-1, что избавляет нас от избыточного хранения огромного количества байт с нулями. Если хотя бы один символ не попадает в диапазон символов, которым нужно больше восьми бит для представления, то используется UTF-16.

`char code unit` — 16-битный символ `char`.

`char code point` — символ Юникода. Может состоять из двух `char`-ов.

Все строковые литералы в Java являются объектами класса `String`.

Примеры создания строк:

StringCreation.java

```
String greeting1 = "Hello world!";

char[] chardata = { 'h', 'e', 'l', 'l', 'o', '.'};
String greeting2 = new String(chardata);

String greeting3 = """
        Hello \
        world!
        """;

System.out.println(greeting1); // Hello world!
System.out.println(greeting2); // hello.
System.out.println(greeting3); // Hello world!
```

Для строковых литералов используются двойные кавычки, а для примитивного типа `char` — одинарные.

У класса `String` довольно много конструкторов. Можно, например, создать строку на основе массива байт:

StringCreation.java

```
byte[] byteArray = new byte[] { 0x48,  0x65, 0x6C, 0x6C, 0x6F};
        java.nio.charset.Charset charset
                = java.nio.charset.Charset.forName("ISO-8859-1");
        String str = new String(byteArray, charset);
        System.out.println(str); // Hello
```

Или так:

StringCreation.java

```
try {
    String str2 = new String(byteArray, "ISO-8859-1");
} catch (java.io.UnsupportedEncodingException ex) {
    // Вывод информации об ошибке
}
```

В этих примерах "ISO-8859-1" — это имя кодировки. Существует также вариант аналогичного конструктора, но без указания кодировки. В этом случае до Java 18 использовалась кодировка по умолчанию, что могло приводить к разным результатам на разных платформах. Начиная с Java 18, вариант без указания кодировки использует UTF-8.

Каждая реализация Java поддерживает следующие виды кодировок:

- US-ASCII.
- ISO-8859-1.
- UTF-8.
- UTF-16BE.
- UTF-16LE.
- UTF-16.

Дополнительно могут поддерживаться другие кодировки, например для Windows обязательно будет поддержка кодировки windows-1251.

Когда компилятор видит в коде строковый литерал, например "Hello world!", он создает объект класса `java.lang.String`.

Так как строковые литеры являются объектами, то у них можно вызывать методы:

StringCreation.java

```
System.out.println("Hi, Larry.".length());  // 10
```

В Java нет перегрузки операций, но класс строки имеет особую поддержку со стороны компилятора: строки можно соединять с помощью операции "+" (при этом используется `java.lang.StringBuilder` или `java.lang.StringBuffer`):

StringCreation.java

```
String personName = "Vasya";
System.out.println(personName + " goes"
        + " to school."); // Vasya goes to school.
```

Конкатенации строковых литералов вида " goes" + " to school." компилятор вычисляет на этапе компиляции, поэтому во время выполнения не будет происходить

лишних операций. Можно с помощью "+" разделять строковые константы на несколько строк, компилятор уберет лишние операции сам и превратит все в один строковый литерал:

StringCreation.java

```
String str1 = "Vasya goes to school."
        + " Petya does the same thing too."
        + " Turtle is green.";
```

Начиная с Java 15, вместо конкатенации строковых литералов для такого случая можно использовать текстовые блоки:

StringCreation.java

```
String vasyaSchool2 = """
        Vasya goes to school. \
        Petya does the same thing too. \
        Turtle is green.""";
```

В коде выше использовался символ обратной косой черты, чтобы указать, что в конечной строке `vasyaSchool2` не должно быть переводов строки в этом месте. Если бы мы не использовали этот символ, то результат в строке `vasyaSchool2` был бы многострочным.

Каждый объект в Java может участвовать в операции конкатенации (соединения) строк, в этом случае используется метод `toString()` (если ссылка на объект равна `null`, то используется строка "`null`"):

StringCreation.java

```
Object obj1 = new Object() {
    @Override
    public String toString() {
        return "obj1toString()";
    }
};

Integer integer1 = 23;
java.math.BigDecimal bigDecimal1 = new java.math.BigDecimal("3.45");
Object obj2 = null;
System.out.println("obj1: " + obj1 + "; integer1: "
        + integer1 + "; bigDecimal1: "
        + bigDecimal1 + "; obj2: " + obj2);
```

Выведет в консоль:

```
obj1: obj1toString(); integer1: 23; bigDecimal1: 3.45; obj2: null
```

Класс String НЕ имеет специальной поддержки для ==, поэтому сравнивать строки нужно либо через метод `equals()`, либо `equalsIgnoreCase()`, либо `compareTo()`, либо compare`ToIgnoreCase()`, либо с помощью специального класса `java.text.Collator`. Методы `equals()`, `equalsIgnoreCase()`, `compare()` и `compareToIgnoreCase()` не учитывают региональные настройки конкретного языка и страны, поэтому более корректным способом сравнения строк будет использование `Collator`-а:

StringCompare.java

```
// Получаем экземпляр Collator в соответствии
// с русской локалью:
Collator collator = Collator.getInstance(
        new Locale("ru", "RU"));
// Или можно получить экземпляр для текущей локали:
// Collator.getInstance();

String str1 = "Ёжик";
String str2 = "ежик";

// С помощью значений PRIMARY, SECONDARY, TERTIARY или IDENTICAL
// можно настраивать минимальный уровень различий.
// Для разных языков и стран каждое из значений имеет свой смысл.
// В данном случае для русского языка при PRIMARY буквы "Ё" и "е" будут
// считаться идентичными, а также не будет учитываться регистр букв,
// поэтому str1 будет идентична str2.
collator.setStrength(Collator.PRIMARY);

int result = collator.compare(str1, str2);
if (result < 0) {
    System.out.println(
            String.format("%s должен быть перед %s.", str1, str2));
} else if (result == 0) {
    System.out.println(
            String.format("%s и %s одинаковы.", str1, str2));
} else {
    System.out.println(
            String.format("%s должен быть после %s.", str1, str2));
}
```

В Java используется пул строковых литералов. Одинаковые строковые литералы всегда ссылаются на один и тот же экземпляр класса `String`:

StringPool.java

```
String vasya = "Vasya", ya = "ya";
System.out.println(vasya == "Vasya"); // 1
```

```
System.out.println(vasya == ("Vas" + ya)); // 2
System.out.println(vasya == ("Vas" + "ya")); // 3
System.out.println(vasya == ("Vas" + ya).intern()); // 4
```

Этот код выведет:

```
true
false
true
true
```

Объяснение:

1. Одинаковые строковые литералы всегда ссылаются на один и тот же экземпляр класса `String`.
2. Экземпляры класса `String`, вычисленные во время выполнения, создаются заново, автоматически в пуле не участвуют и потому различны.
3. Строковые литералы в константных выражениях вычисляются на этапе компиляции и затем расцениваются как обычные литералы.
4. С помощью метода `intern()` можно добавить строку в пул либо получить ссылку на такую строку из пула.

Многие методы строки принимают в качестве параметра или возвращают в качестве результата индекс. Индексация начинается с нуля. Первый `char` (имеется в виду `char code unit`, а не символ, разумеется) в строке имеет индекс 0, а последний — индекс `length()` — 1. Если переданный в параметр индекс выходит за пределы строки, то методы генерируют исключение `java.lang.IndexOutOfBoundsException`.

Некоторые методы принимают в качестве параметров начальный индекс и конечный индекс. В этом случае начальный индекс включается в отрезок, а конечный исключается, так что длина отрезка получается равной "конечный индекс минус начальный индекс", как на рис. 14.1.

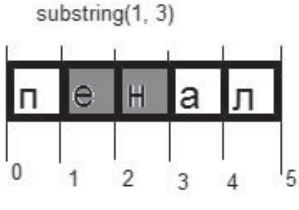


Рис. 14.1. Индексация элементов метода substring

Подобная индексация является стандартом в Java. Всегда, когда указывается отрезок в массиве, строке, списке или где-то еще, начальный индекс включается в отрезок, а конечный исключается.

14.2. Методы класса String

```
public char charAt(int index)
```

Возвращает char (Unicode code unit) по указанному индексу.

```
public int codePointAt(int index)
```

Возвращает символ (Unicode code point) по указанному индексу. Индекс указывается для `char`-ов (Unicode code units) и может принимать значения от 0 до `length() - 1`.

```
public int compareTo(String anotherString)
```

Лексикографическое сравнение строк. Возвращает –1, если текущая строка лексикографически меньше `anotherString`, 0 — строки лексикографически равны, 1 — текущая строка лексикографически больше. Метод полезен для сортировки строк.

```
public int compareToIgnoreCase(String str)
```

Лексикографическое сравнение, игнорирующее разницу в регистрах букв.

```
public String concat(String str)
```

Если длина строки `str` равна 0, то возвращает `this`, иначе возвращает строку, являющуюся конкатенацией (объединением) текущей строки и `str`.

```
public boolean contains(CharSequence s)
```

Возвращает `true`, если текущая строка содержит последовательность символов `s`. Класс `java.lang.String` реализует интерфейс `java.lang.CharSequence`, так что в качестве параметра можно передавать экземпляр класса `String`.

```
public boolean endsWith(String suffix)
```

Возвращает `true`, если текущая строка заканчивается на `suffix` или равна ему. Если suffix является пустой строкой, то возвращает `true`.

```
public byte[] getBytes(Charset charset)
public byte[] getBytes(String charsetName)
                throws UnsupportedEncodingException
```

Возвращает массив байт, содержащий строку в указанной кодировке.

```
public int indexOf(int ch)
public int indexOf(int ch,
                   int fromIndex)
```

Возвращает индекс первого вхождения символа `ch`, начиная с `fromIndex`. Если символ не найден, то возвращает –1.

```
public int indexOf(String str)
public int indexOf(String str,
                   int fromIndex)
```

Возвращает индекс первого вхождения подстроки в строке, начиная с `fromIndex`. Если подстрока не найдена, то возвращает –1.

```
public static String format(Locale l,
                            String format,
                            Object... args)
public static String format(String format,
                            Object... args)
```

Форматированный вывод. Более подробно будет рассмотрен в *главе 28 "Форматирование и парсинг"*.

```
public String intern()
```

Магический метод. В Java существует пул строк. Этот метод проверяет наличие строки в пуле, если в пуле есть такая строка, то метод возвращает ссылку на нее. Если в пуле нет такой строки, то строка добавляется в пул и возвращается ссылка на нее. Все строковые литералы и константы автоматически включаются в пул. Примеры:

```
boolean b1 = "Vasya" == "Vasya"; // true, т. к. в пуле будет одно
                                 // значение Vasya.

String str1 = "Vasya";
String substr1 = "Vas";
String str2 = substr1 + "ya"; // Но если сделать "Vas" + "ya", то будет
                              // ссылаться туда же,
                              // т. к. подобные конкатенации строковых
                              // констант компилятор автоматически убирает
boolean b2 = str1 == str2; // false

String str3 = "Vasya";
boolean b3 = str1 == str3; // true

str2 = str2.intern();
boolean b4 = str1 == str2; // true;

public static String join(CharSequence delimiter,
                          CharSequence... elements)
```

Объединяет несколько `CharSequence` в одну строку, используя в качестве разделителя `delimiter`.

```
public int length()
```

Возвращает длину строки в `char`-ах (Unicode code units). Количество символов в строке (Unicode code points) может отличаться от этого значения.

```
public boolean isEmpty()
```

Возвращает `true`, если `length() == 0`.

```
public String replace(char oldChar,
                      char newChar)
```

Возвращает строку, в которой все `oldChar` заменены на `newChar`. Если `oldChar` в строке нет, то возвращается исходная строка.

```
public String replaceAll(String regex,
                         String replacement)
```

Возвращает строку, в которой все вхождения подстрок согласно регулярному выражению заменены на `replacement`.

```
public String replace(CharSequence target,
                      CharSequence replacement)
```

Возвращает строку, в которой все вхождения последовательности символов `target` заменены на `replacement`.

```
public String substring(int beginIndex)
public String substring(int beginIndex,
                        int endIndex)
```

Возвращает подстроку, начинающуюся с `beginIndex` (включительно) до `endIndex` (исключительно) или конца строки.

```
public String[] split(String regex)
```

Разбивает строку на массив строк по регулярному выражению. Пустые конечные строки не включаются в результирующий массив.

```
public boolean startsWith(String prefix)
```

Возвращает `true`, если строка начинается с `prefix` либо равна ему. Если `prefix` является пустой строкой, то возвращается `true`.

```
public String toLowerCase()
public String toLowerCase(Locale locale)
```

Возвращает строку, где все символы приведены в нижний регистр относительно текущей или указанной локали.

```
public String toUpperCase()
public String toUpperCase(Locale locale)
```

Возвращает строку, где все символы приведены в верхний регистр относительно текущей или указанной локали.

```
public String trim()
```

Возвращает строку, в которой убраны пробелы в начале и в конце строки. Если строка начинается и заканчивается на непробельные символы (больше '\u0020'), то возвращается ссылка на `this`.

```
public static String valueOf(boolean b)
public static String valueOf(char c)
public static String valueOf(int i)
public static String valueOf(long l)
public static String valueOf(float f)
```

```
public static String valueOf(double d)
public static String valueOf(Object obj)
```

Возвращает строковое представление объекта. В случае с `Object` результатом является `obj.toString()`, если `obj != null` и "`null`" в противном случае.

И другие методы.

В Java 11 в класс `String` был также добавлен метод `strip`, который работает почти так же, как и `trim`, но убирает не только пробелы, но и другие невидимые символы, для которых `Character.isWhitespace(char)` возвращает `true`.

```
public String strip();
```

Также добавлены аналогичные методы для удаления невидимых символов в левой и в правой частях строки:

```
public String stripLeading(); // Возвращает строку без начальных
                              // пробельных символов.

public String stripTrailing(); // Возвращает строку без конечных
                               // пробельных символов.
```

В Java 11 в класс строки также был добавлен метод repeat, который возвращает строку, содержащую исходную строку, указанное количество раз:

```
public String repeat(int count)
```

14.3. StringBuilder и StringBuffer

В отличие от String класс `StringBuilder` позволяет менять содержимое своих экземпляров. В большинстве случаев нужно использовать `String`, использование же `StringBuilder` целесообразно в случаях, когда вам нужно соединить большое количество строковых переменных, например перебирая элементы массива или коллекции в цикле.

Так же как и у `String`, у `StringBuilder` есть метод `length()`, позволяющий узнать его длину в `char`-ах.

В отличие от `String`, у `StringBuilder`, кроме длины, есть еще вместимость/емкость (capacity). Вместительность можно узнать с помощью метода `capacity()`, она всегда больше длины или равна ей.

Класс `StringBuilder` имеет довольно большое количество конструкторов.

```
public StringBuilder()
```

Создает пустой `StringBuilder` вместительностью 16 (16 пустых элементов).

```
public StringBuilder(CharSequence cs)
```

Создает `StringBuilder`, содержащий символы из последовательности и 16 дополнительных пустых элементов.

`public StringBuilder(int initCapacity)`

Создает пустой `StringBuilder` с начальной вместительностью в `initCapacity` элементов.

`public StringBuilder(String s)`

Создает `StringBuilder`, который содержит указанную строку и 16 дополнительных пустых элементов.

`StringBuilder` содержит пару дополнительных методов, связанных с длиной, которых нет в `String`:

`void setLength(int newLength)`

Устанавливает длину последовательности символов. Если `newLength` меньше `length()`, то последние символы обрезаются. Если `newLength` больше `length()`, то в конец последовательности добавляются нулевые символы.

`void ensureCapacity(int minCapacity)`

Обеспечивает, что вместительность будет как минимум равной `minCapacity`.

Некоторые методы могут увеличить длину последовательности символов в `StringBuilder`. Если после выполнения подобного метода длина результирующей последовательности окажется больше вместительности, то вместительность автоматически увеличится.

```
StringBuilder append(boolean b)
StringBuilder append(char c)
StringBuilder append(char[] str)
StringBuilder append(char[] str, int offset, int len)
StringBuilder append(double d)
StringBuilder append(float f)
StringBuilder append(int i)
StringBuilder append(long lng)
StringBuilder append(Object obj)
StringBuilder append(String s)
```

Добавляет аргумент, преобразованный в строку, в конец `StringBuilder`-а.

```
StringBuilder delete(int start, int end)
StringBuilder deleteCharAt(int index)
```

Первый метод удаляет подпоследовательность со `start` до `end-1` включительно. Второй метод удаляет символ по индексу.

```
StringBuilder insert(int offset, boolean b)
StringBuilder insert(int offset, char c)
StringBuilder insert(int offset, char[] str)
StringBuilder insert(int index, char[] str, int offset, int len)
```

```
StringBuilder insert(int offset, double d)
StringBuilder insert(int offset, float f)
StringBuilder insert(int offset, int i)
StringBuilder insert(int offset, long lng)
StringBuilder insert(int offset, Object obj)
StringBuilder insert(int offset, String s)
```

Вставляет второй аргумент, конвертированный в строку, в позицию `index`.

```
StringBuilder replace(int start, int end, String s)
void setCharAt(int index, char c)
```

Заменяет указанный символ(ы).

```
StringBuilder reverse()
```

Меняет порядок символов в `StringBuilder` на обратный. Первый символ становится последним и т. д..

```
String toString()
```

Возвращает строку, содержащую последовательность символов из `StringBuilder`.

Пример использования `StringBuilder`:

StringBuilderExample.java

```
String andStr = " and Petya";
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Vasya");
stringBuilder.append(andStr);
stringBuilder.append(" go to school.");
System.out.println(stringBuilder);
```

14.4. Задания

1. Напишите программу, заменяющую в тексте все вхождения машинописного минуса "-" (который есть на клавиатуре) на символы типографического тире "—" (код символа \u2014), но только в тех случаях, когда символ машинописного минуса с обоих сторон обрамляется пробельным символом (не только самим пробелом, но и неразрывным пробелом, табулятором и т. д.)
2. Напишите программу, которая удаляет дублирующиеся пробелы из текста, оставляя только один из идущих подряд.
3. Напишите программу, которая сортирует массив строк с Ф. И. О. на русском языке, игнорируя регистр букв и считая буквы "Ё" и "е" идентичными.
4. Напишите программу, которая создает одну результирующую строку из массива строк. Напишите ее в двух вариантах: с использованием `String.join` и с `StringBuilder` вместе с обходом массива в цикле.

ГЛАВА 15

Автоупаковка и распаковка

15.1. Теория

Автоупаковка (autoboxing) — это автоматическая конвертация из примитивных типов в соответствующий этому типу класс-обертку, вставляемая компилятором Java, например из `float` во `Float`, из `int` в `Integer`.

Примеры автоупаковки:

Autoboxing1.java

```
Character ch = 'a';
Integer i1 = 220;
Double d1 = 300.0;
Boolean b1 = false;
```

Компилятор Java применяет автоупаковку в следующих случаях:

- При передаче примитивного типа в параметр метода, ожидающего соответствующий ему класс-обертку.
- При присвоении значения примитивного типа переменной соответствующего класса-обертки.

Распаковка (unboxing) — конвертация класса-обертки в соответствующий ему примитивный тип. В процессе распаковки может произойти исключение `java.lang.NullPointerException`, если значение переменной равно `null`.

Компилятор Java автоматически применяет распаковку в следующих случаях:

- При передаче объекта класса-обертки в метод, ожидающий соответствующий примитивный тип.
- При присвоении экземпляра класса-обертки переменной соответствующего примитивного типа.
- В выражениях, в которых один или оба аргумента являются экземплярами классов-оберток (кроме операции `==` и `!=`).

Примеры:

Autoboxing2.java

```
class Autoboxing2 {

    public static void method1(int x) {
    }

    public static void main(String[] args) {
        Integer i1 = 100;

        method1(i1); // распаковка

        Double d1 = 2.3;
        Double d2 = 3.3;
        double d3 = i1 + d1 + d2; // распаковка

        System.out.println(d3);
    }
}
```

Таблица 15.1. *Соответствие примитивных типов и классов-оберток*

Примитивный тип	Класс-обертка	Упаковка	Распаковка
boolean	Boolean	Boolean.valueOf(booleanValue)	booleanObject.booleanValue()
byte	Byte	Byte.valueOf(byteValue)	byteObject.byteValue()
char	Character	Character.valueOf(charValue)	characterObject.charValue()
float	Float	Float.valueOf(floatValue)	floatObject.floatValue()
int	Integer	Integer.valueOf(integerValue)	integerObject.integerValue()
long	Long	Long.valueOf(longValue)	longObject.longValue()
short	Short	Short.valueOf(shortValue)	shortObject.shortValue()
double	Double	Double.valueOf(doubleValue)	doubleObject.doubleValue()

Если оба операнда являются классами-обертками, то для операций сравнения <, >, <=, >= компилятор автоматически вставляет распаковку, но в случае операций == и != происходит сравнение ссылок объектов.

Autoboxing3.java

```
Integer x = 10;
Integer y = 20;
```

```
System.out.println("x > y : " + (x > y)); // false
System.out.println("x < y : " + (x < y)); // true

Integer x1 = new Integer(10);
Integer x2 = new Integer(10);
// x1 и x2 ссылаются на разные экземпляры объектов
System.out.println("x1 >= x2 : " + (x1 >= x2)); //true
System.out.println("x1 <= x2 : " + (x1 <= x2)); // true
System.out.println("x1 == x2 : " + (x1 == x2)); // false происходит
                                                // сравнение ссылок
System.out.println("x1 != x2 : " + (x1 != x2)); // true происходит
                                                // сравнение ссылок

Integer x3 = Integer.valueOf(10);
Integer x4 = 10; // Здесь неявно вызывается Integer.valueOf
// Теперь x3 и x4 указывают на один и тот же объект из-за кеширования.
System.out.println("x3 == x4 : " + (x3 == x4)); // true из-за кеширования
                                                // см. метод Integer.valueOf
System.out.println("x3 != x4 : " + (x3 != x4)); // false из-за кеширования
                                                // см. метод Integer.valueOf
```

15.2. Задания

1. Создайте метод, который принимает класс-обертку, например `Integer`. Вызовите его, передав примитивный тип. Попытайтесь объяснить, что произошло на самом деле.
2. Создайте метод, который принимает `double`. Вызовите его, передав `Double`. Попытайтесь объяснить, что произошло на самом деле.
3. Расскажите, в каких случаях можно сравнивать между собой примитивные типы и классы-обертки и почему.

ГЛАВА 16

Optional

16.1. Теория

В Java 8 появился полезный класс `java.util.Optional`. Он используется для избавления от большого числа проверок значений на `null`, которые затрудняют чтение кода.

Пусть, например, у нас есть класс:

```
package ru.urvanov.javaindynamics2022.optional;

import java.util.Optional;

class Satyr {
 public void myMethod() {

 }

 public Integer getSomeValue() {
   return 2;
 }

}
```

Пример кода с экземпляром этого класса и проверкой на `null`:

```
public static Integer testMethod(Satyr myClass) {
    if (myClass != null) {
        myClass.myMethod();
        return myClass.getSomeValue();
    }
    return null;
}
public static void main(String[] args) {
    Satyr myClass = new Satyr();

    Integer result1 = testMethod(myClass);
}
```

Аналогичный код с использованием модного класса `Optional`:

```
public static Integer testMethodWithOptional(Optional<Satyr> myClass) {
    myClass.ifPresent(Satyr::myMethod);
```

```
    return myClass.map(Satyr::getSomeValue).orElse(null);
}
public static void main(String[] args) {
    Satyr myClass = new Satyr();

    int result2 = testMethodWithOptional(Optional.ofNullable(myClass));
}
```

Как видно из этого примера, код метода `testMethodWithOptional` стал на две строки короче, чем код `testMethod`. Если в исходном примере было бы еще больше проверок на `null`, то сокращение размера кода было бы гораздо заметнее.

Давайте теперь разберемся, что же тут происходит. Сначала обратите внимание на вызов `Optional.ofNullable(myClass)`. Этот вызов статического метода создает экземпляр класса `Optional`. Созданный экземпляр выступает в качестве контейнера, который может либо содержать, либо не содержать не `null` значение. Для проверки того, содержит ли контейнер значение, нужно использовать метод `isPresent()`, который возвращает `true`, если контейнер содержит значение. В примере выше мы его не использовали, но метод весьма полезен.

В дальнейшем вся работа происходит через методы этого контейнера. Метод `ifPresent` позволяет вызвать какой-либо код, использующий объект из контейнера, но этот код будет выполняться только в том случае, если контейнер содержит значение. В данном случае мы передаем туда ссылку на метод `myMethod`.

Если же нам нужно вызвать какой-то код, результатом которого нам нужно воспользоваться в дальнейшем, то нам следует использовать метод map, который возвращает экземпляр Optional, содержащий возвращенное значение, если оно НЕ `null`. В примере выше внутри метода map мы вызываем наш метод `getSomeValue` у класса `Satyr`.

Но из нашего метода `testMethodWithOptional` нам нужно вернуть не сам контейнер, а хранящееся в нем значение или `null`, если значения нет. Для этого мы вызываем метод `orElse`, который возвращает хранящееся в контейнере Optional значение, если оно есть, либо значение, которое мы передадим в качестве аргумента метода. В данном случае в качестве аргумента мы передаем `null`.

Сокращение размера кода и проверок на `null` становится очень сильно заметно, если нам нужно подряд вызвать сначала один метод объекта, затем, если вернулось НЕ `null`, вызвать другой метод над возвращенным объектом и т. д. Пример:

OptionalExample1.java

```
package ru.urvanov.javaindynamics2022.optional;

import java.util.Arrays;
import java.util.Optional;

public class OptionalExample1 {
    static class A {
```

```
        public B getB() {
            return new B();
        }
    }

    static class B {
        public C getC() {
            return new C();
        }
    }

    static class C {
        public D getD() {
            return new D("from C");
        }
    }

    static class D {
        private String value;
        public D(String value) {
            this.value = value;
        }
        @Override
        public String toString() {
            return "D [value=" + value + "]";
        }
    }

    static class E {}

    public static void main(String[] args) {
        D d = Optional.ofNullable(new A()).map(A::getB)
                .map(b -> b.getC()).map(C::getD)
                .orElse(new D("from orElse"));
        System.out.println("d = " + d);
    }
}
```

Попробуйте поиграть с кодом выше. Например, поправить код класса `C` или `B`, чтобы они возвращали `null`, тогда вы увидите, что экземпляр класса `D` будет содержать строку "`from orElse`", т. к. он создан в методе `orElse`, где ему в конструктор передалось именно это значение. В текущем же варианте кода экземпляр класса `D` будет содержать строку "`from C`".

Если метод вашего класса уже возвращает экземпляр `Optional`, то использование метода map приведет к тому, что в результирующем значении будет `Optional` внутри другого `Optional`. Для того чтобы этого избежать, используйте метод `flatMap`:

OptionalExample2.java

```
package ru.urvanov.javaindynamics2022.optional;

import java.util.Optional;

public class OptionalExample2 {

    static class MyClass {

        public Optional<Integer> getSomeValue() {
            return Optional.ofNullable(2);
        }
    }

    public static void main(String[] args) {
        Integer result = Optional.ofNullable(new MyClass())
                .flatMap(MyClass::getSomeValue)
                .orElse(Integer.valueOf(-1));
    }
}
```

Если же мы вызываем цепочкой какие-то методы, и на самом последнем этапе нам нужно либо вернуть значение, либо бросить какое-то конкретное исключение, если значения нет, то нужно использовать метод `orElseThrow`:

OptionalExample3.java

```
package ru.urvanov.javaindynamics2022.optional;

import java.util.Optional;

public class OptionalExample3 {

    static class MyClass {

        public Optional<Integer> getSomeValue() {
            return Optional.ofNullable(2);
        }
    }

    static class NoResultValueException extends Exception {
    }

    public static void main(String[] args)
            throws NoResultValueException {
```

```
        Integer result = Optional.ofNullable(new MyClass())
                .flatMap(MyClass::getSomeValue)
                .orElseThrow(NoResultValueException::new);
    }
}
```

Обратите внимание, что существуют два похожих метода создания экземпляра `Optional`:

```
public static <T> Optional<T> of(T value)
```

а также

```
public static <T> Optional<T> ofNullable(T value)
```

Какая между ними разница? Разница в том, что `Optional.of` бросит исключение `NullPointerException`, если ему передать значение `null` в качестве параметра. `Optional.ofNullable` вернет `Optional`, не содержащий значение, если ему передать `null`.

Примеры:

```
Integer value = null;
Optional<Integer> optional1 = Optional.of(value); // throws
                                                  // NullPointerException

Integer value = null;
Optional<Integer> optional1 = Optional.ofNullable(value); // returns
                                                          // Optional with
                                                          // empty value
```

В каких случаях нам нужно использовать `Optional`? Этот класс появился только в Java 8, до этого мы много лет жили без него.

В JavaDoc-ах к `Optional` точно сказано, что класс `Optional` создан в первую очередь для использования в качестве возвращаемого значения для методов, которые могут вернуть `null`.

```
Optional<MyEntity> findById(Long id); // OK.
```

Не стоит принимать `Optional` в качестве параметра. Вот так не нужно:

```
void myMethod(Optional<MyObject> arg0); // Плохо!
```

Почему не нужно принимать его в качестве параметра? Все очень просто. Если мы будем принимать `Optional` в качестве параметра метода, то нам придется вставлять в метод дополнительные проверки. Вместо одной проверки на `null` нам придется проверять на `null` и пустой `Optional` без значения.

И уж тем более ни в коем случае не стоит использовать `Optional` в качестве поля класса. В этом вообще нет никакого смысла, а также это может привести к проблемам в будущем, т. к. класс `Optional` несериализуемый (см. *главу 21 "Сериализация"*).

16.2. Задания

1. Представьте, что вы пишете код компьютерной игры. Создайте класс пещеры, в которой мог бы жить медведь, но при этом пещера могла бы быть и пустой. Напишите метод получения медведя, живущего в пещере.
2. Создайте заготовку класса, который мог бы обрабатывать клиентов банка. Напишите методы получения клиента банка по уникальным идентификаторам: серии и номеру паспорта, СНИЛС, ИНН. Учтите, что может быть передан такой идентификатор, клиента по которому не найдется.
3. Создайте класс, который мог бы использоваться в ГИБДД для получения информации о владельце машины по ее номеру.
4. Создайте класс человека. Добавьте к нему методы получения папы и мамы. Учтите, что родители могут быть неизвестны.

ГЛАВА 17

Модули

17.1. Теория

Модули появились в Java 9. В свое время они частично поломали обратную совместимость, но зато внутренние классы Java теперь разбиты на несколько модулей, что позволяет создавать свои JRE, которые включают в себя только те модули, которые нужны вашему приложению. Структура JDK и JRE одинакова, начиная с Java 9.

Модули представляют собой еще одну группировку классов и ресурсов над пакетами и jar-никами. Модуль — это имеющая уникальное имя группа связанных пакетов и ресурсов.

Обратите внимание, что проект с примерами `java-in-dynamics-2022` содержит несколько подкаталогов, в каждом из которых содержится свой файл pom.xml. Этот файл используется сборщиком Maven. О нем можно написать отдельную книгу, но нам сейчас для понимания модулей нужно в общих чертах узнать об основных принципах работы с ним.

Откройте файл pom.xml, который содержится внутри `java-in-dynamics-2022/general`:

java-in-dynamics-2022/general/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>java-in-dynamics-2022</artifactId>
        <groupId>ru.urvanov.javaindynamics2022</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>general</artifactId>

</project>
```

Сам проект `java-in-dynamics-2022` является мультимодульным Maven-проектом. Раздел parent указывает на родительский проект. В данном случае это проект с `groupId = ru.urvanov.javaindynamics2022`, `artifactId = java-in-dynamics-2022` и версией 1.0-SNAPSHOT.

`GroupId` обозначает уникальный идентификатор вашего проекта среди всех остальных проектов. Он должен следовать тем же правилам, что и наименования пакетов Java.

ArtifactId обозначает уникальное имя артефакта, который будет сгенерирован. Обычно он генерируется в виде `<artifactId>-<version>.<extension>`, например calculator-0.1-SNAPSHOT.jar.

В файле pom.xml также описываются и зависимости от других проектов в блоках dependencies, но в случае с general используются только классы самой JDK.

> ***Примечание***
>
> Имейте в виду, что модули IntelliJ IDEA, модули Maven и модули из Java 9 — это разные понятия.

Изучать что-либо новое лучше всего на примере. Давайте создадим простой калькулятор, у которого основной код будет находиться в главном модуле, а коды операций, которые он может выполнять, будут разбросаны по разным внешним модулям.

Исходный код всех примеров из этой главы находится в модулях `calculator`, `calculator-sum`, `calculator-minus` проекта с примерами этой книги.

Для начала давайте создадим главный модуль `calculator`. Каждый модуль содержит файл module-info.java, в котором описывается название модуля, экспортируемые пакеты, требуемые пакеты, сервисы, которые он реализует, и сервисы, которые он экспортирует для реализации.

Для нашего главного модуля калькулятора "module-info.java" начальный вариант будет выглядеть так:

module-info.java

```
module ru.urvanov.javaindynamics2022.calculator {
}
```

В этом варианте мы просто описали имя модуля ru.urvanov.javaindynamics2022.calculator. Наша модель будет экспортировать интерфейс математической операции, который будет реализовывать модули calculator-sum, calculator-minus и другие. Пусть этот интерфейс называется Operation:

Operation.java

```
package ru.urvanov.javaindynamics2022.calculator.plugin;

public interface Operation {
    double calculate(double x, double y);
}
```

Он расположен в пакете ru.urvanov.javaindynamics2022.calculator.plugin. Нам нужно экспортировать это содержимое из нашего модуля, чтобы оно было видно другим модулям, для этого используется exports в файле module-info.java:

module-info.java

```
module ru.urvanov.javaindynamics2022.calculator {
    exports ru.urvanov.javaindynamics2022.calculator.plugin;
}
```

Теперь создадим модуль с операцией сложения calculator-sum. В нем нужно создать файл module-info.java, в котором указывается, что он требует наличия основного модуля калькулятора с помощью requires, и с помощью provides указывается, что он предоставляет реализацию Operation.

calculator-sum/src/main/java/module-info.java

```
module ru.urvanov.javaindynamics2022.calculator.sum {
    requires ru.urvanov.javaindynamics2022.calculator;
    provides ru.urvanov.javaindynamics2022.calculator.plugin.Operation
            with ru.urvanov.javaindynamics2022.calculator.sum.Sum;
}
```

В requires мы указали имя модуля основного калькулятора, а в provides указали, что в нашем модуле calculator-sum есть класс Sum, который предоставляет реализа-

цию интерфейса `Operation` из основного модуля. Давайте теперь создадим сам класс `Sum`.

calculator-sum ... Sum.java

```
package ru.urvanov.javaindynamics2022.calculator.sum;

import ru.urvanov.javaindynamics2022.calculator.plugin.Operation;

public class Sum implements Operation {
    @Override
    public double calculate(double x, double y) {
        return x + y;
    }
}
```

Реализация довольно проста. Тут сложно что-то комментировать. Однако мы можем заметить, что наш проект не собирается, т. к. из модуля `calculator-sum` не виден модуль `calculator`. Мы указали через requires, что модуль `calculator-sum` требует наличия модуля `calculator`, но мы также должны указать сборщику Maven, что Maven-проект `calculator-sum` требует наличия модуля `calculator` в `classpath`. Для этого в "`pom.xml`" `calculator-sum` нужно добавить строки:

calculator-sum/pom.xml

```
<dependencies>
    <dependency>
        <groupId>ru.urvanov.javaindynamics2022</groupId>
        <artifactId>calculator</artifactId>
    </dependency>
</dependencies>
```

В блоке `dependency`, кроме `groupId` и `artifactId`, обычно еще указывают версию, но у нас мультимодульный проект Maven, поэтому информацию о версии нам нужно указать не в `pom.xml` от `calculator-sum`, а в корневом `pom.xml` родительского проекта в блоке `dependencyMangement`:

java-in-dynamics-2022/pom.xml

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>ru.urvanov.javaindynamics2022</groupId>
            <artifactId>calculator</artifactId>
            <version>${project.version}</version>
        </dependency>
```

```
    </dependencies>
</dependencyManagement>
```

Теперь мы можем попытаться использовать нашу операцию сложения в основном проекте калькулятора. Для этого нужно поправить файл module-info.java, добавив в него, что основной модуль калькулятора использует реализации интерфейса `Operation`:

calculator/pom.xml

```
module ru.urvanov.javaindynamics2022.calculator {
    exports ru.urvanov.javaindynamics2022.calculator.plugin;
    uses ru.urvanov.javaindynamics2022.calculator.plugin.Operation;
}
```

Теперь напишем основной класс калькулятора, который с помощью `ServiceLoader`-а получает все реализации `Operation`, поставляемые модулями и поочередно использует их.

Calculator.java

```
package ru.urvanov.javaindynamics2022.calculator;

import ru.urvanov.javaindynamics2022.calculator.plugin.Operation;

import java.util.ServiceLoader;
import java.util.stream.StreamSupport;

public class Calculator {
    public static void main(String[] args) {
        double x = 100.1;
        double y = 23.73;
        ServiceLoader<Operation> sl
                = ServiceLoader.load(Operation.class);
        sl.forEach(op -> {
            System.out.println(
                    "Operation: " + op.getClass().getName()
                    + ". Operands: " + x + ", " + y
                    + ". Result = " + op.calculate(x, y));
        });
    }
}
```

Код довольно простой. Мы просто запрашиваем через `ServiceLoader.load` доступные реализации `Operation`, а затем перебираем их через `forEach` и вызываем метод `calculate`.

Мы уже сейчас можем запустить класс `Calculator` на исполнение, но он пока не найдет ни одной реализации `Operation`, т. к. модуль `calculator-sum` ему будет недоступен в `Classpath`. Для исправления этого нам нужно указать дополнительный параметр JVM `--module-path`, в котором через двоеточие перечисляются все дополнительные каталоги, в которых находятся другие модули:

На рис. 17.1 параметр `--module-path` установлен в

```
--module-path calculator/target/classes:calculator-sum/target/
classes:calculator-minus/target/classes
```

В пути находятся три модуля: `calculator`, `calculator-sum`, `calculator-minus`. В этой статье было описано создание `calculator` и `calculator-sum`. Модуль `calculator-minus` создается аналогично.

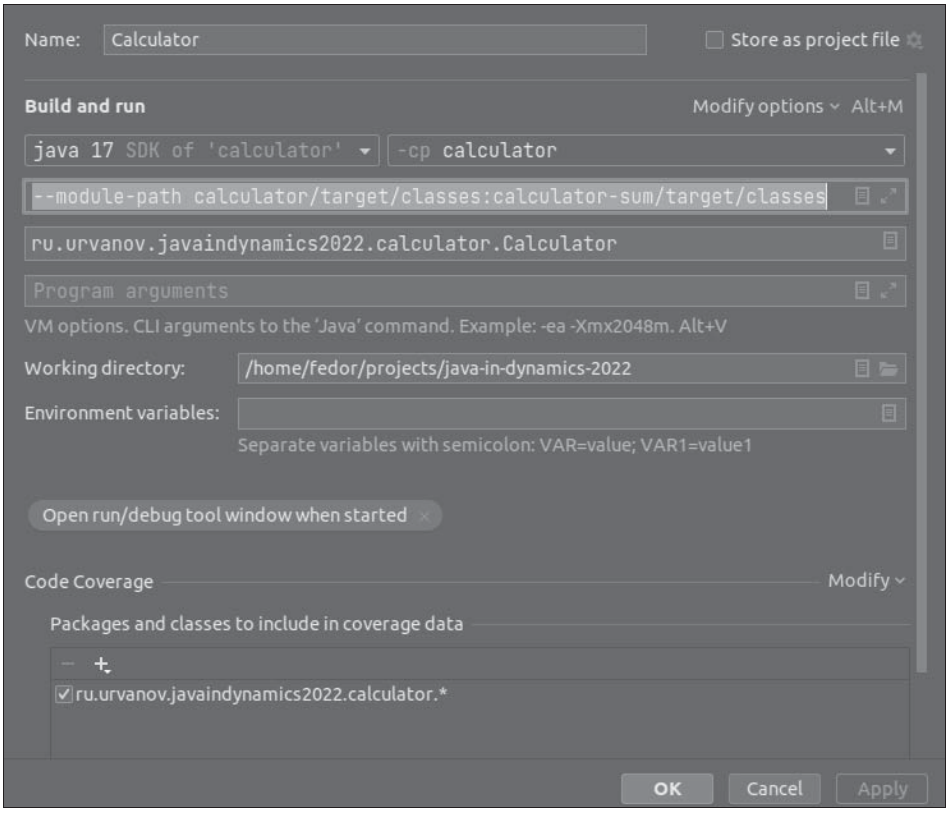


Рис. 17.1. Настройка Run Configuration для запуска Calculator в IntelliJ IDEA

17.2. Задания

1. Расширьте возможности калькулятора из предыдущего раздела. Добавьте операции возведения в степень и взятия остатка от деления. Каждую операцию располагайте в отдельном модуле.

2. Представьте, что вы разрабатываете программное обеспечение для уличных терминалов. Создайте основной модуль и модули с возможными операциями: пополнение телефона, пополнение транспортной карты, оплата ЖКХ, покупка валюты в MMO, оплата штрафов и т. д.
3. Придумайте и создайте модуль с базовым интерфейсом монстра для видеоигры. Напишите как минимум пять модулей, которые предоставляют реализации этого интерфейса. Необязательно делать интерфейс и сами реализации слишком сложными. В этом задании главное — разобраться с модулями в Java.

ГЛАВА 18

Обобщения

18.1. Введение

Обобщения (Generics) позволяют указать ограничения, накладываемые на поведение класса или методов, в терминах неизвестных типов.

> ***Важно***
> Обобщения Java и шаблоны C++ — это НЕ одно и то же. Обобщения могут показаться похожими на шаблоны C++, но это не одно и то же. Они работают совершенно по-другому. Не стоит переносить опыт работы с шаблонами C++ на обобщения в Java.

Java с самой первой версии позволяла использовать обобщенные классы, поскольку все объекты уже наследуются от `Object`. Механизм обобщений, появившийся в Java 5, позволил указывать ограничения на используемый класс, что упростило использование Java Collections Framework, описанный в *главе 26 "Коллекции"*.

18.2. Класс Lair

Посмотрите следующий код:

SimpleLair.java

```
package ru.urvanov.javaindynamics2022.generics;

// Логово
class SimpleLair {
```

```
    // Житель
    Object inhabitant;

    public void setInhabitant(Object inhabitant) {
        this.inhabitant = inhabitant;
    }

    public Object getInhabitant() {
        return this.inhabitant;
    }
}

class Goblin {
}

class Main {
    public static void main(String[] args) {
        SimpleLair lair = new SimpleLair();
        // указываем жителя.
        lair.setInhabitant(new Goblin());
        // Нужно явное приведение типа!
        Goblin goblin = (Goblin) lair.getInhabitant();
    }
}
```

Так как класс работает с `Object`, вы можете свободно поместить в жилище `SimpleLair` абсолютно любой класс. Нет никакого способа проверки использования класса во время компиляции. Одна часть кода может поселить в жилище гоблина `Goblin`, а другая — попытаться вытащить джинна `Genie`, что приведет к ошибке во время выполнения.

18.3. Обобщенная версия класса Lair

Для того чтобы класс `Lair` использовал обобщение, нужно сделать объявление обобщенного класса путем замены `class Lair` на `class Lair<T>`, что создаст переменную типа `T`, которую можно использовать в любом месте класса `Lair`.

С этими изменениями код класса `Lair` станет таким:

Lair.java

```
package ru.urvanov.javaindynamics2022.generics;

// Логово
class Lair<T> {

    // Житель
    T inhabitant;
```

```
    public void setInhabitant(T inhabitant) {
        this.inhabitant = inhabitant;
    }

    public T getInhabitant() {
        return this.inhabitant;
    }
}
```

Как вы видите, все вхождения `Object` заменены на `T`. Переменная типа может быть любым типом, кроме примитивных: любой класс, интерфейс или другая переменная типа.

> ***Примечание***
>
> Обобщенный класс не может быть прямым или косвенным наследником класса `java.lang.Throwable`, используемого при обработке исключений.

18.4. Соглашение об именовании переменных типа

По соглашению переменные типа именуются одной буквой в верхнем регистре. Это сильно отличается от соглашения об именовании переменных, классов и интерфейсов. Без такого отличия было бы трудно отличить переменную типа от класса и интерфейса.

Наиболее часто используемые имена для параметров типа:

- `E` — элемент (Element, обширно используется Java Collections Framework).
- `K` — Ключ.
- `N` — Число.
- `T` — Тип.
- `V` — Значение.
- `S`, `U`, `V` и т. п. — 2, 3, 4-й типы.

18.5. Создание экземпляра обобщенного типа и обращение к нему

При обращении к обобщенному типу нужно заменить параметры типа на конкретные классы или интерфейсы, например `Goblin`:

```
Lair<Goblin> goblinLair;
```

Параметр типа и аргумент типа — это два разных понятия. Когда вы объявляете обобщенный тип `Lair<T>`, то здесь `T` является параметром типа. Когда вы обращаетесь к обобщенному типу, вы передаете аргумент типа, например `Goblin`. Это довольно похоже на различие формальных параметров и аргументов методов.

Примечание

Нельзя использовать примитивные типы в качестве аргумента типа. Будет ошибка компиляции.

Объявление переменной `Lair<Goblin> goblinLair` НЕ создает экземпляр класса `Lair`. Такой код просто объявляет переменную `goblinLair` как `Lair Goblin`-ов.

Обращение к обобщенному типу обычно называется параметризованным типом (parameterized type). `Lair<Goblin>` в нашем случае будет параметризованным типом.

Можно параметризовать обобщенный класс параметризованным типом, тогда мы получим конструкции вида:

```
Lair<Pack<Goblin>> packGoblinsLair;
```

Пример создания экземпляра класса:

Lair.java

```
Lair<Goblin> goblinLair = new Lair<Goblin>();
```

Начиная с Java 7, можно использовать бриллиантовую операцию (diamond operator), которая позволяет указывать пустые аргументы типа <> там, где компилятор может вывести тип из контекста:

Lair.java

```
Lair<Goblin> goblinLair2 = new Lair<>();
```

После создания экземпляра можно обращаться к методам:

Lair.java

```
// указываем жителя.
goblinLair.setInhabitant(new Goblin());
// Приведение типа уже не нужно.
Goblin goblin = goblinLair.getInhabitant();
```

18.6. Бриллиантовая операция (Diamond operator)

Начиная с Java 7, существует также бриллиантовая операция (diamond operator), которая позволяет указывать пустые аргументы типа <> там, где компилятор может вывести тип из контекста:

Lair.java

```
Lair<Goblin> goblinLair2 = new Lair<>();
```

18.7. Несколько параметров типа

Обобщенный тип может иметь несколько параметров типа:

PairLair.java

```
package ru.urvanov.javaindynamics2022.generics;

class PairLair<T, S> {
    T inhabitant1;
    S inhabitant2;

    public void setInhabitant1(T inhabitant1) {
        this.inhabitant1 = inhabitant1;
    }

    public T getInhabitant1() {
        return this.inhabitant1;
    }

    public void setInhabitant2(S inhabitant2) {
        this.inhabitant2 = inhabitant2;
    }

    public S getInhabitant2() {
        return this.inhabitant2;
    }

    public static void main(String[] args) {
        PairLair<Goblin, Genie> goblinGenieLair = new PairLair<>();
        goblinGenieLair.setInhabitant1(new Goblin());
        goblinGenieLair.setInhabitant2(new Genie());
        Goblin goblin = goblinGenieLair.getInhabitant1();
        Genie genie = goblinGenieLair.getInhabitant2();
    }
}

class Genie {
}
```

18.8. Сырой тип (Raw type)

Сырой тип (raw type) — это имя обобщенного класса или интерфейса без аргументов типа (type arguments).

Например, параметризованный тип создается так:

RawType.java

```
Lair<Goblin> goblinLair1 = new Lair<>();
```

Если убрать аргументы типа, то будет создан сырой тип:

RawType.java

```
Lair lair1 = new Lair();
```

Поэтому `Lair` — это сырой тип обобщенного типа `Lair<T>`. Однако необобщенный класс или интерфейс НЕ являются сырыми типами.

Вы можете часто увидеть использование сырых типов в старом коде, поскольку многие классы, например коллекции, до Java 5 были необобщенными. Когда вы используете сырые типы, вы по сути получаете то же самое поведение, которое было до введения обобщений в Java.

Для совместимости со старым кодом допустимо присваивать параметризованный тип своему собственному сырому типу:

RawType.java

```
Lair<Goblin> goblinLair2 = new Lair<>();
Lair lair2 = goblinLair2; // OK
```

Но если вы попытаетесь присвоить параметризованному типу сырой тип, то будет предупреждение (warning):

RawType.java

```
Lair lair3 = new Lair();
Lair<Goblin> goblinLair3 = lair3; // warning
```

Вы также получите предупреждение (warning), если попытаетесь вызвать обобщенный метод в сыром типе:

RawType.java

```
Lair<Goblin> goblinLair4 = new Lair<>();
Lair lair4 = goblinLair4;
lair4.setInhabitant(new Goblin()); // warning
```

Предупреждение показывает, что сырой тип обходит проверку обобщенного типа, что откладывает обнаружение ошибки на выполнение программы.

18.9. Сообщения об ошибках "unchecked"

Как упоминалось выше, при использовании сырого типа вы можете столкнуться с предупреждениями вида:

```
Note: Main.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Термин "unchecked" означает "непроверенные", т. е. компилятор не имеет достаточного количества информации для обеспечения безопасности типов. По умолчанию этот вид предупреждений выключен, поэтому компилятор дает подсказку. Чтобы видеть все "unchecked" предупреждения, нужно перекомпилировать код с опцией `-Xlint:unchecked`:

```
javac -Xlint:unchecked Main.java
```

Предупреждения будут такого вида:

```
Main.java:48 warning: [unchecked] unchecked call to setInhabitant(T) as member
of the raw type Lair
        lair.setInhabitant(new Goblin()); //warning
                           ^
  where T is type-variable:
    T extends Object declared in class Lair
1 warning
```

Чтобы полностью отключить подобные предупреждения, нужно перекомпилировать с опцией `-Xlint:-unchecked`. Можно также использовать аннотацию `@SuppressWarnings("unchecked")`, чтобы отключить эти предупреждения для поля, метода, параметра, конструктора или локальной переменной.

18.10. Обобщенные методы

Обобщенные методы похожи на обобщенные классы, но параметры типа относятся к методу, а не к классу. Допустимо делать обобщенными статические и нестатические методы, а также конструкторы.

Синтаксис обобщенного метода включает параметры типа внутри угловых скобок, которые указываются перед возвращаемым типом.

Utils.java

```
package ru.urvanov.javaindynamics2022.generics;

// Пример использования обобщенного метода
class Utils {
    // Обобщенный метод.
    static <T> void setIfNull(Lair<T> lair, T t) {
        if (lair.getInhabitant() == null) {
            lair.setInhabitant(t);
        }
    }
}
```

Пример вызова обобщенного метода:

Utils.java

```
Lair<Goblin> goblinsLair = new Lair<>();
Utils.<Goblin>setIfNull(goblinsLair, new Goblin());
```

Здесь тип `Goblin` указан явно, но обычно он может быть опущен, и компилятор выведет тип из контекста:

Utils.java:

```
Utils.setIfNull(goblinsLair, new Goblin());
```

Более подробно выведение типа будет описано ниже.

18.11. Ограниченные параметры типа

Иногда нужно ограничить типы, которые можно использовать в качестве аргументов в параметризованных типах. Например, в жилище `Lair` могут жить только наследники класса `Monster`. Подобное ограничение можно сделать с помощью ограниченного параметра типа (bounded type parameters).

Чтобы объявить ограниченный параметр типа, нужно после имени параметра указать ключевое слово `extends`, а затем — верхнюю границу (upper bound), которой в данном примере является класс `Monster`. В этом контексте `extends` означает как `extends`, так и `implements`.

LairForMonster.java

```
package ru.urvanov.javaindynamics2022.generics;

// У параметра типа указываем верхнюю границу Monster.
class LairForMonster<T extends Monster> {

    T inhabitant;

    public void setInhabitant(T inhabitant) {
        this.inhabitant = inhabitant;
    }

    public T getInhabitant() {
        return this.inhabitant;
    }

    public void tick() {
        if (inhabitant != null) {
```

```
            // Можно вызывать методы интерфейса или класса,
            // указанного в качестве верхней границы параметра типа.
            inhabitant.doSomething();
        }
    }

    public static void main(String[] args) {
        LairForMonster<Barghest> barghestLair = new LairForMonster<>();
        barghestLair.setInhabitant(new Barghest());
        barghestLair.tick();
    }
}

class Monster {
    public void doSomething() {
        System.out.println("Doing something.");
    }
}

class Barghest extends Monster {
}
```

В этом примере мы ограничили возможные типы, которые можно использовать в параметризованных классах `Lair`, наследниками класса `Monster`. Если попытаться указать, например, `Lair<Integer>`, то возникнет ошибка компиляции. В дополнение мы получили возможность вызывать в обобщенном классе методы класса `Monster` (`inhabitant.doSomething();`).

Можно указать несколько верхних границ, перечисляя их через символ `&`, но при этом только один класс может быть указан в списке верхних границ, и он должен стоять первым:

LairForDreadfulEnemyMonster.java

```
package ru.urvanov.javaindynamics2022.generics;

interface Enemy {}

interface Dreadful {}

// Указываем несколько верхних границ.
// Если в списке верхних границ есть класс,
// то он обязательно должен идти первым.
class LairForDreadfulEnemyMonster<T extends Monster & Enemy & Dreadful> {

}
```

Аналогичным образом можно создавать обобщенные методы с ограничением:

BoundedGenericMethod.java

```
package ru.urvanov.javaindynamics2022.generics;

public class BoundedGenericMethod {
    // Обобщенный метод.
    static <T> void setIfNull(Lair<T> lair, T t) {
        if (lair.getInhabitant() == null) {
            lair.setInhabitant(t);
        }
    }

    // Ограниченный обобщенный метод
    public static <T extends Monster & Enemy & Dreadful>
            void doSomething(T[] monsters) {
        T result = null;
        for (T obj : monsters) {

            obj.doSomething();
        }
    }

    static class FoxSpirit extends Monster implements Enemy, Dreadful {

    }

    public static void main(String[] args) {
        // Вызов ограниченного обобщенного метода.
        BoundedGenericMethod.FoxSpirit[] foxSpirits =
                                        new BoundedGenericMethod.FoxSpirit[10];
        for (int n = 0; n < foxSpirits.length; n++) {
            foxSpirits[n] = new BoundedGenericMethod.FoxSpirit();
        }
        BoundedGenericMethod.doSomething(foxSpirits);
    }
}
```

18.12. Обобщения, наследование и дочерние типы

Как вы уже знаете, можно присвоить объекту одного типа объект другого типа, если эти типы совместимы. Например, вы можете присвоить объект типа `Integer` переменной типа `Object`, т. к. `Object` является одним из супертипов `Integer`:

```
Object someObject = new Object();
Integer someInteger = Integer.valueOf(10);
someObject = someInteger;   // OK
```

В объектно-ориентированной терминологии это называется связью "является" ("is a"). Так как `Integer` является `Object`-ом, то такое присвоение разрешено. Но `Integer` также является и `Number`, поэтому следующий код тоже корректен:

```
public void someMethod(Number n) { /* ... */ }

someMethod(Integer.valueOf(10));   // OK
someMethod(Double.valueOf(10.1));   // OK
```

Это также верно для обобщений. Вы можете осуществить вызов обобщенного типа, передав `Number` в качестве аргумента типа, и любой дальнейший вызов будет разрешен, если аргумент совместим с `Number`:

Box.java

```
package ru.urvanov.javaindynamics2022.generics;

public class Box<T> {

    public void add(T value) {
        // добавление value в Box
    }
    public static void main(String[] args) {
        Box<Number> box = new Box<Number>();
        box.add(Integer.valueOf(10));   // OK
        box.add(Double.valueOf(10.1));  // OK
    }
}
```

Теперь рассмотрим метод:

Box.java

```
public void boxTest(Box<Number> n) { /* ... */ }
```

Какой тип аргумента он будет принимать? Если посмотрите на сигнатуру, то вы можете увидеть, что он принимает один аргумент с типом `Box<Number>`. Но что это означает? Можете ли вы передать `Box<Integer>` или `Box<Double>`, как вы могли бы ожидать? Нет, не можете, т. к. `Box<Integer>` и `Box<Double>` не являются потомками `Box<Number>`.

Это частое недопонимание принципов работы обобщений, и это важно знать.

> ***Запомните***
>
> Для двух типов `A` и `B` (например, `Number` и `Integer`) `MyClass<A>` не имеет никакой связи или родства с `MyClass<B>`, независимо от того, как `A` и `B` связаны между собой. Общий родитель `MyClass<A>` и `MyClass<B>` — это `Object`.

Информацию о способах создания связи, подобной потомок-родитель, между двумя обобщенными типами, когда параметры типа связаны, см. в *разделе 18.21 "Подстановочные символы и дочерние типы"*.

Вы можете указать обобщенный класс или интерфейс в качестве родительского для своего класса или интерфейса. Связь между параметрами типа одного класса или интерфейса и другого определяются ключевыми словами `extends` и `implements`.

Для классов коллекций, например, `ArrayList<E> implements List<E>` и `List<E> extends Collection<E>`. Также `ArrayList<String>` является дочерним типом для `List<String>`, который является дочерним типом `Collection<String>`. Наследование между типами сохраняется, пока вы не меняете аргумент типа.

Давайте представим, что мы хотим определить свой собственный интерфейс списка `PayloadList`, который связывает необязательное значение `P` с каждым элементом. Объявление может выглядеть так:

```
interface PayloadList<E,P> extends List<E> {
    void setPayload(int index, P val);
    ...
}
```

Следующие параметризованные типы являются дочерними типами для `List<String>`, но не связаны между собой:

- `PayloadList<String,String>`
- `PayloadList<String,Integer>`
- `PayloadList<String,Exception>`

18.13. Выведение типов

Выведение типов (type inference) — это возможность компилятора Java автоматически определять аргументы типа на основе контекста, чтобы вызов получился возможным. Алгоритм выведения типов определяет типы аргументов и, если есть, тип, в который присваивается результат или в котором возвращается результат. Далее алгоритм пытается найти наиболее конкретный тип, который работает со всеми аргументами.

В приведенном ниже примере выведение типов определяет, что второй аргумент, передаваемый в метод `pick`, имеет тип `Serializable`:

TypeInferenceSimple.java

```
package ru.urvanov.javaindynamics2022.generics;

import java.io.Serializable;
import java.util.ArrayList;

public class TypeInferenceSimple {
    static <T> T pick(T a1, T a2) { return a2; }
```

```
    public static void main(String[] args) {
        Serializable s = pick("d", new ArrayList<String>());
    }
}
```

18.14. Выведение типов и обобщенные методы

В описании обобщенных методов уже рассказывалось о выведении типов, которое делает возможным вызов обобщенного метода так, будто это обычный метод, без указания типа в угловых скобках <>. Рассмотрим этот пример:

LairDemo.java

```
package ru.urvanov.javaindynamics2022.generics;

class Chimera {

    private String name;

    public Chimera(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return this.name;
    }

}

class LairDemo {

    public static <U> void addLair(U u, java.util.List<Lair<U>> lairs) {
        Lair<U> lair = new Lair<>();
        lair.setInhabitant(u);
        lairs.add(lair);
    }

    public static <U> void outputLairs(java.util.List<Lair<U>> lairs) {
        int counter = 0;
        for (Lair<U> lair : lairs) {
            U lairInhabitant = lair.getInhabitant();
            System.out.println("Lair #" + counter + " contains ["
                    + lairInhabitant.toString() + "]" );
            counter++;
        }
    }
```

```
    public static void main(String[] args) {
        java.util.ArrayList<Lair<Chimera>> listOfChimerasLairs
                = new java.util.ArrayList<>();
        LairDemo.<Chimera>addLair(new Chimera("Michael"), listOfChimerasLairs);
        LairDemo.addLair(new Chimera("Rafael"), listOfChimerasLairs);
        LairDemo.addLair(new Chimera("Pushkin"), listOfChimerasLairs);
        LairDemo.outputLairs(listOfChimerasLairs);
    }

}
```

Этот код выведет в консоль следующее:

```
Lair #0 contains [Michael]
Lair #1 contains [Rafael]
Lair #2 contains [Pushkin]
```

Обобщенный метод `addLair` объявляет один параметр типа `U`. В большинстве случаев компилятор Java может вывести параметры типа вызова обобщенного метода, в результате вам чаще всего вовсе не обязательно их указывать. Например, чтобы вызвать обобщенный метод `addBox`, вы можете указать параметры типа так:

LairDemo.java

```
LairDemo.<Chimera>addLair(new Chimera("Michael"), listOfChimerasLairs);
```

Либо вы можете опустить их, и тогда компилятор Java автоматически выведет тип `Chimera` из аргументов метода:

LairDemo.java

```
LairDemo.addLair(new Chimera("Rafael"), listOfChimerasLairs);
```

18.15. Выведение типов и создание экземпляра обобщенного класса

Рассмотрим следующее объявление переменной:

DiamondOperator.java

```
Map<String, List<String>> myMap1 = new HashMap<String, List<String>>();
```

Вы можете заменить параметризованный тип конструктора пустыми угловыми скобками (<>), так называемой бриллиантовой операцией (diamond operator):

DiamondOperator.java

```
Map<String, List<String>> myMap2 = new HashMap<>();
```

Обратите внимание: для того чтобы воспользоваться выведением типов при создании экземпляра обобщенного класса, вы должны использовать бриллиантовую операцию (diamond operator). В примере ниже компилятор сгенерирует предупреждение unchecked conversion warning, т. к. конструктор `HashMap()` обращается к сырому типу `HashMap`, а не к `Map<String, List<String>>`:

DiamondOperator.java

```
Map<String, List<String>> myMap3 = new HashMap(); // unchecked
                                                  // conversion warning
```

18.16. Выведение типа и обобщенные конструкторы обобщенных и необобщенных классов

Конструкторы могут быть обобщенными как в обобщенных, так и в необобщенных классах. Рассмотрим пример:

GenericConstructor.java

```
package ru.urvanov.javaindynamics2022.generics;

class GenericConstructor<X> {
    <T> GenericConstructor(T t) {
        // ...
    }
}
```

Рассмотрим создание экземпляра класса GenericConstructor:

GenericConstructor.java

```
new GenericConstructor<Integer>("");
```

Эта инструкция создает экземпляр параметризованного типа `GenericConstructor<Integer>`. Инструкция явно указывает Integer в качестве формального параметра типа `X` обобщенного класса `MyClass<X>`. Обратите внимание, что конструктор этого обобщенного класса содержит параметр типа `T`. Компилятор выводит тип `String` для этого формального параметра `T`, т. к. фактически переданный аргумент является экземпляром класса `String`.

Компилятор Java 7 и более поздней версии может вывести аргументы типа создаваемого экземпляра обобщенного класса с помощью бриллиантовой операции (diamong operator). Пример:

GenericConstructor.java

```
GenericConstructor<Integer> myObject = new GenericConstructor<>("");
```

В этом примере компилятор выводит Integer для параметра типа X обобщенного класса MyClass<X>. Он выводит тип String для параметра T обобщенного конструктора обобщенного класса.

Важно запомнить, что алгоритм выведения типа использует только аргументы вызова, целевые типы и, возможно, очевидный ожидаемый возвращаемый тип для выведения типов. Алгоритм выведения не использует последующий код программы.

18.17. Целевые типы

Компилятор Java пользуется целевыми типами для вывода параметров типа вызова обобщенного метода. Целевой тип выражения — это тип данных, который компилятор Java ожидает в зависимости от того, в каком месте находится выражение. Рассмотрим метод Collections.emptyList(), который объявлен так:

```
static <T> List<T> emptyList();
```

Рассмотрим следующую инструкцию присвоения:

TargetType.java

```
List<String> listOne = Collections.emptyList();
```

Эта инструкция ожидает экземпляр List<String>. Этот тип данных является целевым типом. Поскольку метод emptyList возвращает значение типа List<T>, компилятор выводит, что аргумент типа T будет типом String. Это работает как в Java 7, так и в Java 8. Вы также можете указывать аргумент типа напрямую:

TargetType.java

```
List<String> listTwo = Collections.<String>emptyList();
```

Но в данном случае в этом нет необходимости. Это может быть необходимо в других случаях. Рассмотрим метод:

TargetType.java

```
void processStringList(List<String> stringList) {
    // process stringList
}
```

Представьте, что вы хотите вызвать метод processStringList с пустым списком. В Java 7 следующий код не будет работать:

TargetType.java

```
processStringList(Collections.emptyList());
```

Компилятор Java 7 сгенерирует примерно такую ошибку:

```
List<Object> cannot be converted to List<String>
```

Компилятору необходимо значение аргумента типа для `T`, и он начинает с Object. В результате вызов `Collections.emptyList` возвращает тип `List<Object>`, который несовместим с методом `processStringList`. Таким образом, в Java 7 вы должны указать аргумент типа так:

TargetType.java

```
processStringList(Collections.<String>emptyList());
```

В Java 8 в этом больше нет необходимости. Термин "целевой тип" расширен и включает аргументы методов. В этом случае `processStringList-y` необходим аргумент типа `List<String>`. Метод `Collections.emptyList` возвращает значение типа `List<T>`, компилятор выводит аргумент типа `T` как `String`, используя целевой тип `List<String>`. Таким образом, в Java 8 следующая инструкция успешно скомпилируется:

TargetType.java

```
processStringList(Collections.emptyList());
```

18.18. Подстановочный символ (wildcard)

В обобщенном коде знак вопроса (`?`), называемый подстановочным символом, означает неизвестный тип. Подстановочный символ может использоваться в разных ситуациях: как параметр типа, поля, локальной переменной, иногда в качестве возвращаемого типа. Подстановочный символ никогда не используется в качестве аргумента типа для вызова обобщенного метода, создания экземпляра обобщенного класса или супертипа.

18.19. Подстановочный символ, ограниченный сверху (Upper bounded wildcard)

Вы можете использовать подстановочный символ, ограниченный сверху, чтобы ослабить ограничения переменной. Например, если вы хотите написать метод, который работает с `List<Monster>`, `List<Double>` и `List<Number>`, вы можете достичь этого с помощью ограниченного сверху подстановочного символа.

Чтобы объявить ограниченный сверху подстановочный символ, используйте символ вопроса `?` с последующим ключевым словом `extends` и последующим ограничением сверху. Запомните, что в этом контексте `extends` означает как расширение класса, так и реализацию интерфейса.

Чтобы написать метод, который работает со списками `UpperMonster` и дочерними типами от `UpperMonster`, например `UpperDaemon` и `UpperGreatDaemon`, вы можете указать `List<? extends UpperMonster>`. `List<UpperMonster>` вводит более жесткое ограничение, чем `List<? extends UpperMonster>`, потому что оно соответствует только списку типа `UpperMonster`, а `List<? extends UpperMonsterr>` соответствует списку типа `UpperMonster` и спискам всех его подклассов.

Рассмотрим следующий метод process:

UpperBoundedWildcard.java

```
public static void process(List<? extends Monster> list) { /* ... */ }
```

Ограниченный сверху подстановочный символ `<? extends Monster>`, где `Monster` — любой тип, соответствует `Monster` и любому подтипу `Monster`. Метод `process` может обращаться к элементу списка как к типу `Monster`:

UpperBoundedWildcard.java

```
public static void process(List<? extends Monster> list) {
    for (Monster elem : list) {
        // ...
    }
}
```

18.20. Неограниченный подстановочный символ (Unbounded wildcard)

Если просто использовать подстановочный символ, то получится подстановочный символ без ограничений. `List<?>` означает список неизвестного типа.

Неограниченный подстановочный символ полезен в двух случаях:

- Если вы пишете метод, который может быть реализован с помощью функциональности класса `Object`.
- Когда код использует методы обобщенного класса, которые не зависят от параметра типа. Например, `List.size()` или `List.clear()`. В реальности `Class<?>` используется так часто, потому что большинство методов `Class<T>` не зависят от `T`.

Рассмотрите следующий метод printList:

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

Цель метода printList — вывод в консоль списка любого типа, но сейчас он ее не выполняет, т. к. он может вывести в консоль только список объектов типа `Object`. Он не может принимать в качестве параметра `List<Monster>`, `List<Spirit>`, `List<FireDaemon>` и какие-либо еще, т. к. они не являются дочерними типами для `List<Object>`. Чтобы сделать этот метод более общим, нужно использовать `List<?>`:

UnboundedWildcard.java

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

`List<A>` является дочерним типом для `List<?>` для любого конкретного типа `A`, поэтому вы можете использовать `printList` для вывода в консоль списков любого типа:

UnboundedWildcard.java

```
List<BigDecimal> li = Arrays.asList(
   new BigDecimal("100.01"), new BigDecimal("200.3"));
List<String> ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

> ***ЗАМЕТКА***
>
> Метод `Arrays.asList` конвертирует массив и возвращает список фиксированного размера.

Важно запомнить, что `List<Object>` и `List<?>` — это НЕ одно и то же. Вы можете вставить `Object` или любой дочерний тип от `Object` в `List<Object>`. Но вы можете вставить только `null` в `List<?>`.

Ограниченный снизу подстановочный символ ограничивает неизвестный тип так, чтобы он был либо указанным типом, либо одним из его предков.

Вы можете указать либо только верхнюю границу для подстановочного символа, либо только нижнюю, но вы не можете указать оба ограничения сразу.

Например, чтобы написать метод, который работает со списком `GreatDaemon`-ов и супертипами `GreatDaemon`-а (в примере ниже это будут `LowerDaemon` и `LowerMonster`), вы можете указать `List<? super LowerGreatDaemon>`. Вариант `List<LowerGreatDaemon>` более ограничен, чем `List<? super LowerGreatDaemon>`, потому что он позволяет использовать только список объектов типа `LowerGreatDaemon`, тогда как `List<? super LowerGreatDaemon>` соответствует спискам любого родительского класса от `LowerGreatDaemon` и списку `LowerGreatDaemon`-ов.

Следующий код добавляет числа от 1 до 10 в конец списка:

LowerBoundedWildcard.java

```
package ru.urvanov.javaindynamics2022.generics;

import java.util.List;

public class LowerBoundWildcard {

    static class LowerMonster {};

    static class LowerDaemon extends LowerMonster {};

    static class LowerGreatDaemon extends LowerDaemon {};

    public static void processs(List<? super LowerGreatDaemon> list) {
        list.add(new LowerGreatDaemon());
    }
}
```

18.21. Подстановочные символы и дочерние типы

Как было описано в *разделе 18.12 "Обобщения, наследование и дочерние типы"*, обобщенные классы или интерфейсы связаны не только из-за связи между их типами. Однако вы можете использовать подстановочные символы (wildcards) для создания связи между обобщенными классами и интерфейсами.

С данными обычными (необобщенными) классами:

```
class Monster { /* ... */ }
class Goblin extends Monster { /* ... */ }
```

имеет смысл написать вот такой код:

```
Goblin goblin = new Goblin();
Monster monster = goblin;
```

Этот пример показывает, что наследование следует правилу подчиненных типов: класс `Goblin` является подклассом класса `Monster`, если он расширяет его. Это правило НЕ работает для обобщенных типов:

```
List<Goblin> listGoblins = new ArrayList<>();
List<Monster> listMonsters = listGoblins;   // compile-time error
```

> ***Примечание***
>
> Несмотря на то что `Goblin` является подтипом `Monster`, `List<Goblin>` не является подтипом `List<Monster>`. Это разные типы. Общим предком для `List<Monster>` и `List<Goblin>` является `List<?>`.

Для того чтобы создать такую связь между этими классами, нужно использовать подстановочный символ:

```
List<? extends Goblin> goblins = new ArrayList<>();
List<? extends Monster>  monsters = intList;  // OK.
                // List<? extends Goblin> дочерний тип
                // от List<? extends Monster>
                // Его можно передавать в качестве параметра методам,
                // ожидающим List<? extends Monster>.
```

Так как `Goblin` является дочерним типом от `Monster`, и monsters является списком объектов типа `Monster`, теперь существует связь между `goblins` и `monsters` (рис. 18.1).

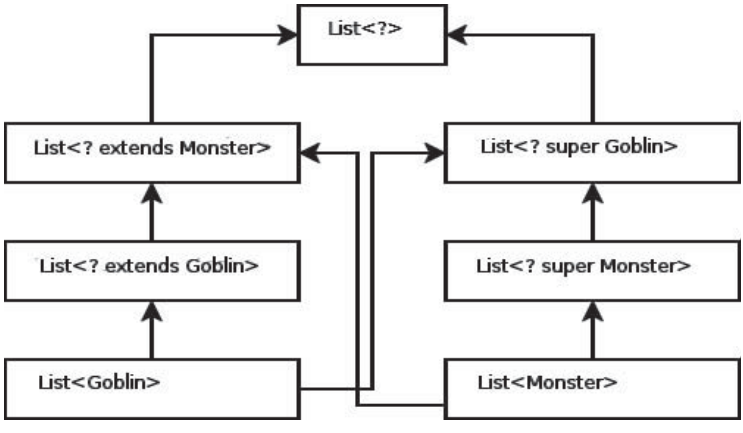


Рис. 18.1. Связь между разными `List`, ограниченными подстановочными символами

18.22. Захват символа подстановки (Wildcard Capture) и вспомогательные методы

В некоторых случаях компилятор может вывести тип подстановочного символа. Список может быть определен как `List<?>`, но при вычислении выражения компилятор выведет конкретный тип из кода. Этот сценарий называется захватом подстановочного символа.

В большинстве случаев вам нет нужды беспокоиться о захвате подстановочного символа, кроме случаев, когда вы видите фразу "capture of" в сообщении об ошибке.

Следующий код при компиляции выводит сообщение об ошибке, связанное с захватом подстановочного символа:

WildcardEngineExample.java

```
package ru.urvanov.javaindynamics2022.generics;

import java.util.List;
```

```
public class WildcardEngineExample {
    class Ant{};
    class WarAnt extends Ant{};
    class WorkerAnt extends Ant{};
    void processAntsError(List<?> ants) {
        ants.set(0, ants.get(0));
    }
}
```

В этом примере компилятор обрабатывает параметр ants как тип `Object`. Когда метод `processAntsError` вызывает `List.set(int, E)`, компилятор не может подтвердить тип объекта, который будет вставляться в список, и генерирует ошибку. Когда возникает этот тип ошибки, это обычно означает, что компилятор верит, что вы присваиваете неправильный тип переменной. Обобщения были добавлены в Java именно для этого: чтобы усилить безопасность типов во время компиляции.

При компиляции в JDK 8 кода, приведенного выше, компилятор выдаст следующее сообщение об ошибке:

```
WildcardEngineExamlpe.java:14:29
Java: incompatible types: Object cannot be converted to capture#1 of ?
        i.set(0, i.get(0));
```

В нашем же примере код пытается выполнить безопасную операцию, тогда как мы можем обойти ошибку компиляции. Вы можете исправить ее, написав приватный вспомогательный метод, который захватывает подстановочный символ. В этом случае вы можете обойти проблему с помощью создания приватного вспомогательного метода `fooHelper()`:

WildcardFixed.java

```
void processAntsFixed(List<?> ants) {
    processAntsHelper(ants);
}

// Вспомогательный метод создан так, чтобы подстановочный символ
// мог быть захвачен через выведение типа.
private <T> void processAntsHelper(List<T> ants) {
    ants.set(0, ants.get(0));
}
```

Благодаря вспомогательному методу компилятор использует выведение типа для определения, что `T` является захваченной переменной в вызове. Пример теперь успешно компилируется.

По соглашению вспомогательные методы обычно называются как `originalMethodNameHelper`.

Теперь рассмотрите более сложный пример, `WildcardErrorBad`:

WildcardEngineExample.java

```
void swapAnts(List<? extends Ant> l1, List<? extends Ant> l2) {
    Ant ant = l1.get(0);

    // Нельзя, ошибка компиляции
    // l1.set(0, l2.get(0));

    // Нельзя, ошибка компиляции
    // l2.set(0, ant);
}
```

В этом примере код пытается выполнить небезопасную операцию.

Представьте, что мы передаем в метод `List<WarAnt>` и `List<WorkerAnt>`; оба удовлетворяют критерию `List<? extends Ant>`, но это совершенно неверно брать элемент из списка объектов типа `WarAnt` и пытаться добавить его в список объектов типа `WorkerAnt`, поэтому такой код компилироваться не будет.

18.23. Руководство по использованию подстановочного символа

Когда использовать ограниченный сверху подстановочный символ (wildcard), а когда использовать ограниченный снизу подстановочный символ, определить зачастую бывает довольно сложно. Здесь собраны советы по выбору необходимого ограничения для подстановочного символа.

В этом обсуждении будет полезно думать о переменных, будто они представляют две функции:

- Входная переменная. Предоставляет данные для кода. Для метода `copy(src, dst)` параметр `src` предоставляет данные для копирования, поэтому он считается входной переменной.
- Выходная переменная. Содержит данные для использования в другом месте. В примере с `copy(src, dst)` параметр `dst` принимает данные и является выходной переменной.

Некоторые переменные могут быть одновременно входными и выходными, такой случай тоже здесь рассматривается.

Руководство:

- Входная переменная определяется с ограниченным сверху подстановочным символом, используя ключевое слово `extends`.
- Выходная переменная определяется с ограниченным снизу подстановочным символом, используя ключевое слово `super`.

- Если ко входной переменной можно обращаться, только используя методы класса `Object`, используйте неограниченный подстановочный символ.
- Если переменная должна использоваться как входная и как выходная одновременно, то НЕ используйте подстановочный символ.

Это руководство не охватывает использование подстановочных символов в возвращаемых из методов типах. Не используйте подстановочные символы в возвращаемых типах, потому что это будет принуждать других программистов разбираться с подстановочными символами.

Список, объявленный как `List<? extends ...>`, может неформально считаться как список только для чтения, но это нестрогое ограничение. Предположим, что у вас есть следующие два класса:

CryptoCoin.java

```
package ru.urvanov.javaindynamics2022.generics;

import java.util.ArrayList;
import java.util.List;

public class CryptoCoin {
    private String name;

    public CryptoCoin(String name) {
        this.name = name;
    }
}

class ElvenCoin extends CryptoCoin {
    public ElvenCoin() {
        super("ELC");
    }
}
```

Рассмотрите следующий код:

```
List<ElvenCoin> ec = new ArrayList<>();
List<? extends CryptoCoin> cc = ec;
// ошибка компиляции
// cc.add(new CryptoCoin("DGC"));
```

Так как `List<ElvenCoin>` является дочерним типом от `List<? extends CryptoNumber>`, то вы можете присвоить его переменной `cc`. Но вы не можете использовать cc для добавления `CryptoCoin` в список объектов `ElvenCoin`. Над списком возможны следующие операции:

- Вы можете добавить в список `null`.
- Вы можете вызвать `clear`.

- Вы можете получить iterator и вызвать `remove`.
- Вы можете захватить подстановочный символ и записывать элементы, которые вы прочитали из списка.

Вы можете увидеть, что список `List<? extends CryptoCoin>` НЕ является списком только для чтения, но можно считать его таким, потому что туда нельзя добавить новый элемент или заменить существующий элемент в списке.

18.24. Стирание типа (Type Erasure)

Обобщения были введены в язык программирования Java для обеспечения более жесткого контроля типов во время компиляции и для поддержки обобщенного программирования. Для реализации обобщения компилятор Java применяет стирание типа (type erasure):

- Заменяет все параметры типа в обобщенных типах их границами или `Object`-ами, если параметры типа не ограничены. Сгенерированный байт-код содержит только обычные классы, интерфейсы и методы.
- Вставляет приведение типов, где необходимо, чтобы сохранить безопасность типа.
- Генерирует связующие методы, чтобы сохранить полиморфизм в расширенных (extended, наследующиеся от других) обобщенных типах.

Стирание типа обеспечивает, что никакие новые классы не создаются для параметризованных типов, следовательно, обобщения не приводят к накладным расходам во время выполнения.

> ***Важно***
>
> Обобщения в Java — это только "синтаксический сахар", позволяющий нам писать более безопасный код. На этапе выполнения программы нет никакой разницы между обобщенным классом / методом и классом / методом, использующим `Object`

TypeErasure.java

```
Lair<Monster> monsterLair = new Lair<>();
Lair<Integer> integerLair = new Lair<>();
// Выведет true
System.out.println(monsterLair.getClass() == integerLair.getClass());
```

18.25. Стирание типа в обобщенных типах

Во время процесса стирания типов компилятор Java стирает все параметры типа и заменяет каждый из них его ограничением, если параметр типа ограничен, либо `Object`-ом, если параметр типа неограничен.

Например, `Lair<T>` станет `Lair<Object>`, а `Lair<? extends Monster>` станет `Lair<Monster>`.

18.26. Стирание типа в обобщенных методах

Компилятор Java также стирает параметры типа обобщенных методов. Рассмотрите следующий обобщенный метод.

Предположим, что объявлены следующие классы:

```
class Monster { /* ... */ }
class Goblin extends Monster { /* ... */ }
class Viy extends Monster { /* ... */ }
```

Вы можете написать метод, рисующий разных монстров:

```
public static <T extends Monster> void draw(T monster) { /* ... */ }
```

компилятор Java заменит `T` на `Monster`:

```
public static void draw(Monster monster) { /* ... */ }
```

18.27. Получение аргумента типа родительского класса

Компилятор стирает информацию о параметрах типа, но в некоторых случаях во время выполнения программы можно получить аргументы типа суперкласса. Для этого используется метод `getGenericSuperclass()`, который у нас есть еще с Java 1.5.

Для лучшего понимания надо привести пример.

GetGenericSuperclassExample.java

```
private class MySuperclass<T> {
 public void method1(T arg0) {
   System.out.println(arg0);
 }
}
```

В дочернем классе в качестве аргумента типа укажем строку:

GetGenericSuperclassExample.java

```
private class MyClass extends MySuperclass<String> {
}
```

Пример программы, использующей метод `getGenericSuperclass()`:

GetGenericSuperclassExample.java

```
package ru.urvanov.javaindynamics2022.generics;

import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
```

```
public class GetGenericSuperclassExample {

    public static void main(String[] args) {
        Type t = MyClass.class.getGenericSuperclass();
        ParameterizedType p = (ParameterizedType) t;
        Type[] a = p.getActualTypeArguments();
        for (Type type : a) {
            System.out.println("аргумент типа = " + type);
        }
    }
}
```

18.28. Влияние стирания типа и методы-мосты (bridge methods)

Иногда стирание типа приводит к ситуации, которой вы не ожидали. Следующие примеры показывают, как это может произойти.

Пусть есть два класса:

DarkLair.java

```
package ru.urvanov.javaindynamics2022.generics;

public class DarkLair<T> {

    // Житель
    T inhabitant;

    public void setInhabitant(T inhabitant) {
        System.out.println("DarkLair.setInhabitant");
        this.inhabitant = inhabitant;
    }

    public T getInhabitant() {
        return this.inhabitant;
    }

}
```

MonsterDarkLair.java

```
class MonsterDarkLair extends DarkLair<Monster> {
    public void setInhabitant(Monster inhabitant) {
        System.out.println("MonsterDarkLair.setInhabitant");
        super.setInhabitant(inhabitant);
    }
```

```
    public static void main(String[] args) {
        MonsterDarkLair mdl = new MonsterDarkLair();
        DarkLair dl = mdl; // сырой тип - компилятор генерирует
                           // предупреждение unchecked warning

        // Исключение ClassCastException
        dl.setInhabitant(new Chimera("Greenus Maximus"));
    }
}
```

18.29. Методы-мосты (Bridge Methods)

При компиляции класса или интерфейса, который расширяет параметризованный класс или реализует параметризованный интерфейс, компилятор требует создания синтетического метода, называемого методом-мостом (bridge method). Вам не нужно о них беспокоиться, но вы можете увидеть их в трассировке стека:

После стирания типа классы `Lair` и `MonsterLair` станут такими:

```
public class Lair {

    public Object data;

    public void setInhabitant(Object data) {
        System.out.println("Lair.setData");
        this.inhabitant = inhabitant;
    }
}

public class MonsterLair extends Lair {

    public void setInhabitant(Monster data) {
        System.out.println("MonsterLair.setData");
        super.setData(data);
    }
}
```

После стирания типа сигнатуры методов не совпадают. Метод из класса `Lair` становится `setInhabitant(Object)`, а метод из класса `IntegerNode` становится `setInhabitant(Monster)`, поэтому метод `setInhabitant` из класса `MonsterLair` не переопределяет метод `setInhabitant` из класса `Lair`.

Чтобы исправить проблему и сохранить полиморфизм обобщенных типов после стирания типа, компилятор Java генерирует методы-мосты, чтобы расширение работало как ожидается. Для класса `MonsterLair` компилятор генерирует следующий метод-мост `setInhabitant`:

```
class MonsterLair extends Lair {

    // Метод-мост, сгенерированный компилятором
    //
    public void setInhabitant(Object data) {
        setData((Inhabitant) data);
    }

    public void setInhabitant(Inhabitant data) {
        System.out.println("InhabitantLair.setData");
        super.setData(data);
    }

}
```

Как вы можете видеть, метод-мост, который имеет ту же сигнатуру, что и метод `setData` у класса `Lair`, делегирует действие к оригинальному методу `setData`.

Я так и не смог нормально перевести слово "Non-Reifiable", но пусть будет переводиться как "нематериализуемые".

В разделе стирания типов обсуждается процесс, где компилятор удаляет информацию, связанную с параметрами типа и аргументами типа. Стирание типа имеет последствия, связанные с произвольным количеством параметров (varargs).

Материализуемые типы (reifiable types) — это типы, информация о которых полностью доступна во время выполнения: примитивы, необобщенные типы, сырые типы, обращения к неограниченным подстановочным символам.

Нематериализуемые типы (non-reifiable types) — это типы, информация о которых удаляется во время компиляции стиранием типов: обращения к обобщенным типам, которые не объявлены с помощью неограниченных подстановочных символов. Во время выполнения о нематериализуемых типах (Non-reifiable types) нет всей информации. Примеры нематериализуемых типов: `List<String>` и `List<Number>`. Виртуальная машина Java не может узнать разницу между ними во время выполнения. В некоторых ситуациях нематериализуемые типы не могут использоваться, например в выражениях `instanceof` или в качестве элементов массива.

18.30. Загрязнение кучи (Heap pollution)

Загрязнение кучи (heap pollution) возникает, когда переменная параметризованного типа ссылается на объект, который не является параметризованным типом, например `Lair<Integer>` хранит экземпляр `String`. Такая ситуация возникает, если программа выполнила некоторую операцию, которая генерирует предупреждение unchecked warning во время компиляции. Предупреждение unchecked warning генерируется, если правильность операции, в которую вовлечен параметризованный тип (например, приведение типа или вызов метода), не может быть проверена. Например, загрязнение кучи возникает при смешивании сырых типов и параметризованных типов или при осуществлении непроверяемых преобразований типа.

В обычных ситуациях, когда код компилируется в одно и то же время, компилятор генерирует unchecked warning, чтобы привлечь ваше внимание к загрязнению кучи. При компиляции разных частей кода отдельно становится трудно определить место, которое могло бы привести к загрязнению кучи. Если вы исправите все предупреждения компилятора, то загрязнение кучи (heap pollution) не сможет произойти.

> ***Примечание***
> Обобщенные методы, которые включают произвольное число параметров, могут привести к загрязнению кучи.

Рассмотрим следующий класс `FantasyWorldEngine`:

FantasyWorldEngine.java

```
package ru.urvanov.javaindynamics2022.generics;

import java.util.Arrays;
import java.util.List;

public class FantasyWorldEngine {

    public static <T> void union(List<T> listArg, T... elements) {
        for (T x : elements) {
            listArg.add(x);
        }
    }

    public static void invalid(List<Nymph>... lists) {
        Object[] objectArray = lists;     // OK
        objectArray[0] = Arrays.asList("просто любой объект");
        Nymph s = lists[0].get(0);       // Здесь генерируется
                                         // ClassCastException
    }
}

class Nymph {

    private String name;

    public Nymph(String name) {
        this.name = name;
    }
}
```

Следующий пример использует класс `FantasyWorldEngine`:

FantasyWorldEngine.java

```
public static void main(String[] args) {

 List<Nymph> stringListA = new ArrayList<>();
 List<Nymph> stringListB = new ArrayList<>();
```

```
 FantasyWorldEngine.union(stringListA,
     new Nymph("Эльза"),
     new Nymph("Мария"),
     new Nymph("Яна"),
     new Nymph("Оксана"));
 FantasyWorldEngine.union(stringListB,
     new Nymph("Джамиля"),
     new Nymph("Конни"),
     new Nymph("Екатерина"),
     new Nymph("Алиса"));
 List<List<Nymph>> listOfNymphLists =
     new ArrayList<>();
 FantasyWorldEngine.union(listOfNymphLists,
     stringListA, stringListB);

 // Выйдет ошибка, т. к. внутри метода вставляем строку
 // внутрь списка Нимф, а потом при попытке вытащить его оттуда
 // получаем ошибку.
 FantasyWorldEngine.invalid(
     List.of(
         new Nymph("Конни"),
         new Nymph("Эльвира")),
     List.of(new Nymph("Джамиля"),
         new Nymph("Яна")));
}
```

Если компилировать класс `FantasyWorldEngine` с опцией `-Xlint:unchecked`, то возникает следующее предупреждение: Possible heap pollution from parameterized vararg type (Возможно загрязнение кучи от типа с `vararg`).

Когда компилятор сталкивается с произвольным числом параметров (`varargs`-метод), он преобразует `varargs`-параметр в массив. Однако Java не запрещает создание массивов с параметризованными типами. В методе `FantasyWorldEngine.union` компилятор преобразует формальный параметр `T... elements` в формальный параметр `T[] elements`. Но из-за стирания типа компилятор конвертирует формальный параметр в `Object[] elements`, поэтому возникает загрязнение кучи.

> ***Примечание***
>
> Загрязнение кучи (heap pollution) в Java — это ситуация, когда переменная параметризованного типа ссылается на объект, не являющийся объектом этого параметризованного типа.

Следующая инструкция присваивает `varargs`-параметр `lists` в массив объектов `objectArgs`:

```
Object[] objectArray = lists;
```

Эта инструкция потенциально приводит к загрязнению кучи. Значение, которое не соответствует параметризованному типу `varargs`-параметра `lists`, может быть при-

своено переменной в `objectArray` и таким образом может быть присвоено в `lists`. Но компилятор не генерирует unchecked warning в этой инструкции, т. к. он уже сгенерировал предупреждение, когда преобразовывал `varargs`-параметр `List<Nymph>... lists` в формальный параметр `List[] lists`. Инструкция корректна, переменная `lists` имеет тип `List[]`, который является дочерним типом для `Object[]`.

В результате компилятор не сгенерирует никаких ошибок или предупреждений, если вы присвоите объект `List` объектов любого типа к элементу массива `ObjectArray`, как показано здесь:

```
objectArray[0] = Arrays.asList("просто любой объект");
```

Эта инструкция присваивает первому элементу массива `objectArray` список `List`, содержащий объекты типа String.

Предположим, что вы вызываете метод `FantasyWorldEngine.invalid` следующим образом:

```
FantasyWorldEngine.invalid(
   List.of(
       new Nymph("Конни"),
       new Nymph("Эльвира")),
   List.of(new Nymph("Джамиля"),
       new Nymph("Яна")));
```

Во время выполнения виртуальная машина Java генерирует `ClassCastException` на следующей инструкции:

```
Nymph s = lists[0].get(0);
```

Объект, сохраненный в первом элементе массива переменной `lists`, имеет тип `List<String>`, но инструкция ожидает объект типа `List<Nymph>`.

18.31. Подавление предупреждений для методов с произвольным количеством параметров с нематериализуемыми формальными параметрами

Если вы объявили метод с произвольным числом параметров параметризованного типа и обеспечили то, что тело метода не бросает исключение `ClassCastException` или другое похожее исключение, связанное с неправильной обработкой `varargs`-параметра, то вы можете отключить предупреждение, которое компилятор генерирует для этих методов при помощи добавления аннотации к статическому методу и методу-неконструктору:

```
@SafeVarargs
```

Аннотация `@SafeVarargs` относится к документируемой части объявления метода. Эта аннотация говорит, что реализация метода корректно обрабатывает `varargs`-параметр. Для нее есть дополнительные ограничения. Например, аннотацию нельзя

указывать для методов, которые не указаны как `static` или `final`, либо для методов с фиксированным числом параметров.

PredefinedAnnotations.java

```
package ru.urvanov.javaindynamics2022.annotation;

import java.util.List;

public class PredefinedAnnotations extends ParentPredefinedAnnotations{

    @SafeVarargs
    public final void varargsMethod(List<String>... arrayOfListStrings) {

    }
}
```

Также возможно, но менее желательно, использовать следующую аннотацию для подавления этих предупреждений:

```
@SuppressWarnings({"unchecked", "varargs"})
```

Но этот способ не подавляет предупреждения со стороны вызова метода.

18.32. Ограничения обобщений

Ограничение

Нельзя создавать экземпляры обобщенных типов с примитивными типами в качестве аргументов типа.

Нельзя создавать `Lair<int>` или `Lair<double>`, но можно использовать классы-обертки `Integer`, `Double` и т. д.: `Lair<Integer>`, `Lair<Double>`. Компилятор будет использовать автоупаковкку и автораспаковку при попытках передать элемент примитивного типа в объекты таких классов.

Ограничение

Нельзя создавать экземпляры параметров типа.

Вы не можете создать экземпляр параметра типа. Например, следующий код приведет к ошибке компиляции:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

В качестве обходного пути вы можете создать объект параметра типа с помощью отражения (reflection):

```
public static <E> void append(List<E> list, Class<E> cls)
        throws Exception {
```

```
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

Вы можете вызвать метод `append` вот так:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Ограничение

Нельзя объявлять статические поля с типом параметра типа

Статические поля класса являются общими для всех объектов этого класса, поэтому статические поля с типом параметра типа запрещены.

Ограничение

Нельзя использовать приведения типа или instanceof с параметризованными типами.

Так как компилятор Java стирает все параметры типа из обобщенного кода, то вы не можете проверить во время выполнения, какой параметризованный тип используется для обобщенного типа:

```
public static <E> void process(Lair<E> lair) {
    if (lair instanceof Lair<Goblin>) {  // compile-time error
        // ...
    }
}
```

Самое большое, что вы можете сделать, — это использовать подстановочный символ для проверки, что объект является типом `Lair`:

```
public static void process(List<?> list) {
    if (lair instanceof Lair<?>) {  // OK; instanceof requires
                                    // a reifiable type
        // ...
    }
}
```

Обычно вы не можете использовать приведение типа к параметризованному типу, если он не использует неограниченный подстановочный символ. Например:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // ошибка компиляции
```

Однако в некоторых случаях компилятор знает, что параметр типа всегда верный, и позволяет использовать приведение типа. Например:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Ограничение

Невозможно создавать массивы параметризованных типов.

Вы не можете создавать массивы параметризованных типов. Например, следующий код не будет компилироваться:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // ошибка компиляции
```

Следующий код показывает, что случится при вставке различных типов в массив:

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // исключение ArrayStoreException
```

Если вы попробуете то же самое с обобщенным списком, то будет такая проблема:

```
Object[] stringLists = new List<String>[];  // ошибка компиляции, но допустим,
что это возможно
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // должно быть исключение
                                  // ArrayStoreException,
                                  // но среда выполнения не может его заметить.
```

Если бы массивы с параметризованными типами были разрешены, то предыдущий код не смог бы бросить исключение `ArrayStoreException`.

> ***Ограничение***
>
> Нельзя создавать, ловить (catch) или бросать (throw) объекты параметризованных типов.

Обобщенный класс не может расширять класс `Throwable` напрямую или ненапрямую. Например, следующие классы не компилируются:

```
// Расширяет Throwable ненапрямую
class MathException<T> extends Exception { /* ... */ }    // ошибка компиляции

// Расширяет Throwable напрямую
class QueueFullException<T> extends Throwable { /* ... */ // ошибка компиляции
```

Метод не может ловить (catch) экземпляр параметра типа:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) {   // ошибка компиляции
        // ...
    }
}
```

Однако вы можете использовать параметр типа в клаузе `throws`:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {     // OK
        // ...
    }
}
```

Ограничение

Нельзя перегружать метод так, чтобы формальные параметры типа стирались в один и тот же сырой тип.

Класс не может иметь два перегруженных метода, которые будут иметь одинаковую сигнатуру после стирания типов:

```
public class Example {
    public void updateLair(Lair<Goblin> goblinLair) { }
    public void updatelair(Lair<Ghoul> ghoulLair) { }
}
```

Этот код не будет компилироваться.

18.33. Задания

1. Создайте класс `Vector`, с помощью которого можно было бы обрабатывать вектор в трехмерном пространстве (x, y, z) и совершать с помощью него основные операции над ними: сложение, умножение на число, вычисление нормы (длины) и т. д. Сделайте его обобщенным, чтобы он мог работать с любым из наследников `Number`.
2. Создайте обобщенный класс `Matrix`, который позволял бы обрабатывать матрицы произвольного типа данных и размера. Реализуйте операции умножения, сложения, умножения матрицы на число, транспонирование, вычисление определителя.

ГЛАВА 19

Исключения

19.1. Введение

Исключение (exception) — это событие, которое возникает во время выполнения программы и прерывает нормальный поток выполнения инструкций.

Когда возникает какая-нибудь ошибка внутри метода, он создает специальный объект, называемый объектом-исключением или, просто, исключением (exception object), который передается системе выполнения. Этот объект содержит информацию об ошибке, включая тип ошибки и состояние программы, в которой произошла

ошибка. Создание объекта-исключения и передача его системе выполнения называется броском исключения (throwing an exception).

После броска исключения система пытается найти его обработчик. Система выполнения проходит по стеку вызовов от текущего метода вверх, ища подходящий обработчик исключений.

Выбранный обработчик ловит это исключение.

Если системе не удается найти подходящий обработчик исключения, то программа завершает свое выполнение.

В Java все классы-исключения являются наследниками от класса `java.lang.Throwable`, который, в свою очередь, имеет подклассы `java.lang.Error` и `java.lang.Exception`. Класс `java.lang.Exception` имеет дочерний класс `java.lang.RuntimeException` (рис. 19.1).

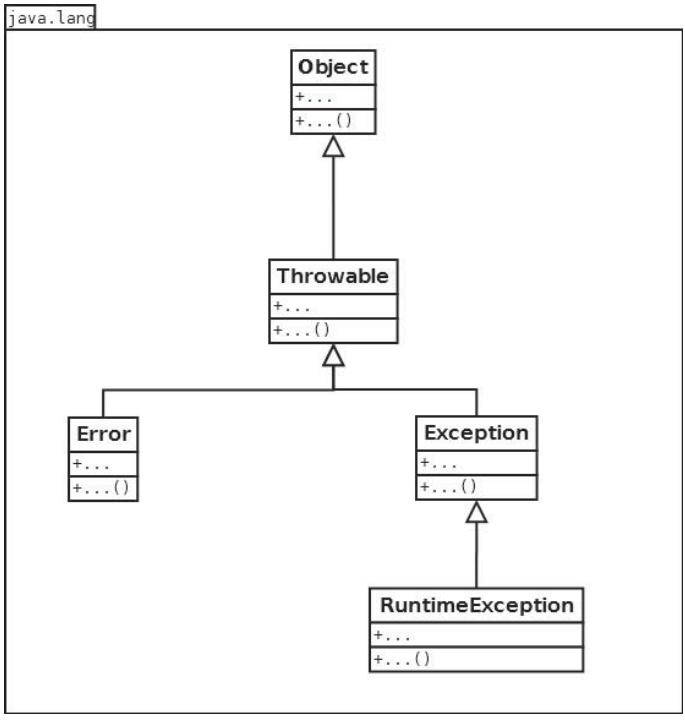


Рис. 19.1. Иерархия основных исключений Java

По соглашению все бросаемые исключения являются наследниками трех классов: `java.lang.Error`, `java.lang.Exception` или `java.lang.RuntimeException`. Технически можно бросить исключение, которое не является наследником этих трех классов, но является наследником `java.lang.Throwable`, но так делать не принято.

Из описанных выше трех классов выходит три вида исключений в Java:

- Наследники `java.lang.Error`. Эти исключения возникают при серьезных ошибках, после которых невозможно нормальное продолжение выполнения програм-

мы. Это могут быть различные сбои аппаратуры и т. д. В обычных ситуациях ваш код не должен перехватывать и обрабатывать этот вид исключений.

- Наследники `java.lang.RuntimeException`. Это непроверяемый тип исключений вроде выхода за границу массива или строки, попытка обращения к методу на переменной, которая содержит `null`, неправильное использование API и т. д. В большинстве своем программа не может ожидать подобных ошибок и восстановиться после них. Подобные исключения возникают из-за ошибок программиста. Приложение может их перехватывать, но в большинстве случаев имеет гораздо больше смысла исправить ошибку, приводящую к подобным исключениям.
- Наследники `java.lang.Exception`, которые НЕ являются наследниками `java.lang.RuntimeException`. Подобный тип исключений называется проверяемыми исключениями (checked exceptions). Это такой тип исключений, который может ожидать хорошо написанная программа и из которых она может восстановить свой обычный ход выполнения. Это может быть попытка открыть файл, к которому нет доступа или которого не существует; проблемы с доступом по сети и т. д. Все исключения являются проверяемыми, кроме наследников `java.lang.Error` и `java.lang.RuntimeException`. Любой метод, который может бросить проверяемое исключение, должен указать это исключение в клаузе `throws`. Для любого кода, который может бросить проверяемое исключение, это исключение должно быть указано в throws метода либо должно быть перехвачено с помощью инструкции `try-catch`.

19.2. Перехватывание и обработка исключений

Рассмотрите следующий код:

```
import java.io.*;

class Main {

    public static void main(String[] args) {
        byte[] bytesToWrite = new byte[100];
        OutputStream os = new FileOutputStream("output.file");
        os.write(bytesToWrite);
        os.close();
    }
}
```

На текущий момент этот код не будет компилироваться, т. к. конструктор `FileOutputStream(String name)` может бросать проверяемое исключение `FileNotFoundException`, вызов метода `os.write(byte[] b)` — проверяемое исключение `IOException`, и вызов метода `os.close()` тоже может бросать проверяемое исключение.

Проверяемые исключения, которые может бросать метод или конструктор, указываются с помощью `throws`:

```
public FileOutputStream(String name)
                throws FileNotFoundException

public void write(byte[] b) throws IOException

public void close()
           throws IOException
```

Все проверяемые исключения, которые может бросать код, должны быть указаны в `throws` этого метода, либо должны быть перехвачены и обработаны с помощью `try-catch-finally`.

Чтобы пример выше компилировался, нужно поместить в блок try весь код, который может бросить исключения:

BlockTry.java

```
package ru.urvanov.javaindynamics2022.exception;

import java.io.*;

class BlockTry {

    public static void main(String[] args) {
        byte[] bytesToWrite = new byte[100];

        try {
            OutputStream os = new FileOutputStream("output.file");
            os.write(bytesToWrite);
            os.close();
        } catch (FileNotFoundException fnfe) {
            System.out.println("Cannot find the file.");
        } catch (IOException ioex) {
            System.out.println("Error writing file: "
                + ioex.getMessage());
        }

    }
}
```

Теперь в случае возникновения исключения `FileNotFoundException` управление будет передаваться на блок:

```
catch (FileNotFoundException fnfe) {
    System.out.println("Cannot find the file.");
}
```

Этот блок выведет в консоль строку "Cannot find the file", после чего управление передастся на следующую инструкцию за блоком `try-catch`.

Если же возникнет исключение `IOException` либо один из его потомков (`FileNotFoundException` тоже является потомком `IOException`, но поскольку мы указали его блок первым, то в случае `FileNotFoundException` будет выполняться его специфичный блок), то будет выполняться блок, который выведет в консоль строку `"Error writing file: ..."`, после чего управление передастся на следующую инструкцию за блоком `try-catch`.

Мы также могли сделать свой, отдельный блок `try-catch` для каждой инструкции, которая может бросить исключение.

При возникновении исключения смотрятся блоки `catch` ближайшего блока `try` в том порядке, в котором они объявлены. Среда исполнения пытается сопоставить тип объекта-исключения с типом объекта-исключения, указанного в каждом из блоков `catch`, и выполняется первый блок `catch`, тип обрабатываемого исключения которого совпадает с типом брошенного исключения. Если подходящего блока `catch` не нашлось, то смотрится вышестоящий блок `try` и т. д.

Как уже было показано выше, обработчик исключения может обращаться к методам объекта исключения, который передается ему в качестве параметра.

Начиная с Java 7, можно в одном блоке `catch` перехватывать несколько различных типов исключений, что позволяет уменьшить количество кода:

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    // ... other code
}
```

Если блок `catch` перехватывает несколько типов исключений, то тогда его параметр неявно `final`, т. е. в примере выше вы не можете присвоить параметру `ex` что-либо в блоке `catch`.

Посмотрим еще раз на фрагмент кода из примера:

```
OutputStream os = new FileOutputStream("output.file");
os.write(bytesToWrite);
os.close();
```

Последняя строка `os.close()` закрывает файл и освобождает ресурсы системы. Но она сработает только при нормальном ходе выполнения программы. Если какое-нибудь исключение возникнет в конструкторе или в методе `write()`, то метод закрытия потока не выполнится, а значит, будет утечка ресурсов.

Чтобы избежать подобных проблем, освобождение ресурсов нужно осуществлять в блоке `finally`. Код в блоке `finally` выполняется ВСЕГДА после завершения блока `try`, даже в случае возникновения исключения.

Исправленный код:

BlockFinally.java

```
package ru.urvanov.javaindynamics2022.exception;

import java.io.*;
```

```
class BlockFinally {

    public static void main(String[] args) {
        byte[] bytesToWrite = new byte[100];
        OutputStream os = null;
        try {
            os = new FileOutputStream("output.file");
            os.write(bytesToWrite);
            System.out.println("end try");
        } catch (FileNotFoundException fnfe) {
            System.out.println("Cannot find the file.");
        } catch (IOException ioex) {
            System.out.println("Error writing file: "
                + ioex.getMessage());
        } finally {
            System.out.println("finally.");
            if (os != null) {
                // Метод close тоже может бросить исключение.
                try {
                    os.close();
                } catch (IOException closeException) {
                    System.out.println("closeException: "
                            + closeException.getMessage());
                }
            }
        }

        System.out.println("End of program.");
    }
}
```

При обычном ходе выполнения этот код выведет в консоль следующее:

```
end try
finally.
End of program.
```

Если же во время открытия файла на запись произойдет ошибка, то вывод может стать таким:

```
Cannot find the file.
finally.
End of program.
```

Как видите, блок `finally` отработал после блока обработки исключений.

Если виртуальная машина Java завершит свое выполнение во время выполнения кода `try` или `catch`, то блок `finally` может НЕ выполниться. Также, если поток будет прерван внутри кода `try` или `catch`, блок `finally` может НЕ выполниться, хотя программа продолжит свое выполнение.

Приведенный выше код можно сделать более понятным, если использовать оператор `try-with-resources`. Любой объект, реализующий интерфейс `java.lang.AutoCloseable`, который включает все объекты, реализующие `java.io.Closeable` (`Closeable` расширяет `AutoCloseable`), например `FileOutputStream`, можно использовать в `try-with-resources`:

TryWithResources.java

```
package ru.urvanov.javaindynamics2022.exception;

import java.io.*;

class TryWithResources {

    public static void main(String[] args) {
        byte[] bytesToWrite = new byte[100];

        try (OutputStream os = new FileOutputStream("output.file")) {
            os.write(bytesToWrite);
            System.out.println("end try");
        } catch (FileNotFoundException fnfe) {
            System.out.println("Cannot find the file.");
        } catch (IOException ioex) {
            System.out.println("Error writing file: "
                    + ioex.getMessage());
        }
        // Блок finally уже не нужен, но можно
        // использовать, если хочется.

        System.out.println("End of program.");
    }
}
```

В этом примере `try-with-resources` автоматически вызовет метод `close()`, что освободит ресурсы.

В блоке `try-with-resources` можно указывать несколько ресурсов, тогда они будут открываться слева направо, как указано в блоке, а закрываться справа налево (т. е. в обратном порядке):

```
try (OutputStream os = new FileOutputStream("output.file");
        FileReader fr = new FileReader("input.txt")) {
    // ...
}
```

Блок `try-with-resources` может не иметь ни секции `catch`, ни `finally`, но обычный блок `try` должен обязательно иметь либо секцию `catch`, либо секцию `finally`, либо обе секции.

Ресурсы блока `try-with-resources` закрываются перед выполнением блоков `catch` и `finally`.

Рассмотрите следующий код:

TryWithResources.java

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                                        throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Если метод `readLine()` бросит исключение, а затем метод `close()` тоже бросит исключение, то метод `readFirstLineFromFileWithFinallyBlock()` бросит исключение из блока `finally`, а исключение из блока `try` будет подавлено.

Если же исключение возникнет в блоке `try` и в блоке, освобождающем ресурсы для `try-with-resources`, то конечное исключение, бросаемое методом, — исключение из блока `try`, т. е. исходное. Это еще одно преимущество использования `try-with-resources`. Исключения, которые были подавлены в блоке `try-with-resources`, можно получить с помощью метода `public final Throwable[] getSuppressed()`.

Метод `close()` в интерфейсе `java.lang.AutoCloseable` объявляет в клаузе throws исключение `Exception`, а метод `close()` в `java.io.Closeable` объявляет `IOException`, что позволяет наследникам `AutoCloseable` определять свои, специфичные для своей области, исключения.

19.3. Указание типов исключений, бросаемых методом

Если какой-нибудь код внутри метода может бросать проверяемые исключения, то либо эти исключения должны перехватываться и обрабатываться внутри метода, либо метод должен указывать, что он может бросить исключение подобного вида с помощью ключевого слова throws:

WildWorld.java

```
class WildWorld {
    public void someCalculation(int arg1, double arg2)
            throws java.io.IOException, java.sql.SQLException,
                    java.lang.IndexOutOfBoundsException {
    }
}
```

Исключение `IndexOutOfBoundsException` является наследником `RuntimeException`, поэтому указывать его необязательно и даже не нужно:

WildWorld.java

```
class WildWorld {
    public void someCalculation(int arg1, double arg2)
            throws java.io.IOException, java.sql.SQLException {
    }
}
```

19.4. Как бросить исключение

Перед тем как вы сможете перехватить исключение, какой-нибудь код должен его бросить/сгенерировать. Любой код может бросить исключение: ваш код, код из пакета, написанного кем-то другим, сама среда Java. Исключение всегда бросается с помощью инструкции throw, независимо от того, кто его бросает:

```
throw objThrowable;
```

Пример:

```
if (x == null)
    throw new IllegalStateException("Что-то пошло не так");
```

19.5. Цепочки исключений

Приложения часто отвечают на исключение бросанием другого исключения. Цепочки исключений позволяют узнать, какое исключение привело к появлению другого.

Следующие методы и конструкторы класса java.lang.Throwable помогают работать с цепочками исключений:

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

Аргумент `Throwable` метода `initCause` и `Throwable` в конструкторах — это исключения, которые привели к текущему исключению. Метод `getCause()` возвращает исключение, которое стало причиной текущего исключения, а метод `initCause` устанавливает причину текущего исключения:

Пример использования цепочки исключений:

```
try {

} catch (IOException e) {
    throw new SampleException("Другое IOException", e);
}
```

В этом примере при обработке `IOException` создается новое исключение `SampleException`, причина этого присоединяется к цепочке исключений, и цепочка бросается в следующий уровень обработчиков исключений.

Если какой-нибудь код с верхнего уровня обработчиков исключений захочет вывести стек вызовов, то ему нужно будет использовать метод `getStackTrace()`:

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int n = 0; n < elements.length; n++) {
        System.err.println(elements[n].getFileName()
            + ":" + elements[n].getLineNumber()
            + "-> "
            + elements[n].getMethodName() + "()");
    }
}
```

19.6. Создание своих объектов-исключений

Когда вы выбираете исключение, которое будет бросать ваш код в какой-либо ситуации, вы можете выбрать создание своего нового класса исключения.

Свои исключения обычно создаются в том случае, если стандартные исключения Java не подходят для вашей ситуации либо если вам нужно создать свою иерархию исключений для вашей библиотеки или приложения.

Предположим, что если вы описываете игровую логику, то вам, возможно, потребуется различать следующие виды исключений (рис. 19.2).

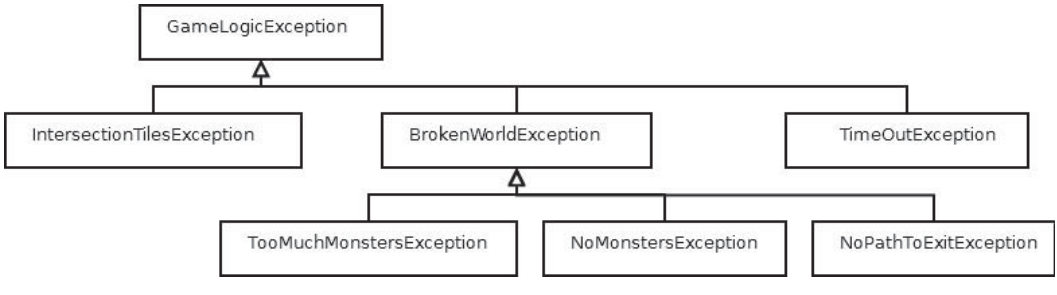


Рис. 19.2. Возможная иерархия исключений видеоигры

В качестве родительского класса для своего исключения `GameLogicException` логичнее всего выбрать класс `Exception`, т. к. в нашем случае нужны именно проверяемые исключения.

По соглашению о кодировании в Java имена исключений должны заканчиваться на `Exception`.

Пример возможного кода для исключения `GameLogicException`:

GameLogicException.java

```
public class GameLogicException extends Exception {

    // Конструкторы, вызывающие конструкторы базового класса.

    public GameLogicException() {
        super();
    }

    public GameLogicException(String message) {
        super(message);
    }

    public GameLogicException(String message, Throwable cause) {
        super(message, cause);
    }

    public GameLogicException(Throwable cause) {
        super(cause);
    }

    // остальные методы.
}
```

19.7. Преимущества исключений

- Разделение кода обработки ошибок от обычного кода.
- Распространение информации о произошедшей ошибке вверх по стеку вызовов.
- Группировка и разделение различных типов ошибок.

19.8. Задания

1. Представьте, что наш банкомат может принимать в качестве наличных только рубли. Создайте класс с методом `takeCash`, который принимал бы в качестве параметров стандартный класс `java.util.Currency` (тип валюты), и вторым параметром double money (сумма денег). Создайте исключение, которое можно бы было бросить, если в метод пришла валюта, отличная от рублей, а также исключение, которое можно было бы бросить при попытке положить наличными сумму больше допустимой (например, больше 50 000). Какой тип исключений вы использовали и почему?
2. Придумайте класс, который мог бы проверять логин и пароль. Пусть для упрощения правильный логин и пароль будут зашиты в константах в классе. Создай-

те иерархию исключений: базовое исключение, неправильный логин или пароль, пользователь заблокирован и т. д. Подумайте над тем, какой тип исключений использовать: проверяемые или непроверяемые.

ГЛАВА 20

Потоки ввода/вывода

20.1. Введение

Поток ввода/вывода (I/O Stream) представляет собой источник данных или место их назначения. Потоки могут представлять собой абсолютно различные источники и места назначения: файлы на диске, устройства, сеть, другие программы, массивы в памяти т. д.

Потоки поддерживают большое количество различных типов данных: байты, примитивные типы, локализованные символы, объекты. Некоторые потоки просто передают данные, другие изменяют в соответствии со своими потребностями.

20.2. Потоки байт

Все классы, работающие с потоками байт, наследуются от абстрактных классов `java.io.InputStream` или `java.io.OutputStream`.

На моем сайте **https://urvanov.ru** можно найти диаграмму классов-наследников как для `InputStream`, так и для `OutputStream` — базовых классов для работы с потоками. Дочерние классы реализуют работу с файлами, сетью и тому подобным либо добавляют дополнительные возможности наподобие буферизации данных.

20.3. InputStream

Это абстрактный класс, являющийся базовым для всех классов, представляющих поток ввода.

```
public int available()
              throws IOException
```

Возвращает количество байт, которое может быть прочитано из потока без блокировки. Некоторые реализации `InputStream` возвращают полное количество байт

в потоке, но не все. Не стоит использовать этот метод для определения размера буфера, который будет хранить все данные из потока.

```
public void close()
          throws IOException
```

Закрывает поток и освобождает все ресурсы.

```
public void mark(int readlimit)
```

Помечает текущую позицию во входной строке. Работает, только если `markSupported()` возвращает `true`. Смысл этого метода в том, что поток каким-нибудь образом запоминает все считанные после вызова этого метода данные и может вернуть те же самые данные еще раз после вызова метода `reset()`. Если после вызова метода `mark(int readLimit)` из потока было прочитано больше `readLimit` байт, то поток не обязан запоминать что бы то ни было.

```
public void reset()
          throws IOException
```

Если метод `markSupported()` возвращает `true`, то:

- Если метод `mark()` не был вызван ни разу либо количество байт, которые были прочитаны из потока после вызова `mark()`, больше аргумента метода `mark()` в последнем его вызове, то может броситься исключение `IOException`.
- Если исключение `IOException` не было брошено, то поток возвращается в такое состояние, что все вызовы методов `read()` в дальнейшем будут возвращать те же данные, которые они возвращали с момента последнего вызова метода `mark()` (либо с начала потока, если метод `mark()` не был вызван ни разу).

Если метод `markSupported()` возвращает `false`, то:

- Вызов метода `reset()` может бросить исключение `IOException`.
- Если не бросается исключение `IOException`, то поток сбрасывается в фиксированное состояние, которое зависит от конкретного типа входного потока, и от того, как он был создан. Байты, которые будут прочитаны при последующих вызовах методов `read()`, зависят от конкретного типа входной строки.

```
public boolean markSupported()
```

Возвращает `true`, если реализация `InputStream` поддерживает методы `mark()` и `reset()`.

```
public abstract int read()
                 throws IOException
```

Считывает один байт из потока. Возвращает его в `int`, содержащем значение от 0 до 255. Возвращает –1, если достигнут конец потока. Блокирует выполнение текущего потока программы до тех пор, пока не появятся входные данные, не будет достигнут конец потока либо не бросится исключение.

```
public int read(byte[] b)
         throws IOException
```

Считывает некоторое количество байт из входного потока и сохраняет его в массив байт `b`. Возвращает количество считанных байт, которое может быть меньше длины массива. Метод блокирует выполнение текущего потока программы до тех пор, пока не появятся входные данные, не будет достигнут конец потока либо не бросится исключение.

Если длина массива `b` равна нулю, то байты не считываются и возвращается 0, в противном случае происходит попытка считать хотя бы один байт. Если достигнут конец потока, то возвращается –1.

Метод `read(b)` у класса `InputStream` имеет такой же эффект, что и `read(b, 0, b.length)`, но дочерние классы могут переопределить его, если нужно.

```
public int read(byte[] b,
                int off,
                int len)
         throws IOException
```

Читает до `len` байт из входного потока в массив байт. Пытается считать `len` байт, но может считать и меньше. Количество реально считанных байт возвращается как `int`.

Этот метод блокирует выполнение текущего потока программы до тех пор, пока не появятся данные, не будет достигнут конец потока либо не возникнет исключение.

Если `len` равен нулю, то байты не считываются и возвращается 0. В противном случае происходит попытка считать хотя бы один байт. Если никаких байт нет, т. к. был достигнут конец потока, то возвращается –1, иначе хотя бы один байт считывается и сохраняется в b.

20.4. OutputStream

Абстрактный класс, являющийся базовым для классов, реализующих выходной поток байт:

Основные методы:

```
public void close()
           throws IOException
```

Закрывает выходной поток и освобождает ресурсы.

```
public void flush()
           throws IOException
```

Записывает все байты из буфера. Некоторые реализации выходного потока могут накапливать байты в буфере и лишь потом реально записывать их. Вызов этого метода принудительно записывает данные из буфера и очищает его.

```
public abstract void write(int b)
                    throws IOException
```

Записывает байт в выходной поток.

```
public void write(byte[] b)
           throws IOException
```

Записывает `b.length` байт из указанного массива байт в выходной поток. Аналогично вызову `write(b, 0, b.length)`.

```
public void write(byte[] b,
                  int off,
                  int len)
           throws IOException
```

Записывает в выходной поток `len` байт из массива байт, начиная с `off`. Реализация этого метода в `OutputStream` вызывает в цикле метод `write(int b)`. Дочерние классы могут переопределить его, дав более оптимальную реализацию.

20.5. FileInputStream и FileOutputStream

Предназначены для чтения и записи данных в файл и из файла. Пример использования:

```
package ru.urvanov.javaindynamics2022.io;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * Файловые потоки ввода/вывода
 */
class FileInputOutputStreams {
    public static void main(String[] args) {
        try (InputStream is = new FileInputStream("input.txt");
                OutputStream os = new FileOutputStream("output.txt")) {
            int bytesReaded;
            final int BUFFER_SIZE = 10_000;
            byte[] buff = new byte[BUFFER_SIZE];
            while ((bytesReaded = is.read(buff)) != -1) {
                os.write(buff, 0, bytesReaded);
            }
        } catch (IOException ioex) {
            ioex.printStackTrace();
        }
    }
}
```

В диаграмме наследников `InputStream` и диаграмме наследников `OutputStream`, которые можно найти на сайте **https://urvanov.ru**, показаны конструкторы этих классов.

20.6. ByteArrayInputStream и ByteArrayOutputStream

Класс `ByteArrayInputStream` позволяет создать входной поток, который будет считывать данные из массива байт. Класс `ByteArrayOutputStream` позволяет записывать данные в поток, а по окончании получить записанные данные в виде массива байт с помощью метода `toByteArray()`. Принцип работы с этими классами такой же, как и с остальными наследниками `java.io.InputStream` и `java.io.OutputStream`.

В диаграмме наследников `InputStream` и диаграмме наследников `OutputStream`, которые можно найти на сайте **https://urvanov.ru**, показаны конструкторы и методы этих классов.

20.7. FilterInputStream и FilterOutputStream

Базовые классы для потоков, которые содержат внутри себя другой поток и производят некую трансформацию записываемых и считываемых данных.

В диаграмме наследников `InputStream` и диаграмме наследников `OutputStream` показаны конструкторы и методы этих классов.

20.8. DataInputStream и DataOutputStream

Классы `DataInputStream` и `DataOutputStream` позволяют платформонезависимо записывать в поток и считывать из него примитивные типы языка Java. Класс `DataInputStream` реализует интерфейс `DataInput`, который содержит методы для чтения примитивных типов, а класс `DataOutputStream` реализует интерфейс `DataOutput`, который содержит методы для записи примитивных типов. Эти классы используют в качестве обертки над другими потоками, например, так:

```
DataInputStream dis = new DataInputStream(new FileInputStream("myfile.data"));
```

Методы интерфейса `DataInput` используют исключение `java.io.EOFException` для обозначения конца потока, в отличие от методов класса `InputStream`, которые возвращают –1.

20.9. BufferedInputStream и BufferedOutputStream

Классы `BufferedInputStream` и `BufferedOutputStream` используют буфер, чтобы не нагружать систему операцией считывания и записи при каждом вызове методов `write` и `read`.

20.10. PipedInputStream и PipedOutputStream

Экземпляр класса `PipedInputStream` должен быть связан с экземпляром класса `PipedOutputStream` с помощью метода `connect(PipedOutputStream src)`. С помощью `PipedInputStream` считываются данные, которые в другом потоке записываются в `PipedOutputStream`.

20.11. ObjectInputStream и ObjectOutputStream

Классы `ObjectInputStream` и `ObjectOutputStream` позволяют считывать объекты из потока и записывать в него объекты, т. е. используются для сериализации и десериализации объектов.

20.12. Потоки символов

Все классы потоков символов наследуются от `java.io.Reader` или `java.io.Writer`. Как и у потоков байт, есть два специализированных класса для файлового ввода/вывода: `java.io.FileReader` и `java.io.FileWriter`. Работа с потоками символов аналогична работе с потоками байт.

Любой поток байт можно превратить в поток символов, обернув его в `java.io.InputStreamReader` или в `java.io.OutputStreamWriter` для потока вывода.

```
try (Reader reader = new InputStreamReader(
        new ByteArrayInputStream(byteArray), "windows-1251")) {
    // ... some code
}
```

С помощью класса `java.io.BufferedReader` можно считывать данные построчно, используя метод `readLine()`, который считает за конец строки символ '`\n`' (`LF`), '`\r`' (`CR`) или строку из двух символов "`\r\n`" (`CR LF`).

На сайте **https://urvanov.ru** вы можете посмотреть диаграмму с иерархией классов `Reader` и `Writer`.

Обратите внимание, что вторым параметром конструктора `InputStreamReader` мы указали кодировку windows-1251. У класса `InputStreamReader` существует конструктор с одним параметром, где не нужно указывать кодировку. Рекомендуется всегда использовать конструктор с указанием кодировки, т. к. в противном случае до Java 18 будет использоваться кодировка по умолчанию: windows-1251 для Windows в России, windows-1252 для Windows в Европе и UTF-8 для современных Linux, что может привести к разным неожиданностям при использовании приложения в разных ОС. А начиная с Java 18, все методы и конструкторы из `java.io` и `java.nio` используют UTF-8 по умолчанию.

Аналогично `Charset.defaultCharset()` возвращает UTF-8, начиная с Java 18, а в версиях до Java 18 он возвращает кодировку системы по умолчанию.

20.13. Scanner и PrintStream

Класс `PrintStream` позволяет записывать в поток форматированные данные. Особенно важны его методы `printf` или `format`. Пример `PrintStream` вы уже могли видеть: это `System.out`, с помощью которого мы осуществляли вывод в консоль в примерах.

Класс Scanner позволяет считывать из текста форматированные данные с помощью методов `nextInt`, `nextDouble` и аналогичных.

Работа с этими классами несколько отличается от работы с большинством классов, описанных в этой статье.

ScannerExample.java

```
package ru.urvanov.javaindynamics2022.io;

import java.util.Scanner;

/**
 * Пример использования Scanner
 */
public class ScannerExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Введите целое число");
        int i = sc.nextInt();
        System.out.println("Введите дробное число");
        double d = sc.nextDouble();
        // System.out - это PrintStream
        // Мы используем метод format, в качестве первого аргумента
        // передается строка, содержащая специальные символы
        // форматирования.
        // %d - вывод десятичного целого числа
        // %f - вывод числа с плавающей точкой.
        // Мы дополнительно указываем,
        // что нужно вывести 10 символов до запятой
        // и 2 символа после запятой
        System.out.format("i = %d, d = %10.2f", i, d);
    }
}
```

20.14. Задания

1. Создайте программу, которая считывает файл в указанной кодировке и выводит его на экран.

2. Создайте программу, которая считывает файл и записывает его содержимое в другой файл. Используйте `PipedInputStream` и `PipedOutputStream`.
3. Создайте программу, которая считывает различные типы данных от ввода с клавиатуры при помощи `Scanner`.

ГЛАВА 21

Сериализация

21.1. Теория

Класс `ObjectInputStream` реализует интерфейс `java.io.ObjectInput`, а класс `ObjectOutputStream` — интерфейс `java.io.ObjectOutput`, в которых описаны основные методы для чтения и записи объектов.

Чтение объекта из `ObjectInputStream` осуществляется с помощью метода:

```
public final Object readObject()
                        throws IOException,
                               ClassNotFoundException
```

Запись объекта в `ObjectOutputStream` производится с помощью метода:

```
public final void writeObject(Object obj)
                       throws IOException
```

Эти методы считывают и записывают в поток не только сам передаваемый объект, но и все объекты, на которые он ссылается и которые необходимы для того, чтобы восстановить исходный объект при чтении. Если какой-нибудь объект записывается в поток два раза, то реально он будет записан туда только один раз, но ссылки на него будет две. Если же какой-нибудь объект записывается в два разных потока, то при чтении из этих двух потоков получатся два разных объекта.

Только классы, реализующие интерфейсы `java.io.Serializable` или `java.io.Externalizable`, могут быть считаны из потока и записаны в поток.

Метод `readObject` возвращает тип `Object`, который должен быть приведен к ожидаемому типу с помощью операции приведения типов. Строки и массивы в Java являются объектами. Примитивные типы считываются с помощью методов `DataInput` (`ObjectInput` наследуется от `DataInput`) и записываются с помощью методов `DataOutput` (`ObjectOutput` наследуется от `DataOutput`).

Запомните

Запись объекта называется сериализацией.

Чтение объекта называется десериализацией.

Поля, объявленные как `transient` или `static`, игнорируются в процессе сериализации/десериализации.

При десериализации всегда создаются новые объекты, чтобы защитить уже существующие объекты от изменения.

В процессе чтения объекты создаются аналогично выполнению конструктора для нового объекта. Сначала выделяется и заполняется нулями память. Для несериализуемых классов вызываются конструкторы по умолчанию, а поля сериализуемых классов считываются из потока, начиная от наиболее близкого к `java.lang.Object` класса и заканчивая наиболее специфичным классом.

Если классу нужна особая обработка в процессе сериализации/десериализации, то он должен реализовать методы:

```
private void writeObject(java.io.ObjectOutputStream stream)
    throws IOException;
 private void readObject(java.io.ObjectInputStream stream)
    throws IOException, ClassNotFoundException;
 private void readObjectNoData()
    throws ObjectStreamException;
```

Метод `readObject` отвечает за чтение и восстановление состояния объекта класса, записанного в поток соответствующим методом `writeObject`. Метод не должен беспокоиться о состоянии, принадлежащем его суперклассам или подклассам. Состояние восстанавливается чтением данных из `ObjectInputStream` для каждого поля и присвоением этих значений соответствующим полям объекта. Чтение примитивных типов происходит с помощью `DataInput`.

Любая попытка считать данные, которые выходят за границу данных, записанных методом `writeObject`, приводит к исключению `OptionalDataException`. Методы чтения примитивных типов или возвращают –1 в массив байт или бросают исключение `EOFException` в зависимости от метода.

Метод `readObjectNoData` отвечает за инициализацию состояния объекта в случае, когда поток сериализации не имеет данного класса в качестве суперкласса объекта, десериализация которого происходит. Это может возникнуть, когда при сериализации использовалась другая версия класса, которая не имела в качестве суперкласса того класса, от которого наследуется класс у считывающей стороны. Эта ситуация может также возникнуть в случае подделки сериализованного потока. Метод `readObjectNoData` полезен для инициализации объектов "враждебного" или поврежденного потока.

Сериализация не читает и не присваивает значения полям, которые не реализуют интерфейс `java.io.Serializable`. Подклассы класса `Object`, которые не сериализуются, могут быть сериализуемыми, для этого они должны иметь конструктор по

умолчанию. В этом случае задача сохранения и восстановления их состояния ложится на дочерний класс.

Любое исключение в процессе десериализации перехватывается `ObjectInputStream`-ом и останавливает процесс чтения.

Реализация интерфейса `Externalizable` позволяет полностью контролировать содержимое и формат сериализации. Методы `void writeExternal(ObjectOutput out) throws IOException` и `void readExternal(ObjectInput in) throws IOException, ClassNotFoundException` интерфейса `Externalizable` вызываются при сохранении и восстановлении состояния объектов. При реализации каким-нибудь классом они могут записывать и считывать свое состояние с помощью методов `ObjectOutput` и `ObjectInput`. Ответственность за обработку версионности ложится на сам объект.

Десериализация констант перечислений отличается от обычной сериализации и десериализации объектов. Сериализованная форма констант содержит только их имена, значения полей констант опускаются. При десериализации константы `ObjectInputStream` считывает имя константы из потока, затем вызывает метод `Enum.valueOf(Class, String)` для получения значения. Процесс десериализации перечислений не может быть изменен. Любой метод `readObject`, `readObjectNoData` и `readResolve`, объявленный в перечислении, игнорируется. Также игнорируются поля `serialPersistentFields` и `serialVersionUID`, перечисления имеют фиксированный `serialVersionUID 0L`.

Статическое поле `serialVersionUID` означает версию класса. Оно объявляется так:

```
<модификатор доступа> static final long serialVersionUID = 42L;
```

Если в сериализуемом классе не объявлено поля `serialVersionUID`, то его значение вычисляется автоматически на основе окружения и полей класса. При чтении класса проверяется `serialVersionUID`. Если `serialVersionUID` считываемого класса не равен `serialVersionUID` нашего класса, то возникает исключение `java.io.InvalidClassException`. Старайтесь всегда объявлять вручную `serialVersionUID`, т. к. автоматически вычисленное значение может сильно отличаться в зависимости от платформы и реализации.

Класс, для которого при сериализации нужно использовать другой объект, должен реализовать метод:

```
<модификатор доступа> Object writeReplace() throws ObjectStreamException;
```

Метод `writeReplace()` вызывается в процессе сериализации, если он существует и доступен.

Если класс должен использовать другой объект при десериализации, то он должен реализовать метод:

```
<модификатор доступа> Object readResolve() throws ObjectStreamException;
```

Метод `readResolve()` вызывается в процессе десериализации, если он существует и доступен.

21.2. Задания

1. Создайте какой-нибудь класс. Используйте поля разных типов. Сделайте его сериализуемым. Попытайтесь записать его в файл и считать обратно.
2. Измените класс из задачи 1 так, чтобы он использовал методы `writeObject` и `readObject`.
3. Измените класс из задач 1 и 2 так, чтобы он использовал интерфейс `Externalizable` и его методы `readExternal` и `writeExternal`.

ГЛАВА 22

Файлы (NIO.2)

22.1. Path

`Path` представляет собой путь в файловой системе. Он содержит имя файла и список каталогов, определяющих путь к файлу.

Экземпляры Path отражают путь в конкретной платформе (например /home/fedor/foo для Linux или C:\home\fedor\foo для Windows). Экземпляры Path зависят от платформы. Нельзя сравнивать `Path` из Linux с путем из Windows, даже если структура их каталогов одинаковая и оба этих экземпляра указывают на один и тот же относительный файл.

`Path` может указывать на файл или каталог, которых не существует. Методы `Path` работают только с представлением пути. Они не проверяют существование пути.

Создавать экземпляры `Path` можно разными способами.

Можно использовать методы `java.nio.file.Paths.get()`:

```
public static Path get(String first,
                       String... more)

public static Path get(URI uri)
```

Примеры:

```
Path p1 = Paths.get(URI.create("file:///Users/fedor/Test.java"));
Path p2 = Paths.get("/home/fedor/temp.txt");
```

Эти методы являются сокращенной формой для следующего кода:

```
Path p3 = java.nio.file.FileSystems.getDefault()
        .getPath("/home/fedor/temp.txt");
```

Можно рассматривать `Path` как класс, сохраняющий имена каталогов в пути и имя файла в виде последовательности. Наивысший (ближний к корневому) элемент находится по индексу 0, самый нижний элемент находится по индексу `n - 1`, где `n` — количество элементов в пути.

Получить конкретный элемент пути можно с помощью метода `getName()`:

```
String element0 = path.getName(0)
// Для пути /home/fedor/foo (Linux) вернется home
// Для пути C:\home\fedor\foo (Windows) вернется home.
```

Узнать количество элементов в пути можно с помощью метода `getNameCount()`:

```
int nameCount = path.getNameCount();
// Для /home/fedor/foo (Linux) вернет 3
// Для C:\home\fedor\foo (Windows) вернет 3
```

Можно получить путь родительской директории с помощью метода `getParent()`:

```
Path parentPath = path.getParent();
// Для /home/fedor/foo (Linux) вернется /home/jho
// Для C:\home\fedor\foo Windows вернется C:\home\jho
// Вернет null, если родительского элемента нет.
```

Можно получить корень пути с помощью метода `getRoot()`:

```
Path rootPath = path.getRoot();
// Для /home/fedor/foo (Linux) вернет /
// Для C:\home\fedor\foo (Windos) вернет C:\
// Вернет null, если путь относительный и корня нет.
```

С помощью метода `toString()` можно получить путь в виде строки:

```
String str1 = path.toString()
// Вернет строку "/home/fedor/foo" для пути /home/fedor/foo (Linux).
// Вернет строку "C:\home\fedor\foo" для
// пути C:\home\fedor\foo (Windows).
```

Многие файловые системы используют символ точки "." для обозначения текущего каталога и две точки ".." для обозначения родительского каталога. Например:

```
/home/./fedor/foo
```

```
/home/maria/../fedor/foo
```

Метод `normalize()` удаляет все подобные элементы и приводит к нормализованному пути `/home/fedor/foo/`. Этот метод не проверяет файловую систему. Это чисто синтетическая операция, работающая с элементами `Path`. Если maria является символической ссылкой, то удаление `maria/..` может привести к тому, что `Path` больше не указывает на предыдущий файл.

Если вам нужно преобразовать `Path` к строке, с помощью которой можно открыть файл в браузере, то используйте метод `toUri()`:

```
Path p1 = Paths.get("/home/logfile");
// Result is file:///home/logfile
System.out.format("%s%n", p1.toUri());
```

Метод `toAbsolutePath()` преобразует путь к абсолютному. Способ преобразования зависит от системы. Если переданный путь уже является абсолютным, то возвращается тот же самый объект `Path`.

Метод `toRealPath()` возвращает реальный путь существующего файла. В качестве параметра в метод можно передать константу перечисления `java.nio.file.LinkOption` с единственным возможным значением `NOFOLLOW_LINKS`. Метод бросает исключение, если файл не существует, либо к нему нет доступа. Этот метод убирает все элементы "." и ".." и возвращает всегда абсолютный путь.

```
try {
    Path fp = path.toRealPath();
} catch (NoSuchFileException x) {
    System.err.format("%s: no such file or directory%n", path);
} catch (IOException x) {
    System.err.format("%s%n", x);
}
```

Можно объединять пути с помощью метода `resolve()`, в который передается часть пути, которую нужно добавить к исходному:

```
// Linux
Path p1 = Paths.get("/home/fedor/temp");

// Получим /home/fedor/temp/bar
System.out.format("%s%n", p1.resolve("bar"));
```

Или для Windows:

```
// Microsoft Windows
Path p1 = Paths.get("C:\\home\\fedor\\temp");
// Результат C:\home\fedor\temp
System.out.format("%s%n", p1.resolve("bar"));
```

С помощью метода `relativize()` можно создать относительный путь от одного пути к другому:

```
Path p1 = Paths.get("fedor");
Path p2 = Paths.get("yana");
// Так как нет другой информации, то считается, что
// fedor и yana находятся в одном каталоге.
// Результат ../yana
Path p1_relativize_p2 = p1.relativize(p2);
// Result is ../fedor
Path p2_relativize_p1 = p2.relativize(p1);
```

Относительный путь не может быть получен только в том случае, когда только один из путей имеет корневой элемент. Если оба пути имеют корневой элемент, то возможность построения относительного пути зависит от системы.

Path поддерживает метод `equals()`, что позволяет сравнивать пути. Также есть методы `startsWith(Path other)`, `startsWith(String other)`, `endsWith(Path other)`, `endsWith(String other)`, позволяющие проверять начало и конец пути на совпадение с указанной строкой или частью пути. `Path` реализует интерфейс `Comparable`, что позволяет сортировать строки.

Вы можете найти примеры использования `Path` в проекте `niofilecommander` на сайте **https://urvanov.ru**.

22.2. Что такое Glob?

Некоторые методы класса `java.nio.file.Files` принимают аргумент `glob`. Шаблон glob использует следующие правила:

- Символ `*` обозначает любое количество символов или их отсутствие.
- Две звездочки `**` работают так же, как и одна, но переходят за границы каталогов.
- Для того чтобы избавиться от специального значения символов (`*` или `?`), нужно предварять его символом `\`.
- Символ вопроса `?` обозначает ровно один символ.
- Квадратные скобки позволяют указать набор символов либо диапазон символов:
  - `[abc]` обозначает a, b или c.
  - `[A-Z]` обозначает любую прописную букву.
  - `[a-z,A-Z]` обозначает любую строчную или прописную букву.
  - Внутри квадратных скобок `*`, `?` и `/` обозначают самих себя.
  - `[4-7]` обозначает цифру от 4 до 7.
- Фигурные скобки указывают коллекцию паттернов. Например `{fedor,yana,temp}` совпадает с "fedor" , "yana" или "temp". `{Sir*, Sire*}` совпадает со всеми строками, начинающимися с "Sir" или "Sire".

22.3. Класс Files

Класс `java.nio.file.Files` содержит статические методы для работы с файлами, каталогами и другими типами файлов.

22.4. Проверка существования файла или каталога

Проверить существование пути `Path` можно с помощью методов класса `Files`:

```
public static boolean exists(Path path,
                             LinkOption... options)
```

```
public static boolean notExists(Path path,
                                LinkOption... options)
```

Если передать константу `LinkOption.NOFOLLOW_LINKS`, то метод не будет проходить по символическим ссылкам. Если оба метода `exists()` и `notExists()` возвращают `false`, то существование файла не может быть проверено (например, нет доступа).

22.5. Проверка прав доступа к файлу или каталогу

Проверка доступа к файлу осуществляется с помощью методов класса `Files`:

```
// Проверяет, что файл доступен для чтения
public static boolean isReadable(Path path)

// Проверяет, что файл доступен для записи
public static boolean isWritable(Path path)

// Проверяет, что файл доступен для выполнения
public static boolean isExecutable(Path path)
```

22.6. Один и тот же файл

Можно проверить, что два пути указывают на один и тот же файл на одной и той же файловой системе:

```
public static boolean isSameFile(Path path,
                                 Path path2)
                          throws IOException
```

22.7. Удаление файла или каталога

Метод `delete(Path)` удаляет файл или бросает исключение, если удалить файл не удалось. Можно удалять каталог, но только если он пустой.

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("No such file or directory: %s.%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // Ошибка удаления файла (например, недостаточно прав).
    System.err.println(x);
}
```

Метод

```
public static boolean deleteIfExists(Path path)
                              throws IOExceptio
```

тоже удаляет файл, но не бросает никаких исключений, если файл не существует. Это может быть полезно, если два потока удаляют файл, и вы не хотите получать исключение из-за того, что другой поток уже удалил файл до этого.

Пример использования можно найти в проекте niofilecommander на сайте **https://urvanov.ru**.

22.8. Копирование файла или каталога

Можно копировать файл или каталог с помощью метода `copy(Path, Path, CopyOption...)`, но имейте в виду, что файлы внутри каталога не копируются этим методом. Метод принимает константы `CopyOption`:

- `StandardCopyOption.REPLACE_EXISTING` заменяет существующие файлы.
- `StandardCopyOption.COPY_ATTRIBUTES` копирует атрибуты файла.
- `LinkOption.NOFOLLOW_LINKS` нельзя переходить по символическим ссылкам.

Пример использования можно найти в проекте niofilecommander на сайте **https://urvanov.ru**.

22.9. Перемещение файла или каталога

Можно переместить файл или каталог с помощью метода `move(Path, Path, CopyOption...)`. Можно перемещать пустые каталоги. Возможность перемещения каталогов с содержимым зависит от платформы. Метод принимает `CopyOption`:

- `StandardCopyOption.REPLACE_EXISTING` — заменяет существующий файл, если он существует.
- `StandardCopyOption.ATOMIC_MOVE` — попытка осуществить перемещение файла как единую атомарную операцию. Все остальные опции игнорируются.

Пример использования можно найти в проекте niofilecommander на сайте **https://urvanov.ru**.

22.10. Управление метаданными

Метаданные файлов — это размер, дата создания, дата последнего изменения, владелец, права доступа и прочее.

Метаданные файлов и каталогов часто называют атрибутами файлов.

Методы для работы с метаданными:

```
public static long size(Path path)
                throws IOException
```

Возвращает размер файла в байтах.

```
public static boolean isDirectory(Path path,
                                  LinkOption... options)
```

Проверяет, что path указывает на каталог. Можно указать `LinkOption.NOFOLLOW_LINKS`, чтобы метод не переходил по символическим ссылкам.

```
public static boolean isRegularFile(Path path,
                                    LinkOption... options)
```

Возвращает `true`, если `path` указывает на обычный файл. Можно передать `LinkOption.NOFOLLOW_LINKS`, чтобы метод не переходил по символическим ссылкам.

```
public static boolean isSymbolicLink(Path path)
```

Возвращает `true`, если path указывает на символическую ссылку.

```
public static boolean isHidden(Path path)
                       throws IOException
```

Возвращает `true`, если файл является скрытым. Для Linux файл является скрытым, если его имя начинается с точки. Для Windows файл является скрытым, если установлен соответствующий атрибут.

```
public static FileTime getLastModifiedTime(Path path,
                                           LinkOption... options)
                                    throws IOException
```

Возвращает `FileTime` с датой последнего изменения файла. Можно передать `LinkOption.NOFOLLOW_LINKS`, чтобы метод не переходил по символическим ссылкам.

```
public static Path setLastModifiedTime(Path path,
                                       FileTime time)
                                throws IOException
```

Устанавливает дату последнего изменения файла. Смотрите описание класса `FileTime` в документации Oracle.

```
public static UserPrincipal getOwner(Path path,
                                     LinkOption... options)
                              throws IOException
```

Возвращает владельца файла. Можно использовать метод `String getName()` у возвращенного объекта, чтобы получить имя пользователя.

```
public static Path setOwner(Path path,
                            UserPrincipal owner)
                     throws IOException
```

Меняет владельца файла.

Различные файловые системы имеют различные атрибуты файлов. Можно считывать группы атрибутов:

- `BasicFileAttributeView` — базовые атрибуты файлов, которые должны поддерживаться всеми реализациями файловых систем.
- `DosFileAttributeView` — расширяет стандартные атрибуты 4 битами (скрытый, архивный, только чтение, системный).
- `PosixFileAttributeView` — расширяет базовые атрибуты атрибутами системы Linux.
- `FileOwnerAttributeView` — поддерживается всеми файловыми системами, которые поддерживают владельцев файлов.
- `AclFileAttributeView` — права доступа к файлу, которые реализованы в Windows.
- `UserDefinedFileAttributeView` — пользовательские метаданные.

Получение конкретной группы атрибутов происходит с помощью метода

```
public static <V extends FileAttributeView> V getFileAttributeView(
        Path path,
        Class<V> type,
        LinkOption... options)
```

Пример:

```
Path path = ...
AclFileAttributeView view = Files.getFileAttributeView(path,
                                                       AclFileAttributeView.class);
if (view != null) {
    List<AclEntry> acl = view.getAcl();
    // ...
}
```

Работа с конкретными группами метаданных/атрибутов очень зависит от платформы. Вам вряд ли когда-нибудь придется столкнуться с этим. Но если интересно, то рекомендую рассмотреть мой проект niofilecommander (**https://urvanov.ru/2016/01/08/niofilecommander/**), который использует все группы атрибутов, описанные здесь.

22.11. OpenOption

Многие методы, описанные в этом разделе, используют OpenOption в качестве своих параметров. В этом случае в метод можно передавать следующие значения:

- `LinkOption.NOFOLLOW_LINKS` — не переходить по символическим ссылкам.
- `StandardOpenOption.APPEND` — если файл открыт для записи, то байты будут добавляться в конец файла, а не в начало.
- `StandardOpenOption.CREATE` — создавать новый файл, если его нет.
- `StandardOpenOption.CREATE_NEW` — создавать новый файл. Если файл уже существует, то происходит ошибка.

- `StandardOpenOption.DELETE_ON_CLOSE` — удалять файл при закрытии.
- `StandardOpenOption.DSYNC` — каждое обновление содержимого файла синхронно пишется на устройство хранения (жесткий диск).
- `StandardOpenOption.READ` — открыть для чтения.
- `StandardOpenOption.SPARSE` — разреженный файл (sparse file). Поддерживается в некоторых файловых системах. Для файлов, в которых много блоков, заполненных нулями, разреженный файл хранит вместо этих нулей только информацию об областях, заполненных нулями, что уменьшает занимаемое файлом место.
- `StandardOpenOption.SYNC` — каждое обновление содержимого файла или метаданных синхронно пишется на устройство чтения (жесткий диск).
- `StandardOpenOption.TRUNCATE_EXISTING` — если файл уже существует и открывается для записи, то его длина устанавливается в 0.
- `StandardOpenOption.WRITE` — открывает файл на запись.

22.12. Наиболее часто используемые методы для небольших файлов

```
public static byte[] readAllBytes(Path path)
                            throws IOException
```

Читает содержимое файла и возвращает его в массиве байт.

```
public static List<String> readAllLines(Path path,
                                        Charset cs)
                                  throws IOException
```

Читает содержимое текстового файла и возвращает его в виде списка строк.

```
public static Path write(Path path,
                         byte[] bytes,
                         OpenOption... options)
                  throws IOException
```

Записывает байты в файл.

```
public static Path write(Path path,
                         Iterable<? extends CharSequence> lines,
                         Charset cs,
                         OpenOption... options)
                  throws IOException
```

Записывает строки в файл, преобразуя их в указанную кодировку.

22.13. Буферизированный ввод и вывод в текстовые файлы

```
public static BufferedReader newBufferedReader(Path path,
                                               Charset cs)
                                        throws IOException
```

Возвращает `BufferedReader`, который может быть использован для чтения из текстового файла.

```
public static BufferedWriter newBufferedWriter(Path path,
                                               Charset cs,
                                               OpenOption... options)
                                        throws IOException
```

Открывает или создает файл для записи. Возвращаемый `BufferedWriter` может быть использован для записи в файл. Если `OpenOption` не переданы, то используются опции `CREATE`, `TRUNCATE_EXISTING`, `WRITE`.

22.14. Небуферизированный ввод и вывод

```
public static InputStream newInputStream(Path path,
                                         OpenOption... options)
                                  throws IOException
```

Возвращает поток, который можно использовать для чтения байт из файла.

```
public static OutputStream newOutputStream(Path path,
                                           OpenOption... options)
                                    throws IOException
```

Возвращает поток, который можно использовать для записи байт в файл. Если OpenOption не переданы, то используются опции `CREATE`, `TRUNCATE_EXISTING`, `WRITE`.

22.15. Создание файлов

```
public static Path createFile(Path path,
                              FileAttribute<?>... attrs)
                       throws IOException
```

Создает новый и пустой файл. Бросает исключение, если файл уже существует.

22.16. Создание временных файлов

```
public static Path createTempFile(Path dir,
                                  String prefix,
                                  String suffix,
                                  FileAttribute<?>... attrs)
                           throws IOException
```

Создает временный файл в указанном каталоге. Является только частью работы с временными файлами. Вы можете использовать метод `File.deleteOnExit()` для того, чтобы каталог удалялся автоматически.

```
public static Path createTempFile(String prefix,
                                   String suffix,
                                   FileAttribute<?>... attrs)
                            throws IOException
```

Создает временный файл в специальном каталоге для временных файлов. Является только частью работы с временными файлами. Вы можете использовать метод `File.deleteOnExit()` для того, чтобы каталог удалялся автоматически.

22.17. Java NIO.2 Channels

В `java.nio.file.channels` хранятся классы для работы с файлами, ресурсами сети и другими устройствами через каналы.

В Java NIO Channel есть несколько интерфейсов: `ByteChannel`, `ReadableByteChannel`, `WritableByteChannel`, `SeekableByteChannel` и др.

А также различные классы, реализующие работу с разными сущностями (файлами, сетью и т. д.): `FileChannel`, `ServerSocketChannel`, `SocketChannel`, `DatagramChannel` и т. д.

Интерфейс `ReadableByteChannel` содержит метод `read`:

```
int read(ByteBuffer dst)
  throws IOException
```

Метод `read` считывает данные в `dst` и возвращает количество считанных байт либо –1 при достижении конца файла.

Интерфейс `WritableByteChannel` содержит метод `write`:

```
int write(ByteBuffer src)
   throws IOException
```

Метод write возвращает количество записанных байт.

Файл можно открыть одновременно для чтения и записи и с возможностью перемещения текущей позиции в любое место файла. Подобная функциональность реализуется с помощью интерфейса `java.nio.channels.SeekableByteChannel`.

Получить экземпляр класса, реализующего интерфейс `SeekableByteChannel`, можно с помощью статического метода из класса `Files`:

```
public static SeekableByteChannel newByteChannel(Path path,
                                                 OpenOption... options)
                                          throws IOException
```

либо:

```
public static SeekableByteChannel newByteChannel(
        Path path,
```

```
        Set<? extends OpenOption> options,
        FileAttribute<?>... attrs)
                throws IOException
```

Пример использования (класс `FileChannel` реализует `ReadableByteChannel`, `WritableByteChannel`):

FileChannelExample.java

```
package ru.urvanov.javaindynamics2022.nio;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Paths;

import static java.nio.file.StandardOpenOption.READ;
import static java.nio.file.StandardOpenOption.WRITE;

public class FileChannelExample {
    public static void main(String[] args) {
        String s = "Моя добавленная строчка\n";
        byte data[] = s.getBytes();
        ByteBuffer out = ByteBuffer.wrap(data);

        ByteBuffer copy = ByteBuffer.allocate(12);

        // Вместо output.txt укажите имя существующего файла
        // (его содержимое будет изменено)
        try (FileChannel fc = FileChannel.open(
                Paths.get("output.txt"), READ, WRITE)) {
            // Читаем первые 12 байт из файла.
            int nread;
            do {
                nread = fc.read(copy);
            } while (nread != -1 && copy.hasRemaining());

            // Пишем "Моя добавленная строчка" в начало файла.
            fc.position(0);
            while (out.hasRemaining())
                fc.write(out);
            out.rewind();

            // Перемещаемся в конец файла. Копируем первые 12 байт
            // в конец файла.  Пишем "Моя добавленная строчка" снова.
            long length = fc.size();
            fc.position(length-1);
            copy.flip();
```

```
            while (copy.hasRemaining())
                fc.write(copy);
            while (out.hasRemaining())
                fc.write(out);
        } catch (IOException x) {
            System.out.println("I/O Exception: " + x);
        }
    }
}
```

22.18. Перечисление корневых каталогов файловой системы

Для этого используется специальный метод класса `FileSystem`:

```
public abstract Iterable<Path> getRootDirectories()
```

Пример:

```
Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
for (Path name: dirs) {
    System.err.println(name);
}
```

22.19. Создание каталога

Метод класса Files:

```
public static Path createDirectory(Path dir,
                                   FileAttribute<?>... attrs)
                            throws IOException
```

создает каталог.

```
public static Path createDirectories(Path dir,
                                     FileAttribute<?>... attrs)
                              throws IOException
```

создает каталог вместе со всеми родительскими каталогами, которых еще нет.

Пример использования можно найти в проекте niofilecommander на сайте **https://urvanov.ru**.

22.20. Создание временного каталога

```
public static Path createTempDirectory(Path dir,
                                       String prefix,
                                       FileAttribute<?>... attrs)
                                throws IOException
```

Создает временный каталог. Как и методы, `createTempFile` является только частью работы с временными файлами. Вы можете использовать метод `File.deleteOnExit()` для того, чтобы каталог удалялся автоматически.

```
public static Path createTempDirectory(String prefix,
                                        FileAttribute<?>... attrs)
                                 throws IOException
```

Создает временный каталог в специальном каталоге для временных файлов. Как и методы, `createTempFile` является только частью работы с временными файлами. Вы можете использовать метод `File.deleteOnExit()` для того, чтобы каталог удалялся автоматически.

22.21. Перечисление содержимого каталога

```
public static DirectoryStream<Path> newDirectoryStream(Path dir)
                                                 throws IOException
```

Возвращает поток, который позволяет пройтись по всем элементам каталога.

Пример использования:

DirectoryStreamExample.java

```
// Подставьте сюда путь к каталогу вместо src
Path dir = Paths.get("src");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
    for (Path file: stream) {
        System.out.println(file.getFileName());
    }
} catch (IOException | DirectoryIteratorException x) {
    // IOException не может броситься во время итерации.
    // В этом куске кода оно может броситься только
    // методом newDirectoryStream.
    System.err.println(x);
}
public static DirectoryStream<Path> newDirectoryStream(Path dir,
                                                       String glob)
                                                throws IOException
```

Позволяет посмотреть элементы каталога, названия которых удовлетворяют переданному шаблону Glob.

Например, следующий код возвращает список файлов с расширениями .class, .java, .jar:

DirectoriesByGlob.java

```
// Подставьте сюда путь к каталогу вместо target/classes
Path dir = Paths.get("target/classes");
```

```
try (DirectoryStream<Path> stream =
     Files.newDirectoryStream(dir, "*.{java,class,jar}")) {
    for (Path entry: stream) {
        System.out.println(entry.getFileName());
    }
} catch (IOException x) {
    // IOException никогда не бросится во время итерации.
    // В этом куске кода оно может броситься только
    // методом newDirectoryStream
    System.err.println(x);
}
```

Можно написать свой собственный фильтр для содержимого каталога. Например, этот фильтр будет возвращать только каталоги:

DirectoriesByFilter.java

```
package ru.urvanov.javaindynamics2022.nio;

import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class DirectoriesByFilter {
    public static void main(String[] args) {
        DirectoryStream.Filter<Path> filter =
                new DirectoryStream.Filter<Path>() {
            public boolean accept(Path file) throws IOException {
                return (Files.isDirectory(file));
            }
        };
        // Подставьте сюда путь к каталогу вместо target/classes
        Path dir = Paths.get("target/classes");
        try (DirectoryStream<Path>
                     stream = Files.newDirectoryStream(dir, filter)) {
            for (Path entry: stream) {
                System.out.println(entry.getFileName());
            }
        } catch (IOException x) {
            System.err.println(x);
        }
    }
}
```

22.22. Символические и другие ссылки

Каждый метод класса `Path` автоматически обрабатывает символические ссылки либо предоставляет опции, которые позволяют указать, каким образом их обрабатывать.

Ссылки бывают символическими и жесткими. Жесткие ссылки имеют больше ограничений, чем символические:

- Цель, на которую указывает жесткая ссылка, должна существовать.
- Жесткие ссылки обычно не могут указывать на каталоги.
- Жесткие ссылки не могут указывать на другой раздел или том и не могут указывать на другую файловую систему.
- Жесткие ссылки выглядят и ведут себя так, как обычный файл, поэтому их может быть трудно обнаружить.
- Жесткая ссылка для всех целей и действий является той же сущностью, что и исходный файл. Они имеют те же самые права доступа, временные метки и т. д.

Из-за таких ограничений жесткие ссылки не так часто используются, как символические ссылки.

22.23. Создание символических ссылок

Если ваша файловая система поддерживает символические ссылки, то вы можете создавать их с помощью метода класса `Files`:

```
public static Path createSymbolicLink(Path link,
                                       Path target,
                                       FileAttribute<?>... attrs)
                                throws IOException
```

Этот метод создает символическую ссылку по пути `link`, указывающую на `target`. Параметр `attrs` позволяет задать атрибуты ссылки.

Пример использования можно найти в проекте niofilecommander на сайте **https://urvanov.ru**.

22.24. Создание жестких ссылок

```
public static Path createLink(Path link,
                              Path existing)
                       throws IOException
```

Создает жесткую ссылку по пути `link`, указывающую на `existing`.

Пример использования можно найти в проекте niofilecommander на сайте **https://urvanov.ru**.

22.25. Определение символической ссылки

```
public static boolean isSymbolicLink(Path path)
```

Этот метод класса `Files` возвращает `true`, если файл является символической ссылкой.

22.26. Нахождение цели ссылки

```
public static Path readSymbolicLink(Path link)
                              throws IOException
```

Этот метод класса `Files` возвращает путь, на который указывает символическая ссылка. Если `link` не является ссылкой, то бросается исключение `java.nio.file.NotLinkException`.

22.27. Обход дерева файлов с FileVisitor

Чтобы обходить дерево файлов, нужно сперва написать класс, реализующий интерфейс `java.nio.file.FileVisitor`. Этот интерфейс позволяет описать действия, которые будут происходить в основные моменты обхода дерева файлов: при посещении файла, перед доступом к каталогу, после доступа к каталогу и при возникновении ошибки. `FileVisitor` имеет четыре метода для этих ситуаций:

- `preVisitDirectory` — вызывается перед посещением элементов каталога.
- `postVisitDirectory` — вызывается после посещения всех элементов каталога.
- `visitFile` — вызывается при посещении файла. В метод передаются `BasicFileAttributes`.
- `visitFileFailed` — вызывается в случае, когда не удается получить доступ к файлу.

Если вам не нужны все эти четыре метода, то вы можете вместо реализации интерфейса `FileVisitor` наследоваться от `java.nio.file.SimpleFileVisitor`. Этот класс реализует `FileVisitor` и посещает все файлы в дереве, бросая исключение `IOException` при возникновении ошибки. Вы можете отнаследоваться от этого класса и реализовать только те методы, которые вам нужны.

Пример реализации `SimpleFileVisitor`:

PrintFiles.java

```
import static java.nio.file.FileVisitResult.*;

public static class PrintFiles
    extends SimpleFileVisitor<Path> {

    // Выводит в консоль информацию о каждом типе файлов.
    @Override
```

```
    public FileVisitResult visitFile(Path file,
                                     BasicFileAttributes attr) {
        if (attr.isSymbolicLink()) {
            System.out.format("Symbolic link: %s ", file);
        } else if (attr.isRegularFile()) {
            System.out.format("Regular file: %s ", file);
        } else {
            System.out.format("Other: %s ", file);
        }
        System.out.println("(" + attr.size() + "bytes)");
        return CONTINUE;
    }

    // Пишет в консоль каждый посещаемый каталог
    @Override
    public FileVisitResult postVisitDirectory(Path dir,
                                              IOException exc) {
        System.out.format("Directory: %s%n", dir);
        return CONTINUE;
    }

    // Если возникла какая-нибудь ошибка при доступе к файлу,
    // то выводим эту ошибку.
    // Если вы не переопределите этот метод и возникнет
    // ошибка, то бросится исключение IOException
    @Override
    public FileVisitResult visitFileFailed(Path file,
                                           IOException exc) {
        System.err.println(exc);
        return CONTINUE;
    }
}
```

Запуск процесса обхода дерева файлов

```
public static Path walkFileTree(Path start,
                                FileVisitor<? super Path> visitor)
                        throws IOException
```

или

```
public static Path walkFileTree(Path start,
                                Set<FileVisitOption> options,
                                int maxDepth,
                                FileVisitor<? super Path> visitor)
                        throws IOException
```

Второй метод позволяет задать максимальную глубину обхода и `FileVisitOption.FOLLOW_LINKS`, что позволит заходить в символические ссылки.

Пример для нашего `PrintFiles`:

PrintFiles.java

```
// Вместо general/src подставить путь к своему каталогу
Path startingDir = Paths.get("general/src");
PrintFiles pf = new PrintFiles();
Files.walkFileTree(startingDir, pf)
```

Дерево файлов обходится в глубину, но вы не можете предугадать порядок посещения подкаталогов.

Если ваша программа делает изменения в файловой системе, то вам нужно внимательно реализовывать ваш `FileVisitor`.

Например, если вы пишете рекурсивное удаление, то сначала вам нужно удалить файлы в каталоге, а затем удалять сам каталог. В этом случае удалять каталог нужно в методе `postVisitDirectory`.

Если вы пишете рекурсивное копирование, то вам сначала нужно создавать каталог в `preVisitDirectory`, а лишь потом копировать файлы в `visitFiles`. Если вам нужно скопировать атрибуты из исходного каталога, то это нужно делать после копирования каталога, т. е. в `postVisitDirectory`.

Если вы пишете поиск файла, то осуществляйте сравнение в `visitFile`; если вам нужно находить не только файлы, но и каталоги, то используйте вдобавок метод `preVisitDirectory` или `postVisitDirectory`.

Вам также нужно решить, как обрабатывать символические ссылки. Если вы удаляете файлы, то переходить по символическим ссылкам, скорее всего, не нужно. Если вы копируете дерево файлов, то вам, вероятно, потребуется переходить по ссылкам. По умолчанию метод `walkFileTree` не переходит по символическим ссылкам.

Метод `visitFile` вызывается для файлов. Если вы указали `FOLLOW_LINKS` и ваше дерево файлов имеет циклическую ссылку на родительский каталог, то во время повторения каталога будет сообщено в `visitFileFailed` с помощью исключения `FileSystemLoopException`.

Во время обхода может потребоваться не заходить внутрь какого-либо каталога либо даже прервать обход дерева. Методы `FileVisitor` возвращают `java.nio.file.FileVisitResult`, которое является перечислением со следующими константами:

- `FileVisitResult.CONTINUE` — указывает, что обход дерева файлов должен продолжаться. Если метод `preVisitDirectory` возвращает `CONTINUE`, то каталог посещается.
- `FileVisitResult.TERMINATE` — прерывает обход всего дерева файлов.
- `FileVisitResult.SKIP_SUBTREE` — если `preVisitDirectory` возвращает это значение, то указанный каталог и его подкаталоги пропускаются.

- `FileVisitResult.SKIP_SIBLINGS` — если `preVisitDirectory` возвращает это значение, то указанный каталог не посещается и `postVisitDirectory` не вызывается. Если возвращается из `postVisitDirectory`, то остальные каталоги с тем же родительским каталогом пропускаются.

Смотрите примеры в niofilecommander на сайте (**https://urvanov.ru/2016/01/08/niofilecommander/**).

22.28. Поиск файлов

Каждая реализация файловой системы поставляет свою реализацию интерфейса `java.nio.file.PathMatcher`.

Вы можете получить экземпляр класса, реализующий этот интерфейс для файловой системы:

```
String pattern = ...;
PathMatcher matcher =
    FileSystems.getDefault().getPathMatcher("glob:" + pattern);
```

Строка, которая передается в этот метод, указывает, какие файлы искать. В этом примере используется синтаксис `glob`. Можно использовать регулярные выражения, тогда вместо `glob:` нужно писать `regex:`.

В дальнейшем этот matcher можно использовать при обходе дерева файлов вот так:

```
if (matcher.matches(filename)) {
    System.out.println(filename);
}
```

Пример использования можно найти в проекте niofilecommander на сайте **https://urvanov.ru**.

22.29. Подписываемся на изменения в каталоге

Последовательность шагов, которую нужно проделать для того, чтобы следить за изменениями в каком-нибудь каталоге:

1. Создаем экземпляр `WatchService` для файловой системы.
2. Для каждого каталога, за которым нужно наблюдать, регистрируем наблюдателя (watcher). При регистрации каталога вы указываете события, о которых вам бы хотелось знать. Для каждого каталога вы получаете экземпляр класса `WatchKey`.
3. Делаем бесконечный цикл на входящих событиях. При возникновении события `WatchKey` этого каталога получает сигнал и кладется в очередь наблюдателя.
4. Получаем `WatchKey` из очереди. Вы можете получить имя файла из этого ключа.
5. Получаем каждое событие для ключа (их может быть несколько) и обрабатываем.
6. Сбрасываем `WatchKey` и возвращаемся к ожиданию событий.

7. Закрываем `WatchServce`. Он закрывается при выходе из потока либо при вызове метода.

Экземпляры `WatchKey` потокобезопасны.

```
import static java.nio.file.StandardWatchEventKinds.*;
...

WatchService watcher = FileSystems.getDefault().newWatchService();

Path dir = ...;
try {
    WatchKey key = dir.register(watcher,
                           ENTRY_CREATE,
                           ENTRY_DELETE,
                           ENTRY_MODIFY);
} catch (IOException x) {
    System.err.println(x);
}
```

Для обработки событий сначала мы получаем `WatchKey` с помощью одного из методов WatchService:

```
WatchKey poll()
```

Удаляет из очереди и возвращает следующий `WatchKey` либо `null`, если очередь пуста.

```
WatchKey poll(long timeout,
              TimeUnit unit)
       throws InterruptedException
```

Удаляет из очереди и возвращает следующий `WatchKey`. Ждет, если нужно, указанное количество времени.

```
WatchKey take()
       throws InterruptedException
```

Удаляет из очереди и возвращает следующий `WatchKey`. Если очередь пуста, то ждет до тех пор, пока не появится хоть что-нибудь.

Затем обрабатываются все события из `WatchKey`, которые получаются при вызове его метода:

```
List<WatchEvent<?>> pollEvents()
```

Тип события получается с помощью следующего метода `WatchEvent`:

```
WatchEvent.Kind<T> kind()
```

Независимо от событий, на которые мы подписаны, мы может получить событие `OVERFLOW`.

Получаем имя файла из события с помощью метода `context()`, который возвращает относительный путь к файлу от каталога, на изменения в котором мы подписаны.

Вызываем `reset()` на `WatchKey`. Если он вернул `false`, то он больше не работает и из цикла можно выходить.

Пример обработки событий изменения содержимого каталога можно увидеть в niofilecommander на моем сайте, который обновляет свои панели при добавлении, переименовании и удалении файлов в них.

22.30. Задания

1. Изучите код niofilecommander с сайта **https://urvanov.ru** (**https://urvanov.ru/2016/01/08/niofilecommander/**).
2. Вспомните главу про модули из Java 9. Попытайтесь добавить возможность использования подключаемых модулей в niofilecommander. Напишите какой-нибудь простой модуль. Например, выпадающий список для более быстрого перемещения к корневым каталогам (`getRootDirectories`).
3. Добавьте в niofilecommander внутренний просмотрщик текстовых файлов. Предусмотрите возможность просмотра очень больших файлов. Считывайте не весь файл целиком, а только ту часть, которую реально необходимо считать для отображения фрагмента текста в соответствии с полосой прокрутки. Будет очень хорошо, если вы оформите его в виде подключаемого модуля Java 9.
4. Добавьте еще какие-нибудь возможности в niofilecommander. Например, просмотр содержимого ZIP-файлов.

ГЛАВА 23

Многопоточность

23.1. Класс Thread

Каждый поток ассоциирован с классом `java.lang.Thread`. Есть два основных способа использования объектов `Thread` в многопоточном программировании:

- Создание потоков и управление ими с помощью экземпляров класса `Thread`.
- Абстрагирование от управления потоками и передача задач в `executor`.

Есть два способа запуска кода в другом потоке с помощью `Thread`:

1. Предоставить экземпляр класса, реализующего интерфейс `java.lang.Runnable`. Этот класс имеет один метод `run()`, который должен содержать код, выполняе-

мый в отдельном потоке. Экземпляр класса `java.lang.Runnable` передается в конструктор класса `Thread`:

RunnableImpl.java

```
package ru.urvanov.javaindynamics2022.multithreading;

public class RunnableImpl implements Runnable {

    public void run() {
        System.out.println("Текст из другого потока");
    }

    public static void main(String args[]) {
        (new Thread(new RunnableImpl())).start();
    }

}
```

2. Написать подкласс класса `Thread`. Класс `Thread` сам реализует интерфейс `java.lang.Runnable`, но его метод `run()` ничего не делает. Приложение может унаследовать класс от Thread и переопределить метод `run()`:

ThreadChild.java

```
package ru.urvanov.javaindynamics2022.multithreading;

public class ThreadChild extends Thread {

    public void run() {
        System.out.println("Текст из другого потока");
    }

    public static void main(String args[]) {
        (new ThreadChild()).start();
    }
}
```

В обоих случаях мы используем метод `Thread.start()` для запуска нового потока. Именно он запускает отдельный поток. Если просто вызывать метод `run()`, то код будет выполняться в том же потоке, отдельный поток создаваться не будет.

Первый способ, где предоставляется экземпляр класса, реализующего Runnable, более общий, т. к. в этом случае класс может наследоваться от отличного от Thread класса. Второй способ легче использовать в простых приложениях, но он ограничен тем, что ваш класс будет наследником `Thread`.

Метод `sleep` класса `Thread` останавливает выполнение текущего потока на указанное время. Он используется, когда нужно освободить процессор, чтобы он занялся

другими потоками или процессами либо для задания интервала между какими-нибудь действиями.

Есть два варианта метода `sleep`: первый принимает в качестве параметра количество миллисекунд, на которое нужно остановить текущий поток, второй дополнительно принимает второй параметр, в котором указывается количество наносекунд, на которые нужно дополнительно остановить поток:

```
public static void sleep(long millis)
                  throws InterruptedException

public static void sleep(long millis, int nanos)
                  throws InterruptedException
```

где

`millis` — это количество миллисекунд,

`nanos` — количество наносекунд дополнительно к `millis`.

Время остановки потока неточно, оно зависит от возможностей системы. К тому же состояние ожидания для потока может быть прервано извне.

Пример:

SleepExample.java

```
package ru.urvanov.javaindynamics2022.multithreading;

public class SleepExample {
    public static void main(String args[])
            throws InterruptedException {
        for (int n = 0; n < 10; n++) {
            // Ждем 2 секунды
            Thread.sleep(2_000);
            // Выводим значение счетчика цикла
            System.out.println(n);
        }
    }
}
```

Обратите внимание: метод `main` объявляет, что он `throws InterruptedException`. Это исключение бросается методом `sleep`, если поток прерывается во время ожидания внутри `sleep`. Так как эта программа не объявила никаких других потоков, которые могут прерывать текущий, то ей вовсе не обязательно обрабатывать это исключение.

Прерывание (`interrupt`) — это сигнал для потока, что он должен прекратить делать то, что он делает сейчас, и делать что-то другое. Что должен делать поток в ответ на прерывание, решает программист, но обычно поток завершается.

Поток отправляет прерывание, вызывая метод

```
public void interrupt()
```

класса `Thread`. Для того чтобы механизм прерывания работал корректно, прерываемый поток должен поддерживать возможность прерывания своей работы.

Как поток должен поддерживать прерывание своей работы? Это зависит от того, что он сейчас делает. Если поток часто вызывает методы, которые могут бросить `InterruptedException`, то он просто вызывает `return` при перехвате подобного исключения. Пример:

```
try {
    Thread.sleep(1_000);
} catch (InterruptedException e) {
    // Ожидание было прервано
    return;
}
```

Многие методы, которые бросают `InterruptedException`, например методы `sleep`, останавливают свое выполнение и возвращают управление в вызвавший их код при получении прерывания (`interrupt`).

Если поток выполняется длительное время без вызова методов, которые бросают исключение `InterruptedException`, то он может периодически вызывать метод `Thread.interrupted()`, который возвращает `true`, если получен сигнал о прерывании. Например:

```
for (int i = 0; i < monsters.length; i++) {
    recalculateStats(monsters[i]);
    if (Thread.interrupted()) {
        // Обработка параметров монстров была прервана. Выходим
        return;
    }
}
```

В этом примере код просто проверяет наличие сигнала о прерывании и выходит из потока, если сигнал есть. В более сложных приложениях имеет смысл бросить исключение `InterruptedException`:

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

Это позволяет располагать код обработки прерывания потока в одной клаузе catch.

Механизм прерывания реализован с помощью внутреннего флага, известного как статус прерывания (interrupt status). Вызов `Thread.interrupt()` устанавливает этот флаг. Когда поток проверяет наличие прерывания вызовов `Thread.interrupted()`, то флаг статуса прерывания сбрасывается. Нестатический метод `isInterrupted()`, который используется одним потоком для проверки статуса прерывания другого потока, не меняет флаг статуса прерывания.

По соглашению любой метод, который прерывает свое выполнение, бросая исключение `InterruptedException`, очищает флаг статуса прерывания, когда он бросает это исключение. Однако есть вероятность, что флаг статуса прерывания будет сразу же установлен еще раз, если другой поток вызовет `interrupt()`.

Метод `join` позволяет одному потоку ждать завершения другого потока. Если `t` является экземпляром класса `Thread`, чей поток в данный момент продолжает выполняться, то

```
t.join();
```

приведет к приостановке выполнения текущего потока до тех пор, пока поток `t` не завершит свою работу. Метод `join()` имеет варианты с параметрами:

```
public final void join(long millis)
                throws InterruptedException

public final void join(long millis,
                       int nanos)
                throws InterruptedException
```

Они позволяют задать время в миллисекундах и дополнительно количество наносекунд, в течение которых ждать завершения выполнения потока. Однако, как и с методами `sleep`, методы `join` зависят от возможностей операционной системы, поэтому вы не должны полагаться на то, что `join` будет ждать точно указанное время.

Как и методы `sleep`, методы `join` отвечают на сигнал прерывания, останавливая процесс ожидания и бросая исключение `InterruptedException`.

Пример состоит из двух потоков. Первый является главным потоком приложения, который есть в каждой программе на Java. Главный поток создает новый и ждет его завершения. Если второй поток выполняется слишком долго, то главный прерывает его.

InterruptJoinExample.java

```
package ru.urvanov.javaindynamics2022.multithreading;

public class InterruptJoinExample {

    // Выводим сообщение с именем текущего потока в начале.
    static void writeMessage(String message) {
        String threadName =
                Thread.currentThread().getName();
        System.out.format("[%s]: %s%n",
                threadName,
                message);
    }

    private static class Counter
            implements Runnable {
```

```
        public void run() {
            try {
                for (int n = 0; n < 5; n++) {
                    // Ждем 2 секунды
                    Thread.sleep(2_000);
                    // Выводим значение счетчика цикла
                    writeMessage("counter " + n);
                }
            } catch (InterruptedException interruptedException) {
                writeMessage("Thread interrupted");
            }
        }
    }

    public static void main(String args[])
            throws InterruptedException {

        writeMessage("Starting Counter");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new Counter());
        t.start();

        writeMessage("Waiting for Counter to finish");
        // ждем, пока MessageLoop существует
        while (t.isAlive()) {
            writeMessage("calling join...");
            // Ждем максимум 1 секунду
            // завершения потока Counter
            t.join(1_000);

            // максимально 10 секунд
            // Если поток не успел, то посылаем ему сигнал прерывания
            if (((System.currentTimeMillis() - startTime) > 10000)
                    && t.isAlive()) {
                writeMessage("Interrupting...");
                t.interrupt();
                // Должно быть недолго теперь.
                // -- Ждем до конца
                t.join();
            }
        }
        writeMessage("DONE");
    }
}
```

23.2. Синхронизация

Потоки общаются, в основном разделяя свои поля и поля объектов между собой. При этом может возникнуть два типа ошибок: вмешательство в поток (thread interference) и ошибки консистентности памяти (memory consistency errors). Для того чтобы предотвратить эти ошибки, нужно использовать синхронизацию потоков.

Однако синхронизация может привести к конкуренции потоков (thread contention), которая возникает, когда два или более потока пытаются получить доступ к одному и тому же ресурсу одновременно; это приводит к тому, что среда выполнения Java выполняет один или более этих потоков медленнее или даже приостанавливает их выполнение. Голодание (starvation) и активная блокировка (livelock) — это формы конкуренции потоков.

23.3. Вмешательство в поток (thread interference)

Рассмотрим простой класс Counter:

Counter.java

```
package ru.urvanov.javaindynamics2022.multithreading;

// Пример вмешательства в поток (thread interference)
class Counter {
    private int value = 0;

    public void increment() {
        for (int n = 0; n < 100_000; n++) {
            value++;
        }
    }

    public void decrement() {
        for (int n = 0; n < 100_000; n++) {
            value--;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread threadIncrement = new Thread(counter::increment);
        Thread threadDecrement = new Thread(counter::decrement);
        threadIncrement.start();
        threadDecrement.start();
        threadIncrement.join();
        threadDecrement.join();
```

```
        // Каждый раз будем получать разный результат,
        // из-за того что потоки несинхронизированы.
        // Это последствия thread interference.
        System.out.println(counter.getValue());
    }

    private int getValue() {
        return value;
    }

}
```

Каждый вызов метода `increment` добавляет 1 к `c` в цикле 100 000 раз, а каждый вызов decrement вычитает 1 из `c` в цикле 100 000 раз. Однако, если объект `Counter` используется несколькими потоками, вмешательство в поток может помешать этому коду работать, как ожидалось.

Вмешательство в поток происходит, когда два действия выполняются разными потоками, но используют одни и те же данные. Это означает, что два действия, которые содержат несколько шагов, и последовательность шагов частично перекрываются.

Может показаться, что операции над экземплярами `Counter` не могут перекрываться, т. к. все операции над c являются одиночными простыми инструкциями. Однако даже простые инструкции могут транслироваться виртуальной машиной в несколько шагов. Выражение `c++` может быть разложено на три шага:

1. Получить текущее значение `value`.
2. Увеличить полученное значение на 1.
3. Сохранить увеличенное значение в `value`.

Предположим, что поток `threadIncrement` вызывает `increment`, и в то же самое время поток `threadDecrement` вызывает decrement. Начальное значение c равно 0, их пересеченные действия могут породить следующую последовательность шагов:

1. Поток `threadIncrement` получает `value`.
2. Поток `threadDecrement` получает `value`.
3. Поток `threadIncrement` увеличивает полученное значение, в результате получает 1.
4. Поток `threadDecrement` уменьшает полученное значение, в результате получает –1.
5. Поток `threadIncrement` сохраняет результат 1 в `value`.
6. Поток `threadDecrement` сохраняет результат –1 в `value`.

Результат потока `threadIncrement` потерян, он был перезаписан потоком `threadDecrement`. Такое частичное перекрытие действий — это только один из вариантов. В некоторых других ситуациях может оказаться, что результат потока `threadDecrement` будет потерян либо ошибок не будет совсем. Из-за этого ошибки вмешательства в поток трудно обнаруживать и исправлять.

23.4. Ошибки согласованности памяти (memory consistency errors)

Ошибки согласованности памяти (memory consistency errors, ошибки консистентности памяти) возникают, когда разные потоки видят разные данные в своих разделяемых объектах и их полях. Для исключения ошибок согласованности памяти нужно понимать связи *происходит-до* (happens-before). Эта связь гарантирует, что данные, записанные в память одной инструкцией, видимы в другой. Рассмотрим следующий пример. Предположим, что поле типа `int` объявлено и инициализировано:

```
int value = 0;
```

Поле `value` используется совместно двумя потоками `threadA` и `threadB` аналогично тому, как это было в примере Counter. Предположим, что поток `threadA` увеличивает `value`:

```
value++;
```

Сразу же после этого поток `threadB` выводит в консоль значение `value`:

```
System.out.println(value);
```

Если бы обе инструкции были выполнены одним потоком, то можно было бы смело предположить, что в консоль выведется число 1. Однако, если две инструкции выполняются разными потоками, может быть выведено 0, т. к. нет гарантии, что изменение `value` потоком `threadA` будет видимо потоком `threadB`, до тех пор, пока не будет обеспечена связь *происходит-до* (happens-before) между этими инструкциями.

Подробнее про happens-before читайте в *разделе 23.19 "Java Memory Model"*.

23.5. Синхронизированные (synchronized) методы

Язык программирования Java предоставляет два базовых способа синхронизации: синхронизированные методы (synchronized methods) и синхронизированные инструкции (synchronized statements). Есть другие, более сложные способы синхронизации, они будут рассмотрены в дальнейшем.

Чтобы сделать метод синхронизированным (synchronized), просто добавьте ключевое слово `synchronized` к его объявлению:

SynchronizedCounter.java

```
package ru.urvanov.javaindynamics2022.multithreading;

public class SynchronizedCounter {
    private int value = 0;
```

```
    public void increment100000() {
        for (int n = 0; n < 100_000; n++) {
            increment();
        }
    }

    private synchronized void increment() {
        value++;
    }

    public void decrement100000() {
        for (int n = 0; n < 100_000; n++) {
            decrement();
        }
    }

    private synchronized void decrement() {
        value--;
    }

    public static void main(String[] args) throws InterruptedException {
        SynchronizedCounter counter = new SynchronizedCounter();
        Thread threadIncrement = new Thread(counter::increment100000);
        Thread threadDecrement = new Thread(counter::decrement100000);
        threadIncrement.start();
        threadDecrement.start();
        threadIncrement.join();
        threadDecrement.join();
        // Результат будет 0, т. к. методы
        // increment и decrement синхронизированы.
        System.out.println(counter.getValue());
    }

    private int getValue() {
        return value;
    }
}
```

Синхронизированные методы работают по следующим правилам:

- Два вызова синхронизированных методов на одном и том же объекте не могут пересекаться. Когда один поток выполняет синхронизированный метод объекта, то другие потоки, которые вызывают синхронизированные методы того же самого объекта, блокируются (приостанавливают свое выполнение) до тех пор, пока первый поток не завершит работу с объектом.
- Когда синхронизированный метод завершает свое выполнение, то он автоматически делает связь *происходит-до* (happens-before) со всеми последующими

вызовами синхронизированных методов того же самого объекта. Это гарантирует, что изменения состояния объекта будут видимы для других потоков.

Конструкторы не могут быть синхронизированными, это не имеет смысла, т. к. объект всегда создается в одном конкретном потоке. Использование ключевого слова synchronized для конструктора приведет к ошибке компиляции.

> ***Предупреждение***
>
> Когда создаете объект, который будет совместно использоваться разными потоками, то будьте очень осторожны, чтобы ссылка на объект не "утекла" раньше времени.

Например, предположим, что вы хотите сделать список `List`, который содержит экземпляры каждого класса. Вы можете захотеть добавить следующую строку в ваш конструктор:

```
engine.addGameObject(this);
```

Но тогда другие потоки смогут использовать engine для получения доступа к объекту до того, как его создание будет завершено.

Синхронизированные методы — простая стратегия для предотвращения вмешательства в поток (thread interference) и ошибок согласованности памяти (memory consistency errors). Если объект видим более чем одному потоку, то все чтения и записи полей объекта должны происходить через синхронизированные методы. Использование синхронизированных методов достаточно просто, но в некоторых случаях недостаточно эффективно; в таких случаях можно использовать более эффективные способы синхронизации, которые будут описаны в последующих разделах.

> ***Важное исключение***
>
> Поля с модификатором final, которые не могут быть изменены после создания экземпляра объекта, могут безопасно читаться из несинхронизированных методов после создания конструктора.

Синхронизированные методы построены на внутренних мониторах (блокировках). Когда поток вызывает синхронизированный метод, то он забирает этот монитор, а когда он выходит из синхронизированного метода, то освобождает его.

Использовать синхронизированные методы нужно с пониманием принципов их работы. Они не имеют никакого механизма защиты от deadlock-ов, которые рассмотрены в *разделе 23.9 "Взаимная блокировка (Deadlock)"*.

23.6. Внутренние мониторы и синхронизация

Каждый объект в Java имеет свой внутренний монитор (блокировку). По соглашению поток, которому требуется эксклюзивный и согласованный доступ к полям объекта, должен получить внутренний монитор объекта перед доступом к ним и освободить его после совершения необходимых действий с ними. Поток владеет монитором объекта между временем получения и временем освобождения блокировки.

Только один поток может держать монитор объекта в одно время. Но поток МОЖЕТ получить монитор, которым он уже владеет. Возможность потоков получать один и тот же монитор несколько раз называется "повторная синхронизация" (reentrant synchronization). Это может быть, например, ситуация, когда синхронизированный код напрямую или ненапрямую вызывает метод, который тоже содержит синхронизированный код, и оба кода используют ту же самую блокировку. Без reentrant synchronization синхронизированному коду пришлось бы использовать много предосторожностей, чтобы исключить блокировку потоком самого себя.

Если вызывается статический синхронизированный метод, то поток получает внутреннюю блокировку объекта `Class`, связанного с этим классом. Таким образом, доступ к статическим полям контролируется другой блокировкой, отличной от блокировки любого из экземпляров класса.

В отличие от синхронизированных методов, синхронизированные инструкции должны указать объект, который предоставляет монитор. Вот так можно переделать класс `SynchronizedCounter` на использование синхронизированных инструкций:

```
public void increment100000() {
 for (int n = 0; n < 100_000; n++) {
   synchronized (this) {
     value++;
   }
 }
}
```

Вместо this в синхронизированных инструкциях можно указывать другой объект, что позволяет брать монитор не над всем `SynchronizedCounter`, например, а создать какой-нибудь специальный внутренний объект для этих целей. Это позволяет брать блокировки над разными объектами внутри синхронизированных инструкций одного объекта.

```
private Object lock1 = new Object();
public void increment100000() {
    for (int n = 0; n < 100_000; n++) {
        synchronized (lock1) {
            value++;
        }
    }
}
```

23.7. Атомарный доступ

В программировании атомарное действие — это действие, которое происходит полностью и сразу. Атомарное действие не может остановиться посередине: оно либо завершается полностью, либо не происходит совсем. Никаких эффектов от атомарного действия не видно снаружи до тех пор, пока действие не завершится.

Даже самые простые выражения могут содержать в себе составные действия, которые могут быть разложены на другие действия. Однако следующие действия атомарны:

- Чтение и запись атомарны для ссылочных переменных и большинства примитивных типов (все типы, кроме `long` и `double`)
- Чтение и запись атомарны для всех переменных, объявленных как volatile (включая `long` и `double`).

Атомарные действия не могут пересекаться, и они могут использоваться без опасений о вмешательстве в поток. Однако это не устраняет все потребности синхронизации атомарных действий, т. к. ошибки консистенции памяти все еще возможны. Использование volatile-переменных уменьшает риск ошибок консистенции памяти, потому что любая запись в volatile-переменную делает связь *происходит-до* (happens-before) для последующих чтений из этой переменной. Это означает, что изменения volatile-переменных всегда видны для других потоков. Это также означает, что, когда поток читает volatile-переменную, он видит не только последнее изменение, но и все побочные эффекты кода, которые приводят к этому изменению.

Если volatile-переменных недостаточно, но использование других методов синхронизации слишком накладно, то можно использовать специальные классы `AtomicInteger`, `AtomicLong` и подобные. Подробнее они рассмотрены в *разделе 23.8 "Атомарные переменные"*.

Использование простого атомарного доступа к переменным более эффективно, чем доступ к этим переменным из синхронизированного кода, но он требует большей внимательности от программиста, чтобы исключить ошибки консистентности памяти.

23.8. Атомарные переменные

В этом разделе мы разберем, как работают `AtomicInteger`, `AtomicLong` и остальные подобные классы из пакета `java.util.concurrent.atomic` в Java.

На самом деле, в этих классах гораздо больше методов, чем просто `get`, `set` и `compareAndSet`.

Сами эти классы наследуются от `java.lang.Number`, а значит, они наследуют и реализуют все его методы: `byteValue`, `shortValue`, `intValue`, `longValue`, `floatValue`, `doubleValue`. Однако нам интересны не они, а методы, специфичные для атомарных классов вроде `AtomicInteger`:

```
int addAndGet(int delta)
```

Атомарное увеличение значения на `delta`. Возвращает обновленное значение.

```
boolean compareAndSet(int expect, int update)
```

Сравнивает текущее значение с expect. Если они равны, то сохраняет update и возвращает `true`. В противном случае возвращает `false`.

`int decrementAndGet()`

Атомарно уменьшает хранящееся значение на единицу. Возвращает новое значение.

`int get()`

Возвращает текущее хранящееся значение.

`int getAndAdd(int delta)`

Атомарно добавляет delta к текущему значению. Возвращает предыдущее значение.

`int getAndDecrement()`

Атомарно уменьшает хранящееся значение на единицу. Возвращает предыдущее значение.

`int getAndIncrement()`

Атомарно увеличивает хранящееся значение на единицу. Возвращает предыдущее значение.

`int getAndSet(int newValue)`

Атомарно устанавливает новое значение и возвращает старое.

`int incrementAndGet()`

Атомарно увеличивает хранящееся значение на единицу и возвращает новое значение.

`void lazySet(int newValue)`

Устанавливает новое значение спустя какое-то время. Это новое значение может быть невидимо для других потоков какое-то время, они будут считывать старое значение. Метод может быть полезен для оптимизации производительности, т. к. последующие обращения к значению вполне могут получить старое значение из кеша процессора без обращения к реальному значению в памяти.

`void set(int newValue)`

Устанавливает новое значение.

`boolean weakCompareAndSet(int expect, int update)`

Аналогично `compareAndSet`, но может вернуть `false` и не обновить значение, даже если хранящееся значение равно expected.

Для `AtomicLong` методы аналогичны, но принимают и возвращают `long` вместо `int`.

Работа методов классов `AtomicInteger` и `AtomicLong` основана на специальной команде процессора CAS (compare-and-set), которая сначала сравнивает значение с ожидаемым и лишь потом заменяет его на новое.

Примечание

Имейте в виду, что у нас нет специальных атомарных классов для `byte`, `short`, `float` и `double`!

23.9. Взаимная блокировка (Deadlock)

Взаимная блокировка (deadlock) описывает ситуацию, когда два или более потока блокируются навсегда, ожидая один другого. Ниже описан пример.

Предположим, что у нас есть система, работающая со счетами пользователей. Счета пользователей представлены классом `AccountWithDeadlock`:

AccountWithDeadlock.java

```
public class AccountWithDeadlock {
    private final int id;
    private int amount;

    public AccountWithDeadlock(int id) {
        this.id = id;
    }

    public int getId() {
        return this.id;
    }

    public int getAmount() {
        return this.amount;
    }

    public synchronized void transfer(
            AccountWithDeadlock fromAccount, int transferSum) {
        synchronized (fromAccount) {
            fromAccount.amount -= transferSum;
            this.amount += transferSum;
        }
    }
}
```

На первый взгляд может показаться, что все в порядке. Но на самом деле в этом коде спрятана взаимная блокировка. Если вызвать `transfer` на первом счете, а затем на втором практически одновременно, то может произойти так, что:

1. Берется блокировка `this` на методе `synchronized` у счета 1.
2. Происходит переключение потоков.
3. Берется блокировка `this` на методе `synchronized` у счета 2.
4. Происходит переключение потоков.

5. Поток, уже имеющий блокировку на счете 1, пытается взять блокировку на счете 2 в синхронизированном блоке, но блокировка `this` у счета 2 уже занята, поэтому поток останавливается до тех пор, пока блокировка счета 2 не освободится.
6. Поток, уже имеющий блокировку на счете 2, пытается взять блокировку на счете 1 в синхронизированном блоке, но блокировка `this` у счета 1 уже занята, поэтому поток останавливается до тех пор, пока блокировка счета 1 не освободится.
7. Оба потока ждут освобождения блокировок друг друга, чего никогда не произойдет.

Как избавиться от подобного? В данном случае можно обратить внимание на идентификатор счета id. Понятное дело, что каждый счет имеет уникальный идентификатор. Чтобы избежать взаимных блокировок (deadlock), мы можем всегда брать блокировки строго в порядке сортировки их идентификаторов, т. е. сначала меньший, а затем больший, чем решим проблему взаимных блокировок для этого случая:

AccountFixed.java

```
public class AccountFixed {
    private final int id;
    private int amount;

    public AccountFixed(int id) {
        this.id = id;
    }

    public int getId() {
        return this.id;
    }

    public int getAmount() {
        return this.amount;
    }

    public void transfer(AccountFixed fromAccount, int transferSum) {
        AccountFixed firstBlock;
        AccountFixed secondBlock;
        if (this.id < fromAccount.id) {
            firstBlock = this;
            secondBlock = fromAccount;
        } else {
            firstBlock = fromAccount;
            secondBlock = this;
        }
```

```
        synchronized (firstBlock) {
            synchronized (secondBlock) {
                fromAccount.amount -= transferSum;
                this.amount += transferSum;
            }
        }

    }
}
```

Если же окажется так, что объекты, на которых нужно взять блокировку, не имеют уникальных полей, по которым можно определять меньший и больший объект, то можно ввести подобные поля синтетически.

23.10. Голодание (starvation)

Голодание (starvation) описывает ситуацию, когда поток не может получить доступ к совместно используемым ресурсам и не может продвинуться в своем выполнении дальше. Это возникает, когда совместно используемый ресурс делается недоступным на долгое время "жадными" потоками. Например, предположим, что объект предоставляет синхронизированный метод, который обычно выполняется достаточно долго. Если один поток вызывает этот метод часто, то другие потоки, которым тоже нужен частый синхронизированный доступ к тому же самому объекту, будут часто блокироваться.

23.11. Активная блокировка (livelock)

Поток часто реагирует на события из другого потока. Если действие другого потока тоже является ответом на событие из еще одного потока, то может произойти активная блокировка (livelock). Как и взаимная блокировка (deadlock), активно заблокированные потоки не могут продвинуться дальше в своем выполнении. Однако эти потоки не заблокированы — они просто слишком заняты, отвечая друг другу, чтобы вернуться к работе. Это можно сравнить с двумя людьми, которые пытаются пройти мимо друг друга в коридоре: Алиса двигается влево, чтобы Боб мог пройти, в это же время Боб двигается вправо, чтобы Алиса могла пройти. Видя, что они все еще блокируют друг друга, Боб двигается влево, а Алиса вправо, но они все еще блокируют друг друга.

23.12. Защищенные блокировки (guarded blocks)

Наиболее часто используемый способ согласования потоков — защищенные блокировки (guarded blocks). Такой блок начинается с выбора условия, которое должно быть `true`, перед тем, как может осуществиться блокировка. Есть несколько шагов, которые нужно выполнить, чтобы осуществить блокировку правильно.

Предположим, что `loadCompleted` — это метод, который не должен выполняться до тех пор, пока ресурсы приложения не будут загружены и разделяемая между пото-

ками переменная `resourcesLoaded` не будет установлена другим потоком. В реальном приложении он мог бы, например, начать обработку загруженных ресурсов, изменение размеров изображений для подгонки под размер монитора, вычисление статического освещения и т. д. Такой метод теоретически должен просто выполнять цикл, пока условие не выполнится, но это было бы расточительно, т. к. выполняется в течение всего времени ожидания.

```
public void loadCompleted() {
    // Простой цикл. Тратит процессорное время
    // не делайте так!
    while(!resourcesLoaded) {}
    System.out.println("Loading completed.");
}
```

Наиболее эффективно использовать `Object.wait()`, чтобы приостановить работу текущего потока. Вызов метода wait не возвращает управление до тех пор, пока другой поток не обработает уведомление о том, что произошло некоторое специальное событие, однако не имеет значения, какое событие ожидает поток:

```
public synchronized void loadCompleted() {
    // Этот цикл выполняется только один раз для каждого специального события,
    // которое может быть событием, которое мы ожидаем.
    while(!resourcesLoaded) try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println("Resources loaded. Processing...");
}
```

> ***Замечание***
>
> Всегда вызывайте `wait` внутри цикла, который проверяет условие, которое ожидается. Не предполагайте, что прерывание было вызвано конкретным условием, которое нам нужно, или что это условие до сих пор выполняется. Вполне может оказаться, что прерывание было вызвано совсем другим событием, а не тем, которое мы ожидали.

Как и многие другие методы, которые приостанавливают выполнение, `wait` может бросить `InterruptedException`. В этом примере мы просто выводим в консоль стектрейс ошибки.

> ***Замечание***
>
> Обратите внимание, что мы вызываем метод `wait` внутри синхронизированного метода, а значит, текущий поток владеет монитором объекта `this`. Это обязательное условие. Вызывайте `wait` только тогда, когда уже владеете монитором объекта.

При вызове метода wait поток освобождает блокировку и приостанавливает выполнение. Затем спустя время другой поток получает ту же самую блокировку и вызывает `Object.notifyAll` или `Object.notify`, сообщая всем ожидающим потокам, что произошло что-то существенное. Метод `notifyAll()` пробуждает все потоки, кото-

рые вызывали `wait` для объекта, а метод `notify()` пробуждает только один случайный поток из них. Мы используем `notifyAll()`:

```
public synchronized resourcesLoaded() {
    resouresLoaded = true;
    notifyAll();
}
```

Спустя какое-то время второй поток освобождает блокировку, первый поток снова получает блокировку и возвращается из вызова `wait`.

Давайте сделаем приложение загрузки и обработки ресурсов более полноценным. Загрузчик будет загружать данные, а обработчик будет что-либо делать с ними. Два потока общаются с помощью общего объекта. Согласование их действий очень важно: поток обработчика не должен пытаться получать данные до того, как загрузчик загрузит их, и поток загрузчика не должен пытаться доставить новые данные до того, пока обработчик не получил предыдущие ресурсы.

ResourceLoadProcessExample.java

```
package ru.urvanov.javaindynamics2022.multithreading;

import java.util.Arrays;

public class ResourceLoadProcessExample {
 // Ресурсы. В нашем случае просто массив байт
 private byte[] resource;
 // True, если ресурсы еще не загружены.
 // False, если ресурсы загружены и готовы к обработке
 private boolean empty = true;

 public synchronized byte[] getLoaded() {
   // Ждем, пока нет загруженного ресурса.
   while (empty) {
     try {
       wait();
     } catch (InterruptedException e) {
       e.printStackTrace();
     }
   }
   // Меняем статус
   empty = true;
   // Пробуждаем загрузчик
   notifyAll();
   return resource;
 }

 public synchronized void loaded(byte[] resource) {
   // Ждем, пока предыдущий ресурс не обработан.
   while (!empty) {
```

```
    try {
      wait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  // Переключаем статус.
  empty = false;
  // Загруженный ресурс
  this.resource = resource;
  // Пробуждаем обработчик ресурсов.
  notifyAll();
}

public static void main(String[] args) throws InterruptedException {
  ResourceLoadProcessExample resourceLoadProcessExample
      = new ResourceLoadProcessExample();

  Thread processor = new Thread(() -> {
    boolean workMore = true;
    while (workMore) {
      byte[] resource = resourceLoadProcessExample.getLoaded();

      if (resource == null) {
        workMore = false;
      } else {
        System.out.println("Processing resource: "
            + Arrays.toString(resource));
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
          workMore = false;
        }
      }
    }
  });

  Thread loader = new Thread(() -> {
    for (int n = 0; n < 10; n++) {
      byte[] loadedResource = new byte[10];
      Arrays.fill(loadedResource, (byte) n);
      resourceLoadProcessExample.loaded(loadedResource);
      try {
        Thread.sleep(n * 1_000);
      } catch (InterruptedException e) {}
    }
    resourceLoadProcessExample.loaded(null);

  });
```

```
        loader.start();
        processor.start();
        loader.join();
        processor.join();

    }
}
```

23.13. Неизменяемые объекты (immutable objects)

Объект считается неизменяемым, если его внутреннее состояние не может быть изменено после создания. Использование неизменяемых объектов — широко распространенная стратегия для создания простого и надежного кода.

Неизменяемые объекты особенно полезны в многопоточных приложениях. Так как они не могут менять своего внутреннего состояния, то они не могут быть испорчены вмешательством в поток (thread interference) или прочитаны в некорректном состоянии.

Ниже перечислены правила определения неизменяемых объектов. Не все классы, документированные как "неизменяемые", следуют этим правилам. Это необязательно значит, что создатели этих классов были некомпетентны — они могли иметь веские причины считать, что экземпляры их объектов никогда не будут меняться после создания. Однако такие стратегии требуют сложного анализа и не подходят для начинающих.

- Все поля должны быть final и private.
- Не должно быть методов установки и изменения значений.
- Не позволяйте расширять класс, объявляя дочерние классы. Самый простой способ добиться этого — объявить класс как `final`. Более сложный способ — это сделать конструктор приватным и создавать экземпляры класса с помощью методов фабрик.
- Если поля экземпляров ссылаются на изменяемые объекты, то не позволяйте менять состояние этих объектов: не предоставляйте методов, которые меняют внутреннее состояние изменяемых объектов, не позволяйте стороннему коду получить ссылки на изменяемые объекты (возвращайте копии этих объектов), не используйте те объекты, которые были переданы в конструктор (создавайте их копии, если нужно).

Пример неизменяемого класса:

ImmutablePoint.java

```
package ru.urvanov.javaindynamics2022.multithreading;

/**
 * Пример неизменяемого класса.
 */
```

```
public final class ImmutablePoint {
    private final double x;
    private final double y;

    public ImmutablePoint(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }
}
```

23.14. Объекты Lock

Синхронизированный код полагается на простой тип reentrant lock (блокировка, которую можно брать несколько раз). Этот тип легко использовать, но он имеет определенные ограничения. Более сложные способы блокировки поддерживаются пакетом `java.util.concurrent.locks`.

Объекты, реализующие интерфейс `Lock`, работают очень похоже на внутренние блокировки, используемые синхронизированным кодом. Так же как и для внутренних блокировок, только один поток может держать блокировку объекта `Lock` в одно время. Объекты `Lock` также поддерживают механизм `wait / notify` через ассоциированные с ними объекты `Condition`.

Преимущество объектов `Lock` над внутренними блокировками в том, что они могут отказаться от участия в попытке приобрести блокировку благодаря методам `tryLock`:

```
boolean tryLock()

boolean tryLock(long time,
 TimeUnit unit)
         throws InterruptedException
```

Метод `tryLock` сразу же завершается, если блокировка недоступна при вызове, либо после истечения указанного времени (если время указано).

```
void lockInterruptibly()
                throws InterruptedException
```

Метод `lockInterruptibly` отказывается от попытки получить блокировку, если другой поток отправляет `interrupt` до получения блокировки.

`void lock()`

Метод `lock` приостанавливает поток до тех пор, пока блокировка будет недоступна. Затем захватывает блокировку и возобновляет поток либо сразу забирает блокировку, если она свободна.

`void unlock()`

Освобождает блокировку.

Основные реализации интерфейса `Lock`:

1. `ReentrantLock`. Работает аналогично внутренней блокировке и синхронизированным методам / синхронизированным блокам инструкций, но с расширенными возможностями.
2. `ReentrantReadWriteLock.ReadLock`, `ReentrantReadWriteLock.WriteLock`. Класс `ReentrantReadWriteLock` имеет два внутренних монитора. Один для чтения, а второй для записи. Блокировка на чтение может держаться несколькими потоками одновременно, но блокировка на запись только одним (много читателей, один писатель).

Пример использования ReentrantLock:

```
import  java.util.concurrent.locks.*;
...
Lock myLock = new ReentrantLock();
try {
    boolean myLockSuccess = myLock.tryLock();
    if (myLockSuccess) {
        // Выполняем действия над объектом,
        // который разделяем с другими потоками.
    } else {
        // Кто-то другой уже работает с объектом.
    }
} finally {
    myLock.unlock();
}
```

23.15. Executors

Интерфейс `java.util.concurrent.Executor` предоставляет один метод execute, который является заменой обычного создания потока. Если `r` реализует интерфейс `Runnable`, а `e` реализует интерфейс `Executor`, то вы можете заменить

```
(new Thread(r)).start();
```

следующим кодом:

```
e.execute(r);
```

Однако определение метода `execute` несколько отличается. В зависимости от реализации `Executor`-а метод execute может делать то же самое, но обычно он использует уже существующий рабочий поток для запуска `r`, либо `r` помещается в очередь, где дожидается освобождения рабочего потока.

Реализации `executor`-ов в `java.util.concurrent` созданы для использования с более продвинутыми интерфейсами `ExecutorService` и `ScheduledExecutorService`, но они также работают и с интерфейсом `Executor`.

Интерфейс `java.util.concurrent.ExecutorService` расширяет интерфейс `Executor`, добавляя множество новых методов. Основной метод — `submit`, который принимает как `Runnable`, так и интерфейс `java.util.concurrent.Callable<V>` с единственным методом `V call()`, который позволяет заданиям возвращать значение. Метод `submit` возвращает интерфейс `java.util.concurrent.Future`, который используется для получения результата и контролирования состояния потока.

Интерфейс `Future` позволяет проверять, не закончилось ли выполнение фоновой задачи с помощью метода:

```
boolean isDone();
```

Метод `isDone` возвращает `true`, если поток завершил свои вычисления либо был отменен с помощью метода `cancel` интерфейса `Future`:

```
boolean cancel(boolean mayInterruptIfRunning);
```

Если задача была отменена, то метод `isCancelled` будет возвращать `true`:

```
boolean isCancelled();
```

Получить значение из завершенной задачи можно с помощью метода `get`:

```
V get() throws InterruptedException, ExecutionException;
```

Метод `get` ожидает завершения выполнения и возвращает результат, который вернул метод `call` интерфейса `Callable`. Существует также вариант метода `get` с лимитом времени ожидания:

```
V get(long timeout, TimeUnit unit) throws InterruptedException,
ExecutionException, TimeoutException;
```

Интерфейс `java.util.concurrent.ScheduledExecutorService` расширяет интерфейс `java.util.concurrent.ExecutorService` и добавляет методы `schedule*`, которые позволяют запланировать выполнение задания.

23.16. CompletableFuture

Класс `CompletableFuture` появился в Java 8. Он реализует интерфейс `Future` и позволяет комбинировать задачи друг с другом. `CompletableFuture` содержит специальные статические методы для запуска своих экземпляров. Некоторые из них позволяют передавать `Executor`, но некоторые не имеют такого параметра, и тогда используется `ForkJoinPool.commonPool()`.

Пример простого использования:

SimpleCompletableFuture.java

```
package ru.urvanov.javaindynamics2022.multithreading;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;

public class SimpleCompletableFuture {
    public static void main(String[] args) throws ExecutionException,
                                                   InterruptedException {
        CompletableFuture<String> future1
                = CompletableFuture.supplyAsync(() -> "Test1");
        CompletableFuture<String> future2
                = CompletableFuture.supplyAsync(
                        () -> "Test2", Executors.newCachedThreadPool());
        System.out.println(future1.get());
        System.out.println(future2.get());

        CompletableFuture<Void> future3 = CompletableFuture.runAsync(
                () -> System.out.println("Test3"));
        CompletableFuture<Void> future4 = CompletableFuture.runAsync(
                () -> System.out.println("Test4"),
                Executors.newCachedThreadPool());
        future3.join();
        future4.join();
    }
}
```

Метод `get` блокирует текущий поток. `CompletableFuture` чаще всего используется с методами, позволяющими задать callback-методы, которые будут что-то делать с вычисленным значением также в параллельном потоке. Для этого используются методы `thenApply`, `thenRun` и аналогичные:

SimpleCompletableFuture.java

```
// thenAccept
CompletableFuture<String> completableFutureThenAccept
        = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<Void> futureThenAccept
        = completableFutureThenAccept.thenAccept(
                s -> System.out.println("Computation returned: " + s));

futureThenAccept.get();
```

```
// thenRun
CompletableFuture<String> completableFutureThenRun
        = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<Void> futureThenRun = completableFutureThenRun
        .thenRun(() -> System.out.println("Computation finished."));

futureThenRun.get();
```

Можно выстраивать последовательные цепочки обработки, но использовать придется не `thenRun` и `thenAccept`, а `thenApply`, который принимает `Function` в качестве параметра:

```
// thenApply
CompletableFuture<Integer> futureThenApply
        = CompletableFuture.supplyAsync(() -> 2);

futureThenApply.thenApply(result -> result + 3);

futureThenApply.thenApply(result -> result -1);

Integer futureThenApplyResult = futureThenApply.get();
System.out.println(futureThenApplyResult);
```

Основная мощь `CompletableFuture` заключается в комбинировании задач с помощью `allOf` и `anyOf`:

SimpleCompletableFuture.java

```
// allOf
List<CompletableFuture<Void>> allOf = new ArrayList<>();
allOf.add(CompletableFuture.runAsync(() -> {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("allOfTaskOne");
}));
allOf.add(CompletableFuture.runAsync(() -> {
    try {
        Thread.sleep(200);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("allOfTaskTwo");
}));
```

```
CompletableFuture.allOf(
        allOf.toArray(new CompletableFuture[allOf.size()])).join();

// anyOf
List<CompletableFuture<Void>> anyOf = new ArrayList<>();
anyOf.add(CompletableFuture.runAsync(() -> {
    try {
        Thread.sleep(120);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("anyOfTaskOne");
}));
anyOf.add(CompletableFuture.runAsync(() -> {
    try {
        Thread.sleep(200);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("anyOfTaskTwo");
}));
CompletableFuture.allOf(
        anyOf.toArray(new CompletableFuture[anyOf.size()])).join();
```

Обратите внимание, что в этом примере мы использовали `join` вместо `get` для ожидания завершения задач. Метод `join` делает то же самое, что и метод `get`, но не требует обработки проверяемых исключений.

23.17. Пулы потоков

Многие реализации executor-ов из пакета java.util.concurrent используют пул потоков. Пул потоков — это некий контейнер, содержащий в себе определенное число потоков и использующий их для выполнения заданий. Этот подход минимизирует издержки создания потоков. Они используют много памяти, и в больших приложениях создание и уничтожение большого количества потоков значительно увеличивает потребление памяти.

Пулы потоков бывают с фиксированным количеством потоков, с одним потоком, с кешем потоков и т. д. Наиболее часто использующийся тип пула потоков — это фиксированный пул потоков. Этот тип всегда держит работающими указанное количество потоков, даже если задач больше или меньше их числа. Задачи отправляются в пул через внутреннюю очередь, которая хранит дополнительные задачи, если их больше, чем потоков.

Важное преимущество фиксированного пула потоков в том, что приложения, которые их используют, не подвисают из-за того, что они создали слишком большое количество потоков, превышающее возможности системы.

Для создания пула с фиксированным числом потоков используется метод `newFixedThreadPool` класса `java.util.concurrent.Executors`, для остальных типов используются соответствующие другие методы.

```
public static ExecutorService newFixedThreadPool(int nThreads)
```

Класс `Executors` также имеет следующие фабричные методы:

```
public static ExecutorService newCachedThreadPool()
```

Создает новые потоки по мере надобности. Повторно использует предыдущие потоки, если они свободны.

```
public static ExecutorService newSingleThreadExecutor()
```

Пул потоков, состоящий из одного потока.

Если ни один из стандартных `executor`-ов не удовлетворяет вашим потребностям, то вы можете создать экземпляры `java.util.concurrent.ThreadPoolExecutor` или `java.util.concurrent.ScheduleThreadPoolExecutor`.

23.18. Fork/Join Framework

Fork/Join Framework является реализацией интерфейса `ExecutorService`, который помогает вам получить преимущество при использовании мультипроцессорной системы. Он спроектирован для такой работы, которая может быть разбита рекурсивно на множество маленьких частей. Цель фреймворка — использовать всю доступную мощь, чтобы увеличить производительность вашего приложения.

Как и с любой реализацией `ExecutorService`, fork/join framework распределяет задачи между рабочими потоками в пуле потоков. Fork/join фрейморк отличается тем, что он использует алгоритм воровства работы. Рабочие потоки, для которых кончилась работа, могут воровать задачи других потоков, которые все еще заняты.

Основным классом fork/join фреймворка является `java.util.concurrent.ForkJoinPool`.

Fork/join framework работает в тех случаях, когда работу можно разбить на небольшие части, которые можно выполнить отдельно.

Пример:

ForkAbs.java

```
package ru.urvanov.javaindynamics2022.multithreading;

import java.util.Random;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class ForkAbs extends RecursiveAction {
    private int[] source;
```

```
    private int[] destination;
    private int start;
    private int length;

    public ForkAbs(
            int[] source,
            int start,
            int length,
            int[] destination) {
        this.source = source;
        this.start = start;
        this.length = length;
        this.destination = destination;
    }

    private void computeDirectly() {
        // Это только пример. В реальности здесь
        // могут быть довольно сложные вычисления.
        for (int n = start; n < start + length; n++) {
            destination[n] = Math.abs(source[n]);
        }
    }

...
```

Теперь мы реализуем абстрактный метод `compute()`, который либо реализует вычисление абсолютного значения напрямую, либо делит его на мелкие задачи.

ForkAbs.java

```
private static int threshold = 10;

@Override
protected void compute() {
    if (length < threshold) {
        computeDirectly();
        return;
    }

    int split = length / 2;
    invokeAll(new ForkAbs(source, start, split, destination),
            new ForkAbs(
                    source,
                    start + split,
                    length - split,
                    destination));
}
```

Пример запуска:

ForkAbs.java

```
int[] source = new int[100];
int[] destination = new int[source.length];
Random random = new Random();
System.out.println("source = ");
for (int n = 0; n < source.length; n ++) {
 source[n] = random.nextInt();
 System.out.print(source[n]);
 System.out.print(" ");
}
System.out.println();

ForkAbs forkAbs = new ForkAbs(source, 0, source.length, destination);

ForkJoinPool pool = new ForkJoinPool();
pool.invoke(forkAbs);

System.out.println("destination = ");
for (int n = 0; n < destination.length; n ++) {
 System.out.print(destination[n]);
 System.out.print(" ");
}
System.out.println();
```

23.19. Java Memory Model

Модель памяти Java или Java Memory Model (JMM) описывает поведение программы в многопоточной среде. Она объясняет возможное поведение потоков и то, на что должен опираться программист, разрабатывающий приложение.

В этом разделе приведено достаточно большое количество терминов. Думаю, что большая часть из них пригодится вам только на собеседованиях, но представлять общую картину того, что такое Java Memory Model, все-таки полезно.

Java может работать на разных процессорах и разных операционных системах, что приводит к затруднению синхронизации между потоками. Многие современные процессоры имеют несколько ядер, могут выполнять команды не в той последовательности, в которой они записаны, а компиляторы могут менять последовательность команд для оптимизации.

Неправильно синхронизированные программы могут приводить к неожиданным результатам.

Например, программа использует локальные переменные r1 и r2 и общие переменные `A` и `B`. Первоначально `A == B == 0` (табл. 23.1).

Таблица 23.1. *Соответствие локальных и общих переменных*

Thread 1	Thread 2
1: r2 = A;	3: r1 = B;
2: B = 1;	4: A = 2;

Может показаться, что результат `r2 == 2` и `r1 == 1` невозможен, т. к. первой должна быть либо инструкция 1, либо инструкция 3. Если инструкция 1 будет первой, то она не сможет увидеть число 2, записанное в инструкции 4. Если инструкция 3 будет первой, то она не сможет увидеть результат инструкции 2.

Если какое-то выполнение программы привело бы к такому поведению, то мы бы знали, что инструкция 4 была до инструкции 1, которая была до инструкции 2, которая была до инструкции 3, которая была до инструкции 4, что совершенно абсурдно.

Однако современным компиляторам разрешено переставлять местами инструкции в обоих потоках в тех случаях, когда это не затрагивает исполнение этого потока, не учитывая другие потоки. Если инструкция 1 и инструкция 2 поменяются местами, то мы с легкостью сможем получить результат `r2 == 2` и `r1 == 1` (табл. 23.2).

Таблица 23.2. *Перемена инструкций*

Thread 1	Thread 2
B = 1;	r1 = B;
r2 = A;	A = 2;

Для некоторых программистов подобное поведение может оказаться ошибочным, но здесь нужно сделать замечание, что этот код неверно синхронизирован.

- ❒ у нас есть запись из одного потока;
- ❒ мы читаем ту же переменную из другого потока;
- ❒ чтение и запись не синхронизированы, что не гарантирует правильный порядок.

Ситуация, описанная в примере выше, называется data race.

Ошибка проектирования, когда data race приводит к ошибкам в программе, называется race condition.

Переставлять команды может Just-In-Time компилятор или процессор. Более того, каждое ядро процессора может иметь свой кеш. А значит, у каждого процессора может быть свое значение одной и той же переменной, что может привести к аналогичным результатам.

Модель памяти описывает, какие значения могут быть считаны в каждый момент программы. Поведение потока в изоляции должно быть таким, каким оно описано в самом потоке, но значения, считываемые из переменных, определяются моделью

памяти. Когда мы ссылаемся на это, то мы говорим, что программа подчиняется intra-thread semantic, т. е. семантике однопоточного приложения.

Память, которая может быть совместно использована разными потоками, называется "куча" (shared memory или heap memory).

Все переменные экземпляров, статические поля, массивы элементов хранятся в куче. Дальше в этой главе они будут называться просто переменными.

Локальные переменные, параметры конструкторов и методов, а также параметры блока `catch` никогда не разделяются между потоками.

Два доступа к одной переменной называются конфликтующими, если хотя бы один из них меняет значение переменной (другой может как менять, так и считывать текущее значение).

Inter-thread action (наиболее корректный, на мой взгляд, перевод — "межпоточное действие") — это действие внутри одного потока, которое может повлиять на другой поток или быть замечено им. Существует несколько типов межпоточных действий:

- Чтение (нормальное, не volatile). Чтение переменной.
- Запись (нормальная, не volatile). Запись переменной.
- volatile read. Чтение volatile-переменной.
- volatile write. Запись volatile-переменной.
- Lock. Взятие блокировки монитора.
- Unlock. Освобождение блокировки монитора.
- Синтетические действия — первое и последнее действия в потоке.
- Действия по запуску нового потока или обнаружению остановки потока.
- Внешние действия. Это действия, которые могут быть обнаружены снаружи выполняющегося потока, например взаимодействия с внешним окружением.
- Thread divergence actions. Действия потока, находящегося в бесконечном цикле без синхронизаций, работы с памятью или внешних действий.

Program order (лучше не переводить, чтобы не возникло путаницы) — общий порядок потока, выполняющего действия, который отражает порядок, в котором должны быть выполнены все действия с соответствии с семантикой intra-thread semantic потока.

Действия называются sequentially consistent (тоже лучше не переводить), если все действия выполняются в общем порядке, который соответствует program order, а также каждое чтение переменной видит последнее значение, записанное туда до этого в соответствии с порядком выполнения.

Если в программе нет состояния гонки, то все запуски программы будут sequentially consistent.

Synchronization order (порядок синхронизации, но лучше не переводить) — общий порядок всех действий по синхронизации в выполнении программы.

Действия по синхронизации вводят связь synchronized-with (синхронизировано с):

- Действие освобождения блокировки монитора синхронизируется с (synchronizes-with) всеми последующими действиями по взятию блокировки этого монитора.
- Присвоение значения volatile переменной синхронизируется с (synchronizes-with) всеми последующими чтениями этой переменной любым потоком.
- Действие запуска потока синхронизируется с (synchronizes-with) первым действием внутри запущенного потока.
- Присвоение значения по умолчанию (0, `false`, `null`) каждой переменной синхронизируется с (synchronizes-with) первым действием каждого потока.
- Последнее действие в потоке синхронизируется с (synchronizes-with) любым действием других потоков, которые проверяют, что первый поток завершился.
- Если поток 1 прерывает поток 2, то прерывание выполнения потока 2 синхронизируется с (synchronizes-with) любой точкой, где другой поток (и прерывающий тоже) проверяет, что поток 2 был прерван (`InterruptedException`, `Thread.interrupted`, `Thread.isInterrupted`).

Happens-before ("выполняется прежде" или "произошло-до") — отношение порядка между атомарными командами. Оно означает, что вторая команда будет видеть изменения первой и что первая команда выполнилась перед второй. Happens-before возникает:

- Освобождение монитора happens-before любого последующего взятия блокировки этого монитора.
- Присвоение значение volatile полю happens-before любого последующего чтения значения этого поля.
- Запуск потока happens-before любых действий в запущенном потоке.
- Все действия внутри потока happens-before любого успешного завершения `join()` над этим потоком.
- Инициализация по умолчанию для любого объекта happens-before любых других действий программы.

Все поля `final` должны быть инициализированы либо конструкциями инициализации, либо внутри конструктора. Не стоит внутри конструкторов обращаться к другим потокам. Поток увидит ссылку на объект только после полной инициализации, т. е. по окончании работы конструктора. Так как `final` полям присваивается значение только один раз, то просто не обращайтесь к другим потокам внутри конструкторов и блоков инициализации, и проблем возникнуть не должно.

Однако `final` поля могут быть изменены через Java Reflection API, чем пользуются, например, десериализаторы. Просто не отдавайте ссылку на объект другим потокам и не читайте значение `final` поля до его обновления, и все будет нормально.

Некоторые процессоры не позволяют записывать один байт в ОЗУ, что приводит к проблеме, называемой word tearing. Представьте, что у нас есть массив байт.

Один поток записывает первый байт, а второй поток пытается записать значение в рядом стоящий байт. Но если процессор не может записать один байт, а только целое машинное слово, то запись рядом стоящего байта может быть проблематичной. Если просто считать машинное слово, обновить один байт и записать обратно, то мы помешаем другому потоку.

В JVM нет проблемы word tearing. Два потока, пишущие рядом стоящие байты, не должны мешать друг другу.

23.20. Задания

1. Создайте программу, которая может в одном потоке считывать данные из текстового файла, а во втором потоке — расставлять в нем переводы строк, чтобы не было строк длиннее 80 символов. Переводы строк можно ставить только между словами. Дополнительно пусть второй поток добавляет пробелы между словами в строке, чтобы она полностью умещалась в 80 символов, а не была выровненной по левому краю. Файл может быть очень большой. Считывать полностью в память нельзя, только небольшими кусками.
2. Создайте программу, аналогичную программе из пункта 1, но с использованием fork / join framework. В этой задаче для упрощения можно считывать файл в память целиком.
3. Создайте программу, которая следит за появлением файлов в каталоге in, а затем проводит с ними процедуру, аналогичную процедуре из задания 1, после чего перекладывает его в каталог out. Используйте фиксированный пул потоков.

ГЛАВА 24

Настройки и окружение

24.1. Properties

`java.util.Properties` — это специальный класс для хранения настроек в виде ключ-значение. И ключ, и значение являются строками. Класс предоставляет методы для:

- Загрузки пар ключ-значение из потока.
- Получения значения по ключу.

- Получения списка всех ключей и значений.
- Прохода по всем ключам.
- Сохранения настроек в поток.

Класс `java.util.Properties` расширяет класс `java.util.Hashtable`. Некоторые методы, которые он унаследовал от него, поддерживают следующие действия:

- Проверка существования указанного ключа или значения в объекте `Properties`.
- Получение количества пар ключ-значение в нашем объекте `Properties`.
- Удаление ключа и его значения.
- Добавление ключа со значением.
- Проход по ключам или значениям.
- Получение значения по ключу.
- Проверка Properties на пустоту.

Пример загрузки настроек из файла:

```
. . .
// Загрузка настроек
Properties props = new Properties();
FileInputStream in = new FileInputStream("application.properties");
props.load(in);
in.close();

. . .
```

Пример сохранения настроек в файл:

```
FileOutputStream out = new FileOutputStream("application.properties");
props.store(out, "---No Comment---");
out.close();
```

Полезные методы класса `java.util.Properties`:

```
public boolean contains(Object value)
```

Возвращает `true`, если значение value присутствует в `Properties`.

```
public boolean containsKey(Object key)
```

Возвращает `true`, если ключ `key` присутствует в `Properties`.

```
public String getProperty(String key)
```

Возвращает значение по ключу. Если такой ключ не найден в списке свойств и в списке свойств по умолчанию, то возвращается `null`.

```
public String getProperty(String key,
                          String defaultValue)
```

Возвращает значение по ключу. Если такой ключ не найден в списке свойств и в списке свойств по умолчанию, то возвращается `defaultValue`.

`public void list(PrintStream out)`

Пишет список настроек в `PrintStream`. Полезно для отладки.

`public void list(PrintWriter out)`

Пишет список настроек в `PrintWriter`. Полезно для отладки.

`public Enumeration<V> elements()`

Возвращает объект для последовательного перебора элементов.

`public Enumeration<K> keys()`

Возвращает объект для последовательного перебора ключей.

`public Enumeration<?> propertyNames()`

Возвращает объект для перечисления всех ключей настроек, включая настройки из списка настроек по умолчанию.

`public V remove(Object key)`

Удаляет пару ключ-значение из списка настроек.

`public Set<String> stringPropertyNames()`

Возвращает множество всех ключей настроек, включая настройки из списка настроек по умолчанию.

`public int size()`

Возвращает количество элементов в `Properties`.

`public Object setProperty(String key, String value)`

Устанавливает/добавляет значение для ключа.

Класс `java.lang.System` предоставляет объект `java.util.Properties`, который определяет конфигурацию текущей среды. Их можно получить с помощью метода `System.getProperty` или `System.getProperties`. Пример:

```
String separator = System.getProperty("path.separator");
java.util.Properties props = System.getProperties();
```

С помощью метода `setProperty` можно поменять значение какой-либо из настроек, но это крайне не рекомендуется. Большая часть значений приведена только в информационных целях. Она не перечитывается после инициализации и не сохраняется для последующих запусков при изменении.

Список основных настроек см. в табл. 24.1.

Таблица 24.1. *Основные настройки*

Key	Meaning
`"file.separator"`	Символ, который разделяет компоненты пути файлов в системе. Это "/" в UNIX и "\" в Windows
`"java.class.path"`	Путь, который используется для поиска каталогов и JAR-архивов, содержащих файлы *.class*. Элементы пути разделяются специфичным для платформы символом, указанным в свойстве path.separator
`"java.home"`	Каталог, в который установлена Java Runtime Environment (JRE)
`"java.vendor"`	Производитель JRE
`"java.vendor.url"`	Адрес сайта производителя JRE
`"java.version"`	Версия JRE
`"line.separator"`	Последовательность символов, используемая операционной системой в качестве разделителя строк в текстовых файлах
`"os.arch"`	Архитектура операционной системы
`"os.name"`	Название операционной системы
`"os.version"`	Версия операционной системы
`"path.separator"`	Символ разделителя пути, используемый в *java.class.path*
`"user.dir"`	Рабочий каталог пользователя
`"user.home"`	Домашний каталог пользователя
`"user.name"`	Имя аккаунта пользователя

24.2. Аргументы командной строки

Приложение на Java может принимать любое количество аргументов из командной строки. Это позволяет пользователю указать необходимую конфигурационную информацию при запуске приложения.

Пользователь вводит аргументы командной строки при вызове приложения и указывает их после имени запускаемого класса.

```
java HelloWorld cat dog
```

Приложение получает аргументы командной строки в массиве строк метода main.

```
public static void main(String[] args)
```

В реальных приложениях для работы с аргументами командной строки используется библиотека Apache Commons CLI, как это показано в статье **https://urvanov.ru/2019/06/08/apache-commons-cli/**. Здесь же приведен базовый пример, как это можно сделать средствами самой Java.

24.3. Переменные окружения

Многие операционные системы используют переменные окружения для передачи информации в приложения.

Переменные окружения представляют собой пары ключ-значение, где ключ и значение — строки. Установка и использование переменных окружения значительно различаются в операционных системах и не будут рассматриваться здесь.

С помощью метода System.getenv приложение может получить переменные окружения. Метод без аргумента возвращает экземпляр `java.util.Map`, который можно использовать только для чтения, но нельзя изменять.

Пример:

PrintEnvMap.java

```
package ru.urvanov.javaindynamics2022.environment;

import java.util.Map;

public class PrintEnvMap {
    public static void main (String[] args) {
        Map<String, String> env = System.getenv();
        for (String name : env.keySet()) {
            System.out.println(name + "="  + env.get(name));
        }
    }
}
```

С аргументом типа `String` метод getenv возвращает значение указанной переменной окружения. Если переменная не указана, то возвращается `null`. Пример:

PrintEnv.java

```
package ru.urvanov.javaindynamics2022.environment;

public class PrintEnv {
 public static void main (String[] args) {
   for (String env: args) {
     String value = System.getenv(env);
     if (value != null) {
       System.out.println(env + "=" + value);
     } else {
       System.out.println(env + " is empty");
     }
   }
 }
}
```

При использовании объекта `java.lang.ProcessBuilder` для создания нового процесса множество переменных окружения, передающихся в новый процесс, такое же, что получено из процессов виртуальной машины Java. Программа может поменять это множество с помощью `ProcessBuilder`.environment.

24.4. Методы класса System

```
public static long currentTimeMillis()
```

Возвращает текущее время в миллисекундах с 1 января 1970 года.

```
public static void exit(int status)
```

Аналогично `Runtime.getRuntime().exit(status)`. Завершает работу виртуальной машины Java с указанным статусом. По соглашению 0 означает успешную работу, статус, отличный от нуля, — ошибку.

24.5. Переменная CLASSPATH

Переменная окружения CLASSPATH указывает на пути, в которых будут искаться классы пользователя. Лучший способ указать путь к классам — это использовать аргумент командной строки -cp, что позволяет указать CLASSPATH индивидуально для каждого приложения.

Значение по умолчанию "." означает, что поиск будет происходить только в текущем каталоге. Указание переменной CLASSPATH или аргумента командной строки -cp переопределяет это значение.

24.6. Задания

1. Напишите программу, которая считывала бы из командной строки название файла с настройками, а потом считывала бы настройки из него в объект Properties.
2. Представьте, что у вас есть метод, который выполняется очень долго. С помощью какого метода из класса `System` можно замерить время его выполнения?
3. Выведите в консоль CLASSPATH запущенного приложения.

ГЛАВА 25

Регулярные выражения

25.1. Теория

В Java классы, связанные с регулярными выражениями, находятся в пакете java.util.regex. Три самых основных класса:

- `java.util.regex.Pattern` — скомпилированное представление регулярного выражения. Не имеет публичных конструкторов, для создания нужно использовать один из его фабричных методов `compile`.
- `java.util.regex.Matcher` — интерпретирует шаблон регулярного выражения и осуществляет сравнение с исходной строкой. У него нет публичных конструкторов, для создания нужно использовать метод `matcher` класса `java.util.regex.Pattern`.
- `java.util.regex.PatternSyntaxException` — непроверяемое исключение, которое возникает при наличии синтаксической ошибки в регулярном выражении.

Сами регулярные выражения — это довольно сложная и обширная тема, но они редко используются в полную силу. В этой главе будет дано общее представление об использовании регулярных выражений, которого будет более чем достаточно в большинстве случаев.

В простейшем случае проверка строк по регулярному выражению происходит следующим образом:

SimpleRegex.java

```
package ru.urvanov.javaindynamics2022.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

// Простой пример использования регулярного выражения
public class SimpleRegex {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("\\d\\d [a-zA-Z]{2}");
        Matcher matcher1 = pattern.matcher("12 df");
        if (matcher1.matches()) {
            System.out.println("first matches");
        }
```

```
        Matcher matcher2 = pattern.matcher("89 33");
        if (matcher2.matches()) {
            System.out.println("second matches");
        }
    }
}
```

При запуске эта программа выведет в консоль:

```
first matches
```

В вышеприведенной программе мы сначала создаем объект `Pattern`, который содержит скомпилированное регулярное выражение, а затем проверяем строки на совпадение с ним, получая объекты `Matcher` для каждой строки.

Основной интерес представляет строка в вызове метода `Pattern.compile`, которая содержит само регулярное выражение.

Сначала мы используем предопределенный класс символов `\d`, который означает любую цифру [0–9]. Обратную косую черту приходится указывать два раза, чтобы в регулярное выражение попала одна косая черта, т. к. символ обратной косой черты имеет специальное значение в строках Java, что может запутать новичков.

Существуют и другие предопределенные классы символов:

- `.` — любой символ.
- `\d` — цифра [0–9].
- `\D` — не цифра [^0–9].
- `\s` — пробельный символ `[ \t\n\x0B\f\r]`.
- `\S` — непробельный символ.
- `\w` — любая буква английского алфавита, символ подчеркивания или цифра.
- `\W` — несловарный символ (не совпадающий с `\w` символ).

Затем регулярное выражение содержит пробел, после него мы объявляем свой класс символов, который может совпадать с любой буквой английского алфавита от a до z и от A до Z. Можно перечислять конкретные символы, а не диапазоны, например `[duh]` будет соответствовать любому символу из `d`, `u` или `h`.

Число 2 в фигурных скобках указывает, что предыдущий класс символов должен содержаться в результирующей строке 2 раза. Можно писать ограничения:

- `{2,}` — больше двух.
- `{2,5}` — от двух до пяти.

25.2. Задания

1. Создайте класс, с помощью которого можно бы было проверять формат номера телефона России.

2. Создайте класс для проверки e-mail (ограничьтесь только английскими буквами и символом "собака", сам формат e-mail очень сложен и проверке регулярными выражениями практически не поддается).
3. Создайте класс для проверки формата СНИЛС (формат "ччч-ччч-ччч чч").
4. Создайте класс для проверки правильности серии и номера паспорта.
5. Создайте класс для проверки правильности ИНН человека (последовательность из 12 цифр).

ГЛАВА 26

Коллекции

26.1. Введение

Все коллекции в Java реализуют какой-нибудь основной интерфейс.

Списки реализуют интерфейс `java.util.List`, множества реализуют интерфейс `java.util.Set` или `java.util.SortedSet` и т. д.

На моем сайте **https://urvanov.ru** есть диаграмма интерфейсов для Java Collections Framework, которая может облегчить вам понимание этой главы.

26.2. Интерфейс Collection

Интерфейс `java.util.Collection` содержит наиболее общие свойства всех коллекций. Он используется в качестве параметра в тех методах, где нужно принимать любой тип коллекций. Например, по соглашению все коллекции имеют конструктор, который принимает интерфейс `Collection` в качестве параметра. Такой конструктор инициализирует новую коллекцию всеми элементами из указанной коллекции, т. е. позволяет вам конвертировать один тип коллекций в другой.

Предположим, что у вас есть `Collection<String> c`, который может быть `List`, `Set` или другим типом `Collection`. Следующий код создает новый экземпляр `ArrayList` (реализацию интерфейса `List`), который содержит все элементы из `c`:

```
List<String> list = new ArrayList<>(c);
```

Интерфейс `java.util.Collection` содержит методы для осуществления базовых операций:

`int size()`

Возвращает количество элементов в коллекции.

`boolean isEmpty()`

Возвращает `true`, если в коллекции нет ни одного элемента.

`boolean contains(Object element)`

Возвращает `true`, если коллекция содержит элемент `element`.

`boolean add(E element)`

Добавляет элемент в коллекцию. Возвращает `true`, если элемент был добавлен. Возвращает `false`, если коллекция не может содержать дублирующихся элементов и такой элемент уже в ней есть.

`boolean remove(Object element)`

Удаляет одно вхождение element из коллекции. Возвращает `true`, если элемент был удален, `false` — коллекция не была изменена.

`Iterator<E> iterator()`

Возвращает итератор, позволяющий пройтись по элементам коллекции и удалить некоторые элементы. Iterator представляет собой вот такой интерфейс:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Метод `hasNext()` возвращает `true`, если остались еще элементы. Метод `next()` возвращает следующий элемент. Метод `remove()` позволяет удалить последний возвращенный методом `next()` элемент. Внимание! Во время использования итератора нельзя изменять коллекцию любым другим способом, кроме метода `remove()` этого итератора, иначе возникнет исключение.

Также есть методы, которые оперируют целыми коллекциями:

`boolean containsAll(Collection<?> c)`

Возвращает `true`, если коллекция содержит все элементы из `c`.

`boolean addAll(Collection<? extends E> c)`

Добавляет все элементы из `c` в коллекцию.

`boolean removeAll(Collection<?> c)`

Удаляет из коллекции все элементы, которые присутствуют в `c`.

```
boolean retainAll(Collection<?> c)
```

Оставляет в коллекции только те элементы, которые также присутствуют в коллекции `c`, удаляя все остальные.

```
void clear()
```

Удаляет все элементы из коллекции.

Есть также методы, позволяющие преобразовать коллекцию в массив:

```
Object[] toArray()
<T> T[] toArray(T[] a)
```

Рекомендуется использовать второй метод, т. к. он позволяет преобразовать в массив определенного типа. Пример: `String[] array = collection.toArray(new String[0]);`.

Начиная с JDK 8, интерфейс `Collection` также поставляет методы:

```
Stream<E> stream()
Stream<E> parallelStream()
```

которые используются для получения последовательных потоков и параллельных потоков из коллекции.

Для прохода по элементам коллекции можно использовать `for-each`:

```
for (Object o : collection)
    System.out.println(o);
```

26.3. Интерфейс Set

`java.util.Set` — это коллекция уникальных элементов. Интерфейс `Set` содержит только методы, унаследованные от `java.util.Collection`. `Set` накладывает более строгие соглашения на методы `equals` и `hashCode`, что позволяет сравнивать экземпляры `Set`, даже если они различных реализаций. Два экземпляра `Set` равны, если у них одинаковое количество элементов и они содержат одинаковые элементы.

Платформа Java предоставляет три реализации интерфейса `java.util.Set` общего назначения:

- `java.util.HashSet`.
- `java.util.TreeSet`.
- `java.util.LinkedHashSet`.

`HashSet` — реализация на основе хеш-таблицы. Это наиболее эффективная реализация, но она не гарантирует сохранение порядка элементов при обходе.

`TreeSet` — реализация на основе красно-черных деревьев. Она упорядочивает элементы в соответствии с их значениями, но работает значительно медленнее `HashSet`.

`LinkedHashSet` — реализация на основе хеш-таблицы, но дополнительно пролинкованная связанным списком. Эта реализация избавляет от хаотичного порядка элементов и лишь незначительно хуже `HashSet`-а по эффективности.

Также есть две реализации java.util.Set специального назначения: java.util.EnumSet и java.util.concurrent.CopyOnWriteArraySet. Реализация java.util.EnumSet предназначена специально для перечислений, всегда используйте ее, когда нужно создать Set, содержащий значения одного и того же перечисления. EnumSet внутренне представляет собой битовый вектор, он очень компактный и быстрый. Реализация CopyOnWriteArraySet хранит свои элементы в массиве и создает новый массив при выполнении любой операции добавления, удаления или замены элемента. Она потокобезопасна, при изменении CopyOnWriteArraySet все созданные до этого итераторы остаются в рабочем состоянии. CopyOnWriteArraySet подходит для множеств, которые редко меняются, но по которым часто проходит операция итерации.

С помощью Set можно удалить из любой коллекции дублирующиеся элементы. Например, мы хотим удалить дублирующиеся элементы из коллекции c:

```
Collection<Type> noDups = new HashSet<Type>(c);
```

26.4. Интерфейс List

Интерфейс java.util.List представляет собой упорядоченную коллекцию элементов.

Java содержит два класса, реализующих интерфейсы List: java.util.ArrayList и java.util.LinkedList.

Есть реализация специального назначения java.util.concurrent.CopyOnWriteArrayList. Она работает аналогично CopyOnWriteArraySet, т. е. для каждой операции добавления, замены или удаления элемента создается новый массив, хранящий элементы списка. Итерация по CopyOnWriteArrayList никогда не приводит к java.util.ConcurrentModificationException. Есть еще синхронизированная реализация java.util.Vector. Существует также класс java.util.Stack, расширяющий класс Vector. Класс Stack можно использовать в качестве стека, но рекомендуется использовать интерфейс Deque и его реализацию ArrayDeque.

Методы интерфейса java.util.Collection работают так, как от них ожидается. Метод remove для списка всегда удаляет первое вхождение элемента, а методы add и addAll добавляют элементы в конец.

Так же как и интерфейс Set, интерфейс List предполагает более строгую реализацию equals и hashCode, так что два списка могут сравниваться, даже если они имеют разные реализации. Два списка равны, если они содержат одинаковые элементы в одинаковом порядке.

Ниже описаны операции, работающие с позициями элементов. Все операции, которые принимают индекс элемента в качестве параметра, бросают исключение java.lang.IndexOutOfBoundsException, если индекс выходит за границы списка.

```
E get(int index)
```

Возвращает элемент по индексу.

```
E set(int index,
      E element)
```

Заменяет элемент по указанному индексу. Возвращает старый элемент.

```
void add(int index,
         E element)
```

Вставляет элемент в позицию по указанному индексу.

```
E remove(int index)
```

Удаляет элемент по указанному индексу.

```
boolean addAll(int index,
               Collection<? extends E> c)
```

Вставляет все элементы из коллекции `c` в позицию по указанному индексу.

```
int indexOf(Object o)
```

Возвращает индекс элемента `o` либо –1, если нет элемента с таким индексом.

В List также присутствует метод получения итератора для списков:

```
ListIterator<E> listIterator()
```

Этот итератор наследуется от обычного итератора, но также имеет метод `previous()`, позволяющий вернуться к предыдущему элементу, и метод `hasPrevious()`, позволяющий проверить наличие предыдущего элемента для текущей позиции итерации. Вызовы `next()` и `previous()` можно чередовать, но нужно быть осторожным, т. к. первый вызов `previous()` после череды вызовов `next()` вернет то же значение, что и последний вызов `next()`, а первый вызов `next()` после череды `previous()` вернет то же значение, что и последний вызов `previous()`. Методы `previousIndex()` и `nextIndex()` используются для получения индекса элемента, который будет возвращен следующим вызовом `previous()` и `next()` соответственно. Если мы находимся в самом начале списка, т. е. следующий вызов `next()` вернет первый элемент, то вызов `previousIndex()` вернет –1. Если же мы находимся в конце списка, т. е. `hasNext()` возвращает `false`, то `nextIndex()` вернет значение `list.size()`. Он также имеет дополнительные методы set и add, которые позволяют заменить последний возвращенный элемент и добавить новый элемент соответственно.

Метод `subList`:

```
List<E> subList(int fromIndex,
                int toIndex)
```

возвращает часть списка между `fromIndex` (включая) и `toIndex` (исключая). Изменения в возвращаемом списке отражаются в исходном списке. Например, с помощью кода `list.subList(from, to).clear()` можно удалить элементы из исходного списка. Поведение возвращенного списка становится непредсказуемым, если элементы в этом диапазоне будут удалены или добавлены в исходный список; все изменения нужно делать через возвращенный список, будьте внимательны.

26.5. Интерфейс Queue

Интерфейс `java.util.Queue` представляет собой очередь.

Каждый метод в `Queue` существует в двух вариантах: первый бросает исключение, если операцию выполнить не удается, а второй возвращает специальное значение (`null` или `false`). Вариант, возвращающий значение, используется в основном для очередей с ограничением на количество элементов.

Методы, бросающие исключение:

- `add(e)` — добавление элемента;
- `remove()` — удаление элемента;
- `element()` — проверка элемента.

Методы, возвращающие специальное значение:

- `offer(e)` — добавление элемента;
- `poll()` — удаление элемента;
- `peek()` — проверка элемента.

Основные реализации: `java.util.LinkedList`, `java.util.PriorityQueue`.

Пакет `java.util.concurrent` содержит интерфейс `java.util.concurrent.BlockingQueue`, который расширяет интерфейс `Queue` и добавляет методы, ожидающие появления элемента при получении элемента, и методы, ожидающие появления свободного места при добавлении элемента.

Классы, реализующие `BlockingQueue`:

- `java.util.concurrent.LinkedBlockingQueue`;
- `java.util.concurrent.ArrayBlockingQueue`;
- `java.util.concurrent.PriorityBlockingQueue`;
- `java.util.concurrent.DelayQueue`;
- `java.util.concurrent.SynchronousQueue`;
- `java.util.concurrent.TransferQueue`.

По названию, как видно, определить тип метода невозможно. Можно либо запомнить, либо добавить эту страницу в закладки и все время сверяться с ней.

Методы `add` и `offer` добавляют элементы в хвост очереди.

Методы `remove()` и `poll()` удаляют головной элемент из очереди и возвращают удаленный элемент.

Методы `element()` и `peek()` возвращают головной элемент, но не удаляют его.

`Queue` обычно реализует FIFO (first-in-first-out), но это необязательно. Некоторые реализации могут использовать очередь на основе приоритетов и другие. Некоторые реализации ограничивают количество элементов в очереди, тогда `add` бросает исключение `java.lang.IllegalStateException` при превышении предела, а `offer` возвращает `false`.

Метод `remove()` бросает исключение `java.util.NoSuchElementException`, если очередь пуста, а метод `poll()` возвращает в этом случае `null`.

Обычно `Queue` не позволяет хранить элементы со значением `null`, т. к. `null` используется в качестве специального возвращаемого значения некоторых методов. Исключением является `LinkedList`, который позволяет хранить `null` по историческим причинам.

Реализации `Queue` обычно не предлагают методов `hashCode()` и `equals()`, основанных на элементах. Вместо этого они обычно наследуют стандартные реализации этих методов от `Object`.

26.6. Интерфейс Deque

Интерфейс `java.util.Deque` представляет собой очередь с двумя концами, т. е. это линейная коллекция элементов с возможностью добавления и удаления элементов с обоих концов.

> ***Примечание***
> Deque можно использовать и в качестве очереди, и в качестве стека.

Стандартные реализации: `java.util.ArrayDeque`, `java.util.LinkedList`.

Специальная реализация: `java.util.concurrent.LinkedBlockingDeque`.

Как и для `Queue`, методы существуют в двух экземплярах.

Методы, бросающие исключение:

- `addFirst(e)` — добавляет элемент в начало очереди;
- `addLast(e)` — добавляет элемент в конец очереди;
- `removeFirst()` — удаляет первый элемент и возвращает его;
- `removeLast()` — удаляет последний элемент и возвращает его;
- `getFirst()` — возвращает первый элемент;
- `getLast()` — возвращает последний элемент.

Методы, возвращающие специальное значение:

- `offerFirst(e)` — добавляет элемент в начало очереди;
- `offerLast(e)` — добавляет элемент в конец очереди;
- `pollFirst()` — удаляет первый элемент и возвращает его;
- `pollLast()` — удаляет последний элемент и возвращает его;
- `peekFirst()` — возвращает первый элемент;
- `peekLast()` — возвращает последний элемент.

26.7. Интерфейс Map

Интерфейс `java.util.Map` представляет собой карту/отображение/справочник — вариантов перевода много. Он отображает ключи на значения. `Map` не может содержать дублирующиеся ключи. Каждый ключ отображается не более чем на одно значение. В отличие от других интерфейсов коллекций, этот НЕ наследуется от интерфейса `java.util.Collection`.

В платформе Java есть три стандартные реализации `Map`: `HashMap`, `TreeMap`, `LinkedHashMap`. Их производительность и особенности аналогичны реализациям `Set`: `HashSet`, `TreeSet`, `LinkedHashSet`. Также есть устаревший потокобезопасный `Hashtable`, но вместо него рекомендуется использовать реализации интерфейса `ConcurrentMap`.

Есть реализации специального назначения: `java.util.EnumMap`, `java.util.WeakHashMap`, `java.util.IdentityHashMap`. Реализация `EnumMap` предназначена специально для карт со значениями перечисления одного вида в качестве ключей. `WeakHashMap` использует слабые ссылки на свои ключи, что позволяет сборщику мусора удалить пару ключ-значение, если на них нет больше ссылок, кроме как внутри самого `WeakHashMap`. `IdentityHashMap` — реализация на основе хеш-таблицы. При сравнении ключей и значений он использует сравнение ссылок, а не сравнение объектов, в отличие от остальных реализаций.

```
V put(K key,
      V value)
```

Связывает значение `value` с ключом `key`. Возвращает предыдущее связанное значение либо `null`. Метод также может вернуть `null`, если предыдущее связанное значение было `null` (некоторые реализации карт поддерживают значения `null`). Используйте метод `boolean containsKey(Object key)`, чтобы различать подобные ситуации.

```
V get(Object key)
```

Возвращает значение, которое связано с ключом `key`. Возвращает `null`, если нет ассоциации с ключом `key`. Может также вернуть `null`, если `key` связан со значением `null` (некоторые реализации карт поддерживают значения `null`). Используйте метод `boolean containsKey(Object key)`, чтобы различать подобные ситуации.

```
V remove(Object key)
```

Удаляет связку ключ-значение для ключа `key`. Возвращает значение, которое было связано с этим ключом, или `null`, если не было связки с таким ключом. Может также вернуть `null`, если `key` был связан со значением `null` (некоторые реализации карт поддерживают значения `null`). Используйте метод `boolean containsKey(Object key)`, чтобы различать подобные ситуации.

```
boolean containsKey(Object key)
```

Возвращает `true`, если карта содержит связку со значением для ключа `key`.

`boolean containsValue(Object value)`

Возвращает `true`, если карта содержит хотя бы одну связку со значением `value`. Для большинства реализаций этот метод может потребовать линейное относительно размера карты время.

`int size()`

Возвращает количество связок ключ-значение в карте.

`void putAll(Map<? extends K,? extends V> m)`

Копирует все связки из `m` в карту.

`void clear()`

Удаляет все связки из карты.

`Set<K> keySet()`

Возвращает `Set`, содержащий ключи из `Map`. Последующие изменения в `Map` отражаются в возвращенном `Set`. Удаление элемента из `Set` удаляет связку из `Map`.

`Set<Map.Entry<K,V>> entrySet()`

Возвращает `Set`, содержащий все связки из карты. Изменения в возвращенном `Set` отражаются на исходном `Map`, так же как и изменения в исходном `Map` отражаются в этом `Set`. Этот возвращенный `Set` поддерживает удаление элементов, но не поддерживает добавление.

`Collection<V> values()`

Возвращает коллекцию, содержащую значения из `Map`. Изменения в исходном `Map` отражаются в возвращенной коллекции и наоборот. Коллекция поддерживает удаление элементов, что удаляет связку из карты, но не поддерживает операции добавления.

26.8. Интерфейс ConcurrentMap

Интерфейс `ConcurrentMap` гарантирует атомарность и потокобезопасность своих методов. Он наследуется от интерфейса `Map` и получает все его методы.

Одна из основных реализаций — `java.util.concurrent.ConcurrentHashMap`. Класс `ConcurrentHashMap` является основной заменой устаревшему `Hashtable`.

> ***Примечание***
>
> Класс `ConcurrentHashMap`, как и `Hashtable`, не поддерживают `null` в качестве ключа или значения. Однако в `HashMap` можно использовать `null` как в качестве ключа, так и в качестве привязанного к этому ключу значения.

Методы получения значения из него в основном не накладывают блокировок и возвращают самое последнее успешно записанное значение. Методы записи не накла-

дывают блокировку на всю коллекцию "ключ-значение", а лишь на ее часть, что обеспечивает высокий уровень параллелизма.

В плане производительности `ConcurrentHashMap` лучше, чем `Collections.synchronizedMap(myMap)`, т. к. второй вариант просто делает все методы `synchronized`.

26.9. Класс Dictionary и его наследник Hashtable

Абстрактный класс `java.util.Dictionary` раньше выполнял ту же роль, что сейчас выполняет интерфейс `Map` и `ConcurrentMap`. Сейчас он устарел, как и его единственный наследник `java.util.Hashtable`. Однако `Hashtable` теперь реализует интерфейс `Map`, так что потенциально может использоваться как его реализация, но это не рекомендуется.

26.10. Сортировка объектов

Список `List` может быть отсортирован с помощью метода:

```
Collections.sort(l);
```

Если `List` состоит из элементов `String`, то они будут отсортированы в алфавитном порядке. Если он состоит из элементов `Date`, то список будет отсортирован в хронологическом порядке. Но как это происходит? `String` и `Date` реализуют интерфейс `java.lang.Comparable`. Реализации интерфейса `java.lang.Comparable` предоставляют естественный порядок сортировки для класса, который позволяет сортировать объекты автоматически.

Многие стандартные классы платформы Java уже реализуют интерфейс:

- `Byte`;
- `Character`;
- `Long`;
- `Integer`;
- `Short`;
- `Double`;
- `Float`;
- `BigInteger`;
- `BigDecimal`;
- `Boolean (Boolean.FALSE < Boolean.TRUE)`;
- `File` (зависимый от системы лексикографический порядок от пути к файлу);
- `String` (лексикографический без учета региональных настроек);
- `Date`.

Если вы пытаетесь сортировать список, элементы которого не реализуют интерфейс `Comparable`, то `Collections.sort(list)` бросит исключение `java.lang.ClassCastException`. Также `Collections.sort(list, comparator)` бросит исключение

java.lang.ClassCastException, если вы попытаетесь отсортировать список, элементы которого не могут сравниваться посредством этого comparator-а. Элементы, которые могут сравниваться друг с другом, называются взаимно сравнимыми. Несмотря на то что элементы разных типов могут быть взаимно сравнимыми, ни один из классов, перечисленных здесь, не допускает межклассового сравнения.

Интерфейс java.lang.Comparable состоит из следующего метода:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Метод compareTo сравнивает текущий объект с указанным объектом и возвращает отрицательное или положительное число либо ноль, если наш объект меньше, или больше переданного, или равен ему в параметрах соответственно. Если переданный объект не может сравниваться с нашим объектом, то метод бросает исключение java.lang.ClassCastException.

StudentScore.java

```
package ru.urvanov.javaindynamics2022.collection;

import java.util.Objects;

public final class StudentScore implements Comparable<StudentScore> {
    private final String name;
    private final int math;
    private final int physics;
    private final int philosophy;

    public StudentScore(
            String name,
            int math,
            int physics,
            int philosophy) {
        if (name == null) {
            throw new NullPointerException();
        }
        this.name = name;
        this.math = math;
        this.physics = physics;
        this.philosophy = philosophy;
    }

    private double averageScore() {
        return ((double) math + physics + philosophy) / 3.0;
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        StudentScore that = (StudentScore) o;
        return math == that.math
                && physics == that.physics
                && philosophy == that.philosophy
                && Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, math, physics, philosophy);
    }

    @Override
    public int compareTo(StudentScore otherStudentScore) {
        if (otherStudentScore == null) {
            throw new NullPointerException();
        }
        double averageScoreThis = averageScore();
        double averageScoreOther = otherStudentScore.averageScore();
        return Double.compare(averageScoreThis, averageScoreOther);
    }

    @Override
    public String toString() {
        return "StudentScore{" +
                "name='" + name + '\'' +
                ", math=" + math +
                ", physics=" + physics +
                ", philosophy=" + philosophy +
                '}';
    }
}
```

Этот пример показывает следующее:

- Объекты `StudentScore` неизменяемые. Неизменяемые объекты очень полезны, особенно при использовании объектов в качестве элементов `Set` или ключей `Map`. Эти коллекции будут работать некорректно, если вы будете изменять содержимое их элементов или ключей во время выполнения.
- Поля, участвующие в сортировке, не могут быть `null`, т. к. являются примитивными типами, а поле name проверяется на `null`, поэтому никогда не возникнет `java.lang.NullPointerException` при использовании методов `StudentScore`.

- Метод `hashCode` переопределен. Важно переопределять этот метод для любого класса, который переопределяет метод `equals`. (Равные объекты должны иметь одинаковые хеш-коды.)
- Метод `equals` возвращает `false`, если переданный объект равен `null` или относится к неподходящему типу. Метод `compareTo` бросает исключение в подобных ситуациях. Оба этих поведения требуются согласно контракту этих методов.
- Метод `toString` переопределен так, чтобы он выводил объект `StudentScore` в понятном для человека формате. Старайтесь всегда переопределять метод `toString`, особенно если объекты планируется помещать в коллекции. Многие коллекции имеют `toString`, использующий методы `toString` у своих элементов, ключей и значений.
- Метод `compareTo` объекта `StudentScore` реализует сортировку по среднему баллу.

Пример программы, создающей массив элементов класса `StudentScore` и сортирующей их:

StudentSort.java

```
package ru.urvanov.javaindynamics2022.collection;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class StudentSort {
 public static void main(String[] args) {
   StudentScore[] studentScoreArray = new StudentScore[] {
       new StudentScore("Mary", 2, 5, 1),
       new StudentScore("John", 5, 5, 5),
       new StudentScore("Tom", 2, 2, 2)
   };
   List<StudentScore> studentScoreList
       = Arrays.asList(studentScoreArray);
   Collections.sort(studentScoreList);
   System.out.println(studentScoreList);
 }
}
```

Если запустить эту программу, то она выведет в консоль:

```
[StudentScore{name='Tom', math=2, physics=2, philosophy=2},
StudentScore{name='Mary', math=2, physics=5, philosophy=1},
StudentScore{name='John', math=5, physics=5, philosophy=5}]
```

Метод `compareTo` должен следовать четырем важным правилам (здесь `sgn(expression)` означает знак выражения):

- Для всех `x` и `y` должно выполняться равенство `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))`. (Если `x.compareTo(y)` бросает исключение, то `y.compareTo(x)` также должно бросать исключение.)
- Сравнение должно быть транзитивным (`x.compareTo(y)>0 && y.compareTo(z)>0` означает, что `x.compareTo(z)>0`).
- `x.compareTo(y)==0` означает, что `sgn(x.compareTo(z)) == sgn(y.compareTo(z))` для любых `z`.
- Очень рекомендуется, чтобы `(x.compareTo(y)==0) == (x.equals(y))`, т. е. compareTo не должен работать вразнобой с equals.

При необходимости сортировать объекты в отличном от естественного порядке, нужно использовать интерфейс `java.util.Comparator`:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Метод `compare` сравнивает два аргумента, возвращая отрицательное или положительное число либо ноль, если первый аргумент меньше или больше второго либо равен ему соответственно. Если типы переданных аргументов не подходят для этого `Comparator`-а, то метод `compare` бросает исключение `java.lang.ClassCastException`.

Большая часть сказанного про `java.lang.Comparable` относится и к `java.util.Comparator`. Метод `compare` должен придерживаться тех же четырех правил, что и метод `compareTo` у `Comparable`.

Предположим, что у нас есть класс `LadyMidday`:

LadyMidday.java

```
package ru.urvanov.javaindynamics2022.collection;

public class LadyMidday implements Comparable<LadyMidday> {
 private String name;
 private int level;
 private int money;
 ...
}
```

Давайте предположим, что естественный порядок сортировки экземпляров `LadyMidday` — это их сортировка по `level` (уровню монстра). К сожалению, требуется вывести их по имени. Это означает, что нам нужно сделать немного дополнительной работы:

LadyMiddaySort.java

```
package ru.urvanov.javaindynamics2022.collection;

import java.util.*;
```

```
public class LadyMiddaySort {
    static final Comparator<LadyMidday> NAME_ORDER
            = (o1, o2) -> {
        return o1.getName().compareTo(o2.getName());
    };

    public static void main(String[] args) {
        LadyMidday[] ladyMiddayArray = {
                new LadyMidday("Monster1", 10, 3),
                new LadyMidday("Midday killer", 34, 300),
                new LadyMidday("poor", 99, 10),
        };
        List<LadyMidday> ladyMiddays = Arrays.asList(ladyMiddayArray);
        Collections.sort(ladyMiddays, NAME_ORDER);
        System.out.println(ladyMiddays);
    }
}
```

При написании компараторов нужно быть аккуратным. Если нужно получить обратный порядок, чтобы вначале шло большее значение, а затем меньшее, то НЕЛЬЗЯ сначала вычислить значение `compareTo` для обычного порядка, а затем применить к нему знак "минус". Есть одно отрицательное значение для `int`, которое остается отрицательным, если к нему применять унарный минус:

```
-Integer.MIN_VALUE == Integer.MIN_VALUE
```

`Comparator`, который мы написали выше, может сортировать `List`, но он не может использоваться в сортированных коллекциях, например `TreeSet`, т. к. он генерирует порядок сортировки, несовместимый с `equals`. Это означает, что этот `Comparator` указывает равными те объекты, которые методы `equals` указывают как разные. Например, две `LadyMidday` с одинаковыми именами будут равны. При сортировке в списке `List` это не имеет значения, но при использовании `Comparator`-а для упорядоченной коллекции это будет не очень хорошо. Если вы будете использовать этот `Comparator` в `TreeSet`, то `TreeSet` будет работать, как ожидается, но при этом будет нарушена договоренность о самом интерфейсе `Set`, потому что он спроектирован именно на основе метода `equals`.

Если нужно чтобы `TreeSet` с вашим компаратором работал в соответствии с методом `equals`, как и ожидается, то нужно писать компаратор таким образом, чтобы он считал отличающимися объекты, считающиеся отличающимися с точки зрения `equals`, и равными те, которые равны с точки зрения `equals`.

26.11. Интерфейс SortedSet

Интерфейс `java.util.SortedSet` — это `java.util.Set`, который поддерживает свои элементы в отсортированном по возрастанию виде согласно естественному порядку или согласно `Comparator`-у, который был передан в конструктор. В дополнение

к обычным операциям java.util.Set-а, интерфейс java.util.SortedSet добавляет три вида операций:

- Работа с частью Set-а с помощью методов subset(E fromElement, E toElement), headSet(E toElement), tailSet(E fromElement), которые возвращают представление, позволяющее работать с частью исходного Set.
- Получение первого и последнего элементов в множестве с помощью методов first() и last().
- Доступ к Comparator-у, который используется для сортировки множества с помощью метода comparator().

По соглашению все реализации интерфейса java.util.Collection общего назначения имеют стандартный конструктор, который принимает Collection в качестве параметра. Реализации SortedSet не являются исключением. В TreeSet этот конструктор создает экземпляр, который сортирует эти элементы в соответствии с их естественным порядком. Однако было бы гораздо лучше динамически проверять, была ли коллекция экземпляром SortedSet, и сортировать TreeSet в соответствии с тем же Comparator-ом или естественным порядком. Специально для этого есть конструктор, который принимает SortedSet и возвращает новый экземпляр TreeSet, который содержит те же самые элементы, отсортированные в соответствии с тем же самым критерием. Заметьте, что порядок определяется на этапе компиляции, а не на этапе выполнения, т. к. используется перегрузка конструкторов.

Реализации SortedSet также по соглашению имеют конструктор, который принимает java.util.Comparator и возвращает пустое множество, которое сортируется в соответствии с указанным Comparator-ом. Если в конструктор передается null, то используется естественный порядок сортировки.

Операции работы с частью множества аналогичны операциям работы с частью списков, но есть одна большая разница. Подмножество остается валидным даже в том случае, если исходное множество было изменено напрямую. Это осуществимо, т. к. конечные точки подмножества являются абсолютными точками из элементов, а не индексами элементов, как это было для списков.

Сортированные множества имеют три метода получения подмножества. Первый метод — subSet. У него есть два параметра, в которых указываются сами элементы диапазона подмножества. Эти элементы должны быть сравнимы с элементами в множестве при помощи Comparator SortedSet-а или естественного порядка его элементов, смотря, что используется. Начальный элемент включается в подмножество, а конечный — не включается.

```
int count = monsters.subSet("Ghoul", "Succubus").size();
```

Изменения в множестве, полученном с помощью метода subSet, отражаются на исходном множестве. Можно удалить монстров от Ghoul до Succubus:

```
monsters.subSet("Ghoul", "Succubus").clear();
```

Есть также два метода: headSet и tailSet. Эти методы принимают в качестве параметра только один объект. Метод headSet возвращает подмножество от начала до

указанного объекта, исключая указанный объект. Второй возвращает множество от указанного объекта (включительно) до конца SortedSet-а. Пример:

```
SortedSet<String> headMonsters = dictionary.headSet("Ghoul");
SortedSet<String> tailMonsters = dictionary.tailSet("Succubus");
```

26.12. Интерфейс SortedMap

Интерфейс java.util.SortedMap представляет собой java.util.Map, который хранит свои элементы отсортированными по возрастанию ключей в соответствии с естественным порядком или указанным Comparator-ом.

Интерфейс java.util.SortedMap содержит методы из java.util.Map и дополнительные:

- Работа с частью SortedMap.
- Получение первого и последнего ключей карты.
- Доступ к Comparator-у, если он есть.

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Операции SortedMap, унаследованные от Map, действуют аналогично, но имеют две особенности:

- Итератор проходит по элементам в соответствии с порядком сортировки.
- Методы toArray у коллекций ключей, значений и элементов возвращают отсортированные массивы.

По соглашению все реализации Map содержат конструктор, который принимает Map в качестве параметра. Реализации SortedMap тоже имеют такой конструктор. В java.util.TreeMap этот конструктор создает экземпляр, который упорядочивает свои элементы в соответствии с естественным порядком сортировки ключей. Однако было бы лучше динамически проверять, что переданный Map является экземпляром SortedMap, и сортировать его с тем же критерием (Comparator-ом или естественным порядком сортировки). TreeMap содержит еще один конструктор, который принимает SortedMap и создает TreeMap, содержащий связки из переданного SortedMap, отсортированные по тому же критерию. Заметьте, что это определяется на этапе компиляции, т. к. используется перегрузка конструкторов.

26.13. Другие реализации интерфейсов коллекций

Класс `java.util.Collections` имеет специальные методы, позволяющие любую коллекцию обернуть в синхронизированную, которая использует исходную коллекцию, но добавляет синхронизацию:

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

Чтобы гарантировать потокобезопасность, все изменения коллекции должны происходить с полученной синхронизированной коллекцией, т. е. в идеале нужно "потерять" ссылку на исходную коллекцию.

Также можно обернуть коллекцию, получив неизменяемую коллекцию, элементы которой нельзя добавлять, удалять или заменять (возникнет `java.lang.UnsupportedOperationException`). Для этого нужно использовать следующие методы класса `Collections`:

```
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);
public static <T> Set<T> unmodifiableSet(Set<? extends T> s);
public static <T> List<T> unmodifiableList(List<? extends T> list);
public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);
public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<? extends T> s);
public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ?
extends V> m);
```

В Java 9 появились новые способы создания неизменяемых коллекций в дополнение к перечисленным выше.

В классе `List` в Java 9 появились статические методы `of`, в которые можно передать элементы для нового неизменяемого списка:

```
List<String> unmodifiableList = List.of("Vasya", "Petya");
```

В классе `Set` в Java 9 появились аналогичные статические методы `of`:

```
Set<String> unmodifiableSet = Set.of("Vasya", "Petya");
```

В классе `Map` тоже появились аналогичные методы:

```
Map<String, String> unmodifiableMap = Map.of("id1", "Vasya", "id2", "Petya");
```

Можно получить коллекцию с динамически проверяемым во время выполнения типом:

```
public static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> type);
public static <E> Set<E> checkedSet(Set<E> s, Class<E> type);
public static <E> List<E> checkedList(List<E> list, Class<E> type);
public static <K,V> Map<K,V> checkedMap(Map<K,V> m, Class<K> keyType,
                                        Class<V> valueType);
```

```
public static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s, Class<E> type)
public static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K,V> m, Class<K>
keyType, Class<V> valueType);
```

Любой массив можно превратить в неизменяемый список с помощью метода `java.util.Arrays.asList`:

```
@SafeVarargs
public static <T> List<T> asList(T... a)
```

С помощью метода `Collections.nCopies` можно создать неизменяемый список, содержащий указанное количество копий элемента:

```
public static <T> List<T> nCopies(int n,
                                  T o)
```

Можно создать неизменяемое множество, состоящее только из одного элемента `Collections.singleton`:

```
public static <T> Set<T> singleton(T o)
```

С помощью методов `Collections.emptySet()`, `Collections.emptyList()`, `Collections.emptyMap()` можно создать пустые коллекции:

```
public static final <T> Set<T> emptySet();
public static final <T> List<T> emptyList();
public static <E> SortedSet<E> emptySortedSet();
public static final <K,V> SortedMap<K,V> emptySortedMap();
```

26.14. Java Stream API

Предположим, что мы создаем компьютерную игру, в которой пытаемся описать домового классом `Hobgoblin`:

Hobgoblin.java

```
package ru.urvanov.javaindynamics2022.collection;

import java.time.LocalDate;

public class Hobgoblin {

    private String name;
    private Role role;
    private int gold;
    private double health = 100.0;
    private double power = 30.0;

    public enum Role {
        WARRIOR,
        MAGE,
```

```
        ARCHER
    }

    public Hobgoblin(
            String name,
            Role role,
            int gold,
            double health,
            double power) {
        this.name = name;
        this.role = role;
        this.gold = gold;
        this.health = health;
        this.power = power;
    }

    ...
}
```

Следующий пример выводит все имена:

Hobgoblin.java

```
System.out.println("Список домовых:");
for (Hobgoblin hobgoblin : hobgoblins) {
    System.out.println(hobgoblin.getName());
}
```

То же самое с помощью потоков и агрегатных операций:

Hobgoblin.java

```
System.out.println("Список домовых с помощью Java Stream API:");
hobgoblins
        .stream()
        .map(Hobgoblin::getName)
        .forEach(System.out::println);
```

В этом примере версия со `stream`-ами длиннее, но для более сложных задач версии с потоками будут более лаконичными.

Канал (pipeline) — это последовательность агрегатных операций. Канал (pipeline) состоит из следующих компонентов:

- Источник, в роли которого может выступать коллекция, массив, генерирующая функция или канал ввода/вывода.
- Ноль или более промежуточных операций. Промежуточная операция, например `map`, порождает новый `stream`. Операция `map` преобразует элементы.

- Терминальная операция, например `forEach`, `findFirst`, `findAny`, `count`, создает не-`stream` результат, например значение примитивного типа, коллекцию, или не возвращает никаких значений совсем. В этом примере параметром `forEach` является ссылка на метод `System.out.println`.

Следующий пример вычисляет среднее количество золота у всех домовых:

Hobgoblin.java

```
System.out.println("Среднее количество золота у всех домовых:");
OptionalDouble average = hobgoblins
        .stream()
        .mapToInt(Hobgoblin::getGold)
        .average();
```

Метод `mapToInt` возвращает новый `stream` типа `IntStream` (который является `stream`-ом, содержащим только `int`). Метод применяет функцию, переданную в параметр, к каждому элементу `stream`-а. В этом примере в него передается ссылка на метод получения золота, имеющегося в наличии у домового. В результате `mapToInt` возвращает `stream`, содержащий возрасты всех членов мужского пола из коллекции `hobgoblins.java`.

Операция `average` вычисляет среднее значение элементов из `stream`-а типа `IntStream`. Она возвращает объект типа `java.util.OptionalDouble`. Если `stream` не содержит элементов, то `average` возвращает пустой экземпляр `java.util.OptionalDouble`, и вызов метода `getAsDouble` приведет к исключению `java.util.NoSuchElementException`.

Кроме `IntStream`, существуют еще `LongStream` и `DoubleStream` для работы `long` и `double` соответственно. Кроме `average` существуют еще и другие терминальные операции: `sum`, `min`, `max`, `count` и другие, которые возвращают одно значение на основе содержимого `stream`-а. Такие операции называются редукционными операциями (reduction operations). JDK также содержит редукционные методы, которые возвращают коллекцию вместо одного значения. Многие редукционные операции выполняют конкретную задачу, например нахождение среднего значения или группировку элементов по категориям. Однако JDK содержит редукционные операции общего назначения `reduce` и `collect`, которые позволяют создавать собственные терминальные операции.

С помощью `findFirst` и `findAny` можно вернуть один элемент из `stream`. Отличие в том, что `findFirst` возвращает первый элемент при упорядоченном `stream`, а `findAny` возвращает любой первый попавшийся:

Hobgoblin.java

```
System.out.println("findAny result: " + hobgoblins.stream().findAny());
```

Метод `stream.reduce` — редукционная операция (reduction operation) общего назначения.

Рассмотрите следующий канал (pipeline):

Hobgoblin.java

```
System.out.println("Сумма золота у всех домовых:");
int sum1 = hobgoblins
        .stream()
        .mapToInt(Hobgoblin::getGold)
        .sum();
System.out.println("sum1: " + sum1);
```

И сравните его со следующим, использующим операцию `Stream.reduce`:

Hobgoblin.java

```
int sum2 = hobgoblins
        .stream()
        .map(Hobgoblin::getGold)
        .reduce(0, (a, b) -> a + b);
System.out.println("sum2: " + sum2);
```

В этом примере операция `reduce` принимает два аргумента:

- `identity`: начальное значение редукции и результат по умолчанию для случая, когда нет элементов в `stream`. В этом примере 0.
- `accumulator`: функция с двумя параметрами: частичный результат редукции (в этом примере сумма обработанных чисел до этого момента) и следующий элемент `stream`-а. Он возвращает частичный результат. В этом примере функция `accumulator`-а является лямбда-выражением, которое складывает два `Integer`-а и возвращает полученный результат.

Операция `reduce` всегда возвращает новое значение. Однако функция `accumulator` возвращает новое значение каждый раз, когда она обрабатывает элемент из `stream`. Предположим, что мы ходим превратить эти элементы в более сложный объект, например коллекцию. Если мы будем для этого использовать `reduce`, то каждая обработка элемента вызовет создание новой коллекции, что неэффективно.

В отличие от метода `reduce`, который всегда создает новое значение при обработке элемента, метод `collect` меняет существующее значение.

Рассмотрим пример, когда из большого числа домовых нам необходимо создать одного объединенного домового, который получается в результате мутации. Количество золота у результирующего домового должно быть суммой золота у всех исходных домовых, а сила результирующего домового должна быть средним арифметическим от силы всех исходных домовых:

HobgoblinMorph.java

```
package ru.urvanov.javaindynamics2022.collection;

import java.util.function.Consumer;

public class HobgoblinMorph implements Consumer<Hobgoblin> {
 private int gold;
 private double power;
 private int count;

 @Override
 public void accept(Hobgoblin other) {
   count++;
   gold+= other.getGold();
   power+= other.getPower();
 }

 public void combine(HobgoblinMorph hobgoblinMorph) {
   this.gold += hobgoblinMorph.gold;
   this.power += hobgoblinMorph.power;
   this.count += hobgoblinMorph.count;
 }

 public Hobgoblin morph() {
   return new Hobgoblin(
       "WeatlhyHobgoblin",
       Hobgoblin.Role.WARRIOR,
       gold,
       100.0,
       power / count);
 }
}
```

Следующий конвейер использует класс `HobgoblinMorph` и метод `collect` для вычисления среднего возраста всех членов мужского пола:

Hobgoblin.java

```
System.out.println("Преобразуем в объединенного богатого домового:");
HobgoblinMorph hobgoblinMorph = hobgoblins.stream()
                .collect(
                        HobgoblinMorph::new,
                        HobgoblinMorph::accept,
                        HobgoblinMorph::combine);
System.out.println(hobgoblinMorph.morph());
```

Операция `collect` в этом примере принимает три аргумента:

- `supplier`: фабричная функция, которая создает экземпляры. Для операции `collect` она создает экземпляры результирующего контейнера. В этом примере — экземпляры класса `HobgoblinMorph`.
- `accumulator`: функция включает элемент из `steam` в число членов результирующего контейнера. В этом примере она модифицирует контейнер `HobgoblinMorph`, увеличивая переменные для подсчета золота и силы.
- `combiner`: функция, которая принимает два результирующих контейнера и объединяет их содержимое.

Операции `collect` можно использовать с параллельными `stream`-ами. (Если вы запустите метод `collect` с параллельным `stream`, то JDK создаст новый поток, когда функция `combiner` создает новый объект, а значит, вам не нужно беспокоиться о синхронизации.)

Операция `collect` часто используется со стандартными коллекторами, которые преобразуют элементы в список, множество или в `Map` (`Collectors.toList`, `Collectors.toSet`, `Collectors.toMap`).

Hobgoblin.java

```
List<Integer> hobgoblinGolds = hobgoblins.stream()
        .map(Hobgoblin::getGold)
        .collect(Collectors.toList());
```

Эта версия `collect` принимает один параметр типа `Collector`. Этот класс содержит три метода, которые могут использоваться в качестве аргументов операции `collect`.

Пример использует `Collectors.toList`, который собирает элементы `stream`-а в новый экземпляр `List`. Как и большинство методов класса Collectors, метод `toList` возвращает экземпляр `Collector`, а не саму коллекцию.

Java Stream API позволяет производить группировку элементов коллекций по какому-либо полю либо по результату вычисления какого-либо выражения.

Следующий пример группирует элементы `hobgoblins` по полу:

Hobgoblin.java

```
Map<Role, List<Hobgoblin>> groupedHobgoblins = hobgoblins.stream()
                .collect(Collectors.groupingBy(Hobgoblin::getRole));
```

Существует также вариант `groupingBy` с двумя параметрами. Второй параметр принимает операцию reduce:

Hobgoblin.java

```
Map<Role, Integer> groupedHobgoblinsGold = hobgoblins.stream()
                .collect(Collectors.groupingBy(
```

```
                Hobgoblin::getRole,
                Collectors.summingInt(Hobgoblin::getGold)));
```

Java Stream API может работать с параллельными `stream`-ами. Сложность в реализации параллелизма в приложениях, использующих коллекции, — коллекции не являются потокобезопасными; это означает, что несколько потоков не могут манипулировать коллекцией без вмешательства в поток (thread interference) и ошибок консистентности памяти (memory consistency errors). Collections Framework предоставляет специальные обертки, которые делают синхронизацию для любой коллекции, что делает их потокобезопасными. Однако синхронизация приводит к конкуренции потоков. Вы хотите избежать конкуренции потоков, т. к. это мешает их параллельной работе. Агрегатные операции и параллельные потоки позволяют реализовать параллелизм с непотокобезопасными коллекциями, т. к. вы не меняете коллекцию, пока работаете с ней.

Параллелизм не означает, что операции будут выполняться быстрее, чем если бы они выполнялись последовательно, однако он может быть быстрее, если у вас достаточно много данных и много процессорных ядер. Несмотря на то что агрегатные операции позволяют вам легко реализовать параллелизм, решение об этом все еще остается на ваших плечах.

Вы можете выполнять `stream`-ы последовательно или параллельно. При выполнении `stream`-ов параллельно Java разделяет `stream` на несколько `substream`-ов. Агрегатные операции проходят по этим `substream`-ам параллельно и затем объединяют результат.

`Stream` всегда последовательный, если не указано обратное. Для того чтобы создать параллельный `stream`, вызовите метод `java.util.Collection.parallelStream`, либо вы можете вызвать `java.util.stream.BaseStream.parallel`. Например, следующая инструкция вычисляет средний возраст всех мужских участников параллельно:

Hobgoblin.java

```
int parallelSumGold = hobgoblins
                .parallelStream()
                .filter(v -> Role.MAGE == v.getRole())
                .mapToInt(Hobgoblin::getGold)
                .sum();
```

В случае использования параллельных `stream`-ов необходимо также использовать коллекторы, поддерживающие параллельную обработку. Например, для примера с группировкой домовых по роли нужно использовать `Collectors.groupingByConcurrent`:

Hobgoblin.java

```
Map<Role, List<Hobgoblin>> parallelGroupedHobgoblins
        = hobgoblins.stream()
       .collect(Collectors.groupingByConcurrent(Hobgoblin::getRole));
```

26.15. Алгоритмы

Все стандартные реализации алгоритмов находятся в классе `java.util.Collections`, и все они являются статическими методами, которые принимают в качестве первого аргумента коллекцию, над которой будет осуществляться выполнение алгоритма.

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
public static <T> void sort(List<T> list,
                            Comparator<? super T> c)
```

Методы `Collections.sort` сортируют содержимое списка в соответствии с естественным порядком либо в соответствии с указанным компаратором.

Пример:

Sort.java

```
import java.util.*;

public class Sort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

```
public static void shuffle(List<?> list)
```

Метод `Collections.shuffle` перемешивает содержимое списка случайным образом.

```
public static void reverse(List<?> list)
```

Метод `Collections.reverse` меняет порядок элементов в списке на обратный.

```
public static <T> void fill(List<? super T> list,
                            T obj)
```

Метод `Collections.fill` заменяет все элементы списка на `obj`.

```
public static <T> void copy(List<? super T> dest,
                            List<? extends T> src)
```

Метод `Collections.copy` копирует все элементы из списка `src` в список `dest`. Список `dest` должен содержать количество элементов, как минимум равное количеству элементов `src`.

```
public static void swap(List<?> list,
                        int i,
                        int j)
```

Меняет местами два элемента в списке.

```
@SafeVarargs
public static <T> boolean addAll(Collection<? super T> c,
                                 T... elements)
```

Метод `Collections.addAll` добавляет все указанные элементы в коллекцию `c`.

```
public static <T> int binarySearch(
                        List<? extends Comparable<? super T>> list,
                        T key)

public static <T> int binarySearch(List<? extends T> list,
                                   T key,
                                   Comparator<? super T> c)
```

Методы `Collections.binarySearch` ищут вхождение элемента в списке с помощью бинарного поиска и естественного порядка сортировки или компаратора. Список должен быть отсортирован по возрастанию с помощью используемого компаратора или в естественном порядке (зависимости от того, что используется при поиске). Если есть несколько искомых элементов, то может вернуться индекс любого из них. Если искомый объект не найден, то возвращается `(-(`*`место_вставки`*`) - 1)`, где *`место_вставки`* — позиция, куда следовало бы вставить элемент.

```
public static int frequency(Collection<?> c,
            Object o)
```

Метод `Collections.frequency` возвращает количество элементов `o` в коллекции `c`. Для сравнения используется метод `equals`.

```
public static boolean disjoint(Collection<?> c1,
                               Collection<?> c2)
```

Возвращает `true`, если две коллекции не имеют ни одного общего элемента.

26.16. Задания

1. Создайте класс мифического существа. Добавьте ему поле "жизненная сила", а также поле типа существа: демоническое, нежить, призрачное и т. д. Создайте список, содержащий существа разных типов. С помощью Java Stream API сгруппируйте существ по типу и посчитайте суммарное количество жизненной силы у каждого типа.
2. Создайте класс, который мог бы представлять информацию о банковском счете: валюта, сумма, Ф.И.О. владельца и т. д. С помощью Java Stream API подсчитайте суммарное количество денег каждой валюты и среднее количество денег каждой валюты на счете.
3. Создайте класс, который мог бы представлять собой карточку товара в трюме пиратского корабля: вес, название, примерная цена за единицу, количество, редкость, риск при продаже и т. д. Заполните список десятью экземплярами этого

класса. С помощью стандартных алгоритмов отсортируйте его по конечной стоимости.

4. Назовите способы создания неизменяемых коллекций.
5. Какие классы потокобезопасных коллекций вы знаете?

ГЛАВА 27

Дата и время

27.1. Введение

Обработка дат и времени — довольно сложная задача. В мире существует большое количество часовых поясов, которые периодически меняются. Нужно учитывать переход на зимнее и летнее время, секунды координации, високосные года и многое другое.

Java уже довольно много лет, и попыток создать классы для обработки дат было несколько. Здесь я лишь в общих чертах опишу три способа:

- `java.util.Date` — самый древний способ. Большая часть методов отмечена устаревшими, но сам класс до сих пор используется довольно часто.
- `java.util.Calendar` — обычно используется совместно с `java.util.Date`.
- Пакет `java.time` — новый способ, сильно превосходит предыдущие по удобству и функционалу, но используется пока не так часто (по крайней мере, мне редко приходилось сталкиваться с ним).

27.2. Класс Date

Самый древний способ. Если вы посмотрите на описание методов `java.util.Date`, то вы заметите, что большая часть методов указана устаревшими. Реально полезных и работающих немного.

Создание экземпляров:

DateExample.java

```
package ru.urvanov.javaindynamics2022.datetime;

public class DateExample {
 public static void main(String[] args) {
```

```
    // Создаем экземпляр с текущей датой и временем.
    java.util.Date currentDate = new java.util.Date();

    // Создаем экземпляр, содержащий время, получившееся по
    // прошествии указанного количества миллисекунд
    // от 1 января 1970 года 00:00 GMT. Обычно используется
    // для конвертации из других форматов даты и времени.
    java.util.Date dateByMillis = new java.util.Date(29368498236L);

    System.out.println("currentDate = " + currentDate);
    System.out.println("dateByMillis = " + dateByMillis);
  }
}
```

Полезные методы класса `java.util.Date`:

```
public boolean after(Date when)
```

Возвращает `true`, если дата объекта находится позже указанной.

```
public boolean before(Date when)
```

Возвращает `true`, если дата объекта находится перед указанной.

```
public int compareTo(Date anotherDate)
```

Сравнение дат. Возвращает 0, если даты равны. Возвращает –1, если дата объекта находится до `anotherDate`. Возвращает 1, если дата объекта позже `anotherDate`.

```
public static Date from(Instant instant)
public Instant toInstant()
```

Используются для конвертации в Instant из пакета `java.time` и обратно.

```
public long getTime()
public void setTime(long time)
```

Возвращает и устанавливает значение `java.util.Date` в количестве миллисекунд, прошедших с 1 января 1970 года 00:00 GMT.

27.3. Класс Calendar

Класс `java.util.Calendar` является абстрактным классом. Единственная реализация — `java.util.GregorianCalendar`. Новый экземпляр создается фабричным методом:

```
Calendar rightNow = Calendar.getInstance();
```

`Calendar` предоставляет методы установки значений различных полей вроде `YEAR`, `MONTH`, `DAY_OF_MONTH`, `HOUR` и т. д., манипуляции этими полями, например добавление дня или месяца. Дата и время представляются количеством миллисекунд, прошед-

ших с 1 января 1970 года 00:00:00.000 GMT. Обычно класс `java.util.Calendar` используется для создания экземпляров `java.util.Date`, представляющих необходимую дату и время.

`Calendar` имеет два режима интерпретации полей: мягкий (lenient) и жесткий (non-lenient). Когда `Calendar` в мягком (lenient) режиме, то он принимает больший диапазон значений полей, чем возвращает. Когда `Calendar` пересчитывает значения полей, возвращаемых `get()`, то все поля нормализуются. Например, мягкий `GregorianCalendar` интерпретирует `MONTH == JANUARY, DAY_OF_MONTH == 32` как 1 февраля.

Когда `Calendar` в жестком (non-lenient) режиме, то он бросает исключение в подобных случаях. Например, `GregorianCalendar` всегда возвращает значение поля `DAY_OF_MONTH` в диапазоне от 1 до длины месяца. Жесткий `GregorianCalendar` бросает исключение во время выполнения вычислений своего времени или поля, если хотя бы одно поле выходит за допустимые границы.

Полезные методы:

```
public void set(int field,
                int value)
```

Устанавливает новое значения поля.

```
public int get(int field)
```

Возвращает значение поля. Для мягкого режима значения полей нормализуются, а в случае жесткого режима и выхода значения какого-либо поля за допустимый диапазон бросается исключение.

```
public void roll(int field,
                 int amount)
```

Добавляет `amount` (может быть отрицательным) к указанному полю.

```
public final Date getTime()
public final void setTime(Date date)
```

Конвертация в `java.util.Date` и обратно.

```
public void setLenient(boolean lenient)
```

Устанавливает мягкий (lenient) или жесткий (non-lenient) режим.

```
public final Instant toInstant()
```

Конвертирует в `Instant` из пакета `java.time`.

```
public void setTimeZone(TimeZone value)
```

Позволяет установить часовой пояс. По умолчанию используется часовой пояс системы, на которой запущена программа. С помощью этого метода можно указать другой часовой пояс.

Например:

```
calender.setTimeZone(TimeZone.getTimeZone("America/Los_Angeles"));
```

```
public TimeZone getTimeZone()
```

Возвращает часовой пояс, ассоциированный с данным экземпляром `Calendar`.

27.4. Пакет java.time

Наиболее удобный и современный способ работы с датой и временем. Берет свое начало от библиотеки Joda-Time.

Есть два базовых способа представления времени. Один способ представляет время в терминах человека, таких как год, месяц, день, час, минуты и секунды. Второй способ представляет машинное время, измеряя время непрерывно с начала, называемого эпохой, в наносекундах. Пакет Date-Time содержит большое количество классов, представляющих дату и время. Некоторые классы в Date-Time API представляют машинное время, некоторые — человеческое.

Сначала определите, какие аспекты даты и времени вам нужны, затем выберите класс или классы, которые подходят под ваши нужды.

Например, вы можете выбрать `java.time.LocalDate` для хранения даты нерабочего праздничного дня. Если вам нужно указать дату и время какого-либо события без привязки к часовому поясу, то вы можете использовать `java.time.LocalDateTime`. Если вам нужна точная отметка времени события, то вы можете использовать `java.time.ZonedDateTime`, который хранит часовой пояс, дату и время. Если вы создаете временную отметку, то, вероятнее всего, вы захотите использовать `java.time.Instant`, который позволяет сравнивать одну временную отметку с другой.

27.5. Перечисление DayOfWeek

Перечисление `java.time.DayOfWeek` описывает день недели. Целочисленные значения для констант начинаются с 1 (понедельник) и заканчиваются 7 (воскресенье).

Список констант:

- MONDAY (понедельник, 1).
- TUESDAY (вторник, 2).
- WEDNESDAY (среда, 3).
- THURSDAY (четверг, 4).
- FRIDAY (пятница, 5).
- SATURDAY (суббота, 6).
- SUNDAY (воскресенье, 7).

Вы можете использовать метод `public String getDisplayName(TextStyle style, Locale locale)` для получения названий дней недели в соответствии с региональ-

ными настройками пользователя. Перечисление `java.time.format.TextStyle` позволяет указать тип строки: `FULL`, `NARROW` (обычно одна буква), `SHORT` (аббревиатура).

Пример:

DayOfWeekExample.java

```
package ru.urvanov.javaindynamics2022.datetime;

import java.time.DayOfWeek;
import java.time.format.TextStyle;
import java.util.Locale;

public class DayOfWeekExample {
 public static void main(String[] args) {
   DayOfWeek dow = DayOfWeek.MONDAY;
   // Можно использовать Locale.getDefault()
   // Либо вариант getDisplayName без указания локали
   Locale locale = new Locale("ru", "RU");
   System.out.println(dow.getDisplayName(TextStyle.FULL, locale));
   System.out.println(dow.getDisplayName(TextStyle.NARROW, locale));
   System.out.println(dow.getDisplayName(TextStyle.SHORT, locale));
 }
}
```

С помощью методов `plus` и `minus` можно получить день недели, который будет через определенное количество дней или был несколько дней назад:

DayOfWeekExample.java

```
// Прибавляем два дня
DayOfWeek dowPlusTwo = dow.plus(2);
System.out.println(dowPlusTwo.getDisplayName(TextStyle.FULL, locale));
```

27.6. Перечисление Month

Перечисление `java.time.Month` описывает двенадцать месяцев, пронумерованных от 1 до 12:

- JANUARY (январь, 1).
- FEBRUARY (февраль, 2).
- MARCH (март, 3).
- APRIL (апрель, 4).
- MAY (май, 5).
- JUNE (июнь, 6).
- JULY (июль, 7).

- AUGUST (август, 8).
- SEPTEMBER (сентябрь, 9).
- OCTOBER (октябрь, 10).
- NOVEMBER (ноябрь, 11).
- DECEMBER (декабрь, 12).

Перечисление `java.time.Month` содержит несколько полезных методов. Например, метод `maxLength()` возвращает максимально возможное количество дней в месяце:

```
System.out.printf("%d%n", Month.FEBRUARY.maxLength());
```

Также есть метод public `String getDisplayName(TextStyle style, Locale locale)`, позволяющий получить текстовое название месяца в соответствии с указанной локалью:

MonthExample.java

```
package ru.urvanov.javaindynamics2022.datetime;

import java.time.Month;
import java.time.format.TextStyle;
import java.util.Locale;

public class MonthExample {
    public static void main(String[] args) {
        Month month = Month.AUGUST;
        // Можно использовать Locale.getDefault()
        // Либо вариант getDisplayName без указания локали
        Locale locale = new Locale("ru", "RU");
        System.out.println(month.getDisplayName(TextStyle.FULL, locale));
        System.out.println(
                month.getDisplayName(TextStyle.NARROW, locale));
        System.out.println(
                month.getDisplayName(TextStyle.SHORT, locale));
    }
}
```

С помощью методов `plus` и `minus` можно вычислить месяц, который будет через определенное количество месяцев или был определенное количество месяцев назад.

MonthExample.java

```
// Месяц, который был три месяца назад
Month monthBeforeThree = month.minus(3);
System.out.println(monthBeforeThree.getDisplayName(
        TextStyle.FULL, locale));
```

27.7. Класс LocalDate

Класс `java.time.LocalDate` хранит год, месяц и день. Он используется для хранения и обработки даты без времени (например, даты рождения). Примеры создания:

LocalDateExample.java

```
LocalDate date = LocalDate.of(2022, Month.NOVEMBER, 20);
```

С помощью различных методов `plus*` и `minus*` можно получать даты, которые отстоят от этого экземпляра `localDate` на определенное количество месяцев, дней, лет или недель.

27.8. Класс LocalTime

Класс `java.time.LocalTime` оперирует только временем. Он полезен для хранения времени открытия/закрытия магазина и т. д. Пример:

LocalTimeExample.java

```
// 10 часов 30 минут
LocalTime localTime0 = LocalTime.of(10, 30);

// 10 часов 30 минут 45 секунд
LocalTime localTime1 = LocalTime.of(10, 30, 45);

// 10 часов 30 минут 45 секунд 100 наносекунд
LocalTime localTime2 = LocalTime.of(10, 30, 45, 100);
```

Класс `LocalTime` не сохраняет информацию о часовом поясе и летнем/зимнем времени.

LocalTimeExample.java

```
// Методы plus* и minus*
LocalTime localTime3 = localTime0.plusHours(2).minusMinutes(10).plusSeconds(15);
```

27.9. Класс LocalDateTime

Класс `java.time.LocalDateTime` хранит дату и время. Он является чем-то вроде комбинации `LocalDate` и `LocalTime`. В дополнение к методу `now()`, который есть у каждого временного класса, класс `LocalDateTime` содержит большое количество методов `of`, которые позволяют создать экземпляры `LocalDateTime`. Метод `from` конвертирует экземпляр другого класса в `LocalDateTime`. Также есть методы для добавления и вычитания часов, минут, дней и недель.

Пример:

LocalDateTimeExamle.java

```
LocalDateTime localDateTimeNow = LocalDateTime.now();
System.out.println(localDateTimeNow);

LocalDateTime localDateTime = LocalDateTime.of(
        2022, Month.NOVEMBER, 10, 12, 30, 0);
System.out.println(localDateTime);
```

С помощью методов `plus*` и `minus*` можно получать новые экземпляры `LocalDateTime`:

LocacIDateTimeExample.java

```
LocalDateTime localDateTime2 = localDateTimeNow
        .plusDays(3)
        .minusMonths(4)
        .plusYears(2);
System.out.println(localDateTime2);
```

27.10. Класс YearMonth

Класс `java.time.YearMonth` представляет месяц с годом. Следующие примеры используют `YearMonth.lengthOfMonth()`, чтобы определить количество дней в конкретном годе и месяце:

YearMonthExample.java

```
YearMonth date = YearMonth.now();
System.out.printf("%s: %d%n", date, date.lengthOfMonth());

YearMonth date2 = YearMonth.of(2010, Month.FEBRUARY);
System.out.printf("%s: %d%n", date2, date2.lengthOfMonth());

YearMonth date3 = YearMonth.of(2012, Month.FEBRUARY);
System.out.printf("%s: %d%n", date3, date3.lengthOfMonth());
```

Этот код выведет в консоль:

```
2013-06: 30
2010-02: 28
2012-02: 29
```

С помощью методов `plus*` и `minus*` можно получать экземпляры `YearMonth`, отстоящие на определенное количество месяцев и лет:

YearMonthExample.java

```
YearMonth date4 = date.plusMonths(2).minusYears(3);
System.out.println(date4);
```

27.11. Класс MonthDay

Класс `java.time.MonthDay` содержит день с месяцем.

MonthDayExample.java

```
MonthDay date = MonthDay.of(Month.JUNE, 15);
```

27.12. Класс Year

Класс `java.time.Year` хранит год.

Year.java

```
Year year = Year.of(2022);
System.out.println(year);
```

С помощью методов `plusYears` и `minusYears` можно получать экземпляры `Year`, отстоящие на определенное количество лет:

YearExample.java

```
Year year2 = year.plusYears(10);
System.out.println(year2);
```

27.13. Классы ZoneId и ZoneOffset

Часовой пояс — это участок земной поверхности, на котором используется одно и то же стандартное время. Каждый часовой пояс определяется идентификатором, имеющим формат регион/город (Europe/Moscow), и смещением от Гринвича/UTC. Например, смещение для Москвы это +3:00.

Date-Time API содержит два класса для указания часового пояса или смещения:

- `java.time.ZoneId` указывает идентификатор часового пояса и поставляет правила для конвертирования `java.time.Instant` в `java.time.LocalDateTime`.
- `java.time.ZoneOffset` указывает смещение часового пояса от Гринвича/UTC.

Смещения от Гринвича/UTC обычно определяются в полных часах, но есть исключения. Следующий код выводит в консоль все часовые пояса, которые используют смещение от Гринвича/UTC, указанные не в полных часах.

AllZones.java

```
package ru.urvanov.javaindynamics2022.datetime;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Set;

public class AllZones {
    public static void main(String[] args) {
        Set<String> allZones = ZoneId.getAvailableZoneIds();
        LocalDateTime dt = LocalDateTime.now();

       // Создаем список с зонами и сортируем его.
        List<String> zoneList = new ArrayList<String>(allZones);
        Collections.sort(zoneList);

        for (String s : zoneList) {
            ZoneId zone = ZoneId.of(s);
            ZonedDateTime zdt = dt.atZone(zone);
            ZoneOffset offset = zdt.getOffset();
            int secondsOfHour = offset.getTotalSeconds() % (60 * 60);
            String out = String.format("%35s %10s%n", zone, offset);

            // Пишем только часовые пояса, которые используют смещение
            // в неполных часах.
            if (secondsOfHour != 0) {
                System.out.printf(out);
            }
        }
    }
}
```

Этот пример выведет в консоль:

```
America/Caracas     -04:30
     America/St_Johns     -02:30
        Asia/Calcutta     +05:30
         Asia/Colombo     +05:30
            Asia/Kabul     +04:30
       Asia/Kathmandu     +05:45
        Asia/Katmandu     +05:45
```

```
         Asia/Kolkata     +05:30
         Asia/Rangoon     +06:30
          Asia/Tehran     +04:30
   Australia/Adelaide     +09:30
Australia/Broken_Hill     +09:30
     Australia/Darwin     +09:30
      Australia/Eucla     +08:45
        Australia/LHI     +10:30
  Australia/Lord_Howe     +10:30
      Australia/North     +09:30
      Australia/South     +09:30
 Australia/Yancowinna     +09:30
  Canada/Newfoundland     -02:30
        Indian/Cocos     +06:30
                Iran     +04:30
             NZ-CHAT     +12:45
      Pacific/Chatham     +12:45
    Pacific/Marquesas     -09:30
      Pacific/Norfolk     +11:30
```

27.14. Класс ZonedDateTime

Класс `java.time.ZonedDateTime` можно рассматривать как комбинацию `java.time.LocalDateTime` и `java.time.ZoneId`. Он представляет собой полную дату, время и часовой пояс. С помощью методов `plus*` и `minus*` можно получать новые экземпляры, отстоящие на определенное количество дней, часов, секунд и других частей даты и времени:

ZonedDateTimeExample.java

```
LocalDateTime localDate = LocalDateTime.of(
        2022, Month.JANUARY, 30, 19, 30, 0);
ZoneId zoneId = ZoneId.of("Europe/Moscow");

ZonedDateTime zonedDateTime = ZonedDateTime.of(localDate, zoneId);
System.out.println(zonedDateTime);

System.out.println(zonedDateTime.plusMonths(2).minusYears(3));
```

27.15. Класс OffsetDateTime

Класс `java.time.OffsetDateTime` можно рассматривать как комбинацию `java.time.LocalDateTime` и `java.time.ZoneOffset`. Он содержит полную дату, время и смещение от Гринвича/UTC (`+/-часы:минуты`).

OffsetDateTimeExample.java

```
LocalDateTime localDate = LocalDateTime.of(
        2022, Month.JANUARY, 30, 19, 30, 0);
ZoneOffset offset = ZoneOffset.of("+04:00");

OffsetDateTime offsetDate = OffsetDateTime.of(localDate, offset);
System.out.println(offsetDate);

OffsetDateTime offsetDate3 = offsetDate.plusDays(3).minusHours(4);
System.out.println(offsetDate3);
```

27.16. Класс OffsetTime

Класс `java.time.OffsetTime` можно рассматривать как комбинацию `java.time.LocalTime` и `java.time.ZoneOffset`. Он содержит время и смещение от Гринвича/UTC (`+/-часы:минуты`).

27.17. Класс Instant

Класс `java.time.Instant` — это один из самых основных классов Date-Time API, который представляет наносекунды от 1 января 1970 года 00:00:00 по UTC. Класс java.time.Instant может содержать отрицательное значение, если он представляет время до 1 января 1970 года.

Пример создания:

InstantExample.java

```
import java.time.Instant;
Instant timestamp = Instant.now();
```

Класс содержит полезные константы: `Instant.EPOCH` (1 января 1970 00:00:00Z), `Instant.MIN` (самое прошлое время из возможных), `Instant.MAX` (самое будущее время из возможных).

Есть методы для сравнения экземпляров `Instant`, например `isAfter` и `isBefore`. Метод `until` возвращает время между двумя экземплярами `Instant`. Следующий код сообщает количество секунд, прошедших с начала эпохи:

InstantExample.java

```
long secondsFromEpoch = Instant.ofEpochSecond(0L).until(Instant.now(),
                        ChronoUnit.SECONDS);
```

Класс `java.time.Instant` работает с машинным представлением времени. Он не работает с годами, месяцами, часами и т. д. Если вам нужно работать с ними,

то следует преобразовать его в java.time.Instant в другой класс, например java.time.LocalDateTime или java.time.ZonedDateTime, связав java.time.Instant с часовым поясом. Пример:

InstantExample.java

```
Instant timestamp;
...
LocalDateTime ldt = LocalDateTime.ofInstant(timestamp, ZoneId.systemDefault());
System.out.printf("%s %d %d at %d:%d%n", ldt.getMonth(), ldt.getDayOfMonth(),
                  ldt.getYear(), ldt.getHour(), ldt.getMinute());
```

Этот код выводит в консоль что-то вроде этого:

```
MAY 30 2013 at 18:21
```

Классы java.time.ZonedDateTime и java.time.OffsetDateTime могут быть преобразованы в экземпляр java.time.Instant, т. к. они указывают на конкретный момент во времени. Однако для обратного преобразования нужно указать часовой пояс или смещение.

27.18. Форматирование и преобразование из строки

Классы из Date-Time API содержат методы parse и format для разбора даты и/или времени из строки и форматированного вывода в строку. Эти методы принимают экземпляр класса java.time.format.DateTimeFormatter. Класс DateTimeFormatter содержит большое количество предопределенных форматировщиков, вы также можете определить свой.

Методы parse и format бросают исключение, если во время конвертации возникает какая-нибудь проблема. Ваш код разбора строки должен отлавливать исключение java.time.format.DateTimeParseException, а форматирующий код должен отлавливать java.time.format.DateTimeException. Эти исключения непроверяемые, так что компилятор не будет требовать их обязательной обработки.

Класс java.time.format.DateTimeFormatter неизменяемый и потокобезопасный. Его можно присвоить константе, если нужно.

Классы из java.time можно использовать и в java.util.Formatter и в String.format, как и старые классы java.util.Date и java.util.Calendar.

DateTimeFormatterExample.java

```
ZonedDateTime zonedDateTime = ZonedDateTime.of(
        2022, // Год
        1, // Месяц
        31,  // День месяца
        8, // час
        51, // Минуты
```

```
        0,  // Секунды
        0, // Наносекунды
        ZoneId.of("Europe/Moscow"));
String formattedIso = zonedDateTime.format(DateTimeFormatter.ISO_ZONED_DATE_TIME);
System.out.println(formattedIso);

DateTimeFormatter fullFormatter = DateTimeFormatter
        .ofLocalizedDateTime(FormatStyle.FULL, FormatStyle.FULL)
        .withLocale(new Locale("ru", "RU"));
String formattedFull = zonedDateTime.format(fullFormatter);
System.out.println(formattedFull);

DateTimeFormatter shortFormatter = DateTimeFormatter.ofLocalizedDateTime(
                FormatStyle.SHORT, FormatStyle.SHORT)
        .withLocale(new Locale("ru", "RU"));
String formattedShort = zonedDateTime.format(shortFormatter);
System.out.println(formattedShort);

// С локалью по умолчанию
String formattedWithSystemLocale = zonedDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(
                FormatStyle.FULL, FormatStyle.FULL));
System.out.println(formattedWithSystemLocale);

ZonedDateTime parsedZonedDateTime = ZonedDateTime.parse(
        "2022-01-31T08:51:00+03:00[Europe/Moscow]",
        DateTimeFormatter.ISO_ZONED_DATE_TIME);
System.out.println(parsedZonedDateTime);
```

27.19. Интерфейс TemporalAdjuster

Интерфейс `java.time.temporal.TemporalAdjuster` содержит методы, которые принимают значение времени и возвращает его выровненное по какому-либо принципу значение.

Класс `java.time.temporal.TemporalAdjusters` содержит предопределенные выравниватели даты и времени для поиска первого или последнего дня в месяце, первого или последнего дня года, последней среды месяца и т. д.

TemporalAdjusterExample.java

```
LocalDate date = LocalDate.of(2022, Month.JANUARY, 31);
DayOfWeek dotw = date.getDayOfWeek();
System.out.printf("%s выпадает на %s%n", date, dotw);

System.out.printf("первый день месяца: %s%n",
        date.with(TemporalAdjusters.firstDayOfMonth()));
```

```
System.out.printf("первая пятница месяца: %s%n",
        date.with(TemporalAdjusters.firstInMonth(DayOfWeek.FRIDAY)));
System.out.printf("последний день месяца: %s%n",
        date.with(TemporalAdjusters.lastDayOfMonth()));
System.out.printf("первый день следующего месяца: %s%n",
        date.with(TemporalAdjusters.firstDayOfNextMonth()));
System.out.printf("первый день следующего года: %s%n",
        date.with(TemporalAdjusters.firstDayOfNextYear()));
System.out.printf("первый день года: %s%n",
        date.with(TemporalAdjusters.firstDayOfYear()));
```

Этот код выведет в консоль следующее:

```
2022-01-31 выпадает на MONDAY
первый день месяца: 2022-01-01
первая пятница месяца: 2022-01-07
последний день месяца: 2022-01-31
первый день следующего месяца: 2022-02-01
первый день следующего года: 2023-01-01
первый день года: 2022-01-01
```

27.20. Интерфейс TemporalQuery

Интерфейс `java.time.temporal.TemporalQuery` может использоваться для получения информации об объекте времени.

Класс `java.time.temporal.TemporalQueries` содержит предопределенные `TemporalQuery`. С помощью него можно получить, например, минимальный шаг изменения какого-нибудь объекта из `java.time`.

В реальной жизни вам вряд ли когда-либо придется использовать `TemporalQuery` и `TemporalQueries`, но знать об их существовании может быть полезно.

27.21. Класс Duration

Класс `java.time.Duration` используется для измерения промежутка времени между двумя машинными временными объектами, например `Instant`-ами.

Примеры:

ZonedDateTimeExample.java

```
ZonedDateTime t1 = ZonedDateTime.of(
        2022, 1, 1, 10, 30, 0, 0, ZoneId.of("Europe/Moscow"));
ZonedDateTime t2 = ZonedDateTime.of(
        2022, 12, 1, 10, 30, 0, 0, ZoneId.of("Europe/Moscow"));

long days = Duration.between(t1, t2).toDays();
System.out.println("days = " + days);
```

```
Instant start = Instant.now();

Duration gap = Duration.ofSeconds(33);
Instant later = start.plus(gap);
```

> ***Запомните***
>
> `Duration` не связан с датой и временем, он не хранит информацию о зонах и переходе на летнее/зимнее время. Добавление `Duration`, эквивалентного 1 дню, к `ZonedDateTime` добавит ровно 24 часа, независимо от часового пояса и зимнего/летнего времени.

27.22. Перечисление ChronoUnit

Перечисление `java.time.temporal.ChronoUnit` объявляет единицы измерения для времени. Метод `ChronoUnit.between` используется, когда вам нужно измерить количество времени только в днях, только в секундах или в иных единицах измерения времени. Метод `between` работает с объектами дат и времени, но возвращает только количество в определенных единицах. Пример:

ChronoUnitExample.java

```
package ru.urvanov.javaindynamics2022.datetime;

import java.time.Instant;
import java.time.temporal.ChronoUnit;

public class ChronoUnitExample {
    public static void main(String[] args) {
        Instant previous = Instant.now();
        Instant current = Instant.now().plusMillis(10_000L);
        long gap = ChronoUnit.SECONDS.between(previous,current);
        System.out.println("difference in seconds: " + gap);
    }
}
```

27.23. Класс Period

Класс `java.time.Period` определяет период времени в человеческих единицах: месяцах, днях, годах.

Общий период времени представляется суммой всех трех частей: месяцев, дней, годов. Чтобы получить период только в одной единице времени, используйте `ChronoUnit.between`.

Следующий год сообщает ваш возраст в годах, месяцах, днях, а затем общее количество прожитых дней.

PeriodExample.java

```
LocalDate localDateFrom = LocalDate.of(2022, 3, 1);
LocalDate localDateTo = LocalDate.of(2022, 6, 30);
Period period = Period.between(localDateFrom, localDateTo);
System.out.println("years = " + period.getYears()
        + " months = " + period.getMonths()
        + " days = " + period.getDays());
```

Код выводит в консоль следующее:

```
years = 0 months = 3 days = 29
```

27.24. Класс Clock

Многие классы с датой и временем содержат метод `now()`, который создает объект с текущей датой и временем, используя системные часы и часовой пояс по умолчанию. Эти же объекты также содержат метод `now(Clock)`, который позволяет передать другой `java.time.Clock`.

Текущая дата и время зависят от часового пояса, и для глобальных приложений необходим `Clock`, чтобы гарантировать создание даты/времени с корректным часовым поясом. Несмотря на то что использование `java.time.Clock` необязательно, эта возможность помогает тестировать ваш код с другими часовыми зонами или с фиксированным временем.

Класс `Clock` абстрактный, поэтому вы не можете создать его экземпляров. Следующие фабричные методы могут быть полезны для тестирования:

- `Clock.systemUTC()` возвращает clock, содержащий часовой пояс Гринвич/UTC.
- `Clock.fixed(Instant, ZoneId)` всегда возвращает один и тот же `Instant`. Для этого `Clock` время не идет, оно остановилось.

27.25. Задания

1. Выведите график нерабочих дней в этом году, используя форматирование в русской локали.
2. Создайте программу, которая хранит (например, в массиве, а лучше во внешнем файле) дни рождения сотрудников. Пусть при запуске она выводит имя сотрудника, день рождения которого будет ближе всего, а также количество дней, которое осталось до этого события.
3. Создайте программу, которая конвертировала бы Unix-время в объект `ZonedDateTime`, а потом выводила его на экран. Имейте в виду, что Unix-время — это количество СЕКУНД, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года.
4. Создайте программу, подсчитывающую количество дней, оставшихся до конца года.

ГЛАВА 28

Форматирование и парсинг

28.1. Введение

В мире огромное количество различных языков, религий, культур и стран. В каждой стране и в каждом языке зачастую приняты свои формы записи чисел, дат и денежных единиц. Локализация приложения в конкретной стране — это далеко не самый тривиальный процесс, подразумевающий не только перевод на язык страны, но и запись чисел и дат в формате, принятом в этой стране, удаление иконок и изображений, нарушающих законодательство этой страны и т. д.

Например, в русском языке принята следующая запись дат:

12.01.2016 — двенадцатое января две тысячи шестнадцатого года;
10 июня 2016 — 10 июня 2016 года.

Но если мы локализуем приложение в США, то даты будут выглядеть так:

01/12/2016 — the twelfth of January year twenty sixteen;
June 10, 2016 — the tenth of June year twenty sixteen.

С записью чисел тоже все далеко не так просто. Вот числа для русского языка:

10 000 000,34;
3 454,456.

А вот те же числа для США:

10,000,000.34;
3,454.456.

Все современные языки поддерживают конвертацию дат и числовых переменных в строку и обратно в соответствии с указанными региональными настройками (локалью) или региональными настройками по умолчанию.

Для понимания дальнейшего текста главы рекомендуется ознакомиться с классом `java.util.Locale`, представляющим собой локаль (региональные настройки).

Вы можете получить экземпляр текущей локали с помощью кода:

```
Locale locale = Locale.getDefault()
```

28.2. Класс NumberFormat

Класс `java.text.NumberFormat` предназначен для форматирования и парсинга чисел. Это абстрактный класс, экземпляры которого можно получить с помощью методов `getInstance()`:

```
public static final NumberFormat getInstance()
public static NumberFormat getInstance(Locale inLocale)
```

Полученный экземпляр `NumberFormat` можно использовать для форматирования чисел с помощью метода `format` и парсинга чисел с помощью метода `parse`:

```
public final String format(double number)

public final String format(long number)

public Number parse(String source)
             throws ParseException
```

Пример:

```
String str1 = java.text.NumberFormat.getInstance()
        .format(10_000_000.34);
String str2 = java.text.NumberFormat.getInstance().format(8000);
String str3 = java.text.NumberFormat.getInstance()
        .format(new java.math.BigDecimal("34000.56"));
System.out.println(str1);
System.out.println(str2);
System.out.println(str3);
try {
    Number var1 = java.text.NumberFormat.getInstance().parse(str1);
    Number var2 = java.text.NumberFormat.getInstance().parse(str2);
    Number var3 = java.text.NumberFormat.getInstance().parse(str3);
    System.out.println(var1);
    System.out.println(var2);
    System.out.println(var3);
} catch (java.text.ParseException pe) {
    pe.printStackTrace();
}
```

Результат:

```
10 000 000,34
8 000
34 000,56
1.000000034E7
8000
34000.56
```

28.3. Класс DecimalFormat

Класс `java.text.DecimalFormat` расширяет класс `java.text.NumberFormat`. Класс `DecimalFormat` предназначен специально для работы с десятичными числами. Он поддерживает различные типы чисел: проценты, денежные единицы и т. д.

Все методы `format` и `parse`, а также способ создания аналогичны `NumberFormat`.

Класс `DecimalFormat` дополнительно содержит настройку:

```
public void setParseBigDecimal(boolean newValue)
```

позволяющую методу `parse` возвращать экземпляры `BigDecimal`.

Класс `java.text.DecimalFormat` содержит два дополнительных конструктора, принимающих строку с форматом числа:

```
public DecimalFormat(String pattern)

public DecimalFormat(String pattern,
                     DecimalFormatSymbols symbols)
```

Метод, принимающий `DecimalFormatSymbols`, позволяет полностью настроить форматирование числа. Строка `pattern` содержит шаблон вида "0.00" и может содержать следующие специальные символы (табл. 28.1).

Таблица 28.1. *Значения символов*

Специальный символ	Значение
0	Число
#	Число, незначащие нули не показываются
. (символ точки)	Разделитель дробной и целой части
-	Знак "минус"
,	Разделитель групп
E	Разделяет мантиссу и экспоненту в научной записи. В префиксе или в суффиксе нужно заключать в кавычки
;	Разделяет положительный шаблон числа и отрицательный шаблон числа
%	Умножается на 100 и показывается в процентах
\u2030	Умножается на 1000 и показывается как в милях
¤ (\u00A4)	Денежный знак. Заменяется денежным символом локали. Если задвоен, то указывается международный денежный символ. Если указан, то используется разделитель денег вместо разделителя дробной и целой части
'	Кавычки. "'#'#" преобразует 123 в "#123" . Чтобы записать саму одинарную кавычку, используйте две одинарные кавычки

Пример:

```
System.out.println(new java.text.DecimalFormat("0000.000").format(10));
```

Результат:

```
0010,000
```

28.4. Класс DateFormat

Класс `java.text.DateFormat` предназначен для форматирования дат. Получить экземпляр этого класса можно с помощью одного из методов:

```
public static final DateFormat getInstance()
```

Если нужно форматировать/парсить только дату без времени, то можно использовать один из следующих методов:

```
public static final DateFormat getDateInstance()

public static final DateFormat getDateInstance(int style)

public static final DateFormat getDateInstance(int style,
                                                Locale aLocale)
```

Здесь `style` может быть одно из значений: `DateFormat.FULL`, `DateFormat.LONG`, `DateFormat.SHORT`, `DateFormat.MEDIUM`, `DateFormat.DEFAULT`.

Если нужно форматировать/парсить только время, то можно использовать один из методов:

```
public static final DateFormat getTimeInstance()

public static final DateFormat getTimeInstance(int style)

public static final DateFormat getTimeInstance(int style,
                                                Locale aLocale)
```

Здесь `style` может быть одно из значений: `DateFormat.FULL`, `DateFormat.LONG`, `DateFormat.SHORT`, `DateFormat.MEDIUM`, `DateFormat.DEFAULT`.

Если нужно форматировать/парсить дату с временем, то можно использовать один из методов:

```
public static final DateFormat getDateTimeInstance()

public static final DateFormat getDateTimeInstance(int dateStyle,
                                                    int timeStyle)

public static final DateFormat getDateTimeInstance(int dateStyle,
                                                    int timeStyle,
                                                    Locale aLocale)
```

Форматирование и парсинг также происходят с помощью методов `format` и `parse`. Пример:

```
System.out.println(java.text.DateFormat.getInstance()
        .format(new java.util.Date());
```

28.5. Класс DateTimeFormatter

Класс `java.time.format.DateTimeFormatter` используется совместно с классами даты и времени из пакета `java.time`. Он был подробно описан в *главе 27 "Дата и время"*.

28.6. Класс SimpleDateFormat

Класс `java.text.SimpleDateFormat` наследуется от `java.text.DateFormat` и позволяет указать пользовательский шаблон форматирования.

Конструкторы:

```
public SimpleDateFormat(String pattern)

public SimpleDateFormat(String pattern,
                        Locale locale)

public SimpleDateFormat(String pattern,
                        DateFormatSymbols formatSymbols)
```

Конструктор с `DateFormatSymbols` позволяет создать форматировщик, используя особые правила.

Шаблон `pattern` может содержать следующие специальные символы, приведенные в табл. 28.2.

Таблица 28.2. *Значения символов*

Буква	Компонент даты и времени	Представление	Примеры
G	Эра	Text	AD
y	Год	Year	1996; 96
Y	Год	Year	2009; 09
M	Месяц в году (зависит от контекста)	Month	July; Jul; 07
L	Месяц в году (самостоятельная форма)	Month	July; Jul; 07
w	Неделя в году	Number	27
W	Неделя в месяце	Number	2
D	День в году	Number	189
d	День в месяце	Number	10
F	День недели в месяце	Number	2
E	Название дня недели	Text	Tuesday; Tue
u	Номер дня недели (1 = Понедельник, ..., 7 = Воскресенье)	Number	1
a	am/pm	Text	PM
H	Час в дне (0-23)	Number	0

Таблица 28.2 *(окончание)*

Буква	Компонент даты и времени	Представление	Примеры
k	Час в дне (1–24)	Number	24
K	Час в дне am/pm (0–11)	Number	0
h	Час в дне am/pm (1–12)	Number	12
m	Минуты	Number	30
S	Секунды	Number	55
S	Миллисекунды	Number	978
z	Часовой пояс	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Часовой пояс	RFC 822 time zone	-0800
X	Часовой пояс	ISO 8601 time zone	-08; -0800; -08:00

Описание столбца "Представление":

- Text: если в шаблоне 4 буквы или более, то используется полная форма, в противном случае — сокращенная. При парсинге принимаются обе формы, независимо от количества букв в шаблоне.
- Number: количество букв в шаблоне — это минимальное количество цифр, более короткие числа добиваются нулями. При парсинге количество букв игнорируется, если только оно не требуется для разделения соседних полей.
- Year: если Calendar форматировщика является григорианским календарем, то применяются следующие правила. При форматировании, если количество букв равно двум, год усекается до двух цифр, в противном случае интерпретируется как число. При парсинге, если количество букв больше двух, год интерпретируется буквально, независимо от количества цифр. Поэтому при использовании шаблона "MM/dd/yyyy" и строки "01/11/12" получается 11 января 12 года нашей эры. При парсинге с сокращенной формой года ("y" или "yy") `SimpleDateFormat` интерпретирует сокращенный год относительно какого-либо века. Он выравнивает даты так, чтобы они были в диапазоне от 80 лет до даты создания `SimpleDateFormat` и до 20 лет после даты создания `SimpleDateFormat`. Во время парсинга только строки, состоящие строго из двух цифр, интерпретируются в текущий век. Любые другие числовые строки, состоящие из одной цифры или трех и более, интерпретируются как полный год.
- Month: если количество букв в шаблоне равно 3 или более, то месяц интерпретируется как текст, в противном случае интерпретируется как число. Буква `M` создает имена месяцев, зависимые от контекста. Если в конструктор был передан `DateFormatSymbols` или был использован метод `setDateFormatSymbols`, то имена месяцев берутся из `DateFormatSymbols`. Буква `L` создает самостоятельную форму имен месяцев.

- General time zone: часовые пояса интерпретируются по текстовым именам. При использовании смещения часовой пояс указывается в виде GMT +01:30 или GMT-12:33.
- RFC 822 time zone: используются четыре цифры: -0800 или +1200.
- ISO 8601 time zone: используются две цифры, четыре цифры или с разделением часов и минут двоеточием: -08; -0800; -08:00.

28.7. Класс PrintStream

Класс `java.io.PrintStream` позволяет писать форматированные данные в любой поток. Вам вряд ли когда-нибудь придется создавать экземпляры этого класса вручную, гораздо чаще вы будете использовать готовые классы, вроде возвращаемых System.out. PrintStream имеет методы `print` и `println`, перегруженные для любого примитивного типа и для класса `Object` (в этом случае используется его метод `toString()`).

Класс `PrintStream` никогда не бросает `IOException`, вместо этого он устанавливает свой внутренний флаг, который может быть проверен с помощью метода `public boolean checkError()`.

Особого внимания заслуживают методы:

```
public PrintStream format(String format,
                          Object... args)

public PrintStream format(Locale l,
                          String format,
                          Object... args)

public PrintStream printf(Locale l,
                          String format,
                          Object... args)

public PrintStream printf(String format,
                          Object... args)
```

Эти методы позволяют писать в поток форматированные данные. Здесь `format` — это шаблон строки, который подробно описывается в *разделе 28.8 "Класс Formatter"*.

28.8. Класс Formatter

Класс `java.util.Formatter` используется во всех методах, принимающих строку форматирования: `java.io.PrintStream.format`, `System.out.format`, `String.format` и т. д.

Каждый метод, принимающий строку форматирования, нуждается в шаблоне и списке аргументов. Пример:

FormatterExample.java

```
Calendar c = Calendar.getInstance();
c.set(2000, Calendar.FEBRUARY, 12, 10,0, 0);
String s = String.format("Джонни родился: %1$te.%1$tm.%1$tY", c);
System.out.println(s);
```

Строка форматирования является первым аргументом метода `format`. Она содержит три спецификатора формата: "`%1$tm`", "`%1$te`" и "`%1$tY`", которые указывают на способ обработки аргументов и на то, как они будут вставлены в текст. Остальные части строки содержат фиксированный текст, включающий в себя "Джонни родился: " и любые другие пробелы и знаки препинания. Список аргументов состоит из всех аргументов, переданных в метод после строки форматирования. В примере выше список аргументов содержит только один объект `java.util.Calendar`.

Спецификаторы формата для общих, символьных и числовых типов имеют следующий синтаксис:

```
%[argument_index$][flags][width][.precision]conversion
```

Необязательный `argument_index` показывает позицию аргумента в списке. К первому аргументу ссылаются "`1$`", ко второму — "`2$`", нумерация аргументов начинается с единицы.

- Необязательный `flags` — это набор символов, модифицирующих выходной формат. Допустимый набор символов зависит от `conversion`.
- Необязательный `width` — это положительное десятичное число, указывающее минимальное количество символов, которое будет записано в выходную строку.
- Необязательный `precision` — это неотрицательное десятичное число, обычно используемое для ограничения количества символов. Поведение `precision` зависит от `conversion`.
- Обязательное `conversion` — это символ, указывающий на способ форматирования аргумента. Допустимый набор символов зависит от типа аргумента.

Спецификаторы форматов, используемые для дат и времени, имеют следующий синтаксис:

```
%[flags][width]conversion
```

Бывают следующие виды `conversion`:

- `General` — может быть применен к любому типу аргумента.
- `Character` — может быть применен к базовым типам, представляющим символы Юникода: `char`, `Character`, `byte`, `Byte`, `short`, `Short`. Этот тип `conversion` может быть также применен к типам `int` и `Integer`, если `Character.isValidCodePoint(int)` возвращает `true`.
- `Integral` — может быть применен к любому целочисленному типу Java: `byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`, `BigInteger` (но не к `char` или `Character`).

- `Floating point` — может быть применен к типам с плавающей точкой: `float`, `Float`, `double`, `Double` и `BigDecimal`.
- `Date/Time` — применяется к типам Java, которые могут содержать дату или время: `long`, `Long`, `Calendar`, `Date` или `TemporalAccessor`.
- `Percent` — создает литеру '`%`' (`'\u0025'`).
- `Line separator` — специфичный для платформы разделитель строк.

Следующая таблица (табл. 28.3) содержит поддерживаемые `conversion`. Заглавные буквы имеют то же значение, что и прописные, но результат преобразуется в верхнем регистре с помощью `String.toUpperCase()`.

Таблица 28.3. *Результаты форматирования*

Conversion	Категория аргумента	Описание
'b', 'B'	general	Если аргумент `null`, то результат `false`. Если аргумент `boolean` или `Boolean`, то результатом будет результат вызова `String.valueOf(arg)`. В противном случае результат "`true`"
'h', 'H'	general	Если аргумент `null`, то результатом будет "`null`". В противном случае результат получается вызовом `Integer.toHexString(arg.hashCode())`
's', 'S'	general	Если аргумент `null`, то результат "`null`". Если аргумент реализует интерфейс java.util.Formattable, то вызывается метод arg.formatTo. В противном случае результат получается вызовом `arg.toString()`
'c', 'C'	character	Результатом будет символ Юникода
'd'	integral	Результат форматируется как десятичное целое
'o'	integral	Результат форматируется как восьмеричное целое
'x', 'X'	integral	Результат форматируется как шестнадцатеричное целое
'e', 'E'	floating point	Результат форматируется как десятичное число в научной записи
'f'	floating point	Результат форматируется как десятичное число
'g', 'G'	floating point	Результат форматируется, используя научную запись или десятичный формат, в зависимости от точности и значения после округления
'a', 'A'	floating point	Результат форматируется в шестнадцатеричное число с плавающей точкой, мантиссой и показателем степени. Не поддерживается для `java.math.BigDecimal`
't', 'T'	date/time	Префикс для даты и времени
'%'	percent	Результатом будет символ '`%`' (`'\u0025'`)
'n'	line separator	Разделитель линий, принятый в текущей платформе

Следующие символы используются в качестве суффиксов к 't' и 'T' и применяются для дат и времени (табл. 28.4).

Таблица 28.4. *Значения символов*

'H'	Час в 24-часовом дне. Форматируется как две цифры с предваряющим нулем, если нужно, т. е. 00–23
'I'	Час от единицы до двенадцати. Форматируется как две цифры с предваряющим нулем, если нужно, т. е. 01–12
'k'	Час в 24-часовом дне, 0–23
'l'	Час от единицы до двенадцати, 1–12
'M'	Минуты в часе. Форматируются как две цифры с предваряющим нулем, если нужно, т. е. 00–59
'S'	Секунды в минуте, форматируются как две цифры с предваряющим нулем, если нужно, т. е. 00–60 ("60" — специальное значение, необходимое для поддержки секунды координации)
'L'	Миллисекунды в секунде. Форматируются как три цифры с предваряющими нулями, если нужно, т. е. 000–999
'N'	Наносекунды в секунде. Форматируются как девять цифр с предваряющими нулями, если нужно, т. е. 000000000–999999999
'p'	До обеда или после обеда ("am" или "pm"). Используйте префикс 'T', чтобы результат был в верхнем регистре
'z'	RFC 822 смещение часовой зоны от GMT, например –0800. Это значение выравнивается для учета перехода на зимнее/летнее время. Для long, Long, Date используется часовой пояс по умолчанию для текущего экземпляра виртуальной машины Java
'Z'	Строка, содержащая аббревиатуру часового пояса. Это значение выравнивается, чтобы учесть переход на зимнее/летнее время. Для long, Long, Date используется часовой пояс по умолчанию для текущего экземпляра виртуальной машины Java
's'	Секунды с начала эпохи от 1 января 1970 года 00:00:00 UTC. От Long.MIN_VALUE/1000 до Long.MAX_VALUE/1000
'Q'	Миллисекунды с начала эпохи 1 января 1970 года 00:00:00 UTC. От Long.MIN_VALUE до Long.MAX_VALUE

Символы для форматирования дат — табл. 28.5.

Таблица 28.5. *Значения символов*

'B'	Полное имя месяца в соответствии с региональными настройками, например "январь", "февраль", "January", "February"
'b'	Сокращенное имя месяца в соответствии с региональными настройками, например "янв", "фев", "Jan", "Feb"
'h'	То же, что и 'b'
'A'	Полное имя дня недели в соответствии с региональными настройками, например "воскресенье", "понедельник", "Sunday", "Monday"
'a'	Короткое название дня недели в соответствии с региональными настройками, например, "Пн", "Вт", "Sun", "Mon"
'C'	Четыре цифры года, поделенные на 100 и форматированные как две цифры с предваряющим нулем, если нужно, т. е. 00–99

Таблица 28.5 *(окончание)*

'Y'	Год, форматированный как минимум четырьмя цифрами с предваряющим нулем, если нужно, т. е. 0092 равен 92 CE для григорианского календаря
'y'	Последние две цифры года, форматированные с предваряющим нулем, если необходимо, т. е. 00–99
'j'	День года, форматированный как три цифры с предваряющими нулями, если необходимо, т. е. 001–366 для григорианского календаря
'm'	Месяц, форматированный двумя цифрами с предваряющим нулем, если необходимо, т. е. 01–13
'd'	День месяца, форматированный как две цифры с предваряющими нулями, если необходимо, т. е. 01–31
'e'	День в месяце, форматированный как две цифры, т. е. 1–31

Символы для форматирования наиболее часто используемых сочетаний дат и времени см. в табл. 28.6.

Таблица 28.6. *Значения символов*

'R'	Время 24-часовой день "%tH:%tM"
'T'	Время в формате "%tH:%tM:%tS"
'r'	Время в формате "%tI:%tM:%tS %Tp"
'D'	Дата в формате "%tm/%td/%ty"
'F'	ISO 8601 форматированная дата "%tY-%tm-%td"
'c'	Дата и время в формате "%ta %tb %td %tT %tZ %tY"

Поддерживаемые флаги — табл. 28.7.

Таблица 28.7*. Флаги*

Флаг	General	Character	Integral	Floating Point	Date/Time	Описание
'-'	Да	Да	Да	Да	Да	Результат будет выровнен по левому краю
'#'	Да(1)	Нет	Да(3)	Да	Нет	Результат должен использовать альтернативную форму, зависимую от `conversion`
'+'	Нет	Нет	Да(4)	Да	Нет	Результат всегда будет содержать знак
' '	Нет	Нет	Да(4)	Да	Нет	Результат будет содержать лидирующий пробел для положительных значений
'0'	Нет	Нет	Да	Да	Нет	Результат будет выровнен нулями

Таблица 28.7 (окончание)

Флаг	General	Character	Integral	Floating Point	Date/Time	Описание
','	Нет	Нет	Да(2)	Да(5)	Нет	Результат будет содержать разделители групп указанной локали
'('	Нет	Нет	Да(4)	Да(5)	Нет	Результат будет заключать отрицательные значения в скобки

1 — зависит от определения `Formattable`.

2 — только для `conversion` 'd'.

3 — только для `conversion` 'o', 'x' и 'X'.

4 — для `conversion` 'd', 'o', 'x' и 'X', примененных к `java.math.BigInteger`, или 'd', примененной к `byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`.

5 — только для `conversion` 'e', 'E', 'f' , 'g' и 'G'.

`width` указывает минимальное количество символов, которые будут записаны в выходную строку. Для разделителя линий `width` не применяется.

`precision` указывает максимальное количество символов, которые будут записаны в выходную строку. Для `conversion` 'a', 'A', 'e', 'E', 'f' `precision` — количество символов после точки. Если `conversion` равен 'g' или 'G', то `precision` указывает общее количество значимых чисел после округления. Для `character`, `integral`, `date/time`, `percent`, `line separator` этот параметр не указывается.

`argument index` — десятичное число, указывающее позицию аргумента в списке аргументов. Первый аргумент "`1$`", второй — "`2$`" и т. д. Можно использовать флаг '`<`' ('`\u003c`'), который использует еще раз предыдущий аргумент.

28.9. Класс Scanner

Класс `java.util.Scanner` предназначен для разбиения форматированного ввода на токены и конвертирования токенов в соответствующий тип данных.

По умолчанию сканер использует пробельные символы (пробелы, табуляторы, разделители линий) для разделения токенов. Рассмотрите следующий код:

ScannerExample.java

```
package ru.urvanov.javaindynamics2022.formatparse;
import java.io.*;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) throws IOException {
```

```
        Scanner s = null;

        try {
            s = new Scanner(new BufferedReader(
                    new FileReader(
                            "scanner-data.txt",
                            StandardCharsets.UTF_8)));

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Если файл scanner-data.txt содержит следующий текст:

scanner-data.txt

```
За ужином объелся я,
А Яков запер дверь оплошно -
Так было мне, мои друзья,
И кюхельбекерно и тошно.
```

То результатом работы программы будет вывод:

```
За
ужином
объелся
я,
А
Яков
запер
дверь
оплошно
-
Так
было
мне,
мои
друзья,
И
кюхельбекерно
и
тошно.
```

Чтобы использовать другой разделитель токенов, примените метод `useDelimiter`, в который передается регулярное выражение. Например, предположим, что мы хотим использовать в качестве разделителя запятую, после которой может идти, а может не идти пробел:

```
s.useDelimiter(",\\s*");
```

Класс `java.util.Scanner` поддерживает все примитивные типы Java, `java.math.BigInteger` и `java.math.BigDecimal`. Scanner использует экземпляр `java.util.Locale` для преобразования строк в эти типы данных.

Пример:

ScannerNumber.java

```
package ru.urvanov.javaindynamics2022.formatparse;

import java.util.Scanner;

public class ScannerNumber {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Ожидаем ввод с консоли сначала числа,
        // а затем вещественного числа
        int i = scanner.nextInt();
        double d = scanner.nextDouble();
    }
}
```

28.10. Задания

1. Создайте программу, выводящую числа в формате рублей, например для чеков. Используйте `DecimalFormat`.
2. Создайте программу, считывающую файл, содержащий строки с именами сотрудников и их датами рождения, разделенными запятыми.
3. Создайте программу, считывающую файл, содержащий информацию с одного чека в продуктовом магазине: название, цена, количество, стоимость. Каждая единица купленного товара располагается на одной строке. Значения разделены символом ";".

ГЛАВА 29

Работа с консолью

29.1. Теория

Стандартный поток ввода `java.lang.System.in`, стандартный поток вывода `java.lang.System.out`, стандартный поток ошибок `java.lang.System.err` — все эти потоки по умолчанию обычно работают с консолью, но в некоторых случаях могут быть связаны с файлом или еще чем-нибудь. Потоки `java.lang.System.out` и `java.lang.System.err` являются экземплярами класса `java.io.PrintStream`. Входной поток `java.lang.System.in` является экземпляром `java.io.InputStream`. Мы уже рассматривали все эти классы и видели примеры их использования.

Класс `java.io.Console` содержит методы для доступа к консоли, если есть консоль, связанная с текущей виртуальной машиной.

Имеет ли виртуальная машина консоль или нет, зависит от платформы и способа запуска виртуальной машины. Если виртуальная машина запущена со стандартной интерактивной консоли без перенаправления стандартных потоков ввода и вывода, то консоль есть, и обычно она связана с клавиатурой и экраном, из которого запущена виртуальная машина. Если виртуальная машина была запущена автоматически, например как фоновый процесс, то обычно она не имеет консоли.

Если виртуальная машина имеет консоль, то может быть получен уникальный экземпляр этого класса с помощью вызова `java.lang.System.console()`. Если нет доступной консоли, то метод вернет `null`.

> ***Примечание***
>
> При запуске в IDE метод `java.lang.System.console()` возвращает `null`. Примеры, использующие его, нужно запускать из консоли самой ОС, а не из IDE.

Операции ввода и вывода синхронизированы, чтобы гарантировать атомарное выполнение критических операций, поэтому вызов методов `readLine()`, `readPassword()`, `format()`, `printf()`, а также операции чтения, форматирования и записи из объектов, возвращенных из `reader()` и `writer()`, могут привести к блокировке при многопоточности.

Вызов `close()` для объектов, возвращенных из `reader()` и `writer()`, не закрывает поток этих объектов.

Методы чтения из консоли возвращают `null`, если был достигнут конец потока ввода, например ввод Ctrl+D в Unix или Ctrl+Z в Windows. Последующие операции

чтения будут успешны, если в консоль позже были введены дополнительные символы.

Если приложению нужно считать пароль или любую другую секретную информацию, то оно должно использовать `readPassword()` или `readPassword(String, Object..)`, а затем после обработки вручную забить пробелами считанный пароль, чтобы минимизировать время пребывания секретных данных в памяти.

ReadPassword.java

```
package ru.urvanov.javaindynamics2022.console;

import java.io.Console;

// Этот код запускать нужно НЕ из IDE.
// Иначе System.console будет возвращать null
public class ReadPassword {
    public static void main(String[] args) {
        Console cons;
        char[] passwd;
        if ((cons = System.console()) != null &&
                (passwd = cons.readPassword(
                        "[%s]", "Password:")) != null) {
            // ...
            java.util.Arrays.fill(passwd, ' ');
        }
    }
}
```

Подробнее про очистку паролей из памяти после использования можно прочесть в статье на моем сайте: **https://urvanov.ru/2019/02/06/remove-password-from-memory-in-java-after-using/**.

Полезные методы класса `java.io.Console`:

```
public void flush()
```

Принудительно очищает буферы, выводя содержащуюся в них информацию на экран.

```
public Console format(String fmt,
                      Object... args)
```

Форматированный вывод в консоль. Смотрите описание `java.util.Formatter` в *главе 28 "Форматирование и парсинг"*.

```
public Console printf(String format,
                      Object... args)
```

Форматированный вывод в консоль. Смотрите описание `java.util.Formatter` в *главе 28 "Форматирование и парсинг"*.

```
public Reader reader()
```

Возвращает `java.io.Reader`, связанный с консолью. Смотрите описание в *главе 20 "Потоки ввода/вывода"*.

```
public String readLine()
```

Читает одну строку текста из консоли.

```
public String readLine(String fmt,
                       Object... args)
```

Выводит форматированную строку в консоль, а затем считывает одну строку введенного пользователем текста.

```
public char[] readPassword()
```

Используется для считывания паролей из консоли. Не отображает вводимые пользователем символы. Обратите внимание, что возвращается массив `char`-ов, это сделано специально из соображений безопасности.

```
public char[] readPassword(String fmt,
                           Object... args)
```

Выводит форматированную строку, а затем считывает пароль с экрана. Не отображает вводимые пользователем символы.

```
public PrintWriter writer()
```

Возвращает `java.io.PrintWriter`, связанный с этой консолью. Смотрите описание `java.io.PrintWriter` в *главе 20 "Потоки ввода/вывода"*.

29.2. Задание

Создайте программу, хранящую список логинов, паролей и дат рождения в массиве. При запуске программа должна запрашивать логин и пароль, а в случае успешного ввода выводить в консоль дату рождения и возраст.

ГЛАВА 30

Локализация

30.1. Теория

Локализация частично уже рассматривалась в *главе 27 "Дата и время"* и в *главе 28 "Форматирование и парсинг"*.

В этой статье будет рассмотрена локализация ресурсов приложения: строк, картинок, аудиофайлов и т. д.

Для разделения ресурсов, специфичных для каждого языка, страны или региона, используется класс `java.util.ResourceBundle` или один из его потомков `java.util.ListResourceBundle`, `java.util.PropertyResourceBundle`.

Концептуально каждый `java.util.ResourceBundle` — это набор подклассов, которые используют одно и то же базовое имя. Например, пусть `ButtonLabel` — базовое имя. Символы, которые следуют за базовым именем, указывают код языка, код страны и вариант `Locale`. Например, `ButtonLabel_en_GB` указывает на английский язык (en) и Великобританию (GB).

```
ButtonLabel
ButtonLabel_de
ButtonLabel_en_GB
ButtonLabel_fr_CA_UNIX
```

Чтобы выбрать подходящий `ResourceBundle`, вызовите метод `ResourceBundle.getBundle`. Следующий пример выбирает `ButtonLabel ResourceBundle` для `Locale` с французским языком, страной Канада и платформой UNIX:

```
Locale currentLocale = new Locale("fr", "CA", "UNIX");
ResourceBundle introLabels = ResourceBundle.getBundle(
                                "ButtonLabel", currentLocale);
```

Если класса `ResourceBundle`, подходящего для указанной `Locale`, нет, то `getBundle` пытается найти наиболее близкое совпадение. Например, если ищется `ButtonLabel_fr_CA_UNIX`, а `Locale` по умолчанию `en_US`, то `getBundle` ищет классы в следующем порядке:

```
ButtonLabel_fr_CA_UNIX
ButtonLabel_fr_CA
ButtonLabel_fr
ButtonLabel_en_US
```

```
ButtonLabel_en
ButtonLabel
```

Заметьте, что метод `getBundle` ищет классы, основанные на `Locale` по умолчанию, перед поиском базового класса. Если `getBundle` не может найти совпадения в этом списке классов, то он бросает исключение `java.util.MissingResourceException`. Всегда создавайте базовый класс, чтобы избежать подобного исключения.

Абстрактный класс `java.util.ResourceBundle` имеет два дочерних класса: `java.util.PropertyResourceBundle` и `java.util.ListResourceBundle`.

Класс `PropertyResourceBundle` использует файлы настроек (properties) для хранения текста. Эти файлы не являются частью кода Java, и они могут содержать только объекты `String`.

Класс `ListResourceBundle` использует список локализованных ресурсов, хранящихся в классах Java.

Класс `java.util.ResourceBundle` очень гибкий. Если вы сначала использовали `PropertyResourceBundle`, чтобы хранить локализованные строки в файлах properties, а позже решили использовать `ListResourceBundle`, то это не отразится на коде. Например, следующий `getBundle` получает `ResourceBundle` для `Locale` независимо от способа хранения этого `ResourceBundle`.

```
ResourceBundle introLabels = ResourceBundle.getBundle(
                                    "ButtonLabel", currentLocale);
```

Метод `getBundle` сначала ищет класс с указанным базовым именем, а затем, если его не находит, ищет файл properties.

Класс `ResourceBundle` содержит массив пар ключ-значение. Вы указываете ключ, который должен быть `String`, когда вам нужно достать значение из `ResourceBundle`. Пример:

ButtonLabel.java

```
package ru.urvanov.javaindynamics2022.localization;

import java.util.ListResourceBundle;

class ButtonLabel extends ListResourceBundle {
    public ButtonLabel() {
    }

    // English version
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
            {"OkKey", "OK"},
            {"CancelKey", "Cancel"},
    };
}
```

ButtonLabel_ru.java

```
package ru.urvanov.javaindynamics2022.localization;

import java.util.ListResourceBundle;

class ButtonLabel_ru extends ListResourceBundle {
    public ButtonLabel_ru() {
    }

    // Russian version
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
            {"OkKey", "Угу"},
            {"CancelKey", "Отмена"},
    };
}
```

Получение значения:

LocalizationExample.java

```
package ru.urvanov.javaindynamics2022.localization;

import java.util.Locale;
import java.util.ResourceBundle;

public class LocalizationExample {
    public static void main(String[] args) {
        ResourceBundle buttonLabels = ResourceBundle.getBundle(
                "ru.urvanov.javaindynamics2022.localization.ButtonLabel",
                new Locale("ru", "RU"));
        String okLabel = buttonLabels.getString("OkKey");
        System.out.println(okLabel);
    }
}
```

30.2. Задание

Ваших знаний уже должно быть достаточно для написания небольшой консольной игры с поддержкой нескольких языков, форматированным выводом сообщений. Создайте подобную игру. В игре персонаж игрока сражается с монстрами и повышает свой уровень. Игра должна поддерживать как минимум два языка: английский и русский. Игрок может путешествовать по миру с помощью консольных команд:

North, East, South, East. После перемещения игра выводит текст с описанием ячейки, в которой он оказался. В ячейке могут быть монстры, с которыми можно сразиться командой fight, предметы, которые можно подобрать командой `pick`, сундуки, которые можно открыть командой `open`. Сражение с монстрами происходит посредством игры "камень-ножницы-бумага". Проигравший теряет одно очко здоровья. Если заканчивается здоровье монстра, то он погибает, а из него выпадают предметы: оружие, зелья лечения и т. д. Если заканчивается здоровье игрока, то игра считается проигранной. Вы можете расширить это задание в соответствии с вашим энтузиазмом и воображением, например добавив квесты, подземелья и т. д.

ГЛАВА 31

Пример сервиса со Spring

31.1. Что за сервис мы напишем

Если вы посмотрите современные вакансии, то увидите, что знания только Java без библиотек и фреймворков не особо полезны. Никто не пишет приложения на чистой Java. Для того чтобы найти работу, нужно знать о целой экосистеме библиотек, утилит, фреймворков и подходов к разработке. Java в большинстве случаев используется совместно со Spring Framework (хотя есть и другие фреймворки, но на текущий момент они не так популярны). Также нужно знать Maven либо Gradle, какую-нибудь базу данных, Spring Data, Hibernate, иметь общее представление о протоколе HTTP и много чего еще.

> ***Важно***
>
> Информация в этой главе может служить отправной точкой в изучении экосистемы Java, но никак не полноценным источником информации о том, что нужно делать и как. Сама Java и мир созданных для этого языка библиотек и фреймворков очень обширны и никоим образом не могут быть уложены в одну книгу или в какие-нибудь жесткие рамки. В процессе чтения этой главы рекомендуется обращаться к официальной документации: **https://spring.io**, **https://junit.org**, **https://maven.apache.org**, **https://gradle.org, https://projectreactor.io** и др., которая вам может потребоваться.

В этом разделе мы попытаемся создать небольшое приложение в качестве примера. Представьте, что вы работаете в фирме, которая создает конструктор виртуального

мира. Мы создадим сервис по размещению существ на карте мира. Для упрощения наш функционал будет состоять из четырех операций:

- Размещение существа на карте в заданных координатах с заданными параметрами.
- Обновление координат и параметров существа по ID.
- Получение информации о существе по ID.
- Удаление существа по ID.

Сам Spring Framework очень большой, и даже внутри него существует множество разных подходов к разработке приложений. В рамках нашего примера мы будем использовать Spring вместе со Spring WebFlux. Сама информация о размещенных в мире существах будет храниться в базе данных H2 в памяти приложения.

31.2. Spring Initializr

Скелет приложения создается с помощью сервиса Spring Initializr. Можно воспользоваться веб-версией **https://start.spring.io**. Она прекрасно сгенерирует скелет приложения в соответствии с вашими настройками, а затем этот сгенерированный скелет можно будет импортировать в вашу IDE.

> На сайте **https://urvanov.ru** можно найти инструкцию по использованию Spring Initializr для Eclipse, NetBeans, IntelliJ IDEA с цветными скриншотами.

Аналогично можно использовать Spring Initializr и из самой IntelliJ IDEA, но это поддерживается только в Ultimate версии IDEA, в версии Community такой возможности по умолчанию нет, но можно установить плагин стороннего разработчика, если есть желание. Плагин стороннего разработчика выглядит очень похоже.

Далее используется Ultimate версия IntelliJ IDEA. Да, это платная версия, но вам, скорее всего, выдадут на нее лицензию на работе, так что лично вам платить не придется.

Создаем новый проект на Spring (рис. 31.1). Для этого в Ultimate версии IntelliJ IDEA кликаем на **File \ New \ Project...**

Если вы используете многомодульный проект, как этой книге, то нужно создавать новый модуль внутри уже существующего проекта, кликнув на **File\New\Module...** (рис. 31.2)

В открывшемся диалоговом окне выберите тип проекта Spring **Initializr** и заполните значения, как на изображении на рис. 31.3.

- **Name** — название проекта.
- **Location** — каталог на диске, где будут храниться файлы проекта.
- **Language** — язык проекта. Нужно выбрать Java.
- **Type** — тип менеджера зависимостей. Чаще всего используется Maven, его и нужно выбрать.

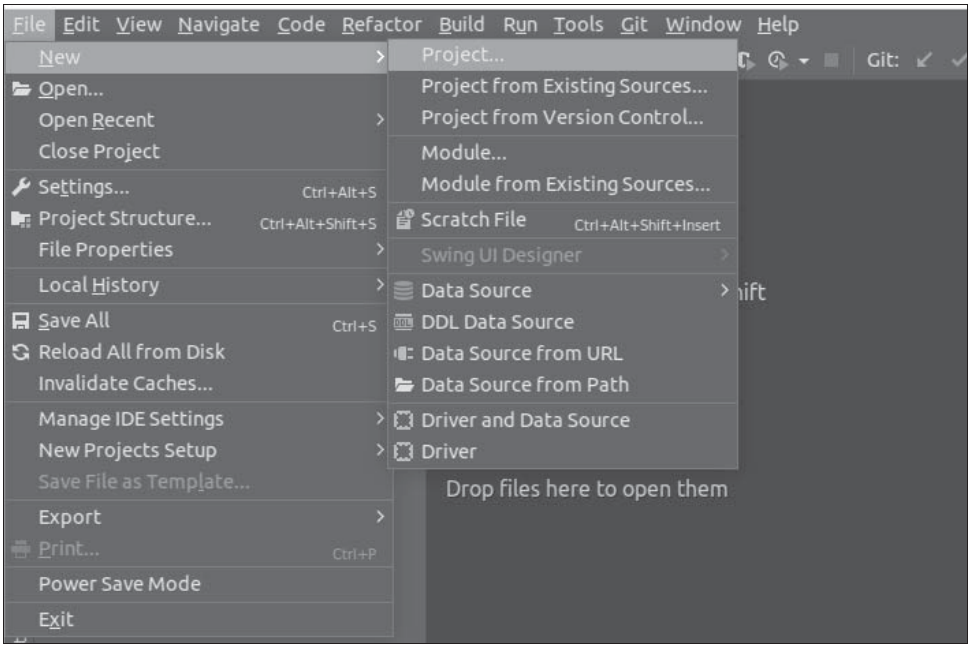


Рис. 31.1. Создание нового проекта в IntelliJ IDEA

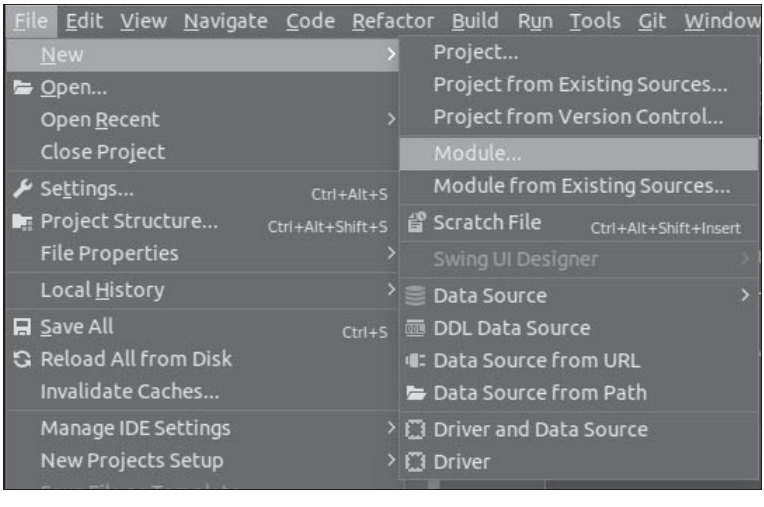


Рис. 31.2. Создание нового модуля в IntelliJ IDEA

- **Group** — имя группы Maven-проекта.
- **Artifact** — название артефакта. Конечный JAR-файл будет иметь название <artifact>-<версия>.jar.
- **Package name** — имя пакета, которое будет создано для проекта.
- **Project SDK** — выберите 17-ю версию, по которой и написана эта книга.
- **Java** — выберите 17.

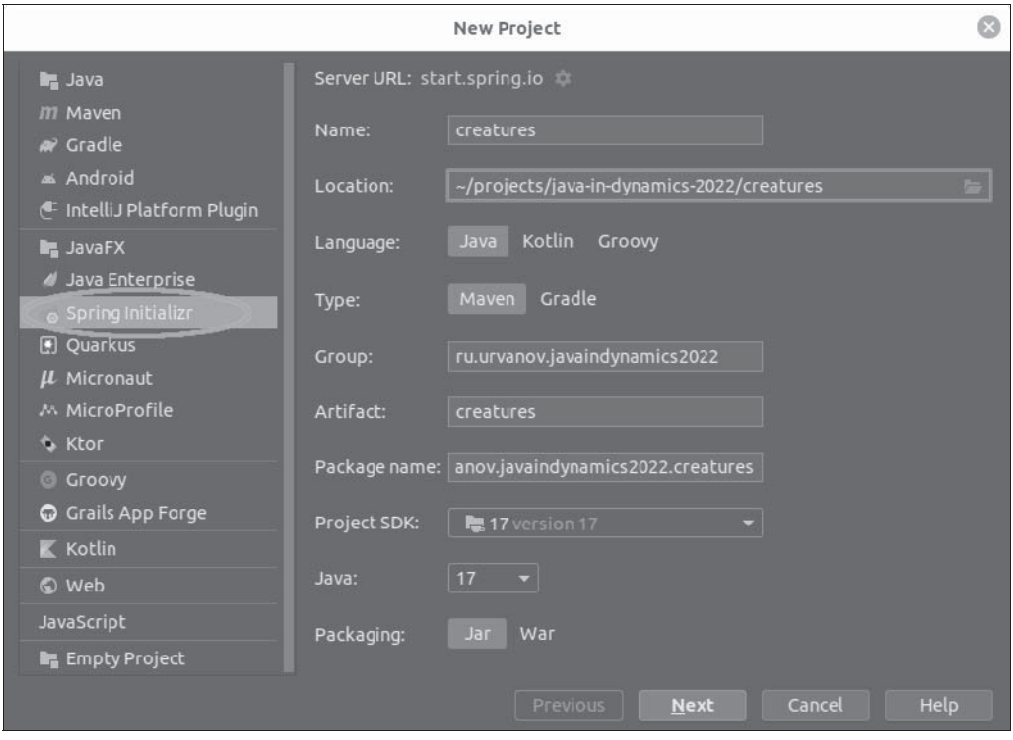


Рис. 31.3. Создание нового проекта Spring Boot с помощью Spring Initializr в IntelliJ IDEA

- **Packaging** — выберите jar. Раньше приложения Spring деплоились внутри контейнеров сервлетов наподобие Apache Tomcat, Wildfly, WebSphere и аналогичных. В одном экземпляре контейнера сервлетов деплоилось несколько веб-приложений (несколько war-файлов). В большинстве современных проектов используется Spring Boot, где конечные jar уже содержат внутри себя контейнер сервлетов и сами настраивают ваше приложение на запуск.

После заполнения полей кликните по кнопке **Next**. Откроется окно выбора зависимостей будущего проекта (рис. 31.4). Разные проекты будут использовать разные зависимости. Для нашего примера в левом дереве нужно проставить галочки на четырех пунктах:

- **Spring Reactive Web** (мы будем писать проект в реактивном стиле с Spring Web Flux).
- **Spring Data R2DBC** (для доступа к базе данных в реактивном стиле).
- **H2 Database** (встроенная база данных).
- **Validation** (для реализации функции проверки).

Кликните **Finish**.

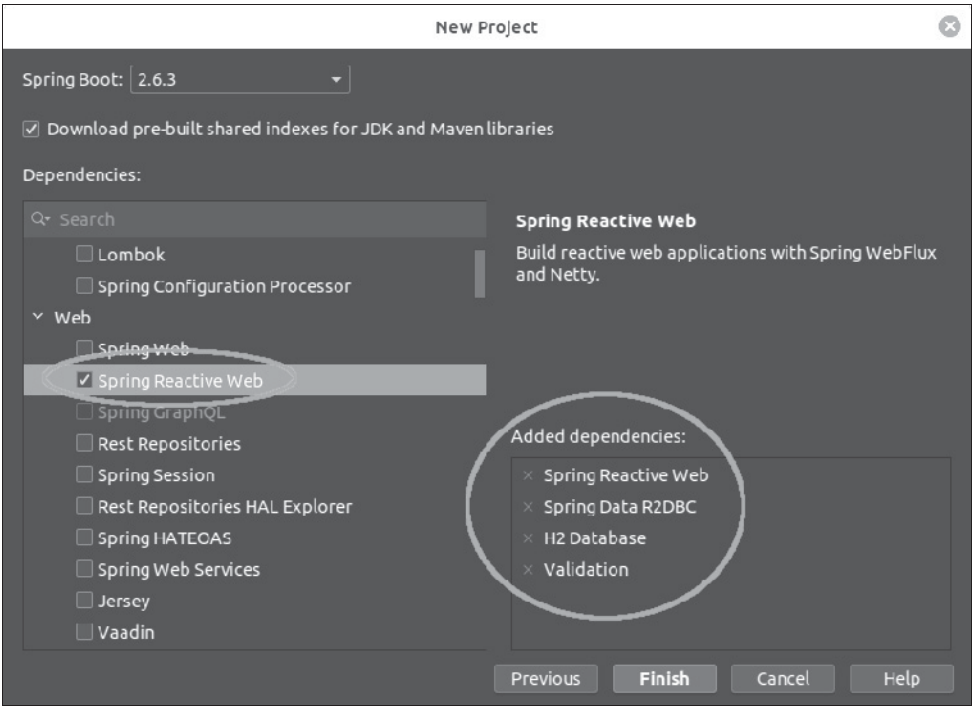


Рис. 31.4. Выбор зависимостей нового проекта Spring Initializr в IntelliJ IDEA

31.3. Разбор сгенерированного скелета приложения

Давайте посмотрим на то, что у нас получилось.

Обратите внимание на следующие файлы:

- .mvn/wrapper/maven-wrapper.jar;
- .mvn/wrapper/maven-wrapper.properties;
- mvnw;
- mvnw.cmd.

Вышеперечисленные файлы — это Maven Wrapper. Он позволяет запускать Maven, не устанавливая его в свою систему. Фактически это зафиксированная версия Maven, которая будет лежать в репозитории вместе с исходными кодами. Подобный подход позволит избежать проблем, связанных с совместимостью различных версий сборщика Maven и необходимостью установки нескольких версий в вашу систему.

Сборка и запуск проекта под Windows:

```
mvnw.cmd clean install
mvnw.cmd spring-boot:run
```

Сборка и запуск проекта под Linux (у вас должна быть заполнена переменная окружения JAVA_HOME значением с путем к каталогу с JDK):

```
./mvnw clean install
./mvnw spring-boot:run
```

Но это все с консоли. Мы же будем работать в основном в IDE, так что для нас Maven Wrapper не особо значим.

На самом деле вы можете запускать на исполнение класс CreaturesApplication из проекта точно так же, как и программу HelloWorld. Однако, если у вас есть IntelliJ IDEA Ultimate Edition, лучше воспользоваться специальной конфигурацией запуска приложений Spring Boot.

В правом верхнем углу IDEA найдите значок молотка. Справа от него будет ниспадающий список с существующими конфигурациями запуска и возможностью добавить новую конфигурацию (рис. 31.5).

При создании новой конфигурации запуска нужно выбрать тип **Spring Boot** (рис. 31.6).

Рис. 31.5. Run Configurations в IntelliJ IDEA

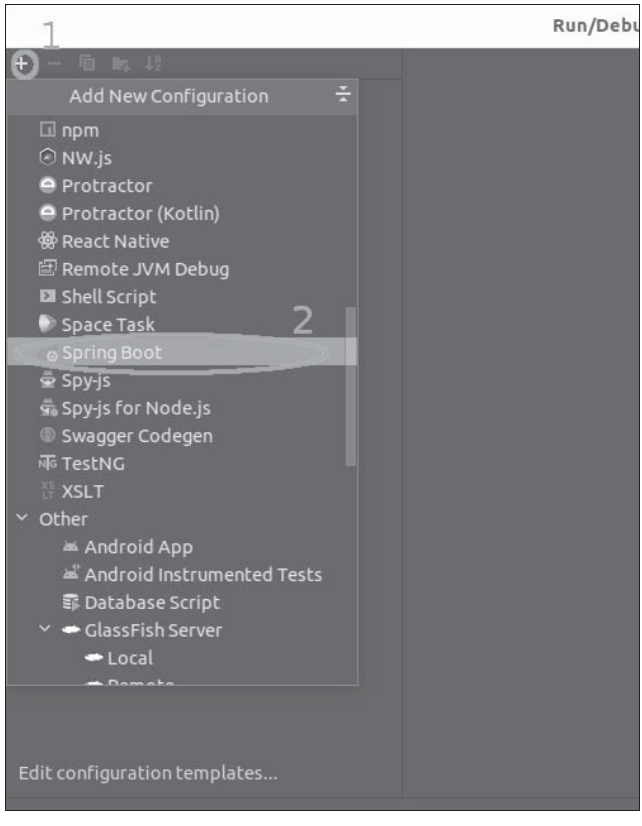


Рис. 31.6. Создание конфигурации для запуска приложения Spring Boot в IntelliJ IDEA

Откроется окно со специальными настройками, характерными для приложения Spring Boot (рис. 31.7).

В этом окне можно задать активные профили, параметры JVM, параметры самого приложения, а также огромное количество других настроек.

Для нас пока достаточно заполнить поля **Name** и **Main class**, как это сделано на рис. 31.7.

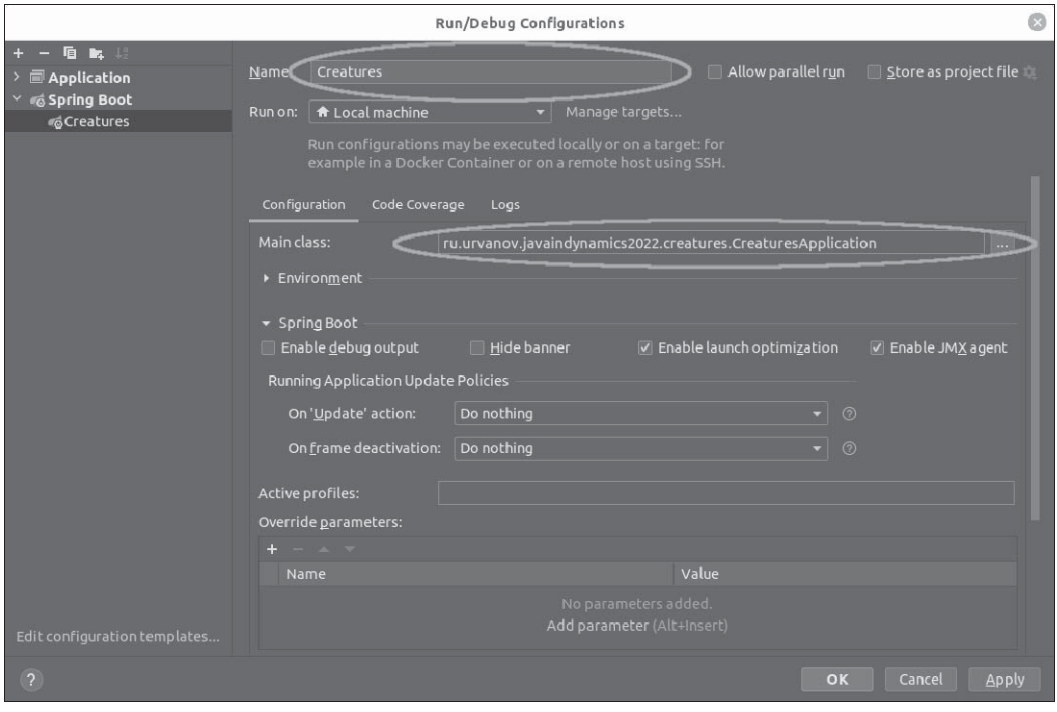


Рис. 31.7. Настройка конфигурации запуска приложения Spring Boot в IntelliJ IDEA

Для запуска нужно использовать кнопку, находящуюся справа от ниспадающего списка, в котором мы создавали новую конфигурацию запуска (рис. 31.8).

Рис. 31.8. Запуск созданного сервиса из IntelliJ IDEA

Пока наше приложение просто прослушивает порт 8080 и больше ничего не делает. Фактически у нас есть два важных файла: pom.xml и CreaturesApplication.java. Файл pom.xml содержит зависимости, которые автоматически добавились в наш проект в соответствии с тем, что мы указали в Spring Initializr.

Мы пока не будем просматривать зависимости. Посмотрите на сгенерированный файл CreaturesApplication.java:

CreaturesApplication.java

```
package ru.urvanov.javaindynamics2022.creatures;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CreaturesApplication {

    public static void main(String[] args) {
        SpringApplication.run(CreaturesApplication.class, args);
    }

}
```

Как видите, минимальное приложение на Spring Boot выглядит довольно просто. Особенно если сравнить его с тем, как мы раньше инициализировали Spring Framework с кучей XML-конфигураций и файлов настроек.

В качестве примера старой инициализации можете посмотреть пример приложения **https://github.com/urvanov-ru/keystore**. Особое внимание уделите файлам src/main/webapp/META-INF/context.xml и src/main/webapp/WEB-INF/web.xml из проекта keystore. Именно так раньше и настраивался Spring Framework.

31.4. Добавление конечных точек

Для каждой из операций с существом мы создадим по одной конечной точке. Создание существа в определенной точке будет происходить HTTP-методом POST, изменение его характеристик и координат — методом PUT, получение информации о существе — методом GET, а удаление существа будет происходить с помощью HTTP-метода DELETE.

Фактически нам нужно создать класс с четырьмя методами. Классам, которые обрабатывают HTTP-запросы, обычно дают название с суффиксом Controller. Пусть наш класс называется CreatureController, и пусть он располагается в пакете `ru.urvanov.javaindynamics2022.creatures.controller`:

CreaturesController.java

```
package ru.urvanov.javaindynamics2022.creatures.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/creatures")
public class CreaturesController {
}
```

Обратите внимание на аннотации `@RestController` и `@RequestMapping`.

Аннотация `@RestController` на самом деле состоит из двух аннотаций: `@Controller` и `@ResponseBody`.

`@Controller` указывает на то, что класс обрабатывает HTTP-запросы.

`@ResponseBody` указывает, что результат, возвращенный методом, должен стать телом ответа на запрос.

С помощью аннотации `@RequestMapping` мы указали, что методы нашего класса будут обрабатывать запросы к пути creatures.

Теперь напишем наши четыре метода:

CreaturesController.java

```
package ru.urvanov.javaindynamics2022.creatures.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/creatures")
public class CreaturesController {

    private static Logger logger
            = LoggerFactory.getLogger(CreaturesController.class);

    @PostMapping
    public void create() {
        logger.debug("Создание существа");
    }

    @PutMapping("{creatureId}")
    public void update(@PathVariable Long creatureId) {
        logger.debug("Обновление информации о существе");
    }

    @GetMapping("{creatureId}")
    public void info(@PathVariable Long creatureId) {
        logger.debug("Получение информации о существе");
    }

    @DeleteMapping("{creatureId}")
    public void delete(@PathVariable Long creatureId) {
        logger.debug("Удаление существа");
    }
}
```

Как видите, мы добавили четыре метода с соответствующими аннотациями:

- `@PostMapping` обрабатывает HTTP POST;
- `@PutMapping` обрабатывает HTTP PUT;
- `@GetMapping` обрабатывает HTTP GET;
- `@DeleteMapping` обрабатывает HTTP DELETE.

Аннотация `@PathVariable` указывает, что значение переменной будет заполняться из параметров запроса. Например, для метода `delete` URL-адреса будут такими:

```
creatures/1
creatures/34
creatures/99
```

Здесь 1, 34 и 99 — это значения, которые будут передаваться в параметр `creatureId` метода `delete`.

Вы можете прямо сейчас попробовать запустить проект на исполнение. Он начнет слушать порт 8080. Попытайтесь в браузере открыть страницу:

http://localhost:8080/creatures/3

Если вы поставите точку остановки на метод `info`, то увидите, что вы туда попадаете, метод `log.debug` отрабатывает, но в консоли пока ничего не происходит, т. к. у нас по умолчанию уровень логирования `info`.

> ***Примечание***
>
> Исходные коды готового сервиса creatures выложены на сайте **https://urvanov.ru**. Если по какой-либо причине у вас не получается запустить сервис на текущем этапе, то вы можете скачать готовый и посмотреть, чего вам не хватает.

А теперь обратите внимание на файл src/main/resources/application.properties. На текущий момент он пустой. В этом файле хранятся настройки, в том числе и настройки логирования. Мы можем включить уровень debug для нашего приложения, добавив туда строчку:

```
logging.level.ru.urvanov.javaindynamics2022=debug
```

После этого снова попытайтесь открыть страницу **http://localhost:8080/creatures/3**. В консоли будет выведено сообщение: "Получение информации о существе".

Настройку уровня логирования не обязательно заполнять в файле application.properties. Можно определить ее для конфигурации запуска в IntelliJ IDEA, как это показано на рис. 31.9.

31.5. Слой бизнес-сервисов

Обычно контроллеры только вызывают методы из слоя бизнес-сервисов, которые, собственно, и занимаются основной логикой.

Для нашего проекта достаточно одного сервиса. Логики особо никакой нет, нам нужно только сохранять данные в БД и вытаскивать из нее (так называемый CRUD — create, read, update, delete).

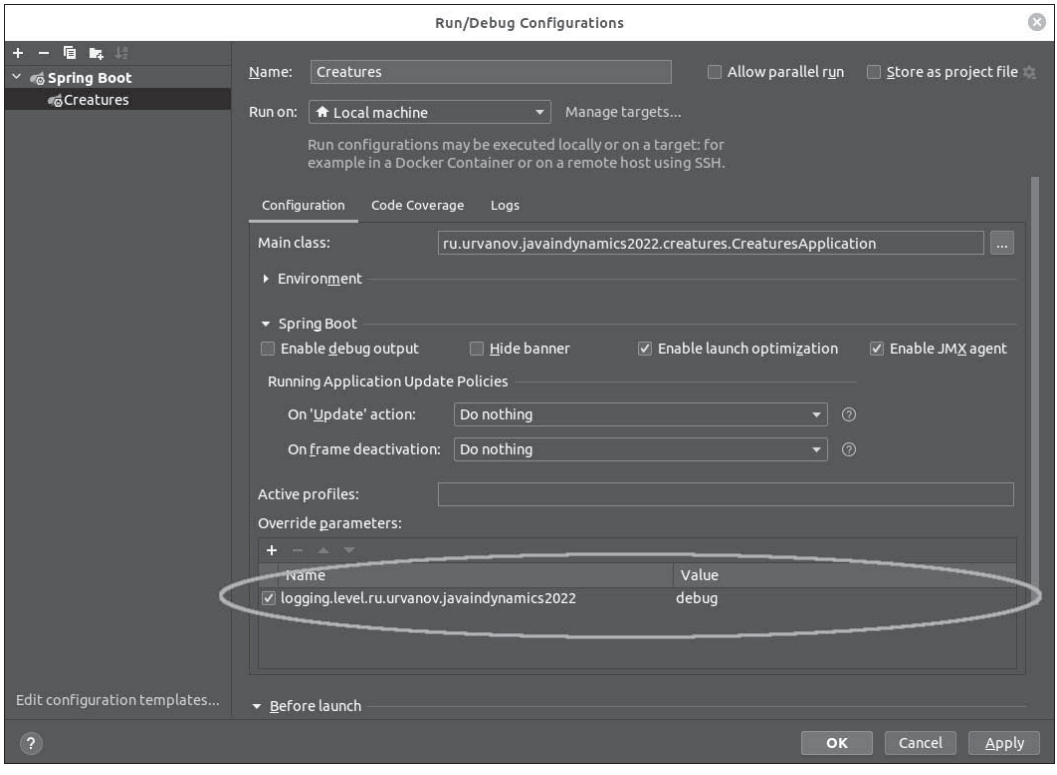


Рис. 31.9. Настройка уровня логирования

Поскольку мы уже дошли до слоя сервисов, то пора бы договориться о параметрах существ, которые мы будем хранить в БД. Класс "существо" будет иметь такой вид:

Creature.java

```
package ru.urvanov.javaindynamics2022.creatures.domain;

import java.util.Objects;

public class Creature {
    private long id;
    private double health;
    private double x;
    private double y;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }
```

```
    public double getHealth() {
        return health;
    }

    public void setHealth(double health) {
        this.health = health;
    }

    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 "Creature{" +
                "id=" + id +
                ", health=" + health +
                ", x=" + x +
                ", y=" + y +
                '}';
    }
}
```

Мы будем хранить идентификатор существа, уровень здоровья и координаты существа в мире. Все методы в классе просто сгенерированы IDE.

В идеале нам нужно бы было переопределить `equals` и `hashCode` таким образом, чтобы они использовали комбинацию уникальных полей из `Creature`. Но в нашем классе сейчас нет подходящих полей, к тому же мы не используем коллекции `Set` в нашем приложении, да и само приложение достаточно простое, поэтому проблем не возникнет.

> ***Важно***
>
> Методы `equals` и `hashCode` для сущностей нужно переопределять. В реальном приложении нужно переопределять `equals` и `hashCode` для сущностей таким образом, чтобы они использовали только неизменяемые уникальные поля, например идентификатор сотрудника.

Обратите внимание на наименование пакета, в котором хранится класс `Creature`. Мы храним его в подпакете `domain`. Обычно идет следующее разделение на пакеты:

- `controller` — для хранения контроллеров, обрабатывающих HTTP-запросы;
- `service` — для хранения классов, обрабатывающих бизнес-логику;
- `domain` — для хранения сущностей предметной области;
- `repository` или `dao` (data access objects) — для хранения классов, предназначенных для информации о состоянии сущностей.

В бизнес-слое для нашего случая будет только один класс `CreatureServiceImpl` вместе с интерфейсом `CreatureService`, который мы создадим в подпакете `service`.

CreatureService.java

```
package ru.urvanov.javaindynamics2022.creatures.service;

import reactor.core.publisher.Mono;
import ru.urvanov.javaindynamics2022.creatures.domain.Creature;

public interface CreatureService {
    Mono<Long> create(Creature creature);

    Mono<Creature> update(Creature creature);

    Mono<Creature> getById(Long creatureId);

    Mono<Void> delete(Long creatureId);
}
```

Реализацию пока напишем самую простую:

CreatureServiceImpl.java

```
package ru.urvanov.javaindynamics2022.creatures.service;

import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import ru.urvanov.javaindynamics2022.creatures.domain.Creature;

@Service
public class CreatureServiceImpl implements CreatureService {
    @Override
    public Mono<Long> create(Creature creature) {
        return Mono.empty();
    }

    @Override
    public Mono<Creature> update(Creature creature) {
```

```
        return Mono.empty();
    }

    @Override
    public Mono<Creature> getById(Long creatureId) {
        return Mono.empty();
    }

    @Override
    public Mono<Void> delete(Long creatureId) {
        return Mono.empty();
    }
}
```

В текущем варианте реализации методы пока ничего не делают.

Особое внимание обратите на аннотацию `@Service`. Она означает, что класс нужно добавить как бин в контекст Spring.

Для того чтобы разрабатывать приложения на Spring Framework, нужно иметь хотя бы общее представление о контексте Spring и его бинах. Мы не создаем сами экземпляры классов и не присваиваем сами значения зависимостям одного класса от другого. Вместо этого мы помещаем классы в контекст Spring и берем зависимости из него же.

Наш основной класс `CreaturesApplication` находится в пакете `ru.urvanov.javaindynamics2022.creatures`. Он помечен аннотацией `@SpringBootApplication`, которая фактически включает в себя три аннотации:

- `@EnableAutoConfiguration` включает автоматическую настройку Spring Boot, основываясь на jar-зависимостях, которые были добавлены.
- `@ComponentScan` включает сканирование классов в пакете, в котором расположен класс, и в подпакетах.
- `@SpringBootConfiguration` включает регистрацию дополнительных бинов в контексте для дополнительных конфигурационных классов.

Нам сейчас нужно обратить внимание на `@ComponentScan`. Он означает, что все классы пакета `ru.urvanov.javaindynamics2022.creatures`, помеченные аннотацией `@Component` и ее производными, будут добавлены в контекст Spring-а.

Аннотация `@Service` сама помечена как `@Component`. Фактически она делает абсолютно то же самое, что и `@Component`, и существует только для смыслового выделения бизнес-сервисов.

Аннотация `@RestController` помечена как `@Controller`, которая помечена как `@Component`, поэтому наш `CreaturesController` тоже попадет в контекст Spring Framework (рис. 31.10).

Теперь нам нужно добавить в `CreaturesController` вызовы методов `CreatureService`. Для этого используется аннотация `@Autowired`, которая производит поиск подходящего по типу бина и внедряет его:

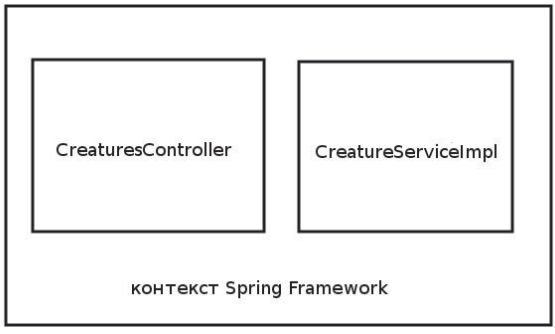


Рис. 31.10. Бины в контексте Spring Framework

CreaturesController.java

```
package ru.urvanov.javaindynamics2022.creatures.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Mono;
import ru.urvanov.javaindynamics2022.creatures.domain.Creature;
import ru.urvanov.javaindynamics2022.creatures.service.CreatureService;

@RestController
@RequestMapping("/creatures")
public class CreaturesController {

    private static Logger logger
            = LoggerFactory.getLogger(CreaturesController.class);


    @Autowired
    private CreatureService creatureService;

    @PostMapping
    public Mono<Long> create(@RequestBody Creature creature) {
        logger.debug("Создание существа");
        return creatureService.create(creature);
    }

    @PutMapping("{creatureId}")
    public Mono<Void> update(@PathVariable Long creatureId,
                                          @RequestBody Creature creature) {
        logger.debug("Обновление информации о существе");
        creature.setId(creatureId);
```

```
        creatureService.update(creature);
        return Mono.empty();
    }

    @GetMapping("{creatureId}")
    public Mono<Creature> info(@PathVariable Long creatureId) {
        logger.debug("Получение информации о существе");
        return creatureService.getById(creatureId);
    }

    @DeleteMapping("{creatureId}")
    public Mono<Void> delete(@PathVariable Long creatureId) {
        logger.debug("Удаление существа");
        return creatureService.delete(creatureId);
    }
}
```

Благодаря аннотации `@Autowired` в поле `creatureService` будет внедрен бин `CreatureService`. На самом деле это будет не сам класс `CreatureServiceImpl`, а бин, который реализует интерфейс `CreatureService` и в дополнение к своим действиям вызывает методы `CreatureServiceImpl` при вызове своих методов.

31.6. Работа с базой данных

Осталось доделать работу с базой данных. Мы используем Spring Data R2DBC с базой данных H2 Database, которая сохраняет данные в ОЗУ. В нашем приложении обычные методы чтения, сохранения, изменения и удаления данных. Они реализуются с помощью интерфейса `ReactiveCrudRepository`, от которого нам нужно унаследовать свой интерфейс:

CreatureRepository.java

```
package ru.urvanov.javaindynamics2022.creatures.repository;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import ru.urvanov.javaindynamics2022.creatures.domain.Creature;

public interface CreatureRepository
        extends ReactiveCrudRepository<Creature, Long> {
}
```

Когда при сканировании классов Spring обнаружит интерфейс, наследующийся от `ReactiveCrudRepository`, он создаст подходящий бин с реализацией базовых методов из этого интерфейса. Нам же достаточно использовать `CreatureRepository` из `CreatureServiceImpl`:

CreatureServiceImpl.java

```
package ru.urvanov.javaindynamics2022.creatures.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import reactor.core.publisher.Mono;
import ru.urvanov.javaindynamics2022.creatures.domain.Creature;
import ru.urvanov.javaindynamics2022.creatures.repository.CreatureRepository;

@Service
public class CreatureServiceImpl implements CreatureService {

    @Autowired
    private CreatureRepository creatureRepository;

    @Override
    @Transactional
    public Mono<Long> create(Creature creature) {
        return creatureRepository.save(creature).map(Creature::getId);
    }

    @Override
    @Transactional
    public Mono<Creature> update(Creature creature) {
        return creatureRepository.save(creature);
    }

    @Override
    @Transactional(readOnly = true)
    public Mono<Creature> getById(Long creatureId) {
        return creatureRepository.findById(creatureId);
    }

    @Override
    @Transactional
    public Mono<Void> delete(Long creatureId) {
        return creatureRepository.deleteById(creatureId);
    }
}
```

Мы просто вызываем методы из `CreatureRepository`. Сама реализация остается на Spring Data R2DBC.

Обратите внимание на аннотацию `@Transactional`. Она управляет транзакциями в нашем приложении. При вызове метода бина, помеченного этой аннотацией,

будет автоматически начата новая транзакция либо будет использована текущая транзакция, если она есть.

Следующим шагом нужно дать небольшие подсказки, чтобы поля класса `Creature` могли правильно мапиться на поля из H2 Database:

Creature.java

```
package ru.urvanov.javaindynamics2022.creatures.domain;

import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

import javax.annotation.processing.Generated;
import java.util.Objects;

@Table("creature")
public class Creature {
    @Id
    private long id;
    private double health;
    private double x;
    private double y;

...
```

Мы добавили две аннотации:

- `@Table` для указания наименования таблицы;
- `@Id` для указания, что поле `id` является первичным ключом.

Также нужно создать таблицу creature в базе данных. Hibernate может генерировать сами необходимые таблицы, но в реальных приложениях эту возможность не используют. Скрипты создания разместим в src/main/resources/schema.sql:

schema.sql

```
CREATE TABLE creature (
    id SERIAL PRIMARY KEY,
    health DOUBLE PRECISION,
    x DOUBLE PRECISION,
    y DOUBLE PRECISION
);

INSERT INTO creature(health, x, y)
VALUES(100.0, 10.2, -3.4);
```

```
INSERT INTO creature(health, x, y)
VALUES(200.0, 50.2, -34.4);

INSERT INTO creature(health, x, y)
VALUES(150.0, 3.2, 54.4);
```

31.7. Вызов методов с Postman

В этом разделе нам понадобится установить Postman с сайта **https://www.postman.com**. Для обучения хватает бесплатной версии.

Дальше нам нужно проверить наши методы, вызвав их на выполнение. Внутри проекта рядом с файлом pom.xml лежит файл creatures-postman.json. Импортируйте его содержимое в Postman через **File ➢ Import**. В коллекции запросов должен появиться пункт **java-in-dynamics-2022-creatures** как минимум с четырьмя запросами: **get**, **create**, **update**, **delete** (рис. 31.11). Каждый из них проверяет один из методов нашего сервиса.

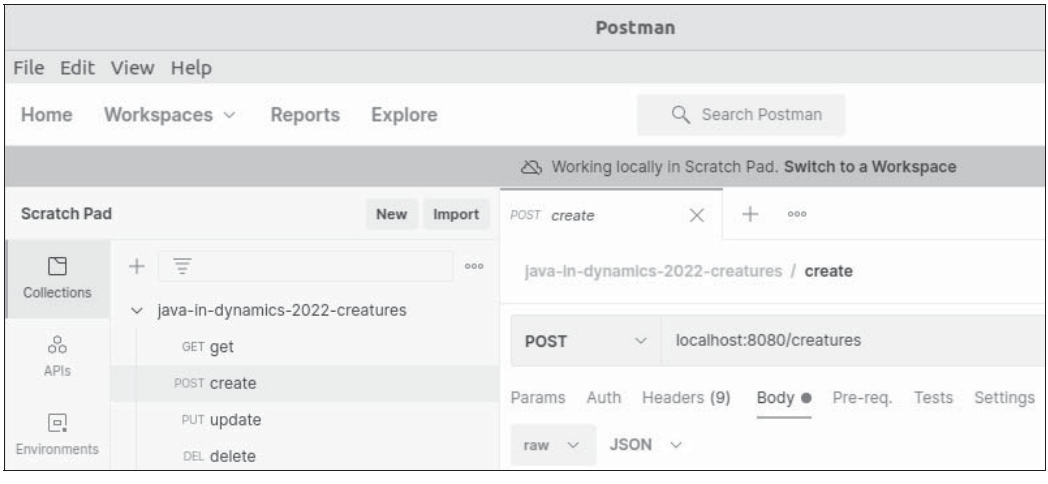


Рис. 31.11. Тестирование методов сервиса с Postman

31.8. Docker

В большинстве случаев приложение Spring Boot упаковывается в какой-нибудь контейнер вместе со всеми своими зависимостями, это облегчает его развертывание и запуск. Одной из платформ для разработки и запуска контейнерных приложений является Docker.

Docker состоит из следующих компонентов:

- Docker daemon — это сервис, работающий в фоновой режиме, через который осуществляется все взаимодействие с контейнерами.
- Docker client — интерфейс командной строки для отправки команд в Docker daemon.

- Docker image (образ) — неизменяемый образ, из которого разворачиваются контейнеры, содержащие ОС и запускаемое приложение.
- Docker container — работающее приложение, развернутое из образа. Его можно представлять как маленький виртуальный компьютер с какой-нибудь операционной системой (чаще всего Linux) и нашим приложением.
- Docker registry (реестр) — репозиторий с образами. Есть публичные и приватные реестры. Распространенный публичный репозиторий **https://hub.docker.com**.
- Dockerfile — инструкция для сборки образа. Простой текстовый файл, содержащий команды.

Docker можно использовать и под Linux, и под Windows, и под MacOS.

Для начала нам нужно установить и настроить Docker согласно инструкции на официальном сайте (**https://docs.docker.com/engine/install/**). Установка и настройка сильно отличаются в зависимости от вашей операционной системы.

После установки Docker на нашем компьютере добавим поддержку создания Docker image в наш проект. В самом простом варианте нам достаточно добавить файл Dockerfile со следующим содержимым:

Dockerfile

```
FROM openjdk:17-alpine
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

После этого мы уже можем создавать Docker image и запускать контейнер на исполнение.

Сначала давайте соберем наше приложение, чтобы у нас были итоговые файлы для запуска Docker, для этого нужно либо выполнить команду:

```
./mvn package
```

из каталога с исходными кодами, либо кликнуть на вкладке **Maven** в правой части IntelliJ IDEA и запустить цель package (рис. 31.12).

Сборка docker image и простановка метки `urvanovru/java-in-dynamics-2022-creatures`.

```
docker build -t urvanovru/java-in-dynamics-2022-creatures .
```

Аналогичного результата можно добиться с помощью Spring Boot Maven Plugin:

```
./mvnw spring-boot:build-image -Dspring-boot.build-
image.imageName=urvanovru/java-in-dynamics-2022-creatures
```

Развертывание и запуск контейнера из docker image с меткой `urvanovru/java-in-dynamics-2022` с пробрасыванием порта 8080 с компьютера хоста (на котором мы работаем) на развернутый контейнер:

```
docker run -p 8080:8080 urvanovru/java-in-dynamics-2022-creatures
```

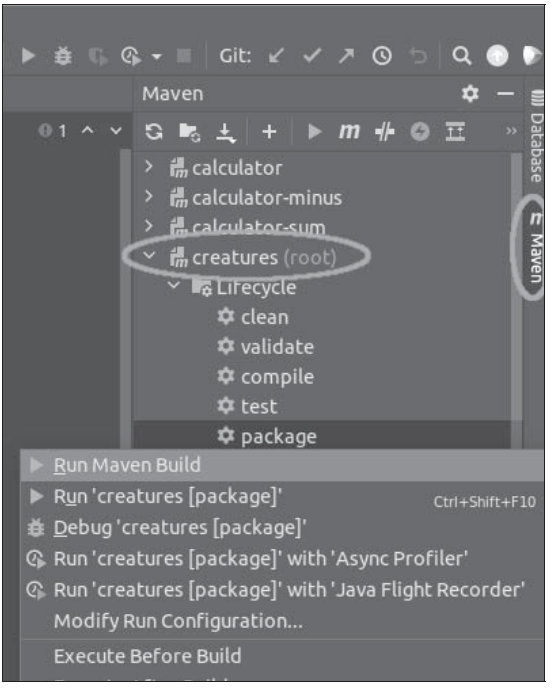


Рис. 31.12. Запуск mvn package из IntelliJ IDEA

Сам Dockerfile содержит описание image:

- FROM указывает базовый docker image, из которого будет строиться новый docker image с нашим приложением. В данном случае у нас будет дистрибутив Alpine Linux с предустановленной Java 17.
- С помощью `ARG` объявляем переменную `JAR_FILE`, содержащую шаблон поиска собранного jar-файла нашего приложения из каталога target.
- Команда `COPY` копирует jar-файл с нашим приложением в файл app.jar внутри нашего docker image. Этот содержит наше приложение, упакованное со всеми зависимостями.
- `ENTRYPOINT` указывает команду, которая должна запускать наше приложение внутри развернутого контейнера (`java -jar /app.jar`).

Безопаснее запускать приложение не от пользователя `root`, а от непривилегированного пользователя, для чего нужно модифицировать Dockerfile:

Dockerfile

```
FROM openjdk:17-alpine
RUN addgroup -S creatures && adduser -S creatures -G creatures
USER creatures:creatures
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Если мы пересоберем Docker image и запустим его, то в консоли можно будет увидеть, что запуск приложения был от пользователя creatures (/app.jar started by creatures):

```
Starting CreaturesApplication v0.0.1-SNAPSHOT using Java 17-ea on f75ba7215d05
with PID 1 (/app.jar started by creatures in /)
```

31.9. Kubernetes

Мы уже рассмотрели Docker, который облегчает перенос и развертывание приложения и необходимого ему окружения. Современное приложение обычно содержит большое количество сервисов, которые нужно не только развернуть, но и поддерживать в рабочем состоянии, следить за работоспособностью, перезапускать и масштабировать при необходимости.

Основные понятия в Kubernetes, которые нам понадобятся в этой главе:

- Node — это физическая машина в кластере Kubernetes.
- Pod — контейнер, можно рассматривать как один небольшой виртуальный компьютер с установленной операционной системой, нашим приложением и необходимыми зависимостями.
- Persistence Volume — постоянное хранилище, которое могут использовать несколько pod-ов.
- Service — абстракция, которая логически объединяет несколько pod-ов и доступ к ним.
- ReplicaSet — абстракция, поддерживающая указанное количество копий pod-ов в рабочем состоянии, уничтожая и создавая новые при необходимости.
- Deployment — описание желаемого состояния, которое Kubernetes будет поддерживать.
- Kubectl — консольный клиент Kubernetes, с помощью которого мы будем им управлять.

Minikube — это локальный Kubernetes, который обычно используется для разработки и изучения. Именно его вам и нужно будет установить по инструкции для вашей операционной системы:

https://minikube.sigs.k8s.io/docs/start

После установки и настройки Minikube у вас должны работать команды его запуска и остановки:

```
$ minikube start
$ minikube stop
```

А следующая команда должна отображать версию kubectl:

```
$ kubectl version
```

Для того чтобы наш проект мог существовать в Kubernetes, нам нужно добавить новую зависимость:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

в файл pom.xml нашего сервиса creatures.

Зависимость `spring-boot-starter-actuator` добавит новый endpoint/actuator/health в проект, благодаря которой Kubernetes сможет узнавать, что наш сервис в рабочем состоянии и его не надо перезапускать. Вызовите с помощью Postman метод `GET` для localhost:8080/actuator/health с запущенным проектом, и вы получите ответ:

```
{
    "status": "UP"
}
```

В главе про Docker мы уже собирали docker image. Но docker registry и Minikube хранят docker image в разных местах, которые никак не связаны. Minikube не сможет просто забрать docker image из прошлой статьи. Нам нужно настроить Docker так, чтобы он собирал свои docker image в Minikube:

```
$  minikube docker-env
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.49.2:2376"
export DOCKER_CERT_PATH="/home/fedor/.minikube/certs"
export MINIKUBE_ACTIVE_DOCKERD="minikube"

# To point your shell to minikube's docker-daemon, run:
# eval $(minikube -p minikube docker-env)
$ eval $(minikube -p minikube docker-env)
```

> ***Настройка Docker на сборку docker image в Minikube***
>
> Обратите внимание, что `eval $(minikube -p minikube docker-env)` нужно запускать в каждом запущенном терминале, в котором вы собираетесь собирать образы Docker.

Сборка образа Docker с помощью плагина Maven от Spring (вызывать из каталога с проектом creatures):

```
./mvnw spring-boot:build-image -Dspring-boot.build-
image.imageName=urvanovru/java-in-dynamics-2022-creatures
```

А теперь давайте проверим, работает ли Minikube:

```
$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   12m
```

Проверка подов в пространстве urvanovru (обычно каждый проект развертывается в свое пространство имен, а не в пространство имен по умолчанию):

```
$ kubectl get pods --namespace urvanovru
No resources found in urvanovru namespace.
```

Пока никаких pod-ов нет.

А теперь, собственно, займемся развертыванием приложения в Kubernetes. Для этого нам нужно создать deployment. Консольная утилита kubectl (не забудьте сделать alias для него, если используете Minikube, как это указано в инструкции по установке) сделает это за нас (команды вызвать из каталога с проектом creatures):

```
$ kubectl create deployment java-in-dynamics-2022-creatures
--image=urvanovru/java-in-dynamics-2022-creatures --dry-run=client -o=yaml >
deployment.yaml
$ echo --- >> deployment.yaml
$ kubectl create service clusterip java-in-dynamics-2022-creatures
--tcp=8080:8080 --dry-run=client -o=yaml >> deployment.yaml
```

После выполнения этих команд создастся файл deployment.yaml. Откройте его в IDE и добавьте imagePullPolicy, как показано ниже:

deployment.yaml

```
...
    spec:
      containers:
      - image: urvanovru/java-in-dynamics-2022-creatures
        name: java-in-dynamics-2022-creatures
        imagePullPolicy: Never
        resources: {}
...
```

Мы указали Never в ImagePullPolicy, чтобы Kubernetes не пытался выкачать docker image из центрального Docker Hub, так наш docker image существует только на нашем компьютере.

Для запуска проекта в Kubernetes нужно применить сгенерированный deployment.yaml следующей командой:

```
$ kubectl apply -f deployment.yaml --namespace urvanovru
```

Проверим, что проект запустился:

```
$ kubectl get pods --namespace urvanovru
```

Если что-то не получилось, то можно удалить deployment из Kubernetes:

```
$ kubectl delete deployment java-in-dynamics-2022-creatures --namespace
urvanovru
```

А потом проверить, что делали неправильно, и заново создать его:

```
$ kubectl apply -f deployment.yaml --namespace urvanovru
```

Можно смотреть лог пода:

```
$ kubectl get pods --namespace urvanovru
NAME                                                READY STATUS    RESTARTS  AGE
java-in-dynamics-2022-creatures-56cb9d5444-7rvhk  1/1  Running   0         7m7s
$ kubectl logs java-in-dynamics-2022-creatures-56cb9d5444-7rvhk --namespace
                                                                    urvanovru
```

```
Setting Active Processor Count to 12
WARNING: Unable to convert memory limit "max" from path "/sys/fs/cgroup/
memory.max" as int: memory size "max" does not match pattern
"^([\\d]+)([kmgtKMGT]?)$"
Calculating JVM memory based on 16104212K available memory
Calculated JVM Memory Configuration: -XX:MaxDirectMemorySize=10M -Xmx15707271K
-XX:MaxMetaspaceSize=89740K -XX:ReservedCodeCacheSize=240M -Xss1M (Total
Memory: 16104212K, Thread Count: 50, Loaded Class Count: 13430, Headroom: 0%)
...
```

Также в Minikube есть своя визуальная панель управления, доступная по команде:

```
$ minikube dashboard
```

После нее откроется браузер по умолчанию с запущенной административной панелью. Не забудьте там выбрать нужное пространство имен, как показано на рис. 31.13.

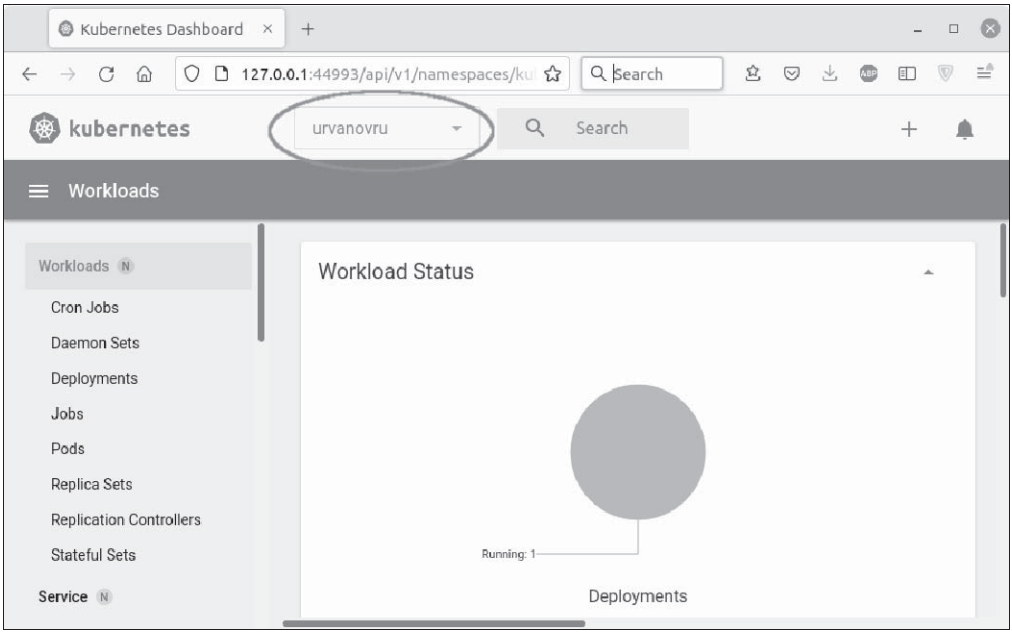


Рис. 31.13. Minikube dashboard

31.10. Задания

1. В идеале REST-сервис должен возвращать не статус HTTP 200 OK во всех случаях, а статус 201 Created при успешном создании существа и 204 No Content при успешном удалении. Измените код созданного сервиса так, чтобы он возвращал нужные статусы. Вам может помочь документация Spring Boot, расположенная на сайте **spring.io**.

2. Добавьте какие-нибудь тесты JUnit в созданное приложение. Используйте документацию Spring Boot на сайте **spring.io** и документацию JUnit на сайте **https://junit.org**. Это задание может быть довольно длительным, в зависимости от качества тестов.
3. Замените H2 Database на какую-нибудь реальную базу данных, например PostgreSQL. Добейтесь работоспособности сервиса.
4. Добавьте Flyway (**https://flywaydb.org**) или Liquibase (**https://www.liquibase.org**) и напишите пару миграций.
5. Добавьте проверку полей для класса Creature с помощью аннотаций из javax.validation. Подробнее про добавление проверки читайте в документации Spring на сайте **https://spring.io**.

ГЛАВА 32

Заключение

В данной книге были рассмотрены возможности современного языка Java, а также их изменения от версии к версии. Также были разобраны подводные камни, с которыми могут столкнуться разработчики, пришедшие из других языков программирования.

Многие темы были рассмотрены достаточно подробно до такого уровня, какой необходим для понимания самих принципов работы Java и для разбора проблем, к которым может привести неправильное применение ее возможностей.

Java — это платформонезависимый на уровне байт-кода язык программирования. В современном мире он чаще всего используется для написания серверной части высоконагруженных приложений. В большинстве случаев используется Spring Boot, JUnit, TestNG, Gradle, Maven, Mockito, PowerMock, Docker, Kubernetes и другие библиотеки, фреймворки и утилиты.

Для продолжения изучения мира Java рекомендую следующие темы:

1. Git;
2. Apache Maven;
3. JUnit;
4. Mockito;

5. PowerMock;
6. Spring Boot и остальные проекты Spring;
7. Slf4j и Logback (про логирование в Java на моем сайте **https://urvanov.ru** есть отличная статья, описывающая текущее положение дел: **https://urvanov.ru/2019/07/08/логирование-c-slf4j-и-logback/**, которой хватит для старта);
8. PostgreSQL;
9. JPA и Hibernate;
10. Lombok (можно изучать по русской версии документации **https://urvanov.ru/2015/09/22/project-lombok/**);
11. Apache Commons;
12. Jackson;
13. Testcontainers;
14. по ситуации.

Дальше — как получится. Очень много ответвлений того, что нужно учить. Смотрите вакансии и изучайте то, что требуется в них.

Многие устаревшие темы и те возможности, которые вряд ли пригодятся при коммерческой разработке, были пропущены для уменьшения количества материала: Java-апплеты, Java Web Start, Java Network Launch Protocol, AWT, Swing, JavaFX, J2ME и многие другие. Если они вам потребуются, то вы можете изучить их самостоятельно.

Java-апплеты устарели и помечены для удаления в Java 17. Раньше это был аналог Flash и Silverlight, и они использовались для создания веб-приложений, запускающихся в песочнице браузера. Сейчас JavaScript и CSS покрывают все их возможности. Если вы даже их встретите, то в какой-нибудь древней корпоративной сети. Ни один современный браузер их все равно не поддерживает.

Java Web Start и Java Network Launch Protocol вы тоже можете встретить только в каких-нибудь старых проектах, т. к. в связи с развитием возможностей JavaScript необходимость в них отпала.

AWT, Spring и JavaFX полезно знать, но в коммерческой разработке вам вряд ли доведется их использовать, т. к. в большинстве случаев современная разработка Java подразумевает именно серверную часть, а не приложения для настольных компьютеров.

Вакансий J2ME тоже уже не встретишь, т. к. в связи с развитием смартфонов на Android и iOS технология перестала быть актуальной.