

**М. Г. Городничев, М. С. Мосева, К. Р. Харрасов,  
А. Д. Водиченков**

# **Информационные технологии и программирование на языке Java**

*Рекомендовано Ученым советом ордена Трудового  
Красного Знамени федерального государственного  
бюджетного образовательного учреждения высшего  
образования «Московский технический университет  
связи и информатики» (МТУСИ) в качестве учебного  
пособия для студентов вузов, обучающихся по  
направлению подготовки 09.03.01 – «Информатика и  
вычислительная техника» и 09.03.04 – «Программная  
инженерия»*

**Москва  
Горячая линия – Телеком  
2025**

УДК 004.774.6 (076.5)

ББК 32.973.4

И74

Р е ц е н з е н т ы :

доктор техн. наук, профессор, профессор кафедры «Математическая кибернетика и информационные технологии» МТУСИ *Ю. Л. Леохин*,  
канд. техн. наук, доцент кафедры «Инженерия и математика прикладных систем искусственного интеллекта» МАДИ *А. С. Доткулова*;

доктор физ.-мат. наук, доцент,

зав. кафедрой «Высшая математика» ПГУТИ *О. В. Осипов*.

А в т о р ы :

М. Г. Городничев, М. С. Мосева, К. Р. Харрасов, А. Д. Водиченков

**И74 Информационные** технологии и программирование на языке Java.

Учебное пособие для вузов / М. Г. Городничев, М. С. Мосева,  
К. Р. Харрасов и др. – М.: Горячая линия – Телеком, 2025. – 112 с.:  
ил.

**ISBN 978-5-9912-1157-4.**

Представлены 13 лабораторных работ, охватывающих ключевые аспекты языка программирования Java и современные технологии разработки. Каждая лабораторная работа включает теоретический материал, практические задания и контрольные вопросы для самостоятельной подготовки. Данное пособие позволит освоить современные инструменты разработки и приобрести практические навыки программирования на Java. В пособии последовательно рассмотрены основы Java, типы данных и объектно-ориентированное программирование (ООП); работа с классами, объектами, исключениями, строками и регулярными выражениями; коллекции, многопоточность и современные API, включая Stream API; инструменты сборки Maven и Gradle; основы работы с фреймворком Spring, включая Inversion of Control (IoC) и Dependency Injection (DI).

Практические задания в каждой лабораторной работе направлены на закрепление теоретических знаний и формирование навыков разработки на Java. Контрольные вопросы помогают проверить усвоение материала.

Для студентов вузов, обучающихся по направлениям подготовки 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия». Будет полезна широкому кругу читателей, желающим углубить свои знания в Java-разработке.

**ББК 32.973.4**

ISBN 978-5-9912-1157-4

© М. Г. Городничев, М. С. Мосева,  
К. Р. Харрасов, А. Д. Водиченков, 2025  
© Издательство «Горячая линия – Телеком», 2025

# Введение

Данное учебное пособие предназначено для изучения языка программирования Java и его современных возможностей. Оно охватывает ключевые темы, начиная с истории языка и базовых типов данных, заканчивая сложными концепциями, такими как многопоточность, работа с коллекциями, аннотации и современные инструменты сборки.

Основной упор сделан на практическое освоение материала: каждая глава сопровождается лабораторными заданиями и контрольными вопросами, что позволит читателю не только закрепить теоретические знания, но и приобрести навыки реального программирования.

В первом разделе рассматриваются основы Java: его история, типы данных и принципы объектно-ориентированного программирования. Далее уделяется внимание работе с классом `Object`, хэш-таблицами, обработке исключений и строками, включая регулярные выражения.

В разделах 2–8 описываются более продвинутые темы: работа с коллекциями, многопоточность и использование Stream API. Значительное внимание уделяется современным инструментам сборки Maven и Gradle, которые помогают автоматизировать процессы разработки.

Последние разделы посвящены фреймворку Spring — одному из самых популярных инструментов для разработки корпоративных приложений. Читатель познакомится с концепциями Inversion of Control (IoC) и Dependency Injection (DI), которые являются основой гибкой и масштабируемой архитектуры программных решений.

Пособие предназначено для студентов, начинающих разработчиков и всех, кто хочет углубить свои знания в программировании на Java. Применение знаний, полученных из данного материала, поможет не только лучше разобраться в языке, но и успешно использовать его в реальных проектах.

# 1 История языка программирования Java. Типы данных

---

Язык Java был разработан в начале 1990-х годов компанией Sun Microsystems. Основным разработчиком языка является Джеймс Гослинг. Изначально язык создавался для управления устройствами и интерактивного телевидения. Принцип WORA (Write Once, Run Anywhere) — это ключевая концепция, связанная с языком программирования Java и его экосистемой. Этот принцип подразумевает, что программа, написанная на Java, может быть скомпилирована в байт-код и затем выполнена на любой платформе, где установлена виртуальная машина Java (Java Virtual Machine, JVM), без необходимости в изменениях или повторной компиляции.

Первая публичная версия языка была выпущена в 1995 году (Java 1.0). Язык Java быстро завоевал популярность благодаря своей платформенной независимости, обеспечиваемой виртуальной машиной Java. В последующие годы язык претерпел множество изменений и обновлений, включая добавление новых библиотек и улучшение производительности. В 2006 году Sun Microsystems открыла исходный код Java, что способствовало её дальнейшему развитию и распространению. В 2010 году Oracle Corporation приобрела Sun Microsystems и продолжила развивать Java.

Жизненный цикл программы на Java включает в себя следующие этапы.

1. Написание кода в текстовом редакторе или интегрированной среде разработки (IDE) с использованием синтаксиса Java.
2. Компиляция: исходный код (.java) компилируется в байт-код (.class) с помощью компилятора javac. Байт-код является платформенно независимым и может выполняться на любой платформе, где установлена JVM.
3. Интерпретация: байт-код выполняется виртуальной машиной Java. JVM интерпретирует байт-код в машинный код во время выполнения. Для ускорения процесса интерпретации

байт-кода в машинный код был введен механизм JIT-компиляции. JIT-компиляция (Just-In-Time compilation) — это метод компиляции, который используется в виртуальных машинах, таких как JVM и .NET Common Language Runtime (CLR). Этот подход позволяет выполнять код, написанный на высокогуровневых языках программирования, более эффективно, чем просто интерпретация. Во время выполнения программы JIT-компилятор анализирует часто исполняемые участки кода и компилирует их в нативный код для конкретной платформы. Этот нативный код затем кешируется, чтобы его можно было повторно использовать при последующих вызовах, при этом увеличивается количество потребляемой памяти. Схема представлена на рис. 1.1.

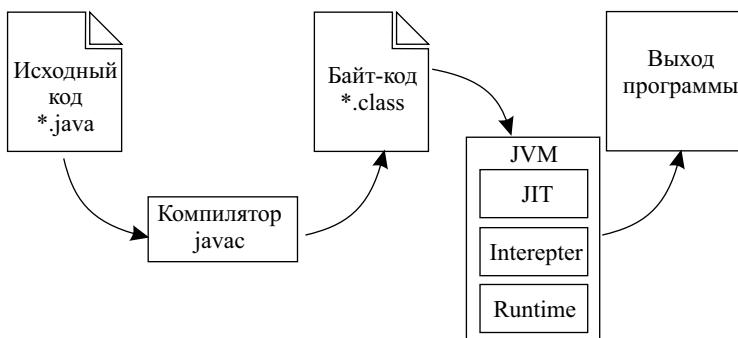


Рис. 1.1. Жизненный цикл программы на языке Java

Укажем особенности языка Java.

**Простой.** Java был разработан программистами для программистов с акцентом на простоту и удобство использования на основе языков С и С++.

**Объектно-ориентированный.** Java является строго объектно-ориентированным языком, что означает, что все в Java рассматривается как объект (за исключением примитивных типов). Это позволяет использовать такие принципы ООП, как инкапсуляция (скрытие деталей реализации), наследование (переиспользование кода), полиморфизм (возможность использовать один интерфейс для различных объектов) и абстракция (концентрация на основных характеристиках объекта, игнорируя несущественные детали; создавать моделей реального мира в виде объектов, которые представляют собой комбинацию данных и методов).

**Надежный.** Java обеспечивает высокую надежность благодаря строгой типизации, обработке исключений и автоматическому управлению памятью с помощью сборщика мусора. Эти особенности помогают предотвратить многие распространенные ошибки, такие как утечки памяти и ошибки времени выполнения. Java также имеет механизмы проверки кода на этапе компиляции, что способствует созданию более безопасных и стабильных приложений.

**Безопасный.** Java предоставляет множество механизмов безопасности, включая управление доступом к памяти и защиту от вредоносного кода. Система безопасности Java основана на концепции «песочницы» (sandbox), которая ограничивает возможности выполнения небезопасного кода.

**Не зависящий от архитектуры компьютера.** Java разработан так, чтобы быть независимым от аппаратной платформы. Программы на Java компилируются в байт-код, который может выполняться на любой платформе с установленной JVM. Это обеспечивает возможность запуска одного и того же кода на различных устройствах без необходимости его модификации.

**Переносимый.** Java следует принципу «напиши один раз, запускай везде» (WORA). Это означает, что после компиляции программы в байт-код она может выполняться на любой платформе, поддерживающей JVM. Это делает Java идеальным выбором для разработки кроссплатформенных приложений.

**Многопоточный.** Java имеет встроенную поддержку многопоточности.

В Java существует несколько типов данных, которые можно разделить на две основные категории: примитивные типы и ссылочные типы. Виды типов данных представлены на рис. 1.2.

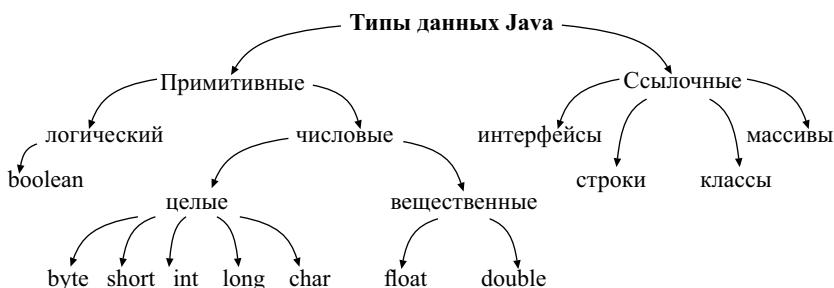


Рис. 1.2. Виды типов данных в Java

Примитивные типы представляют собой простые значения и не имеют методов. В Java есть восемь примитивных типов:

- byte: 8-битное целое число. Диапазон: от -128 до 127;
- short: 16-битное целое число. Диапазон: от -32\_768 до 32\_767;
- int: 32-битное целое число. Диапазон: от -2\_147\_483\_648 до 2\_147\_483\_647;
- long: 64-битное целое число. Диапазон: от -9\_223\_372\_036\_854\_775\_808 до 9\_223\_372\_036\_854\_775\_807;
- float: 32-битное число с плавающей запятой. Используется для представления дробных чисел;
- double: 64-битное число с плавающей запятой. Более точный, чем float;
- char: 16-битный символ Unicode. Используется для представления одиночных символов (например, 'а', '1', '\$');
- boolean: Логический тип данных, который может принимать только два значения: true или false.

Ссылочные типы представляют собой объекты и массивы. Они хранят ссылки на места в памяти, где находятся данные. К основным ссылочным типам относятся классы, интерфейсы, массивы, перечисления.

Примитивные типы данных хранят свои значения непосредственно в памяти. Они занимают фиксированное количество байтов (например, int занимает 4 байта). Примитивные переменные хранятся на стеке, что обеспечивает быструю доступность и освобождение памяти после завершения работы.

В Java существует несколько способов преобразования между примитивными типами данных. Это называется приведением (casting). В зависимости от направления приведения различают явное и неявное преобразование типов. При неявном приведении меньший тип данных преобразуется в больший без потери информации. Например:

- byte → short;
- short → int;
- int → long;
- float → double.

1. int a = 10;
2. long b = a; // Автоматически преобразуем int в long

Когда нужно преобразовать больший тип данных в меньший, необходимо использовать явное приведение, так как может произойти потеря точности. При этом программист берет на себя ответственность за возможные ошибки:

- int → short;
  - long → int;
  - double → float.
1. long c = 1000000L;
  2. int d = (int)c; // Явное приведение long к int
  - 3.
  4. double e = 123.456;
  5. float f = (float)e; // Приведение double к float

Целые числа могут быть приведены к числам с плавающей точкой неявно:

- int → float;
  - int → double;
  - long → float;
  - long → double.
1. int g = 42;
  2. double h = g; // Преобразование int в double происходит
  3. // автоматически

Для приведения чисел с плавающей точкой к целым числам используется явное приведение, при котором дробная часть отбрасывается:

- float → int;
  - double → int;
  - float → long;
  - double → long.
1. double i = 9.99;
  2. int j = (int)i; // Отбрасывание дробной части
  3. System.out.println(j); // Выведет 9

Важно помнить, что при приведении больших типов данных к меньшим возможна потеря точности:

1. float k = 123456789f;
2. int l = (int)k;
3. System.out.println(l); // Выведет 123456792 (потеря точности)

Логический тип boolean не может быть приведен ни к одному другому типу данных и наоборот. Попытка сделать такое приведение вызовет ошибку компиляции.

При выполнении арифметических операций над значениями разных типов данных результат будет иметь тот же тип, который является наибольшим среди участвующих в выражении типов.

Правила автоматического повышения типов для арифметических операций ('+', '-', '\*', '/', '%'):

- если хотя бы одно из значений имеет тип double, то все значения будут преобразованы в double:

1. double x = 1.5;
2. int y = 2;
3. double result = x + y; // Результат будет типа double

- если нет значений типа double, но есть хотя бы одно значение типа float, то результат также будет типа float:

1. float z = 1.25f;
2. int w = 3;
3. float result2 = z \* w; // Результат будет типа float

- если нет значений типа double и float, но хотя бы одно значение имеет тип int, то результат будет типа int. Если все значения имеют тип byte или short, они сначала будут преобразованы в int, а затем операция выполнится над этими значениями типа int:

1. byte a = 127;
2. short b = 32767;
3. int result3 = a + b; // Оба значения будут преобразованы в int,
4. // результат будет типа int

- если используются операторы побитового сдвига (<<, >>, >>>) и один из операндов имеет тип long, то другой операнд также будет преобразован к типу long.

Ссылочные типы данных (например, объекты классов, массивы, строки) хранят ссылки на объекты в памяти, а сами объекты располагаются в куче. При создании объекта, фактические данные хранятся в куче, а переменная содержит только ссылку на это место в памяти.

void — это специальный тип, который используется для обозначения того, что метод не возвращает никакого значения.

Когда мы начинаем изучать язык программирования, первой программой всегда является вывод Hello World. Установим все необходимое и запустим программу на Java, которая выводит приветствие (листинг 1.1).

**Листинг 1.1.** Класс JavaHelloWorldProgram

```
1. public class JavaHelloWorldProgram {  
2.  
3.     public static void main(String args[]){  
4.         System.out.println("Hello World");  
5.     }  
6. }
```

Сохраните приведенную выше программу под именем JavaHelloWorldProgram.java в каталоге для первой лабораторной работы.

Откройте командную строку и перейдите в каталог, в котором сохранен файл программы. Затем выполните следующую команду:

```
> java JavaHelloWorldProgram.java
```

Программа скомпилируется и запустится, отобразится результат.

Стоит запомнить, что:

- исходный файл Java может содержать несколько классов, но допускается только один public (это модификатор доступа) класс;
- имя исходного файла Java должно совпадать с именем public класса. Поэтому имя файла программы — JavaHelloWorldProgram.java;
- когда мы компилируем код, он генерирует байт-код и сохраняет его с расширением Class\_Name.class. Если посмотреть на каталог, в котором мы скомпилировали java-файл, то можно заметить новый созданный файл JavaHelloWorldProgram.class;
- при выполнении файла класса не нужно указывать полное имя файла. Нужно использовать только публичное имя класса;
- при запуске программы с помощью команды java она загружает класс в JVM, ищет в классе метод main и запускает его. Синтаксис метода main должен быть таким же, как указан в примере, иначе программа не будет запущена и выбросит исключение Exception в потоке "main":

```
java.lang.NoSuchMethodError: main
```

## Задания для выполнения лабораторной работы

**Задание 1.** Создайте программу, которая находит и выводит все простые числа меньше 100.

Создайте файл с именем Primes.java, в этом файле опишите следующий класс:

```
1. public class Primes {  
2.     public static void main(String[] args) {  
3.         }  
4.     }
```

Внутри созданного класса после метода main() опишите метод IsPrime (int n), который определяет, является ли аргумент простым числом или нет. Можно предположить, что входное значение n всегда будет больше 2. Полное описание метода будет выглядеть так:

```
1. public static boolean isPrime(int n) {  
2. }
```

Данный метод можно реализовать по вашему усмотрению, однако простой подход заключается в использовании цикла for. Данный цикл должен перебирать числа, начиная с 2 до (но не включая) корня из n, проверяя, существует ли какое-либо значение, делящееся на n без остатка. Для этого можно использовать оператор остатка «%». Если какая-либо переменная полностью делится на аргумент, сработает оператор return false. Если же значение не делится на аргумент без остатка, то это простое число и сработает оператор return true.

После того как это будет реализовано, приступайте к заполнению основного метода main() другим циклом, который перебирает числа в диапазоне от 2 до 100 включительно. Необходимо вывести на экран те значения, которые метод isPrime() посчитал простыми. После написания программы скомпилируйте и протестируйте её. Убедитесь, что результаты правильные.

**Задание 2.** Создайте программу, которая определяет, является ли введенная строка палиндромом.

Для этой программы создайте класс с именем Palindrome в файле с названием Palindrome.java. На этот раз вы можете воспользоваться следующим кодом:

```
1. public class Palindrome {  
2.     public static void main(String[] args) {
```

```
3.     for (int i = 0; i < args.length; i++) {  
4.         String s = args[i];  
5.     }  
6. }  
7. }
```

Первая задача состоит в том, чтобы создать метод, позволяющий полностью изменить символы в строке. Заголовок метода должен быть следующим:

1. public static String reverseString(String s)

Вы можете реализовать этот метод путем создания локальной переменной, которая является пустой строкой, а затем добавлять символы из входной строки в выходные данные в обратном порядке. Используйте метод length(), который возвращает длину строки, и метод charAt(int index), который возвращает символ по указанному индексу. Индексы начинаются с 0 и увеличиваются на 1.

После того как вы создали метод reverseString(), создайте еще один метод:

1. public static boolean isPalindrome(String s)

Этот метод должен перевернуть слово s, а затем сравнить с первоначальными данными. Используйте метод equals(Object obj) для проверки значения равенства. Например:

1. String s1 = "hello";
2. String s2 = "Hello";
3. String s3 = "hello";
4. s1.equals(s2); // Истина
5. s1.equals(s3); // Ложь

Для сравнения строк нельзя использовать оператор «==».

Скомпилируйте и протестируйте программу. Для того чтобы передать аргументы в программу, необходимо указать их при запуске программы через командную строку:

```
java Palindrome madam racecar apple kayak song noon
```

## Контрольные вопросы

1. Java является компилируемым или интерпретируемым языком?
2. Что такое JVM и для чего предназначается?
3. Каков жизненный цикл программы на языке Java?

4. Какие виды типов данных есть в языке Java?
5. Чем примитивные типы данных отличаются от ссылочных?
6. Как происходит преобразование примитивных типов в Java?
7. Что такое байт-код в Java, и почему он важен для платформенной независимости?
8. Какой тип данных используется для хранения символов в Java? Как представляются символы в памяти?
9. Что такое литералы в Java? Приведите примеры литералов для разных типов данных.
10. Почему Java считается строго типизированным языком?
11. Какие проблемы могут возникнуть при использовании неявного преобразования типов?

## 2 Объектно-ориентированное программирование

---

Объектно-ориентированное программирование (ООП) — это парадигма программирования, которая основана на концепции объектов и классов. В Java ООП является одним из ключевых аспектов языка, и оно основано на четырех основных принципах: инкапсуляция, наследование, полиморфизм, абстракция.

Инкапсуляция заключается в сокрытии внутренней реализации объекта от внешнего мира и предоставлении доступа к данным только через определенные методы класса. Это позволяет защитить данные от несанкционированного изменения и повысить надежность кода. Пример реализации инкапсуляции в Java представлен на листинге 2.1.

**Листинг 2.1.** Реализация инкапсуляции

```
1. public class Account {  
2.     private double balance;  
3.  
4.     public void deposit(double amount) {  
5.         if (amount > 0) {  
6.             balance += amount;  
7.         }  
8.     }  
9.  
10.    public void withdraw(double amount) {  
11.        if (balance >= amount && amount > 0) {  
12.            balance -= amount;  
13.        } else {  
14.            System.out.println("Недостаточно средств для снятия.");  
15.        }  
16.    }  
17.  
18.    public double getBalance() {  
19.        return balance;  
20.    }  
21. }
```

В этом примере переменная `balance` объявлена как приватная (`private`), что означает, что доступ к ней возможен только внутри класса. Внешний мир может взаимодействовать с объектом только через публичные методы (`deposit`, `withdraw`, `getBalance`), которые контролируют доступ к данным.

В Java модификаторы доступа определяют уровень видимости классов, методов и полей внутри программы. Существует четыре уровня доступа:

1) `public` — элементы с этим модификатором доступны везде, то есть они могут использоваться любым классом, вне зависимости от того, где он находится;

2) `protected` — доступ к элементам с таким модификатором возможен только для классов в том же пакете или для подклассов этого класса, даже если они находятся в другом пакете;

3) без модификатора (по умолчанию) — если ни один модификатор не указан, то элементы будут видны только в пределах пакета, но недоступны за его пределами;

4) `private` — элементы с таким модификатором доступны только внутри класса, в котором они объявлены. Они невидимы снаружи, даже для наследников.

**Наследование** позволяет создавать новые классы на основе существующих, наследуя их свойства и поведение. Это помогает избежать дублирования кода и упрощает его поддержку. Пример наследования в Java представлен на листинге 2.2.

**Листинг 2.2.** Реализации наследования

```
1. class Animal {  
2.     protected String name;  
3.  
4.     public void eat() {  
5.         System.out.println(name + " ест.");  
6.     }  
7. }  
8.  
9. class Dog extends Animal {  
10.    public void bark() {  
11.        System.out.println(name + " лает.");  
12.    }  
13. }  
14.  
15. public class Main {  
16.     public static void main(String[] args) {  
17.         Dog dog = new Dog();
```

```
18.     dog.name = "Шарик";
19.     dog.eat(); // Шарик ест.
20.     dog.bark(); // Шарик лает.
21. }
22. }
```

Здесь класс Dog наследует все поля и методы класса Animal. Класс Dog добавляет новый метод bark() и использует унаследованный метод eat().

Java не поддерживает множественное наследование через классы. Это означает, что класс может расширять (наследовать) только один другой класс. Однако Java предоставляет альтернативу множественному наследованию через интерфейсы.

Множественное наследование может привести к проблемам, известным как «проблема ромба» (или проблема бриллианта). Рассмотрим код, представленный на листинге 2.3.

#### Листинг 2.3. Ромбовидная проблема

```
1. class A {
2.     void method() { System.out.println("A"); }
3. }
4.
5. class B extends A {
6.     @Override
7.     void method() { System.out.println("B"); }
8. }
9.
10. class C extends A {
11.     @Override
12.     void method() { System.out.println("C"); }
13. }
14.
15. // Множественное наследование невозможно!
16. // class D extends B, C {}
```

Здесь, если бы класс D мог наследоваться одновременно от B и C, возникла бы неопределенность при вызове метода method() — будет вызван метод из B или из C? Чтобы избежать таких проблем, Java ограничивает возможность множественного наследования через классы.

Интерфейсы позволяют классу реализовать несколько контрактов (интерфейсов), тем самым предоставляя функциональность, схожую с множественным наследованием. Интерфейсы содержат только объявления методов без их реализации, а также

могут содержать статические константы и методы по умолчанию. Пример реализации интерфейсов представлен на листинге 2.4.

**Листинг 2.4.** Реализация интерфейсов

```
1. interface InterfaceA {  
2.     void methodA();  
3. }  
4.  
5. interface InterfaceB {  
6.     void methodB();  
7. }  
8.  
9. class MyClass implements InterfaceA, InterfaceB {  
10.  
11.     // Реализация методов из обоих интерфейсов  
12.     @Override  
13.     public void methodA() {  
14.         System.out.println("Метод А");  
15.     }  
16.  
17.     @Override  
18.     public void methodB() {  
19.         System.out.println("Метод В");  
20.     }  
21. }
```

Класс MyClass реализует оба интерфейса — InterfaceA и InterfaceB. Таким образом он получает все функциональные возможности этих интерфейсов.

**Полиморфизм** позволяет объектам разных типов реагировать по-разному на одни и те же сообщения. Полиморфизм бывает двух видов: **статический** (перегрузка методов) и **динамический** ( переопределение методов).

*Перегрузка методов (статический полиморфизм)*

Перегруженные методы имеют одно имя, но разные списки параметров.

Пример перегрузки метода представлен на листинге 2.5.

**Листинг 2.5.** Реализация перегрузки

```
1. class Calculator {  
2.     public int add(int a, int b) {  
3.         return a + b;  
4.     }  
5. }
```

```

6.     public double add(double a, double b) {
7.         return a + b;
8.     }
9. }
10.
11. public class Main {
12.     public static void main(String[] args) {
13.         Calculator calc = new Calculator();
14.         System.out.println(calc.add(5, 7)); // 12
15.         System.out.println(calc.add(5.5, 7.5)); // 13.0
16.     }
17. }
```

Метод `add` перегружен дважды: один принимает два целых числа, другой — два числа с плавающей точкой.

#### *Переопределение методов (динамический полиморфизм)*

Переопределение методов происходит при наследовании, когда дочерний класс изменяет реализацию метода родительского класса. Пример переопределения метода представлен на листинге 2.6.

#### **Листинг 2.6.** Реализация переопределения

```

1. class Animal {
2.     public void makeSound() {
3.         System.out.println("Животное издает звук.");
4.     }
5. }
6.
7. class Cat extends Animal {
8.     @Override
9.     public void makeSound() {
10.         System.out.println("Мяу!");
11.     }
12. }
13.
14. public class Main {
15.     public static void main(String[] args) {
16.         Animal animal = new Cat();
17.         animal.makeSound(); // Мяу!
18.     }
19. }
```

Класс `Cat` переопределяет метод `makeSound()` класса `Animal`, поэтому при вызове этого метода у объекта типа `Animal`, который

ссылается на объект типа Cat, будет вызван метод makeSound() из класса Cat.

**Абстракция** подразумевает выделение наиболее значимых характеристик объекта, скрывая детали его реализации. Абстрактный класс или интерфейс предоставляет общий шаблон поведения, оставляя конкретные реализации подклассам. Пример абстракции в Java представлен на листинге 2.7.

**Листинг 2.7.** Реализация абстракции

```
1. abstract class Shape {  
2.     abstract double area();  
3. }  
4.  
5. class Circle extends Shape {  
6.     private double radius;  
7.  
8.     public Circle(double radius) {  
9.         this.radius = radius;  
10.    }  
11.  
12.    @Override  
13.    double area() {  
14.        return Math.PI * radius * radius;  
15.    }  
16. }  
17.  
18. class Rectangle extends Shape {  
19.     private double width;  
20.     private double height;  
21.  
22.     public Rectangle(double width, double height) {  
23.         this.width = width;  
24.         this.height = height;  
25.     }  
26.  
27.    @Override  
28.    double area() {  
29.        return width * height;  
30.    }  
31. }  
32.  
33. public class Main {  
34.     public static void main(String[] args) {  
35.         Shape circle = new Circle(5);
```

```
36.     Shape rectangle = new Rectangle(10, 20);
37.
38.     System.out.println(circle.area()); // 78.53981633974483
39.     System.out.println(rectangle.area()); // 200.0
40. }
41. }
```

Абстрактный класс Shape определяет абстрактный метод area(), который должен быть реализован в каждом классе-наследнике. Классы Circle и Rectangle предоставляют свои реализации метода area(), соответствующие их форме.

В данной лабораторной работе необходимо использовать классы, чтобы описать, как эти объекты работают. Код для простого класса, который представляет двумерную точку представлен на листинге 2.8.

**Листинг 2.8.** Класс Point2d

```
1. public class Point2d {
2.     /** координата X */
3.     private double xCoord;
4.     /** координата Y */
5.     private double yCoord;
6.     /** конструктор с параметрами */
7.     public Point2d (double x, double y) {
8.         xCoord = x;
9.         yCoord = y;
10.    }
11.    /** конструктор по умолчанию */
12.    public Point2d () {
13.        this(0, 0);
14.    }
15.    /** получение координаты X */
16.    public double getX () {
17.        return xCoord;
18.    }
19.    /** получение координаты Y */
20.    public double getY () {
21.        return yCoord;
22.    }
23.    /** установка значения координаты X. */
24.    public void setX ( double val) {
25.        xCoord = val;
26.    }
27.    /** установка значения координаты Y. */
28.    public void setY ( double val) {
```

```
29.           yCoord = val;
30.       }
31. }
```

Экземпляр класса можно также создать, вызвав любой из реализованных конструкторов, например:

1. Point2d myPoint = new Point2d (); // создает точку (0,0)
2. Point2d myOtherPoint = new Point2d (5,3); // создает точку (5,3)
3. Point2d aThirdPoint = new Point2d ();

При реализации класса трехмерной точки можно использовать описанный выше класс двумерной точки. Для этого необходимо указать что класс Point3d наследует класс Point2d с помощью ключевого слова `extends`, и далее реализовать функционал, который присущ классу Point3d, но отличается от функционала класса Point2d. Реализация класса Point3D представлена на листинге 2.9.

**Листинг 2.9.** Класс Point3d

```
1. public class Point3d {
2.     /** координата Z */
3.     private double zCoord;
4.     /** конструктор с параметрами */
5.     public Point3d (double x, double y, double z) {
6.         super (x, y);
7.         zCoord=z;
8.     }
9.     /** конструктор по умолчанию */
10.    public Point3d() {
11.        this (0,0,0);
12.    }
13.    /** получение координаты Z */
14.    public double getZ () {
15.        return zCoord;
16.    }
17.    /** установка значения координаты Z. */
18.    public void setZ (double val) {
19.        zCoord = val;
20.    }
21. }
```

Весь функционал класса Point2d присутствует в классе Point3d, поэтому необходимо добавить только взаимодействие с третьей координатой. Попробуйте запустить код и создать экземпляры классов Point2d и Point3d, а также установить (получить) значения всех координат.

Стоит запомнить, что:

- важнейшим преимуществом наследования является повторное использование кода, поскольку подклассы наследуют переменные и методы суперкласса;
- приватные члены суперкласса недоступны для подкласса напрямую, но могут быть косвенно доступны через методы (геттеры и сеттеры);
- члены суперкласса с модификатором доступа по умолчанию доступны подклассу только в том случае, если они находятся в одном пакете;
- конструкторы суперкласса не наследуются подклассом;
- если суперкласс не имеет конструктора по умолчанию, то в подклассе также должен быть определен явный конструктор. В противном случае будет выброшено исключение времени компиляции. В конструкторе подкласса вызов конструктора суперкласса в этом случае обязателен и должен быть первым утверждением в конструкторе подкласса;
- Java не поддерживает множественное наследование, подкласс может расширять только один класс;
- мы можем создать экземпляр подкласса и затем присвоить его переменной суперкласса, это называется upcasting (повышающее преобразование);
- когда экземпляр суперкласса присваивается переменной подкласса, это называется downcasting (поникающее преобразование). Нам необходимо явно привести этот экземпляр к подклассу;
- мы можем переопределить метод суперкласса в классе наследнике. Однако мы всегда должны аннотировать переопределенный метод аннотацией `@Override`. Компилятор будет знать, что мы переопределяем метод, и если что-то изменится в методе суперкласса, то мы получим ошибку компиляции, а не нежелательные результаты во время выполнения;
- мы можем вызывать методы суперкласса и обращаться к переменным суперкласса, используя ключевое слово `super`. Это удобно, когда у нас есть одноименная переменная (метод) в подклассе, но мы хотим получить доступ к переменной (методу) суперкласса. Это также используется, когда в суперклассе и подклассе определены конструкторы и нам необходимо явно вызвать конструктор суперкласса;

- мы можем использовать инструкцию `instanceof` для проверки наследования между объектами;
- в Java мы не можем расширять `final`-классы;
- если вы не собираетесь использовать суперкласс в коде, т. е. ваш суперкласс является просто базой для сохранения многократно используемого кода, то вы можете сделать его абстрактным классом, чтобы избежать ненужного инстанцирования клиентскими классами. Это также ограничит создание экземпляров базового класса.

## Задания для выполнения лабораторной работы

**Задание 1.** Создайте иерархию классов в соответствии с вариантом. Ваша иерархия должна содержать:

- абстрактный класс;
- два уровня наследуемых классов (классы должны содержать в себе минимум 3 поля и 2 метода, описывающих поведение объекта);
- демонстрацию реализации всех принципов ООП;
- наличие конструкторов (в том числе по умолчанию);
- наличие геттеров и сеттеров;
- ввод/вывод информации о создаваемых объектах;
- предусмотрите в одном из классов создание счетчика созданных объектов с использованием статической переменной, продемонстрируйте работу.

### *Варианты для выполнения лабораторной работы*

1. Базовый класс: Животные. Дочерние классы: Кошка, Попугай, Рыбка.

2. Базовый класс: Сотрудник. Дочерние классы: Администратор, Программист, Менеджер.

3. Базовый класс: Человек. Дочерние классы: Студент, Преподаватель, Ассистент преподавателя.

4. Базовый класс: Транспортное средство. Дочерние классы: Легковой автомобиль, Грузовой автомобиль, Мотоцикл.

5. Базовый класс: Велосипед. Дочерние классы: Горный велосипед, Детский велосипед, ВМХ.

6. Базовый класс: Геометрическая фигура. Дочерние классы: Шар, Параллелепипед, Цилиндр.

7. Базовый класс: Книга. Дочерние классы: Аудиокнига, Фильм, Мюзикл.

8. Базовый класс: Мебель. Дочерние классы: Стол, Стул, Кровать.

9. Базовый класс: Монстр. Дочерние классы: Гоблин, Русланка, Дракон.

10. Базовый класс: Гаджеты. Дочерние классы: Часы, Смартфон, Ноутбук.

11. Базовый класс: Бытовая техника. Дочерние классы: Холодильник, Посудомоечная машина, Пылесос.

12. Базовый класс: Приложение. Дочерние классы: Социальная сеть, Игра, Погода.

13. Базовый класс: Оружие. Дочерние классы: Меч, Лук, Волшебная палочка.

14. Базовый класс: Заведение. Дочерние классы: Кафе, Магазин, Библиотека.

15. Базовый класс: Компьютерная периферия. Дочерние классы: Клавиатура, Наушники, Графический планшет.

Предметная область может быть выбрана другая по согласованию с преподавателем.

## Контрольные вопросы

1. Что такое абстракция и как она реализуется в языке Java?
2. Что такое инкапсуляция и как она реализуется в языке Java?
3. Что такое наследование и как оно реализуется в языке Java?
4. Что такое полиморфизм и как он реализуется в языке Java?
5. Что такое множественное наследование и есть ли оно в Java?
6. Для чего нужно ключевое слово final?
7. Какие в Java есть модификаторы доступа?
8. Что такое конструктор? Какие типы конструкторов бывают в Java?
9. Для чего нужно ключевое слово this в Java?
10. Для чего нужно ключевое слово super в Java?
11. Что такое геттеры и сеттеры? Зачем они нужны?
12. Что такое переопределение?
13. Что такое перегрузка?

### 3 Класс Object. Работа с хэш-таблицами

---

Класс `Object` является базовым классом всей иерархии классов в Java. Все остальные классы, включая пользовательские, автоматически наследуют этот класс, даже если явного указания нет. Это значит, что каждый объект в Java имеет методы, определенные в классе `Object`.

Основные методы класса `Object`:

- `toString()` — возвращает строковое представление объекта. По умолчанию возвращает имя класса и хеш-код объекта, но часто его переопределяют для получения более информативного результата;
- `equals(Object obj)` — проверяет равенство двух объектов. По умолчанию сравниваются ссылки на объекты (`==`), поэтому обычно требуется переопределение для сравнения содержимого объектов;
- `hashCode()` — возвращает хеш-код объекта. Хеш-коды используются коллекциями типа `HashMap` и `HashSet` для оптимизации поиска элементов. Важно, чтобы два равных объекта возвращали одинаковый хеш-код;
- `getClass()` — возвращает объект класса `Class`, который представляет текущий класс объекта;
- `clone()` — создает копию текущего объекта. Метод объявлен как `protected`, так что его нужно переопределить, если необходимо использовать;
- `wait()`, `notify()`, `notifyAll()` — методы, используемые для синхронизации потоков. Эти методы работают совместно с ключевым словом `synchronized`;
- `finalize()` — вызывается сборщиком мусора перед удалением объекта. Этот метод редко используется, поскольку поведение сборщика мусора непредсказуемо.

По умолчанию метод `equals()` проверяет идентичность ссылок (`==`), что зачастую недостаточно. Например, у вас есть два

объекта одного класса, содержащие одинаковые данные, но имеющие разные ссылки. В этом случае логично считать такие объекты равными. Для этого метод equals() следует переопределить так, чтобы сравнивать содержимое объектов. Код переопределения метода equals() представлен на листинге 3.1.

#### Листинг 3.1. Переопределение метода equals()

```
1. @Override
2. public boolean equals(Object obj) {
3.     if (this == obj) return true;
4.     if (obj == null || getClass() != obj.getClass()) return false;
5.
6.     YourClass other = (YourClass) obj;
7.     return this.field1.equals(other.field1)
8.         && this.field2 == other.field2;
9. }
```

Когда вы переопределяете метод equals(), важно также переопределить метод hashCode(). Причина заключается в следующем:

- два объекта, которые считаются равными методом equals(), должны иметь одинаковое значение хеш-кода;
- коллекции, основанные на хэш-таблицах (например, HashMap, HashSet), используют хеш-коды для эффективного хранения и поиска данных. Несогласованность между методами equals() и hashCode() приведет к неправильной работе этих структур данных. Пример переопределения метода hashCode() представлен на листинге 3.2.

#### Листинг 3.2. Переопределение метода hashCode()

```
1. @Override
2. public int hashCode() {
3.     int result = field1.hashCode();
4.     result = 31 * result + field2;
5.     return result;
6. }
```

Важно соблюдать контракт между этими двумя методами:

- если два объекта равны согласно equals(), то их хеш-коды должны совпадать;
- если два объекта имеют одинаковый хеш-код, это не обязательно означает, что они равны.

Переопределение этих методов помогает обеспечить корректную работу ваших объектов в различных ситуациях, особенно

когда речь идет о сравнении объектов и использовании их в коллекциях.

Хэш-таблица — это структура данных, которая использует-ся для хранения пар «ключ-значение». Она основана на идее хэш-функции, которая преобразует ключ в индекс массива, где хранится значение.

Например, если мы хотим хранить и получать значения по ключу для следующих пар: («apple», 5), («banana», 3), («orange», 7), то мы можем использовать хэш-таблицу следующим образом:

```
Hash table:  
Index 0:  
Index 1:  
Index 2: («banana», 3)  
Index 3: («orange», 7)  
Index 4:  
Index 5: («apple», 5)  
Index 6:
```

Здесь ключи «apple», «banana» и «orange» были преобра-зованы в индексы массива с помощью хэш-функции. Значения хранятся в связанных списках, которые могут быть добавлены к соответствующему индексу.

Хэш-функция должна быть быстрой и однозначной, то есть каждому ключу должен соответствовать уникальный индекс. Если два разных ключа преобразуются в один и тот же индекс, то это называется коллизией.

Коллизии могут быть разрешены различными способами, например с помощью метода цепочек или открытой адресации.

В методе цепочек каждый индекс массива содержит связанный список пар «ключ-значение». Если происходит коллизия, то новая пара добавляется в конец списка. При поиске значе-ния по ключу нужно просмотреть все элементы списка, начиная с первого.

Например, если мы добавим еще одну пару («pear», 2), то наша хэш-таблица будет выглядеть следующим образом:

```
Hash table:  
Index 0:  
Index 1:  
Index 2: («banana», 3) -> («pear», 2)  
Index 3: («orange», 7)  
Index 4:
```

Index 5: («apple», 5)

Index 6:

Здесь ключ «pear» также преобразован в индекс 2, где уже находится пара «banana» — поэтому новая пара добавляется в конец списка.

В Java хеш-таблицы реализованы в виде классов `HashMap` и `Hashtable`. Они имеют похожий интерфейс, но различаются в том, что `HashMap` не является потокобезопасным, а `Hashtable` является.

## Задания для выполнения лабораторной работы

### Задание 1.

1. Создайте класс `HashTable`, который будет реализовывать хеш-таблицу с помощью метода цепочек.
2. Реализуйте методы `put(key, value)`, `get(key)` и `remove(key)`, которые добавляют, получают и удаляют пары «ключ-значение» соответственно.
3. Добавьте методы `size()` и `isEmpty()`, которые возвращают количество элементов в таблице и проверяют, пуста ли она.

Пример реализации метода `put(key, value)` представлен на листинге 3.3.

**Листинг 3.3.** Реализация метода `put(key, value)`

```
1. public void put(K key, V value) {  
2.     int index = hash(key);  
3.     if (table[index] == null) {  
4.         table[index] = new LinkedList<Entry<K, V>>();  
5.     }  
6.     for (Entry<K, V> entry : table[index]) {  
7.         if (entry.getKey().equals(key)) {  
8.             entry.setValue(value);  
9.             return;  
10.        }  
11.    }  
12.    table[index].add(new Entry<K, V>(key, value));  
13.    size++;  
14. }
```

**Задание 2. Работа с встроенным классом HashMap.****Варианты**

1. Реализация хэш-таблицы для хранения информации о студентах. Ключом является номер зачетной книжки, а значением — объект класса Student, содержащий поля имя, фамилия, возраст и средний балл. Необходимо реализовать операции вставки, поиска и удаления студента по номеру зачетки.

2. Реализация хэш-таблицы для хранения информации о товарах в интернет-магазине. Ключом является артикул товара, а значением — объект класса Product, содержащий поля наименование, описание, цена и количество на складе. Необходимо реализовать операции вставки, поиска и удаления товара по артикулу.

3. Реализация хэш-таблицы для хранения информации о заказах в интернет-магазине. Ключом является номер заказа, а значением — объект класса Order, содержащий поля дата заказа, список товаров и статус заказа. Необходимо реализовать операции вставки, поиска и удаления заказа по номеру. Также необходимо реализовать метод для изменения статуса заказа.

4. Реализация хэш-таблицы для хранения информации о книгах в библиотеке. Ключом будет ISBN книги, а значением — объект класса Book, содержащий информацию о названии, авторе и количестве копий. Необходимо реализовать операции вставки, поиска и удаления книги по ISBN.

5. Реализация хэш-таблицы для учета продуктов на складе. Ключом будет штрихкод товара, а значением — объект класса Product, содержащий информацию о названии, цене и доступном количестве. Необходимо реализовать операции вставки, поиска и удаления продукта по штрихкоду.

6. Реализация хэш-таблицы для хранения контактов в телефонной книге. Ключом будет номер телефона, а значением — объект класса Contact, содержащий имя, адрес электронной почты и дополнительные контактные данные. Необходимо реализовать операции вставки, поиска и удаления контактов по номеру телефона.

7. Реализация хэш-таблицы для учета заказов в интернет-магазине. Ключом будет номер заказа, а значением — объект класса Order, содержащий информацию о товарах, адресе доставки и стоимости заказа. Необходимо реализовать операции вставки, поиска и удаления заказа по номеру заказа.

8. Реализация хэш-таблицы для хранения информации о сотрудниках в компании. Ключом будет идентификационный номер сотрудника, а значением — объект класса Employee, содержащий данные о имени, должности и заработной плате. Необходимо реализовать операции вставки, поиска и удаления сотрудника по ID.

9. Реализация хэш-таблицы для учета автомобилей в автопарке. Ключом будет номерной знак автомобиля, а значением — объект класса Car, содержащий информацию о марке, модели и году выпуска. Необходимо реализовать операции вставки, поиска и удаления автомобиля по знаку.

10. Реализация хэш-таблицы для хранения данных о заказах в ресторане. Ключом будет номер столика или заказа, а значением — объект класса Order, содержащий информацию о блюдах, стоимости и времени заказа. Необходимо реализовать операции вставки, поиска и удаления заказа по номеру столика.

## Контрольные вопросы

1. Для чего нужен класс Object?
2. Для чего нужно переопределять методы equals() и hashCode()?
3. Какие есть правила переопределения методов equals() и hashCode()?
4. Что делает метод toString()? Почему его часто переопределяют?
5. Что делает метод finalize()? Почему его использование считается устаревшим (deprecated)?
6. Что такое коллизия?
7. Какие есть способы разрешения коллизий?
8. Как хранятся данные в хэш-таблице?
9. Что происходит, если в хэш-таблицу добавить элемент с одинаковым значением ключа?
10. Что происходит, если в хэш-таблицу добавить элемент с таким же хэш-кодом ключа, но разными исходными значениями?
11. Как изменяется HashMap при достижении порогового значения?

## 4 Обработка исключений

---

Исключения в Java являются механизмом обработки ошибок и нестандартных ситуаций во время выполнения программы. Они позволяют изолировать ошибки и обрабатывать их в специальном блоке кода, что делает программу более надежной и устойчивой к ошибкам.

Все исключения в Java являются потомками класса Throwable. От него наследуются два класса:

- Error — это ошибки, возникающие при выполнении программы в результате сбоя работы JVM, переполнения памяти или сбоя системы. Обычно они свидетельствуют о серьезных проблемах, устранить которые программными средствами невозможно. Такой вид исключений в Java относится к непроверяемым (unchecked) на стадии компиляции;
- Exception — обычные ошибки, которые возникают во время работы многих методов, например ошибки открытия файла.

Среди наследников класса Exception есть класс RuntimeException — это суперкласс тех исключений, которые могут быть выброшены во время нормальной работы JVM, например IndexOutOfBoundsException — когда индекс находится вне пределов массива.

Класс Exception и все его подклассы, которые не являются также подклассами RuntimeException, являются проверяемыми исключениями. Проверяемые исключения должны быть задекларированы в заголовке метода при помощи ключевого слова throws, если они могут быть выброшены при выполнении метода или конструктора и распространиться за пределы границы метода или конструктора или обработаны в том методе, где они могут возникнуть.

Декларация исключения в методе в Java означает указание типов исключений, которые данный метод может генерировать во время своего выполнения.

```

1. void methodName() throws ExceptionType1,
2.           ExceptionType2, ... {
3.     // тело метода
4. }
```

Error, RuntimeException и его подклассы являются непроверяемыми исключениями. Непроверяемые исключения не нужно объявлять в поле throws метода или конструктора, если они могут быть выброшены при выполнении метода или конструктора и распространяться за пределы границ метода или конструктора.

Иерархия классов исключений представлена на рис. 4.1.

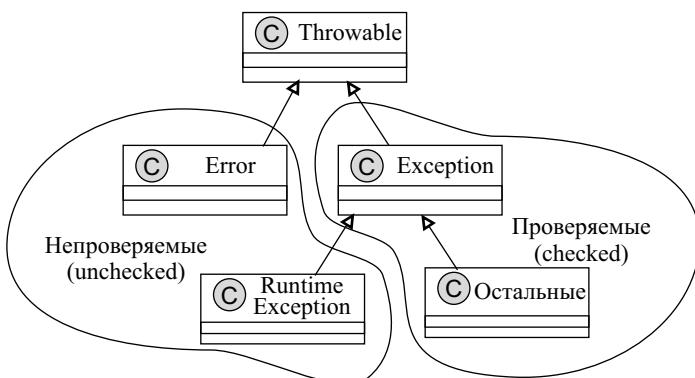


Рис. 4.1. Иерархия классов исключений

Исключения в Java помогают эффективно обрабатывать ошибки и непредвиденные ситуации, делая код более надежным и устойчивым. Правильное использование исключений способствует созданию качественного программного обеспечения.

В Java есть множество типов исключений, которые могут возникнуть в различных ситуациях. Например, классы IOException и FileNotFoundException используются для обработки ошибок ввода-вывода, а классы NullPointerException и ArrayIndexOutOfBoundsException используются для обработки ошибок, связанных с работой с объектами и массивами.

Кроме того, в Java можно создавать свои собственные исключения, наследуясь от класса Exception или его подклассов. Это позволяет создавать более гибкую и точечную обработку ошибок в своих программах.

При работе с исключениями важно учитывать, что блок try-catch не должен заменять проверку условий и предотвращение

возможных ошибок на этапе написания кода. Он должен использоваться только для обработки неожиданных ошибок, которые могут возникнуть в процессе выполнения программы. Код, который может вызвать исключение, помещается в блок `try`. Если исключение возникает внутри блока `try`, то управление передается в блок `catch`, который обрабатывает исключение. Блок `catch` может содержать несколько разделов, каждый из которых обрабатывает определенный тип исключения.

```
1. try {  
2.     // Код, который может вызвать исключение  
3. } catch (ExceptionType1 e) {  
4.     // Обработка исключения типа ExceptionType1  
5. } catch (ExceptionType2 e) {  
6.     // Обработка исключения типа ExceptionType2  
7. } finally {  
8.     // Код, который выполняется всегда, независимо от того,  
9.     // возникло ли исключение или нет  
10. }
```

Блок `finally` необязателен, но если он присутствует, то код внутри него будет выполняться всегда, независимо от того, возникло ли исключение или нет.

## Задания для выполнения лабораторной работы

### Задание 1.

Написать программу, которая будет находить среднее арифметическое элементов массива. При этом программа должна обрабатывать ошибки, связанные с выходом за границы массива и неверными данными (например, если элемент массива не является числом). Пример реализации метода:

```
1. public class ArrayAverage {  
2.     public static void main(String[] args) {  
3.         int[] arr = {1, 2, 3, 4, 5};  
4.         int sum = 0;  
5.         try {  
6.             // Обработка массива  
7.         }  
8.         // Вывод массива  
9.     } catch (*исключение*) {  
10.         // Вывод исключения  
11.     } catch (*исключение*) {
```

```
12.         // Вывод исключения  
13.     }  
14. }  
15. }
```

### Задание 2.

Написать программу, которая будет копировать содержимое одного файла в другой. При этом программа должна обрабатывать возможные ошибки, связанные:

- с открытием и закрытием файлов (вариант 1);
- чтением и записью файлов (вариант 2).

### Задание 3.

Создайте Java-проект для работы с исключениями. Для каждой из восьми указанных ниже задач напишите собственный класс для обработки исключений. *Создайте обработчик исключений, который логирует информацию о каждом выброшенном исключении в текстовый файл.*

#### Варианты задач

1. Создайте класс CustomDivisionException, который будет использоваться для обработки исключений при делении на ноль. Напишите программу, которая делит два числа, и, если происходит деление на ноль, выбрасывайте исключение CustomDivisionException.

2. Создайте класс CustomAgeException, который будет использоваться для обработки недопустимых возрастов. Реализуйте программу, которая проверяет возраст пользователя с использованием этого класса, и, если возраст меньше 0 или больше 120, выбрасывайте исключение CustomAgeException.

3. Создайте класс CustomFileNotFoundException, который будет использоваться для обработки исключения FileNotFoundException. Напишите программу для чтения файла и, если файл не существует, выбрасывайте исключение CustomFileNotFoundException.

4. Создайте класс CustomNumberFormatException, который будет использоваться для обработки исключения NumberFormatException. Реализуйте программу, которая пытается преобразовать строку в число (Integer.parseInt()), и, если строка не является числом, выбрасывайте исключение CustomNumberFormatException.

5. Создайте класс CustomEmptyStackException, который будет использоваться для обработки исключения EmptyStackException.

tion. Напишите класс CustomStack, имитирующий стек, и, если происходит попытка извлечь элемент из пустого стека, выбрасывайте исключение CustomEmptyStackException.

6. Создайте класс CustomInputMismatchException, который будет использоваться для обработки исключения InputMismatchException. Напишите программу, которая считывает целое число с клавиатуры, и, если ввод пользователя не является числом, выбрасывайте исключение CustomInputMismatchException.

7. Создайте класс CustomEmailFormatException, который будет использоваться для обработки недопустимого формата email-адреса. Реализуйте программу, которая проверяет формат email-адреса с использованием этого класса, и, если адрес не соответствует формату, выбрасывайте исключение CustomEmailFormatException.

8. Создайте класс CustomUnsupportedOperationException, который будет использоваться для обработки исключения UnsupportedOperationException. Реализуйте программу, которая выполняет математические операции (сложение, вычитание, умножение, деление) с помощью собственного класса и выбрасывайте исключение CustomUnsupportedOperationException, если операция не поддерживается.

## Контрольные вопросы

1. Что такое исключение в Java?
2. Какие ключевые классы исключений вы знаете?
3. Что такое проверяемые и непроверяемые исключения?
4. Какие исключения и как необходимо обрабатывать?
5. Какие исключения относятся к классу Error и как их обрабатывать?
6. Какие исключения относятся к классу RuntimeException и как их обрабатывать?
7. Как создать собственный класс исключения?
8. Как обрабатываются исключения в Java? Какие конструкции для этого используются?
9. Можно ли использовать try без catch или finally?
10. Что произойдет, если исключение возникнет в блоке finally?
11. Как можно пробросить исключение выше по стеку вызовов?
12. В чем разница между finally и try-with-resources?

13. Какие классы можно использовать в try-with-resources?  
Что такое AutoCloseable?
14. Можно ли в одном try использовать несколько catch-блоков? В каком порядке их нужно располагать?
15. В чем разница между throw и throws?
16. Что такое StackOverflowError и OutOfMemoryError? Можно ли их обработать?

## 5 Строки и регулярные выражения

---

Класс String в Java представляет собой последовательность символов. Он является одним из самых важных и часто используемых классов в стандартной библиотеке Java. После создания объекта типа String его содержимое не может быть изменено. Попытка изменения строки фактически приводит к созданию нового объекта типа String.

1. String str = "Hello"; // Создаем строку "Hello"
2. str += ", World!"; // Пытаемся добавить к строке ", World!"
3. System.out.println(str); // Выведет "Hello, World!"

На первый взгляд кажется, что строка изменилась, однако на самом деле была создана новая строка "Hello, World!", а переменная str теперь указывает на эту новую строку.

Преимущества неизменяемости:

- безопасность многопоточного доступа — так как строки неизменяемы, они безопасны для использования в многопоточных приложениях;
- кэширование — неизменяемые объекты могут быть кэшированы и повторно использованы, что повышает производительность.

Недостатки неизменяемости:

- операции над строками могут приводить к созданию большого количества временных объектов, что увеличивает нагрузку на сборщик мусора.

Для работы со строками, которые часто меняются, лучше использовать класс StringBuilder или StringBuffer.

*Интернирование* — это процесс, при котором компилятор и виртуальная машина JVM пытаются минимизировать количество дублирующихся строковых литералов путем сохранения их в специальном пуле (pool). Когда создается новый строковый литерал, JVM сначала проверяет, есть ли уже такая строка в пу-

ле. Если да, то возвращается ссылка на существующий объект вместо создания нового.

Пример:

1. String s1 = "Hello";
2. String s2 = "Hello";
3. System.out.println(s1 == s2); // Выведет true, так как обе переменные
4. // ссылаются на одну и ту же строку в пул

Интернирование происходит автоматически для строковых литералов, но для других строк оно может быть выполнено вручную с помощью метода `intern()`:

1. String s3 = new String("World").intern(); // Добавляем строку
2. // "World" в пул
3. String s4 = "World";
4. System.out.println(s3 == s4); // Выведет true, так как обе переменные
5. // ссылаются на одну и ту же строку в пул

Строки в Java представляют собой последовательности символов Unicode. Каждый символ Unicode представлен своим уникальным числовым значением, называемым **code point**. В отличие от многих других языков программирования, Java использует UTF-16 для представления строк, что означает, что каждый **code unit** занимает 16 битов (2 байта). Однако некоторые символы Unicode требуют больше, чем 16 битов для представления.

Длина строки в **code points** — это количество уникальных символов Unicode в строке. Для определения этой длины можно воспользоваться методом `length()` класса `String`, который возвращает количество символов в строке.

1. String str = "Привет!";
2. int lengthInCodePoints = str.length();
3. System.out.println(lengthInCodePoints); // Выведет 8

Длина строки в **code units** — это количество 16-битных единиц, необходимых для представления всех символов строки. Для определения этой длины можно воспользоваться методом `codePointCount()` класса `String`.

1. String str = "Привет!";
2. int lengthInCodeUnits = str.codePointCount(0, str.length());
3. System.out.println(lengthInCodeUnits); // Выведет 8

Этот метод учитывает каждую 16-битную единицу отдельно, включая те случаи, когда один символ представлен суррогатной парой.

Рассмотрим строку, содержащую эмодзи  (Земля).

1. String emojiStr = " Привет!";
  2. int lengthInCodePoints = emojiStr.length();
  3. int lengthInCodeUnits = emojiStr.codePointCount(0, emojiStr.length());
  - 4.
  5. System.out.println("Длина в code points: " + lengthInCodePoints); //
- Выведет 9
6. System.out.println("Длина в code units: " + lengthInCodeUnits); //
- Выведет 11
- 7.

Как видно из примера, длина строки в **code points** равна 9, потому что эмодзи  считается одним символом. Но длина в **code units** равна 11, потому что эмодзи  представлен суррогатной парой, занимающей две 16-битные единицы.

Регулярные выражения (Regular expressions) — это специальный язык для поиска и обработки текстовой информации, который широко используется в программировании и компьютерных науках. В Java регулярные выражения реализованы в классе `java.util.regex.Pattern`.

Синтаксис регулярных выражений в Java состоит из специальных символов, которые обозначают определенные шаблоны текста. Например:

- `". "` обозначает любой символ;
- `"*"` обозначает повторение предыдущего символа или группы символов;
- `"+"` обозначает повторение предыдущего символа или группы символов один или более раз;
- `"?"` обозначает, что предыдущий символ или группа символов может быть присутствовать или отсутствовать;
- `"^"` обозначает начало строки;
- `"$"` обозначает конец строки;
- `"[]"` обозначает набор символов, которые могут быть использованы в данной позиции;
- `"[^ ]"` обозначает набор символов, которые не могут быть использованы в данной позиции;
- `"\\b"` обозначает границу слова;
- `"\\d"` обозначает любую цифру;
- `"\\s"` обозначает любой пробельный символ;

- "\w" обозначает любую букву или цифру.

Кроме того, можно использовать скобки для группировки символов и задания приоритета операций. Например:

- "(abc)+" означает повторение группы символов "abc" один или более раз;
- "(abc|def)" означает, что символы "abc" или "def" могут быть использованы в данной позиции.

Также в Java можно использовать специальные символы для задания квантификаторов, которые определяют количество повторений символов или групп символов. Например:

- "{n}" означает точное количество повторений (например, "\d{3}" означает три цифры подряд);
- "{n,}" означает, что символ или группа символов должны повторяться не менее n раз.

В Java регулярные выражения реализованы в классе `java.util.regex.Pattern`, который имеет методы для компиляции регулярного выражения и поиска совпадений в тексте. Класс `Matcher` используется для поиска совпадений в тексте и замены найденных совпадений.

Примеры использования регулярных выражений в Java:

- поиск слова в тексте:

```
1. String text = "The quick brown fox jumps over the lazy dog";
2. Pattern pattern = Pattern.compile("fox");
3. Matcher matcher = pattern.matcher(text);
4. if (matcher.find()) {
5.     System.out.println("Match found!");
6. }
```

- замена слова в тексте:

```
1. String text = "The quick brown fox jumps over the lazy dog";
2. Pattern pattern = Pattern.compile("fox");
3. Matcher matcher = pattern.matcher(text);
4. String result = matcher.replaceAll("cat");
5. System.out.println(result);
```

- проверка корректности email-адреса:

```
1. String email = "example@mail.com";
2. Pattern pattern = Pattern.compile("\b[A-Za-z0-9._%+-]+\@[A-Za-z0-
9.-]+\.\w{2,}\b");
3. Matcher matcher = pattern.matcher(email);
4. if (matcher.matches()) {
5.     System.out.println("Valid email address!");
6. }
```

## Задания для выполнения лабораторной работы

### Задание 1. Поиск всех чисел в тексте

Написать программу, которая будет искать все числа в заданном тексте и выводить их на экран. При этом программа должна использовать регулярные выражения для поиска чисел и обрабатывать возможные ошибки. Пример реализации такого функционала представлен на листинге 5.1.

#### Листинг 5.1. Класс NumberFinder

```
1. import java.util.regex.*;  
2.  
3. public class NumberFinder {  
4.     public static void main(String[] args) {  
5.         String text = "The price of the product is $19.99";  
6.         Pattern pattern = Pattern.compile("\\d+\\.\\d+");  
7.         Matcher matcher = pattern.matcher(text);  
8.         while (matcher.find()) {  
9.             System.out.println(matcher.group());  
10.        }  
11.    }  
12. }
```

### Задание 2. Проверка корректности ввода пароля

Написать программу, которая будет проверять корректность ввода пароля. Пароль должен состоять из латинских букв и цифр, быть длиной от 8 до 16 символов и содержать хотя бы одну заглавную букву и одну цифру. При этом программа должна использовать регулярные выражения для проверки пароля и обрабатывать возможные ошибки.

### Задание 3. Поиск заглавной буквы после строчной

Написать программу, которая будет находить все случаи в тексте, когда сразу после строчной буквы идет заглавная без какого-либо символа между ними и выделять их знаками «!» с двух сторон.

### Задание 4. Проверка корректности ввода IP-адреса

Написать программу, которая будет проверять корректность ввода IP-адреса. IP-адрес должен состоять из 4 чисел, разделенных точками, и каждое число должно быть в диапазоне от 0 до 255. При этом программа должна использовать регулярные выражения для проверки IP-адреса и обрабатывать возможные ошибки.

### Задание 5. Поиск всех слов, начинающихся с заданной буквы

Написать программу, которая будет искать все слова в заданном тексте, начинающиеся с заданной буквы, и выводить их на экран. При этом программа должна использовать регулярные выражения для поиска слов и обрабатывать возможные ошибки.

### Контрольные вопросы

1. Что такое класс String?
2. Почему объект класса String является иммутабельным?
3. Что такое интернирование строк?
4. В чем разница между String, StringBuilder и StringBuffer?
5. Как сравнить две строки? В чем разница между ==, equals() и equalsIgnoreCase()?
6. Как хранятся строки в памяти? Что такое пул строк?
7. Что такое code unit и code point?
8. Что необходимо для использования регулярных выражений в Java?
9. Какие есть режимы работы квантификатора?
10. Как проверить, соответствует ли строка регулярному выражению?
11. Как найти все вхождения подстроки по шаблону?
12. Как разбить строку по регулярному выражению?
13. Как заменить подстроку по шаблону?
14. Как экранировать спецсимволы в регулярных выражениях?

## 6 Работа с коллекциями

---

Коллекции в Java представляют собой классы и интерфейсы, которые служат для хранения и управления группами объектов. Они позволяют удобно и эффективно работать с данными, осуществлять поиск, добавление, удаление и изменение элементов коллекций. Основные интерфейсы и классы коллекций представлены на рис. 6.1, 6.2.

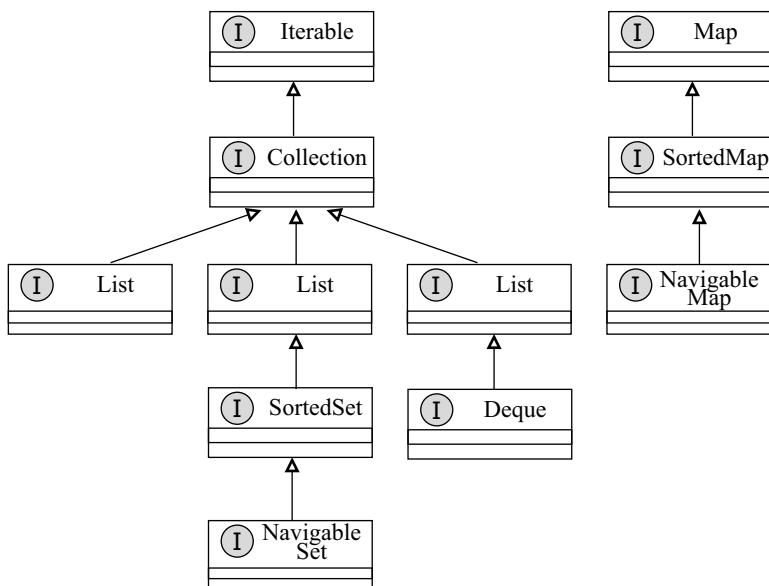


Рис. 6.1. Основные интерфейсы коллекций

В Java существует несколько основных интерфейсов коллекций:

- **Collection** — базовый интерфейс коллекций, который содержит методы для работы с элементами коллекции;

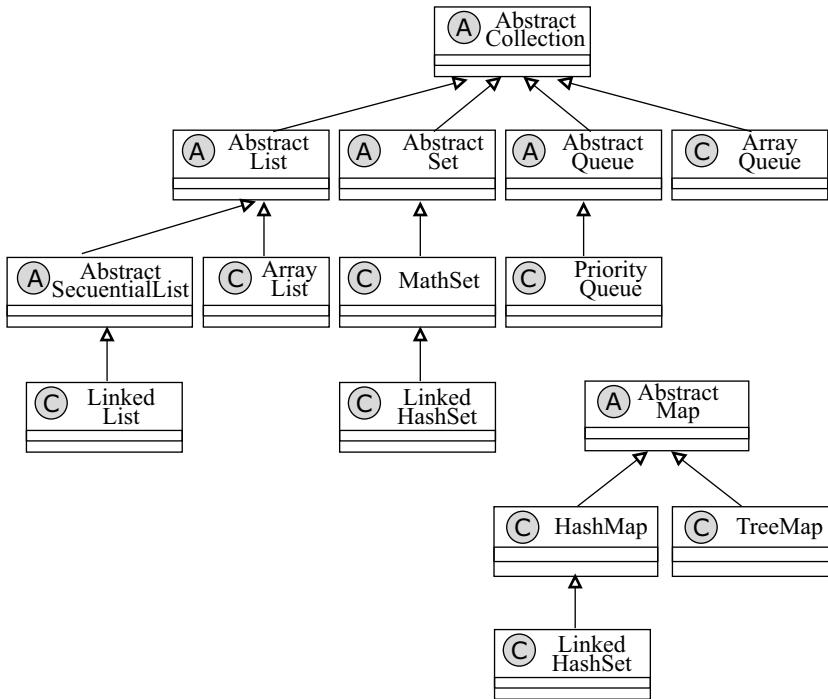


Рис. 6.2. Основные классы коллекций

- **List** — представляет упорядоченную коллекцию элементов, допускающую дубликаты. Основные реализации: **ArrayList**, **LinkedList**;
- **Set** — представляет коллекцию, в которой каждый элемент уникален. Основные реализации: **HashSet**, **TreeSet**;
- **Queue** — представляет очередь, в которой элементы извлекаются в порядке FIFO (First In First Out). Основные реализации: **LinkedList**, **PriorityQueue**;
- **Deque** — представляет двустороннюю очередь, поддерживающую вставку и извлечение элементов с обеих сторон. Основная реализация: **ArrayDeque**;
- **Map** — представляет ассоциативный массив, где каждому ключу соответствует одно значение. Основные реализации: **HashMap**, **TreeMap**.

Коллекции в Java предоставляют мощные инструменты для работы с группами объектов. Выбор конкретной коллекции зависит от требований задачи, таких как необходимость поддержания порядка, уникальности элементов или быстрого доступа к дан-

ным. `ArrayList` и `LinkedList` — это две коллекции в Java, которые используются для доступа по индексу элемента.

`ArrayList` — это список, реализованный на основе массива, а `LinkedList` — это классический связный список, основанный на объектах с ссылками между ними.

Различия между `ArrayList` и `LinkedList` заключаются в том, что `ArrayList` основан на массиве, а `LinkedList` — на связном списке. `ArrayList` обеспечивает быстрый доступ к элементам по индексу, но медленно работает при добавлении или удалении элементов из середины списка. `LinkedList` обеспечивает быстрое добавление и удаление элементов из середины списка, но медленно работает при доступе к элементам по индексу.

Примеры использования коллекций в Java:

- создание списка и добавление элементов:

```
1. List<String> list = new ArrayList<>();  
2. list.add("apple");  
3. list.add("banana");  
4. list.add("orange");
```

- использование цикла для обхода элементов списка:

```
1. for (String fruit : list) {  
2.     System.out.println(fruit);  
3. }
```

- удаление элемента из списка:

```
1. list.remove("banana");
```

*Дженерики* в Java — это механизм, позволяющий создавать классы, интерфейсы и методы, работающие с различными типами данных без необходимости переписывать код для каждого типа. Это улучшает типобезопасность и сокращает количество ошибок времени выполнения.

Тип-параметр — имя, используемое для обозначения типа, который будет заменен конкретным типом при использовании параметризованного типа. Например, в `List<E>` `E` — это тип-параметр.

Можно ограничить тип-параметры, чтобы они могли принимать только определенные типы. Это достигается с помощью ключевых слов `extends` и `super`.

```
1. public class NumberBox< T extends Number> {  
2.     private T item;  
3. }
```

```
4.     public void setItem(T item) {  
5.         this.item = item;  
6.     }  
7.  
8.     public T getItem() {  
9.         return item;  
10.    }  
11. }
```

Теперь NumberBox может работать только с подтипами Number (например, Integer, Double и т. д.).

Стирание типов (type erasure) в Java — это процесс, при котором информация о типах параметров, использующихся в дженериках, стирается во время компиляции. Это означает, что на этапе выполнения программы информация о конкретных типах, указанных в параметрах дженериков, отсутствует. Вместо этого используются необобщённые версии классов и интерфейсов.

При компиляции кода, использующего дженерики, компилятор выполняет следующие шаги.

1. Проверка типов. Компилятор проверяет правильность использования дженериков и соответствие типов параметрам.

2. Замена параметров. Компилятор заменяет параметры дженериков на их границы (если они указаны) или на Object (если границы не указаны).

3. Вставка кастинга. Компилятор добавляет необходимые приведения типов (casts) там, где это необходимо.

4. Удаление информации о типах. Вся информация о типах параметров удаляется, оставляя только необобщённую версию класса или интерфейса.

Рассмотрим следующий код:

```
1. public class Box<T> {  
2.     private T item;  
3.  
4.     public void setItem(T item) {  
5.         this.item = item;  
6.     }  
7.  
8.     public T getItem() {  
9.         return item;  
10.    }  
11. }
```

После компиляции и стирания типов этот код преобразуется в следующее:

```
1. public class Box {  
2.     private Object item;  
3.  
4.     public void setItem(Object item) {  
5.         this.item = item;  
6.     }  
7.  
8.     public Object getItem() {  
9.         return item;  
10.    }  
11. }
```

Ограничения, вызванные стиранием типов:

- невозможность создания массива дженериков — нельзя создать массив параметризованных типов, например new T[10] или new List<String>[10];
- ограничения на создание экземпляров — невозможно создать экземпляр параметризованного типа с использованием оператора new, например new T();
- отсутствие информации о типах на этапе выполнения — на этапе выполнения невозможно узнать конкретный тип параметра дженерика. Например, выражение

```
list instanceof ArrayList<String>
```

не сработает, так как информация о типе String стерта.

Примеры использования дженериков в коллекциях и методах:

- объявление списка с использованием дженериков:

```
1. List<Integer> numbers = new ArrayList<>();
```

- использование дженериков в методе:

```
1. public static <T> T getFirst(List<T> list) {  
2.     return list.get(0);  
3. }
```

- использование дженериков в интерфейсе:

```
1. public interface Pair<K, V> {  
2.     K getKey();  
3.     V getValue();  
4. }
```

*Итераторы* в Java представляют собой объекты, которые позволяют последовательно обходить элементы коллекции. Они позволяют безопасно и эффективно перебирать элементы коллекции, не заботясь о внутреннем устройстве коллекции.

Примеры использования итераторов в коллекциях:

- использование итератора для обхода элементов списка:

```
1. Iterator< String > iterator = list.iterator();
2. while (iterator.hasNext()) {
3.     String fruit = iterator.next();
4.     System.out.println(fruit);
5. }
```

- удаление элемента из списка с помощью итератора:

```
1. Iterator< String > iterator = list.iterator();
2. while (iterator.hasNext()) {
3.     String fruit = iterator.next();
4.     if (fruit.equals("banana")) {
5.         iterator.remove();
6.     }
7. }
```

## Задания для выполнения лабораторной работы

### Задание 1.

Написать программу, которая считывает текстовый файл и выводит на экран топ-10 самых часто встречающихся слов в этом файле. Для решения задачи использовать коллекцию Map, где ключом будет слово, а значением — количество его повторений в файле. Пример реализации данного функционала представлен на листинге 6.1.

#### Листинг 6.1. Пример реализации класса TopWord

```
1. import java.io.File;
2. import java.io.FileNotFoundException;
3. import java.util.*;
4.
5. public class TopWords {
6.     public static void main(String[] args) {
7.         // указываем путь к файлу
8.         String filePath = "C:\\text.txt";
9.         // создаем объект File
10.        File file = new File(filePath);
11.        // создаем объект Scanner для чтения файла
```

```
12.     Scanner scanner = null;
13.     try {
14.         scanner = new Scanner(file);
15.     } catch (FileNotFoundException e) {
16.         e.printStackTrace();
17.     }
18.     // создаем объект Map для хранения слов и их количества
19.     ******
20.     // читаем файл по словам и добавляем их в Map
21.     *****
22.     // закрываем Scanner
23.     *****
24.     // создаем список из элементов Map
25.     *****
26.     // сортируем список по убыванию количества повторений
27.     Collections.sort(list, new Comparator< Map.Entry< String,
28.                         Integer> >() {
29.         @Override
30.         public int compare(Map.Entry< String, Integer> o1,
31.                             Map.Entry< String, Integer> o2) {
32.             *****
33.         }
34.     });
35.     // выводим топ-10 слов
36.     *****
37. }
38. // выводим результат
39. *****
40. }
41. }
42. }
```

### Задание 2:

Написать обобщенный класс `Stack< T >`, который реализует стек на основе массива. Класс должен иметь методы `push` для добавления элемента в стек, `pop` для удаления элемента из стека и `peek` для получения верхнего элемента стека без его удаления. Пример реализации обобщенного класса `Stack< T >` представлен на листинге 6.2.

### Листинг 6.2. Пример реализации класса `Stack< T >`

```
1. public class Stack< T > {
2.     private T[] data;
3.     private int size;
4. }
```

```
5.     public Stack(int capacity) {  
6.         data = (T[]) new Object[capacity];  
7.         size = 0;  
8.     }  
9.  
10.    public void push(T element) {  
11.        *****  
12.    }  
13.  
14.    public T pop() {  
15.        *****  
16.    }  
17.  
18.    public T peek() {  
19.        *****  
20.    }  
21. }
```

Пример использования:

1. Stack< Integer > stack = new Stack< > (10);
2. stack.push(1);
3. stack.push(2);
4. stack.push(3);
5. System.out.println(stack.pop());
6. System.out.println(stack.peek());
7. stack.push(4);
8. System.out.println(stack.pop());

### Задание 3.

Разработать программу для учета продаж в магазине. Программа должна позволять добавлять проданные товары в коллекцию, выводить список проданных товаров, а также считать общую сумму продаж и наиболее популярный товар.

Варианты выполнения задания:

1. Использовать ArrayList для хранения списка проданных товаров.
2. Использовать LinkedList для хранения списка проданных товаров.
3. Использовать HashSet для хранения списка проданных товаров.
4. Использовать TreeSet для хранения списка проданных товаров.
5. Использовать HashMap для хранения пар «товар – количество продаж».

6. Использовать TreeMap для хранения пар «товар – количество продаж».
7. Использовать LinkedHashMap для хранения пар «товар – количество продаж».
8. Использовать ConcurrentHashMap для хранения пар «товар – количество продаж».
9. Использовать ConcurrentHashMap и AtomicInteger для счетчика количества продаж каждого товара.
10. Использовать CopyOnWriteArrayList для хранения списка проданных товаров с возможностью одновременного чтения и записи.

## Контрольные вопросы

1. Какие интерфейсы коллекций есть в Java?
2. Какие классы коллекций есть в Java?
3. Что такое итератор?
4. Как работают коллекции на основе интерфейса Map?
5. Как работают коллекции на основе интерфейса List?
6. Как работают коллекции на основе интерфейса Set?
7. Как можно синхронизировать коллекции в Java?
8. Какие методы предоставляет интерфейс Collection?
9. Какие реализации интерфейса List вы знаете?
10. Какие реализации интерфейса Set вы знаете?
11. Что такое Comparable и Comparator?
12. Что такое параметр типа?
13. Как параметризовать метод, класс?
14. Что такое стирание типов?
15. Как можно обойти ограничения стирания типов?
16. Как работают дженерики с массивами?
17. Можно ли создать массив дженериков?
18. Что такое wildcard тип?
19. В чем разница между <extends T> и <super T>?

## 7 Многопоточность

---

Многопоточность в Java — это способность выполнять несколько задач параллельно, используя несколько потоков выполнения. Потоки позволяют эффективнее использовать ресурсы системы, улучшая производительность приложений. В Java существует несколько способов реализации многопоточности, включая использование классов Thread и Runnable, а также современные подходы с использованием библиотеки Executors и лямбда-выражений.

### Основные концепции

Поток (Thread) — основной строительный блок многопоточности в Java. Каждый поток представляет собой отдельный путь выполнения программы.

Синхронизация — механизм, обеспечивающий безопасный доступ к общим ресурсам несколькими потоками. Включает в себя использование ключевых слов synchronized, volatile и примитивов блокировки (Lock).

Жизненный цикл потока — поток проходит через несколько состояний: создание, выполнение, ожидание, блокировка и завершение.

Самый простой способ создания потока — это расширение класса Thread и переопределение метода run() (листинг 7.1).

#### Листинг 7.1. Расширение класса Thread

```
1. public class MyThread extends Thread {  
2.     @Override  
3.     public void run() {  
4.         // Логика выполнения потока  
5.         System.out.println("Выполнение потока: " + getName());  
6.     }  
7.     public static void main(String[] args) {  
8.         MyThread thread1 = new MyThread();  
9.     }  
}
```

```
10.     MyThread thread2 = new MyThread();
11.
12.     thread1.start();
13.     thread2.start();
14. }
15. }
```

Альтернативный подход — это реализация интерфейса Runnable и передача экземпляра Runnable конструктору класса Thread (листинг 7.2).

**Листинг 7.2.** Реализация интерфейса Runnable

```
1. public class MyRunnable implements Runnable {
2.     @Override
3.     public void run() {
4.         // Логика выполнения потока
5.         System.out.println
6.             ("Выполнение потока: " + Thread.currentThread().getName());
7.     }
8.     public static void main(String[] args) {
9.         Thread thread1 = new Thread(new MyRunnable(), "Thread-1");
10.        Thread thread2 = new Thread(new MyRunnable(), "Thread-2");
11.
12.        thread1.start();
13.        thread2.start();
14.    }
15. }
```

Библиотека Executors предоставляет удобные средства для управления потоками, включая создание пулов потоков и планировщиков (листинг 7.3).

**Листинг 7.3.** Использование библиотеки Executors

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3.
4. public class ExecutorExample {
5.     public static void main(String[] args) {
6.         ExecutorService executor = Executors.newFixedThreadPool(2);
7.
8.         executor.execute(() -> {
9.             // Логика выполнения потока
10.            System.out.println("Выполнение потока: "
11.                + Thread.currentThread().getName());
12.        });
13.        executor.execute(() -> {
```

```
14.     // Логика выполнения другого потока
15.     System.out.println("Выполнение другого потока: "
16.                         + Thread.currentThread().getName());
17.   });
18.   executor.shutdown();
19. }
20. }
```

Для предотвращения состояния гонки и других проблем, связанных с параллельным доступом к общим ресурсам, в Java предусмотрены механизмы синхронизации.

Ключевое слово `synchronized` позволяет заблокировать доступ к методу или блоку кода, пока другой поток не завершит выполнение критической секции (листинг 7.4).

**Листинг 7.4.** Использование ключевого слова `synchronized`

```
1. public class SynchronizedExample {
2.     private int counter = 0;
3.
4.     public synchronized void increment() {
5.         counter++;
6.     }
7.
8.     public int getCounter() {
9.         return counter;
10.    }
11.
12.    public static void main(String[] args) throws InterruptedException {
13.        SynchronizedExample example = new SynchronizedExample();
14.
15.        Thread t1 = new Thread(() -> {
16.            for (int i = 0; i < 1000; i++) {
17.                example.increment();
18.            }
19.        });
20.
21.        Thread t2 = new Thread(() -> {
22.            for (int i = 0; i < 1000; i++) {
23.                example.increment();
24.            }
25.        });
26.
27.        t1.start();
28.        t2.start();
29.    }
```

```
30.     t1.join();
31.     t2.join();
32.
33.     System.out.println(example.getCounter()); // Должно
34.                           // вывести 2000
35. }
36. }
```

Ключевое слово volatile обеспечивает видимость изменений переменной всеми потоками, гарантируя, что значение переменной всегда будет актуальным (листинг 7.5).

**Листинг 7.5.** Использование ключевого слова volatile

```
1. public class VolatileExample {
2.     private volatile boolean flag = false;
3.
4.     public void setFlag(boolean value) {
5.         flag = value;
6.     }
7.
8.     public boolean isFlag() {
9.         return flag;
10.    }
11.
12.    public static void main(String[] args) throws InterruptedException {
13.        VolatileExample example = new VolatileExample();
14.
15.        Thread t1 = new Thread(() -> {
16.            while (!example.isFlag()) {
17.                // Ожидание установки флага
18.            }
19.            System.out.println("Флаг установлен!");
20.        });
21.
22.        t1.start();
23.
24.        Thread.sleep(1000); // Ждем 1 секунду
25.
26.        example.setFlag(true);
27.
28.        t1.join();
29.    }
30. }
```

## Задания для выполнения лабораторной работы

### Задание 1.

Реализация многопоточной программы для вычисления суммы элементов массива.

Вариант 1. Создать два потока, которые будут вычислять сумму элементов массива по половинкам, после чего результаты будут складываться в главном потоке.

Вариант 2. Создать пул потоков с помощью класса ExecutorService и разделить массив на равные части, каждую из которых будет обрабатывать отдельный поток. После завершения работы всех потоков результаты будут складываться в главном потоке.

### Задание 2.

Реализация многопоточной программы для поиска наибольшего элемента в матрице.

Вариант 1. Создать несколько потоков, каждый из которых будет обрабатывать свою строку матрицы. После завершения работы всех потоков результаты будут сравниваться в главном потоке для нахождения наибольшего элемента.

Вариант 2. Создать пул потоков с помощью класса ExecutorService и разделить матрицу на равные части, каждую из которых будет обрабатывать отдельный поток. После завершения работы всех потоков результаты будут сравниваться в главном потоке для нахождения наибольшего элемента.

### Задание 3.

У вас есть склад с товарами, которые нужно перенести на другой склад. У каждого товара есть свой вес. На складе работают 3 грузчика. Грузчики могут переносить товары одновременно, но суммарный вес товаров, переносимый ими за одну итерацию, не может превышать 150 кг. Как только грузчики соберут 150 кг товаров, они отправятся на другой склад и начнут разгружать товары.

Напишите программу на Java, используя многопоточность, которая реализует данную ситуацию.

#### Варианты

1. Использование Thread. Создайте классы Товар, Склад, и Грузчик. Каждый грузчик должен быть представлен в виде отдельного потока.

2. Использование Runnable. Создайте интерфейс Грузчик и реализуйте его в классе LoaderRealization. Используйте ExecutorService для управления потоками.
3. Использование Lock и Condition. Используйте блокировки и условия для синхронизации работы грузчиков.
4. Использование Semaphore. Используйте семафоры для ограничения доступа к складу и контроля над весом товаров.
5. Использование CompletableFuture. Используйте CompletableFuture для асинхронного выполнения задачи переноса товаров.
6. Использование ForkJoinPool. Разделите склад на подзадачи и используйте Fork-Join-пул для обработки этих подзадач.
7. Использование ReentrantLock и Condition. Используйте рентабельные блокировки для управления доступом к ресурсам.
8. Использование CountDownLatch. Используйте CountDownLatch для синхронизации начала и завершения переноса товаров.
9. Использование CyclicBarrier. Используйте барьеры для синхронизации грузчиков перед отправлением на другой склад.
10. Использование Executor и CompletionService. Используйте Executor для управления потоками и CompletionService для получения результатов выполнения.

## Контрольные вопросы

1. Как реализуется многопоточность в Java?
2. Что такое поток?
3. Для чего нужно ключевое слово synchronized?
4. Для чего нужно ключевое слово volatile?
5. Зачем нужно синхронизировать потоки?
6. Какие есть способы синхронизации потоков?
7. В чем разница между Thread и Runnable?
8. Какие состояния может иметь поток? Опишите жизненный цикл потока.
9. Что такое daemon-поток? Как его создать?
10. Как принудительно остановить поток?
11. Как работает метод join()? Для чего он используется?
12. Что такое «гонка данных» (race condition)?
13. Что такое deadlock? Как его избежать?
14. Что такое wait(), notify() и notifyAll()? В каком классе они объявлены?
15. Что такое ThreadPool? Какие реализации ExecutorService есть в Java?

## 8 Аннотации. Stream API

---

Аннотации (Annotations) — это метаданные, которые могут быть присоединены к классам, методам, полям и другим элементам программы в Java. Аннотации позволяют добавлять дополнительную информацию к коду и могут использоваться для анализа и автоматизации процесса компиляции.

Аннотации используются для различных целей, таких как:

- пометка кода для анализа сторонними инструментами;
- отображение информации во время выполнения программы (аннотации с RetentionPolicy.RUNTIME);
- управление поведением компилятора.

Пример аннотации:

```
1. public @interface MyAnnotation {  
2.     String value() default "";  
3. }
```

**Stream API** в Java — это функциональный интерфейс для работы с коллекциями данных, представляющий собой мощное средство для фильтрации, преобразования и агрегирования данных. Он был введен в Java 8 и стал важной частью современного подхода к разработке на Java.

Основные компоненты Stream API:

- источник данных — поток может быть создан из различных источников, таких как коллекции, массивы, генераторы и т. д.;
- промежуточные операции применяются к потоку данных и создают новый поток. Они выполняются лениво, то есть не сразу, а только при необходимости;
- терминальные операции завершают обработку потока и производят результат. После терминальной операции поток закрывается и не может быть использован снова.

Поток можно создать из различных источников данных (листинг 8.1).

**Листинг 8.1.** Создание потока

```
1. import java.util.stream.Stream;
2.
3. // Из коллекции
4. List< String > list = Arrays.asList("apple", "banana", "cherry");
5. Stream< String > streamFromList = list.stream();
6.
7. // Из массива
8. String[] array = {"apple", "banana", "cherry"};
9. Stream< String > streamFromArray = Arrays.stream(array);
10.
11. // Из диапазона чисел
12. IntStream range = IntStream.range(1, 10); // [1, 2, ..., 9]
```

Некоторые примеры промежуточных операций:

- filter — фильтрует элементы потока по заданному условию;
- map — применяет функцию к каждому элементу потока и создает новый поток результатов;
- sorted — сортирует элементы потока;
- distinct — удаляет дубликаты из потока.

```
1. Stream< String > filteredStream = streamFromList
2.         .filter(s -> s.startsWith("a"));
3. Stream< String > mappedStream = filteredStream
4.         .map(String::toUpperCase);
5. Stream< String > sortedStream = mappedStream.sorted();
6. Stream< String > distinctStream = sortedStream.distinct();
```

Примеры терминальных операций:

- forEach — выполняет указанное действие для каждого элемента потока;
- collect — собирает элементы потока в коллекцию;
- count — Подсчитывает количество элементов в потоке;
- reduce — применяет операцию свертки к элементам потока.

```
1. long count = distinctStream.count(); // Подсчет элементов
2.
3. List< String > resultList = distinctStream
4.         .collect(Collectors.toList()); // Сбор в список
5.
6. distinctStream.forEach(System.out::println); // Печать
7.         // каждого элемента
8.
```

Stream API тесно интегрирован с лямбда-выражениями, что делает код более компактным и выразительным:

```
1. List< String> fruits = Arrays.asList("apple", "banana", "cherry");
2.
3. fruits.stream()
4.     .filter(s -> s.startsWith("b"))
5.     .map(String::toUpperCase)
6.     .forEach(System.out::println);
```

Пример Stream API:

```
1. List< Person> people = // Инициализация списка людей
2. List< Person> olderThan30 = people.stream()
3.     .filter(person -> person.getAge() > 30)
4.     .collect(Collectors.toList());
```

Пример сортировки:

```
1. List< String> fruits = Arrays.asList("apple", "banana",
2.         "cherry", "date", "elderberry");
3. List< String> sortedFruits = fruits.stream()
4.     .sorted()
5.     .collect(Collectors.toList());
6. System.out.println(sortedFruits); // Output: [apple, banana,
7.         // cherry, date, elderberry]
```

Пакет **java.util.concurrent** в Java содержит множество классов и интерфейсов, предназначенных для работы с многопоточностью и параллельными вычислениями. Он включает в себя разнообразные утилиты для создания и управления потоками, блокировки, атомарные операции, пулы потоков и многое другое.

#### Ключевые компоненты пакета

Интерфейсы и классы для работы с потоками (листинг 8.2):

- ExecutorService — интерфейс для управления выполнением асинхронных задач;
- Executors — утилитарный класс для создания различных видов ExecutorService;
- Future — интерфейс, представляющий результат асинхронной задачи;
- Callable — интерфейс для задания задачи, которая может вернуть результат.

Синхронизированные структуры данных:

- ConcurrentHashMap — поточно-безопасная карта, оптимизированная для высокопроизводительных сценариев;
- CopyOnWriteArrayList — список, предназначенный для ситуаций, где чтение происходит чаще, чем запись;

- BlockingQueue — очередь, блокирующая потоки до появления свободного места или элементов.  
Механизмы синхронизации (листинг 8.3);
  - Semaphore — семафоры для контроля доступа к общему ресурсу;
  - ReentrantLock — блокировки, позволяющие повторный вход одного и того же потока;
  - Condition — условные переменные для координации потоков.  
Атомарные операции (листинг 8.4);
  - AtomicInteger — атомарный целочисленный счетчик;
  - AtomicReference — атомарная ссылка на объект;
  - LongAdder — оптимизированный для параллельных сценариев накопитель целых чисел.
- Планировщики задач:
- ScheduledExecutorService — Планировщик задач, позволяющий выполнять задачи по расписанию.

**Листинг 8.2.** Создание пула потоков

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3.
4. public class ExecutorExample {
5.     public static void main(String[] args) {
6.         ExecutorService executor = Executors.newFixedThreadPool(5);
7.
8.         for (int i = 0; i < 10; i++) {
9.             executor.submit(() -> {
10.                 System.out.println("Задача выполнена: "
11.                     + Thread.currentThread().getName());
12.             });
13.         }
14.
15.         executor.shutdown();
16.     }
17. }
```

**Листинг 8.3.** Использование семафора

```
1. import java.util.concurrent.Semaphore;
2.
3. public class SemaphoreExample {
4.     public static void main(String[] args) {
5.         Semaphore semaphore = new Semaphore(3);
6.
7.         for (int i = 0; i < 10; i++) {
```

```

8.     new Thread(() -> {
9.         try {
10.             semaphore.acquire();
11.             System.out.println("Доступ получен: "
12.                 + Thread.currentThread().getName());
13.             Thread.sleep(1000);
14.         } catch (InterruptedException e) {
15.             e.printStackTrace();
16.         } finally {
17.             semaphore.release();
18.             System.out.println("Доступ освобожден: "
19.                 + Thread.currentThread().getName());
20.         }
21.     }).start();
22. }
23. }
24. }
```

**Листинг 8.4.** Использование атомарного счетчика

```

1. import java.util.concurrent.atomic.AtomicInteger;
2.
3. public class AtomicCounterExample {
4.     public static void main(String[] args) {
5.         AtomicInteger counter = new AtomicInteger(0);
6.
7.         for (int i = 0; i < 10; i++) {
8.             new Thread(() -> {
9.                 int currentValue = counter.incrementAndGet();
10.                System.out.println("Увеличенный счетчик: "
11.                    + currentValue);
12.            }).start();
13.        }
14.    }
15. }
```

**Задания для выполнения лабораторной работы**

Разработать приложение, которое считывает данные из исходного источника (например, файл, база данных или сетевой ресурс), применяет к ним различные операции с использованием Stream API и сохраняет результаты в новый источник данных.

1. Создайте аннотацию `@DataProcessor`, которая будет использоваться для пометки методов обработки данных.

2. Создайте класс DataManager, который будет отвечать за многопоточную обработку данных. Этот класс должен иметь методы:

- registerDataProcessor(Object processor) — регистрирует объект — обработчик данных с аннотацией @DataProcessor;
- loadData(String source) — загружает данные из исходного источника;
- processData() — запускает многопоточную обработку данных, применяя методы с аннотацией @DataProcessor с использованием Stream API;
- saveData(String destination) — сохраняет обработанные данные в новый источник.

3. Создайте несколько классов, представляющих различные обработчики данных, и пометьте их аннотацией @DataProcessor. Например, можно создать классы для фильтрации, трансформации и агрегации данных.

4. Используйте многопоточность из java.util.concurrent для эффективной обработки данных параллельно.

5. Протестируйте ваше приложение, загрузив данные из исходного источника, применив различные обработчики с помощью Stream API, и сохраните результаты в новый источник.

## Контрольные вопросы

1. Что такое аннотации?
2. Для чего нужны аннотации?
3. Как создать собственную аннотацию?
3. Для чего можно использовать Stream API?
4. С помощью каких инструментов в Java можно обрабатывать данные параллельно?
5. В чем разница между Collection и Stream?
6. Как создать Stream из коллекции? Массива? Отдельных элементов?
7. Что такое Optional в Stream API? Как обрабатывать возможные null значения?
8. Какие существуют терминальные операции Stream API?

## 9 Инструмент сборки Maven

Apache Maven — это инструмент автоматизации сборки проектов на основе концепции проекта (Project Object Model, POM), который помогает разработчикам управлять зависимостями, конфигурацией сборки и жизненным циклом проекта. Maven упрощает управление проектами, обеспечивая стандартизацию процесса сборки и управления зависимостями. Использование Maven делает проекты более переносимыми и легко поддерживаемыми.

В удобной среде разработки выбираем «Создать новый проект», в настройках нового проекта стоит выбрать Build System «Maven» (рис. 9.1), по желанию поменять GroupId и ArtifacId.

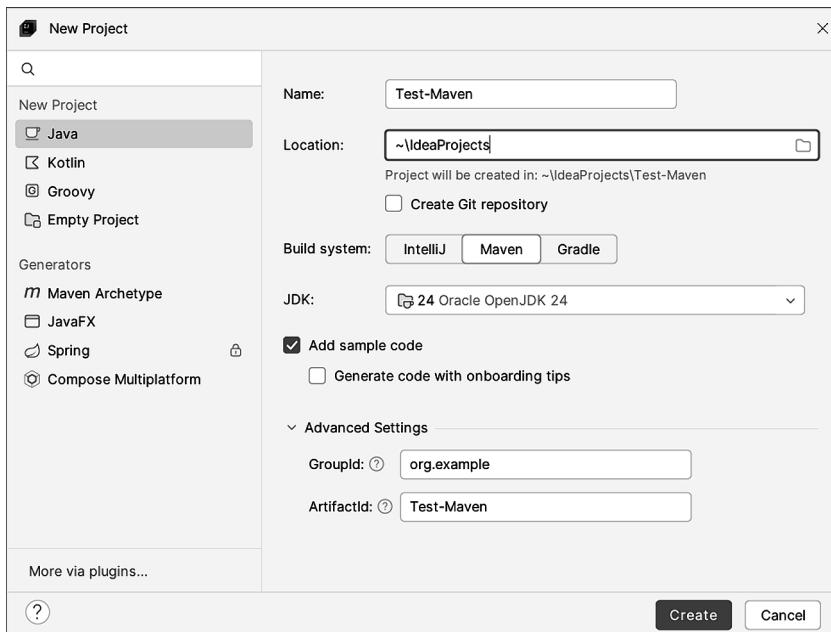


Рис. 9.1. Создание Maven-проекта

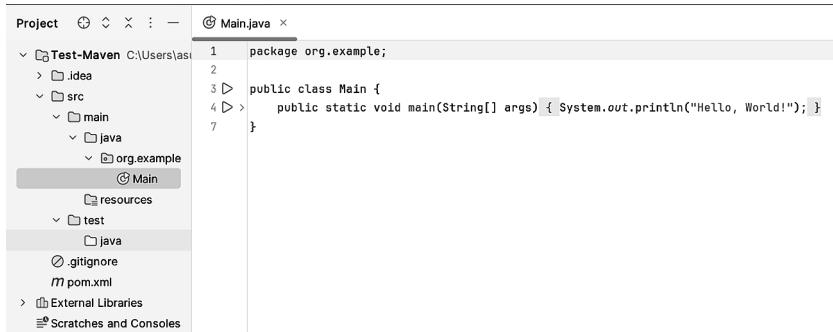


Рис. 9.2. Созданный Maven проект и его структура

Далее нажимаем «Create». После этого видим базовую структуру проекта, подобную изображенной на рис. 9.2.

За сборку проекта при использовании Maven отвечает файл `pom.xml`, который обычно располагается в корне проекта. Что из себя представляет данный файл? Рассмотрим пример такого файла.

Информация для программного проекта, поддерживаемого Maven, содержится в XML-файле с именем `pom.xml`. При исполнении Maven проверяет прежде всего, содержит ли этот файл все необходимые данные и все ли данные синтаксически правильно записаны.

### Пример файла `pom.xml`

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3.           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5.                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
6.   <modelVersion>4.0.0</modelVersion>
7.   <properties>
8.     <project.build.sourceEncoding>UTF-8
9.     </project.build.sourceEncoding>
10.    </properties>
11.    <groupId>com.examclouds</groupId>
12.    <artifactId>courses</artifactId>
13.    <version>1.0-SNAPSHOT</version>
14.
15.    <dependencies>
16.      <dependency>
17.        <groupId>mysql</groupId>
  
```

```
18.      <artifactId> mysql-connector-java</artifactId>
19.      <version>5.1.39</version>
20.    </dependency>
21.
22.    <dependency>
23.      <groupId>com.h2database</groupId>
24.      <artifactId>h2</artifactId>
25.      <version>1.4.196</version>
26.    </dependency>
27.  </dependencies>
28.
29.  <build>
30.    <sourceDirectory>src</sourceDirectory>
31.    <resources>
32.      <resource>
33.        <directory>resources</directory>
34.      </resource>
35.    </resources>
36.    <plugins>
37.      <plugin>
38.        <groupId>org.apache.maven.plugins</groupId>
39.        <artifactId>maven-compiler-plugin</artifactId>
40.        <version>2.0.2</version>
41.        <configuration>
42.          <source>1.8</source>
43.          <target>1.8</target>
44.          <encoding>UTF-8</encoding>
45.        </configuration>
46.      </plugin>
47.      <plugin>
48.        <groupId>org.apache.maven.plugins</groupId>
49.        <artifactId>maven-checkstyle-plugin</artifactId>
50.        <version>2.17</version>
51.        <configuration>
52.          <suppressionsLocation>suppressions.xml
53.          </suppressionsLocation>
54.        </configuration>
55.      </plugin>
56.    </plugins>
57.  </build>
58.  <reporting>
59.    <plugins>
60.      <plugin>
61.        <groupId>org.apache.maven.plugins</groupId>
```

```
62.      <artifactId> maven-compiler-plugin</artifactId>
63.      <version> 2.0.2</version>
64.      <configuration>
65.          <source> 1.8</source>
66.          <target> 1.8</target>
67.          <encoding> UTF-8</encoding>
68.      </configuration>
69.      </plugin>
70.      <plugin>
71.          <groupId> org.apache.maven.plugins</groupId>
72.          <artifactId> maven-checkstyle-plugin</artifactId>
73.          <version> 2.17</version>
74.          <configuration>
75.              <suppressionsLocation> suppressions.xml
76.                  </suppressionsLocation>
77.          </configuration>
78.          <reportSets>
79.              <reportSet>
80.                  <reports>
81.                      <report> checkstyle</report>
82.                  </reports>
83.              </reportSet>
84.          </reportSets>
85.      </plugin>
86.      <plugin>
87.          <groupId> org.apache.maven.plugins</groupId>
88.          <artifactId> maven-pmd-plugin</artifactId>
89.          <version> 3.8</version>
90.      </plugin>
91.      <plugin>
92.          <groupId> org.codehaus.mojo</groupId>
93.          <artifactId> findbugs-maven-plugin</artifactId>
94.          <version> 3.0.4</version>
95.          <configuration>
96.              <xmlOutput> true</xmlOutput>
97.          </configuration>
98.      </plugin>
99.  </plugins>
100. </reporting>
101. </project>
```

### Корневой элемент

В корневом элементе `<project>` прописываются схема, облегчающая редактирование и проверку, и версия РОМ.

### Пример корневого элемента

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3.           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5.                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
6. ...
7. </project>
```

### Заголовок

Внутри тега `project` содержится основная и обязательная информация о проекте.

### Пример заголовка

```
1. <groupId>com.examclouds</groupId>
2. <artifactId>courses</artifactId>
3. <version>1.0-SNAPSHOT</version>
```

### Конфликт имён

В *Maven* разные разработчики могут создавать проекты с одинаковыми названиями. Если не учитывать организацию, можно случайно импортировать не тот проект. Использование доменного имени организации (например, `com.example.project`) позволяет создать уникальный идентификатор, так как доменные имена уникальны в интернете. Таким образом, даже если два проекта называются одинаково, их полные идентификаторы будут разными благодаря уникальным `groupId` и названию проекта `artifactId`.

Внутри тега `version` хранится версия проекта.

Тройкой `groupId`, `artifactId`, `version` (**GAV**) можно однозначно идентифицировать jar-файл приложения или библиотеки. Если состояние кода для проекта не зафиксировано, то в конце к имени версии добавляется метка `"-SNAPSHOT"`, которая обозначает, что версия в разработке и результирующий jar-файл может меняться.

### Тег `packaging`

Тег `<packaging>` определяет тип файла, который будет создан в результате сборки. Возможные варианты: pom, jar, war, ear.

Тег является необязательным. Если его нет, используется значение по умолчанию jar.

### Описание проекта

При желании добавляется информация, которая не используется Maven, но нужна для программиста, чтобы понять, о чём этот проект.

### Пример описания проекта

```
<name> powermock-core </name> название проекта для человека
```

### Описание проекта

```
<description> PowerMock core functionality. </description>
```

```
<url> http://www.powermock.org </url> сайт проекта
```

### Зависимости

**Зависимости** — важная часть pom.xml, где указываются все библиотеки и компоненты, необходимые для работы проекта. Каждая библиотека идентифицируется так же, как и сам проект, — тройкой groupId, artifactId, version (GAV). Объявление зависимостей заключено в теге <dependencies> ... </dependencies> .

Кроме GAV при описании зависимости может присутствовать тег <scope>. Он задаёт, для чего библиотека используется. В данном примере говорится, что библиотека с GAV junit:junit:4.4 нужна только для выполнения тестов.

### Пример зависимости

```
1.  <dependencies>
2.    <dependency>
3.      <groupId> mysql </groupId>
4.      <artifactId> mysql-connector-java </artifactId>
5.      <version> 5.1.39 </version>
6.    </dependency>
7.
8.    <dependency>
9.      <groupId> com.h2database </groupId>
10.     <artifactId> h2 </artifactId>
11.     <version> 1.4.196 </version>
12.   </dependency>
13. </dependencies>
```

### Тег <build>

Тег <build> не обязательный, так как существуют значения по умолчанию. Этот раздел описывает процесс сборки, включая:

- расположение исходных файлов;
- расположение ресурсов;
- используемые плагины.

### Пример тега <build>

```
1. < build>
2.   < sourceDirectory> src/main/java</sourceDirectory>
3.   < resources>
4.     < resource>
5.       < directory> src/main/resources</directory>
6.     </resource>
7.   </resources>
8.   < plugins>
9.     < plugin>
10.       < groupId> org.apache.maven.plugins</groupId>
11.       < artifactId> maven-compiler-plugin</artifactId>
12.       < version> 3.8.1</version>
13.       < configuration>
14.         < source> 11</source>
15.         < target> 11</target>
16.       </configuration>
17.     </plugin>
18.   </plugins>
19. </build>
```

<sourceDirectory> определяет, откуда Maven будет брать файлы исходного кода. По умолчанию это src/main/java, но вы можете определить, где это вам удобно. Директория может быть только одна (без использования специальных плагинов).

<resources> и вложенные в неё теги <resource> определяют одну или несколько директорий, где хранятся файлы ресурсов. Ресурсы, в отличие от файлов исходного кода, при сборке просто копируются. Директория по умолчанию src/main/resources.

<outputDirectory> определяет, в какую директорию компилятор будет сохранять результаты компиляции — файлы с расширением \*.class. Значение по умолчанию — target/classes.

<finalName> — имя результирующего jar (war, ear,...) файла с соответствующим типу расширением, который создаётся на фазе package. Значение по умолчанию — artifactId-version.

Плагины Maven позволяют задать дополнительные действия, которые будут выполняться при сборке. Например, в приведённом ниже примере добавлен плагин, который автоматически делает проверку кода на наличие «плохого» кода и потенциальных ошибок.

Это лишь пример некоторого готового файла pom.xml. Вернемся к нашему проекту и добавим в него плагин для сборки, который поможет нам указать Main Class сразу из кода, чтобы не

прописывать это при запуске. Итоговый файл pom.xml представлен на листинге 9.1.

**Листинг 9.1.** Файл pom.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3.           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5.     <modelVersion>4.0.0</modelVersion>
6.
7.     <groupId>org.example</groupId>
8.     <artifactId>Test-Maven</artifactId>
9.     <version>1.0-SNAPSHOT</version>
10.
11.    <properties>
12.        <maven.compiler.source>17</maven.compiler.source>
13.        <maven.compiler.target>17</maven.compiler.target>
14.        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.    </properties>
16.
17.    <build>
18.        <plugins>
19.            <plugin>
20.                <groupId>org.codehaus.mojo</groupId>
21.                <artifactId>exec-maven-plugin</artifactId>
22.                <version>3.1.0</version>
23.                <configuration>
24.                    <mainClass>org.example.Main</mainClass>
25.                    <skip>false</skip>
26.                </configuration>
27.            </plugin>
28.        </plugins>
29.    </build>
30. </project>
```

```
@Ubuntu:~/IdeaProjects$ ls
Test-Maven
@Ubuntu:~/IdeaProjects$ cd Test-Maven
@Ubuntu:~/IdeaProjects/Test-Maven$ █
```

Рис. 9.3. Перемещение в папку проекта

```

gorshkova@ubuntu:~/IdeaProjects/Test-Maven
```

---

```

kb at 2.0 MB/s)
[INFO] Installing /home/
/IdeaProjects/Test-Maven/target/Test-Maven-1.0-SNAPSHOT.jar to /home/gorshkova/.m2/repository/o
rg/example/Test-Maven/1.0-SNAPSHOT/Test-Maven-1.0-SNAPSHOT.jar
[INFO] Installing /home/
/IdeaProjects/Test-Maven/pom.xml to /home/gorshkova/.m2/repository/org/example/Test-Maven/1.0-S
NAPSHOT/Test-Maven-1.0-SNAPSHOT.pom
```

---

```

[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ Test-Maven ---
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.4.2/plexus-utils-3.4.2.jar
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-component-annotations/2.1.1/plexus-com
ponent-annotations-2.1.1.jar
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-exec/1.3/commons-exec-1.3.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-component-annotations/2.1.1/plexus-comp
onent-annotations-2.1.1.jar (4.1 kB at 36 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.4.2/plexus-utils-3.4.2.jar (267
kB at 2.3 MB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-exec/1.3/commons-exec-1.3.jar (54 kB at
439 kB/s)

Hello and welcome!i = 1
i = 2
i = 3
i = 4
i = 5
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 5.849 s
[INFO] Finished at: 2025-04-01T13:11:51Z
[INFO] -----
```

---

```

@Ubuntu:~/IdeaProjects/Test-Maven$
```

Рис. 9.4. Консольный запуск сборки

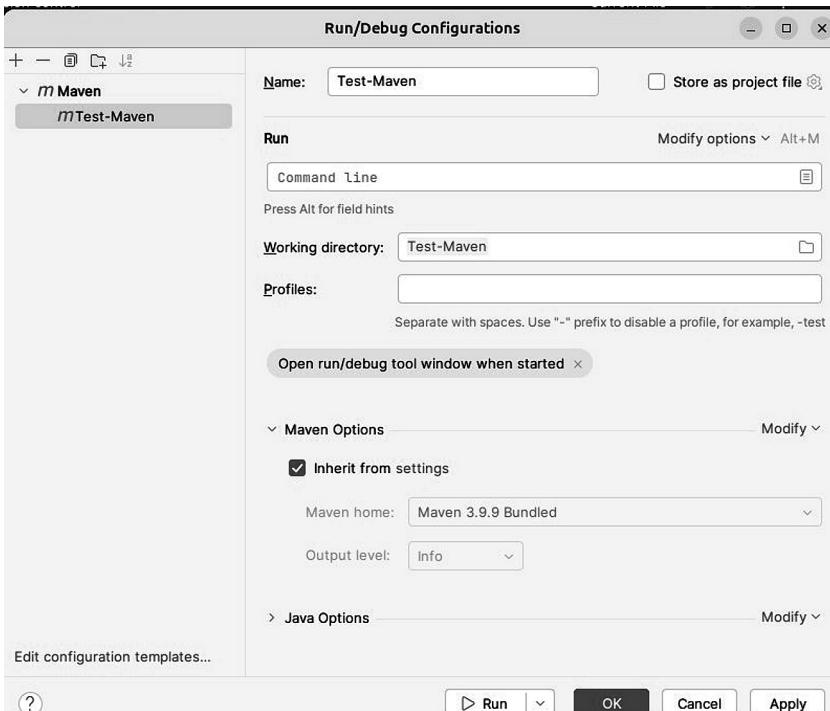


Рис. 9.5. Окно создания новой конфигурации сборки

Далее нам нужно запустить проекта. Это можно сделать как из среды разработки, так и из системной консоли. Для запуска сборки проекта на базе Maven используется команда mvn. В системах на базе Linux mvn можно установить по команде

```
sudo apt install maven
```

где apt — пакетный менеджер в Linux. Затем нам нужно из консоли переместиться в папку проекта (рис. 9.3).

Чтоб создать команду запуска нам нужно определить фазы для запуска. Первой фазой будет clean, чтобы очистить директорию сборки, если проект уже собирался. Обычно clean используется для сборки «начистую». Вторая фаза будет install, в этой фазе компилируется исходный код, выполняются тесты и упаковываются результаты в jar- или war-файл, после чего упакованный файл копируется в локальный репозиторий Maven, что позволяет использовать его как зависимость в других проектах на этой же машине. Третья фаза exec:java, которая требуется для запуска приложения.

Итоговая команда:

```
mvn clean install exec:java
```

Запуск представлен на рис. 9.4.

Теперь рассмотрим второй способ сборки. В среде разработки выбираем «Редактировать конфигурации сборки», в разных средах эта кнопка может отличаться. Затем нажимаем «Добавить новую конфигурацию сборки», «Maven». После чего видим окошко, похожее на изображенное на рис. 9.5.

В этом окне надо в Command line занести наши фазы, а именно clean install exec:java (рис. 9.6). После чего нажимаем «Применить» и «Ок».

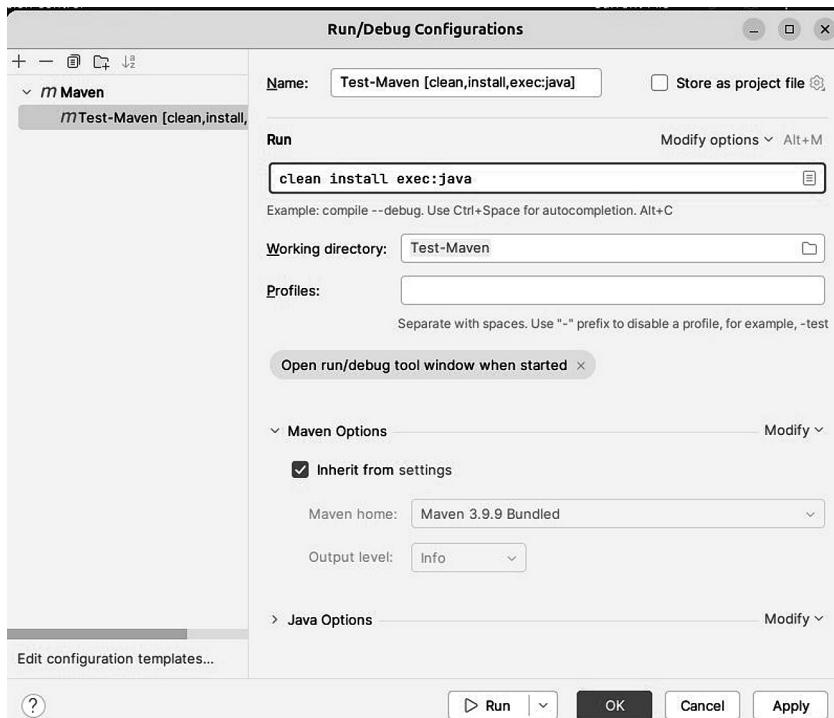


Рис. 9.6. Готовая конфигурация Maven

После этого нажимаем на кнопку запуска конфигурации сборки в среде разработки. Результат запуска приведен на рис. 9.7.

```
Hello and welcome! i = 1
i = 2
i = 3
i = 4
i = 5
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  1.621 s
[INFO] Finished at: 2025-04-01T13:19:57Z
[INFO] -----
```

Process finished with exit code 0

Рис. 9.7. Запуск конфигурации сборки

## Задания для выполнения лабораторной работы

1. Базовый класс: Животные. Дочерние классы: Кошка, Попугай, Рыбка.
2. Базовый класс: Сотрудник. Дочерние классы: Администратор, Программист, Менеджер.
3. Базовый класс: Человек. Дочерние классы: Студент, Преподаватель, Ассистент преподавателя.
4. Базовый класс: Транспортное средство. Дочерние классы: Легковой автомобиль, Грузовой автомобиль, Мотоцикл.
5. Базовый класс: Велосипед. Дочерние классы: Горный велосипед, Детский велосипед, ВМХ.
6. Базовый класс: Геометрическая фигура. Дочерние классы: Шар, Параллелепипед, Цилиндр.
7. Базовый класс: Книга. Дочерние классы: Аудиокнига, Фильм, Мюзикл.
8. Базовый класс: Мебель. Дочерние классы: Стол, Стул, Кровать.
9. Базовый класс: Монстр. Дочерние классы: Гоблин, Русалка, Дракон.
10. Базовый класс: Гаджеты. Дочерние классы: Часы, Смартфон, Ноутбук.
11. Базовый класс: Бытовая техника. Дочерние классы: Холодильник, Посудомоечная машина, Пылесос.
12. Базовый класс: Приложение. Дочерние классы: Социальная сеть, Игра, Погода.

13. Базовый класс: Оружие. Дочерние классы: Меч, Лук, Волшебная палочка.
14. Базовый класс: Заведение. Дочерние классы: Кафе, Магазин, Библиотека.
15. Базовый класс: Компьютерная периферия. Дочерние классы: Клавиатура, Наушники, Графический планшет.

## Контрольные вопросы

1. Что такое Apache Maven и для чего он используется?
2. Как установить Maven на различные операционные системы?
3. Какова структура проекта Maven?
4. Что такое POM-файл и какова его роль в проекте Maven?
5. Какова структура файла POM?
6. Что такое зависимости (dependencies) в Maven и как они определяются?
7. Что такое репозиторий Maven и какие виды репозиториев существуют?
8. Как добавить зависимость в проект Maven?
9. Что такое плагины в Maven и как они используются?
10. Как создать новый проект Maven с помощью команды Maven?
11. Что такое цели (goals) и фазы (phases) в Maven и в чем их отличие?
12. Как выполнить команду сборки проекта в Maven?
13. Что такое жизненный цикл сборки (build lifecycle) в Maven?
14. Как настроить профили (profiles) в Maven для разных сред (например, разработка и продакшн)?
15. Как управлять версиями зависимостей в Maven?
16. Что такое "SNAPSHOT" версии в Maven и как они используются?
17. Как использовать Maven для создания отчета о качестве кода?
18. Какие команды Maven используются для очистки проекта, сборки, тестирования и установки?
19. Как интегрировать Maven с системой контроля версий, такой как Git?
20. Как добавить и настроить сторонний репозиторий в проекте Maven?

# 10 Инструмент сборки Gradle

Gradle — это современный инструмент сборки, написанный на языке Groovy и предназначенный для автоматизации задач сборки программного обеспечения. Он сочетает в себе лучшие черты других инструментов сборки, таких как Ant и Maven, и предлагает гибкость, производительность и простоту использования. Gradle предоставляет мощный и гибкий способ автоматизации процессов сборки, тестирования и развертывания приложений.

В удобной среде разработки выбираем «Создать новый проект», в настройках нового проекта стоит выбрать Build System «Gradle» (рис. 10.1), выбирать Gradle DSL «Kotlin», по желанию поменять GroupId и ArtefactId.

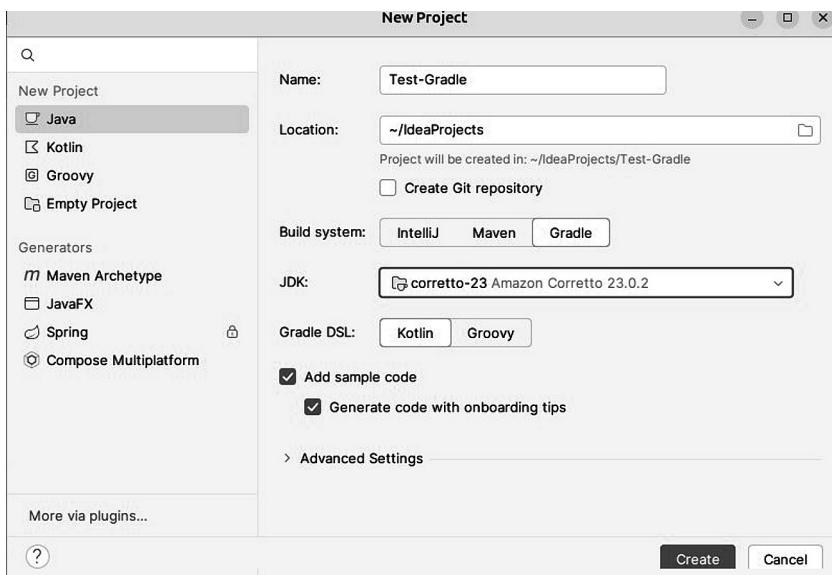


Рис. 10.1. Создание Gradle проекта

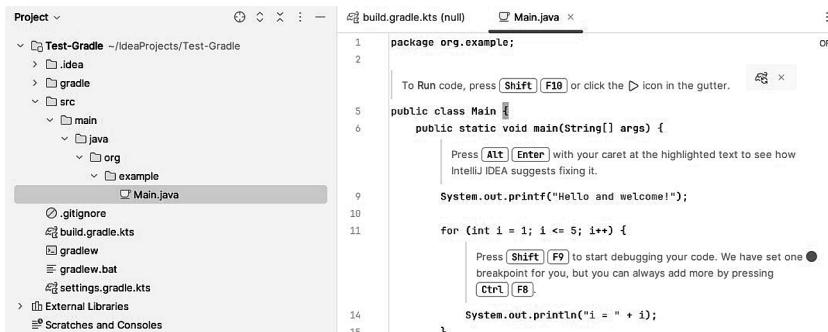


Рис. 10.2. Созданный Gradle проект и его структура

Далее нажимаем «Create». После этого видим базовую структуру проекта, подобную изображенной на рис. 10.2.

За сборку проекта при использовании Gradle отвечают файлы `build.gradle.kts` и `settings.gradle.kts`, которые обычно располагаются в корне проекта. Что из себя представляют данные файлы? Рассмотрим примеры таких файлов.

Файл `build.gradle.kts` — это скрипт на языке Kotlin, используемый для настройки и управления сборкой проекта с помощью системы сборки Gradle. Рассмотрим базовую структуру этого файла.

Пример базовой структуры `build.gradle.kts`

```

1. plugins {
2.   // Применение плагинов
3.   kotlin("jvm") version "1.5.31"
4.   id("application")
5. }
6.
7. group = "com.example"
8. version = "1.0-SNAPSHOT"
9.
10. repositories {
11.   // Репозитории для зависимостей
12.   mavenCentral()
13. }
14.
15. dependencies {
16.   // Зависимости
17.   implementation(kotlin("stdlib"))
18.   testImplementation(kotlin("test"))
19. }
```

```
20.  
21. application {  
22. // Конфигурация для плагина application  
23. mainClass.set("com.example.MainKt")  
24. }  
25.  
26. tasks.withType<Test> {  
27. // Настройка задач тестирования  
28. useJUnitPlatform()  
29. }
```

Описание основных разделов:

- `plugins` — здесь подключаются необходимые плагины. Например, `kotlin("jvm")` для проектов на Kotlin, `id("application")` для приложений;
- `group` и `version` — указываются группа и версия проекта;
- `repositories` — описываются репозитории, откуда будут загружаться зависимости. В данном примере используется центральный репозиторий Maven (`mavenCentral()`);
- `dependencies` — перечисляются зависимости проекта. В примере указываются стандартная библиотека Kotlin (`implementation(kotlin("stdlib"))`) и библиотека для тестирования (`testImplementation(kotlin("test"))`);
- `application` — конфигурация для плагина `application`, где указывается основной класс (`mainClass.set("com.example.MainKt")`), с которого начинается выполнение программы;
- `tasks.withType<Test>` — настройка задач тестирования. В данном примере используется JUnit Platform.

Дополнительные настройки:

- `sourceSets` — можно настроить пути к исходным кодам и ресурсам:

```
1. sourceSets {  
2. main {  
3. kotlin.srcDirs("src/main/kotlin")  
4. resources.srcDirs("src/main/resources")  
5. }  
6. test {  
7. kotlin.srcDirs("src/test/kotlin")  
8. resources.srcDirs("src/test/resources")  
9. }  
10. }
```

- `compileKotlin` и `compileTestKotlin` — настройки компилятора Kotlin:

```
1. tasks.compileKotlin {  
2.     kotlinOptions.jvmTarget = "1.8"  
3. }  
4.  
5. tasks.compileTestKotlin {  
6.     kotlinOptions.jvmTarget = "1.8"  
7. }
```

Это базовая структура и настройки, которые помогут начать работу с 'build.gradle.kts'. В зависимости от требований вашего проекта файл может быть дополнен и расширен различными плагинами и конфигурациями.

Файл settings.gradle.kts используется для настройки и управления мультипроектами в Gradle. В этом файле определяются корневые параметры проекта и указываются модули, которые должны быть включены в сборку. Рассмотрим базовую структуру settings.gradle.kts.

Пример базовой структуры settings.gradle.kts:

```
1. pluginManagement {  
2.     repositories {  
3.         // Репозитории для плагинов  
4.         gradlePluginPortal()  
5.         mavenCentral()  
6.         jcenter()  
7.     }  
8. }  
9.  
10. rootProject.name = "MyProject"  
11.  
12. include(":module1", ":module2")  
13.  
14. dependencyResolutionManagement {  
15.     repositories {  
16.         // Репозитории для зависимостей  
17.         mavenCentral()  
18.         jcenter()  
19.     }  
20. }
```

Описание основных разделов:

- pluginManagement — этот блок используется для настройки репозиториев, откуда будут загружаться плагины. Здесь можно указать различные репозитории, такие как gradlePluginPortal(), mavenCentral() и jcenter();

- `rootProject.name` — устанавливает имя корневого проекта. Это имя будет использоваться в логах сборки и в некоторых других местах;
- `include` — этот метод используется для включения модулей в сборку. В данном примере включены два модуля: `:module1` и `:module2`. Каждый модуль представляет собой подпроект, который будет собираться вместе с основным проектом;
- `dependencyResolutionManagement` — этот блок используется для настройки репозиториев, откуда будут загружаться зависимости. Это может быть полезно, если нужно указать общие репозитории для всех подпроектов.

Примеры дополнительных настроек:

- `includeBuild` — этот метод используется для включения сборок из других проектов:  
`includeBuild("../sharedBuild")`
- `projectDir` — можно указать конкретный каталог для подпроекта:  
`include(":module3")  
project(":module3").projectDir = file("../some/other/dir")`

- `enableFeaturePreview` — включение экспериментальных функций Gradle:  
`enableFeaturePreview("VERSION_CATALOGS")`

Файл `settings.gradle.kts` обеспечивает гибкость и контроль над конфигурацией проекта и его модулей, позволяя задавать различные параметры на уровне корневого проекта.

Это лишь пример некоторого готового файла `build.gradle.kts`. Вернемся к нашему проекту и добавим в него плагин для сборки `application`, который поможет нам указать Main Class сразу из кода, чтоб не прописывать это при запуске. Укажем в конфигурации `application` Main Class. Итоговый файл `build.gradle.kts` представлен на листинге 10.1.

#### Листинг 10.1. Файл `build.gradle.kts`

```
1. plugins {  
2.     id("java")  
3.     application  
4. }  
5.  
6. group = "org.example"  
7. version = "1.0-SNAPSHOT"
```

```
8.  
9. application {  
10.     mainClass = "org.example.Main"  
11. }  
12.  
13. repositories {  
14.     mavenCentral()  
15. }  
16.  
17. dependencies {  
18.     testImplementation(platform("org.junit:junit-bom:5.9.1"))  
19.     testImplementation("org.junit.jupiter:junit-jupiter")  
20. }  
21.  
22. tasks.test {  
23.     useJUnitPlatform()  
24. }
```

Далее нам нужно проделать запуск проекта. Это можно сделать как из среды разработки, так и из системной консоли. Для запуска сборки проекта на базе Gradle используется команда gradle. В системах на базе Linux gradle можно установить командой

```
sudo apt install gradle
```

где apt — пакетный менеджер в Linux. Затем нам нужно из консоли переместиться в папку проекта (рис. 10.3).

```
@Ubuntu:~$ cd IdeaProjects  
@Ubuntu:~/IdeaProjects$ cd Test-Gradle  
@Ubuntu:~/IdeaProjects/Test-Gradle$ █
```

Рис. 10.3. Перемещение в папку проекта

Чтоб создать команду запуска, нужно определить задания (tasks) для запуска. Первой фазой будет clean, чтобы очистить директорию сборки, если проект уже собирался. Обычно clean используется для сборки «начистую». Второй фазой будет build, в этой фазе компилируется исходный код, выполняются тесты и упаковываются результаты в jar- или war-файл, после чего упакованный файл копируется в локальный репозиторий Gradle, что позволяет использовать его как зависимость в других проектах на этой же машине. Третья фаза run, которая требуется для запуска приложения.

Итоговая команда:

gradle clean build run

Запуск представлен на рис. 10.4.

```
@Ubuntu:~/IdeaProjects/Test-Gradle$ gradle clean build run
openjdk version "21.0.6" 2025-01-21
OpenJDK Runtime Environment (build 21.0.6+7-Ubuntu-124.04.1)
OpenJDK 64-Bit Server VM (build 21.0.6+7-Ubuntu-124.04.1, mixed mode, sharing)

> Task :run
Hello world.

BUILD SUCCESSFUL in 1s
9 actionable tasks: 9 executed
@Ubuntu:~/IdeaProjects/Test-Gradle$ █
```

Рис. 10.4. Консольный запуск сборки

Теперь рассмотрим второй способ сборки. В среде разработки выбираем «Редактировать конфигурации сборки», в разных средах эта кнопка может отличаться. Затем нажимаем «Добавить новую конфигурацию сборки», «Gradle». После чего видим окошко, похожее на изображенное на рис. 10.5.

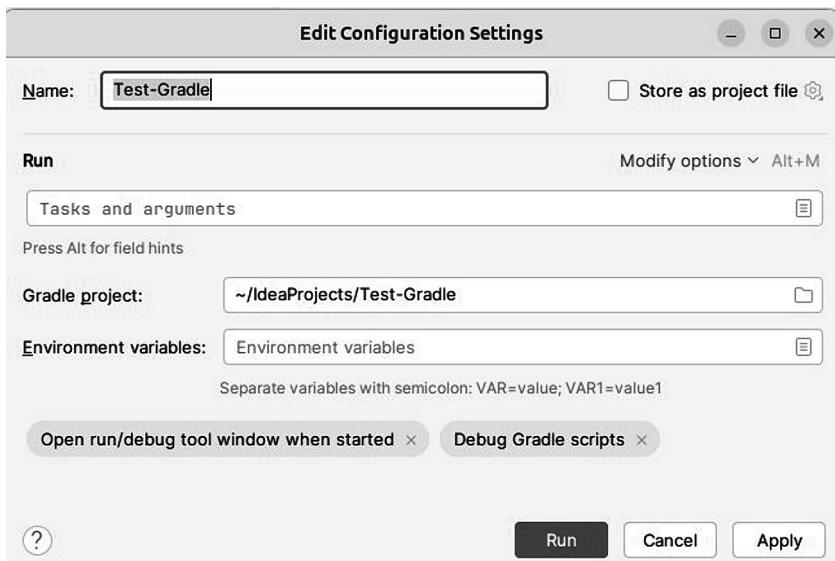


Рис. 10.5. Окно создания новой конфигурации сборки

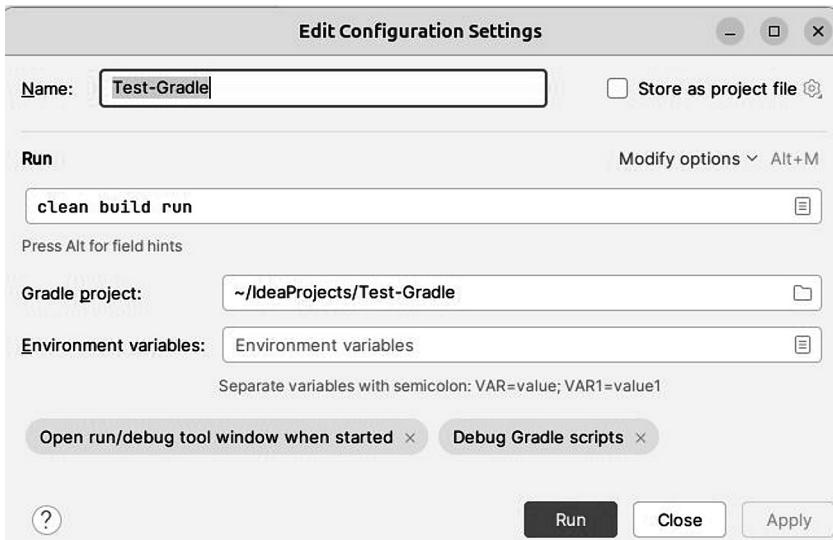


Рис. 10.6. Готовая конфигурация Gradle

В этом окне надо в Command line занести наши tasks, а именно clean build run (рис. 10.6). После чего нажимаем «Применить» и «Ок».

После этого нажимаем на кнопку запуска конфигурации сборки в среде разработки. Результат запуска приведен на рис. 10.7.

```

✓ ✓ Test-Gradle [clean build run]: sui 2 sec, 4 ms
✓ :clean UP-TO-DATE           32 ms
✓ :compileJava                 637 ms
∅ :processResources
✓ :classes
✓ :jar                         65 ms
✓ :startScripts                120 ms
✓ :distTar                      34 ms
✓ :distZip                     11 ms
✓ :assemble
∅ :compileTestJava NO-SOURCE
∅ :processTestResources NO-SOURCE
Task :testClasses UP-TO-DATE
> Task :test NO-SOURCE
> Task :check UP-TO-DATE
> Task :build

> Task :run
Hello and welcome! i = 1
i = 2
i = 3
i = 4
i = 5

BUILD SUCCESSFUL in 1s
7 actionable tasks: 6 executed, 1 up-to-date
2:35:21 PM: Execution finished 'clean build run'.

```

Рис. 10.7. Запуск конфигурации сборки

## Задания для выполнения лабораторной работы

1. Базовый класс: Животные. Дочерние классы: Кошка, Попугай, Рыбка.
2. Базовый класс: Сотрудник. Дочерние классы: Администратор, Программист, Менеджер.
3. Базовый класс: Человек. Дочерние классы: Студент, Преподаватель, Ассистент преподавателя.
4. Базовый класс: Транспортное средство. Дочерние классы: Легковой автомобиль, Грузовой автомобиль, Мотоцикл.
5. Базовый класс: Велосипед. Дочерние классы: Горный велосипед, Детский велосипед, ВМХ.
6. Базовый класс: Геометрическая фигура. Дочерние классы: Шар, Параллелепипед, Цилиндр.
7. Базовый класс: Книга. Дочерние классы: Аудиокнига, Фильм, Мюзикл.
8. Базовый класс: Мебель. Дочерние классы: Стол, Стул, Кровать.
9. Базовый класс: Монстр. Дочерние классы: Гоблин, Русалка, Дракон.
10. Базовый класс: Гаджеты. Дочерние классы: Часы, Смартфон, Ноутбук.
11. Базовый класс: Бытовая техника. Дочерние классы: Холодильник, Посудомоечная машина, Пылесос.
12. Базовый класс: Приложение. Дочерние классы: Социальная сеть, Игра, Погода.
13. Базовый класс: Оружие. Дочерние классы: Меч, Лук, Волшебная палочка.
14. Базовый класс: Заведение. Дочерние классы: Кафе, Магазин, Библиотека.
15. Базовый класс: Компьютерная периферия. Дочерние классы: Клавиатура, Наушники, Графический планшет.

## Контрольные вопросы

1. Что такое Gradle и для чего он используется?
2. В чем основные преимущества использования Gradle по сравнению с Maven и Ant?
3. Какой файл является основным конфигурационным файлом в проекте Gradle?

4. Что такое build.gradle и какие ключевые элементы он содержит?
5. Как запустить сборку проекта с помощью Gradle из командной строки?
6. Что такое Gradle Wrapper и как его использовать?
7. Как определить зависимости в Gradle проекте?
8. Что такое репозиторий в контексте Gradle и как его указать в build.gradle?
9. Как создать новый Gradle проект с нуля?
10. Как добавить и настроить плагины в проекте Gradle?
11. В чем разница между implementation и compile в Gradle?
12. Как в Gradle организована структура проекта?
13. Как настроить Gradle для работы с несколькими модулями (multi-module projects)?
14. Что такое задачи (tasks) в Gradle и как их создать?
15. Как посмотреть список всех доступных задач в проекте Gradle?
16. Как использовать переменные и параметры в build.gradle файле?
17. Что такое Gradle Daemon и какие преимущества он предоставляет?
18. Как обновить версию Gradle в проекте?
19. Как интегрировать тестирование (JUnit, TestNG) в проекте Gradle?
20. Как оптимизировать и ускорить сборку проекта с помощью Gradle?

## 11 Введение в работу с фреймворком Spring

---

Spring Framework — это фреймворк для разработки корпоративных Java-приложений. Он был создан Родом Джонсоном и впервые выпущен в 2003 году. Spring предоставляет инфраструктуру для создания приложений, облегчая разработку и интеграцию различных компонентов.

Основные модули и компоненты:

- Core Container — ядро Spring включает в себя модули, обеспечивающие инверсию контроля (Inversion of Control, IoC) и внедрение зависимостей (Dependency Injection, DI). Эти механизмы позволяют отделить бизнес-логику от инфраструктуры приложения;
- AOP — аспектно-ориентированное программирование (Aspect-Oriented Programming) позволяет добавлять поведение к существующему коду без изменения самого кода. Это полезно для реализации сквозных функций, таких как журналирование, безопасность и транзакции;
- Data Access — обеспечивает работу с базами данных и инструментами объектно-реляционного отображения ORM (Object-Relational Mapping), такими как JDBC, Hibernate, JPA и другие. Он облегчает взаимодействие с данными, абстрагируя низкоуровневые детали доступа к базе данных;
- Web MVC — реализует паттерн Model-View-Controller\* для веб-приложений. Он позволяет разрабатывать веб-интерфейсы, обрабатывающие HTTP-запросы и возвращающие HTML, JSON или XML;

---

\* Паттерн MVC разделяет приложение на три основные компонента: модель — бизнес-логика и данные приложения, представление — отображение данных и пользовательский интерфейс; контроллер — обработка пользовательского ввода и взаимодействие между моделью и представлением.

- Security — предоставляет инструменты для защиты веб-приложений и сервисов. Он включает поддержку аутентификации, авторизации, шифрования и других механизмов безопасности;
- Test — упрощает создание и выполнение модульных и интеграционных тестов. Он интегрируется с популярными тестовыми фреймворками, такими как JUnit и Mockito.

Создаём новый проект в любой удобной среде разработки. После этого важно включить основные зависимости для корректной работы Spring. Итоговый файл для Maven представлен на листинге 11.1, для Gradle — на листинге 11.2.

Листинг 11.1. Файл pom.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3.           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5.                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
6.   <modelVersion>4.0.0</modelVersion>
7.
8.   <groupId>org.example</groupId>
9.   <artifactId>Spring-Test</artifactId>
10.  <version>1.0-SNAPSHOT</version>
11.
12.  <properties>
13.    <maven.compiler.source>17</maven.compiler.source>
14.    <maven.compiler.target>17</maven.compiler.target>
15.    <project.build.sourceEncoding>UTF-8
16.          </project.build.sourceEncoding>
17.  </properties>
18.
19.  <dependencies>
20.    <dependency>
21.      <groupId>junit</groupId>
22.      <artifactId>junit</artifactId>
23.      <version>3.8.1</version>
24.      <scope>test</scope>
25.    </dependency>
26.
27.    <dependency>
28.      <groupId>org.springframework</groupId>
29.      <artifactId>spring-core</artifactId>
30.      <version>6.1.6</version>
31.    </dependency>
```

```
32.
33.    <dependency>
34.        <groupId>org.springframework</groupId>
35.        <artifactId>spring-beans</artifactId>
36.        <version>6.1.6</version>
37.    </dependency>
38.
39.    <dependency>
40.        <groupId>org.springframework</groupId>
41.        <artifactId>spring-context</artifactId>
42.        <version>6.1.6</version>
43.    </dependency>
44. </dependencies>
45.
46. <build>
47.     <plugins>
48.         <plugin>
49.             <groupId>org.codehaus.mojo</groupId>
50.             <artifactId>exec-maven-plugin</artifactId>
51.             <version>3.1.0</version>
52.             <configuration>
53.                 <mainClass>org.example.Main</mainClass>
54.                 <skip>false</skip>
55.             </configuration>
56.         </plugin>
57.     </plugins>
58. </build>
59.
60. </project>
```

**Листинг 11.2.** Файл build.gradle.kts

```
1. plugins {
2.     id("java")
3.     application
4. }
5.
6. group = "org.example"
7. version = "1.0-SNAPSHOT"
8.
9. application {
10.     mainClass = "org.example.Main"
11. }
12.
13. repositories {
14.     mavenCentral()
```

```

15. }
16.
17. dependencies {
18.     testImplementation(platform("org.junit:junit-bom:5.9.1"))
19.     testImplementation("org.junit.jupiter:junit-jupiter")
20.     implementation("org.springframework:spring-context:6.1.6")
21.     implementation("org.springframework:spring-beans:6.1.6")
22.     implementation("org.springframework:spring-core:6.1.6")
23. }
24.
25. tasks.test {
26.     useJUnitPlatform()
27. }

```

Существует 3 варианта создания конфигураций для Spring-приложения:

- 1) XML конфигурация;
- 2) конфигурация с помощью Java-аннотаций;
- 3) комбинирование XML-конфигурации и Java-аннотаций.

Разберем варианты XML-конфигурации и Java-аннотаций, зная их, уже можно будет комбинировать эти способы в крупных проектах.

При использовании XML-конфигурации нам нужно создать директорию resources, где будем хранить конфигурацию Spring-проекта. Расположение данной директории показано на рис. 11.1.



Рис. 11.1. Созданная директория

Содержание файла applicationContext.xml представлено на листинге 11.3.

#### Листинг 11.3. Файл applicationContext.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.   xsi:schemaLocation=

```

```
5.      "http://www.springframework.org/schema/beans
6.      http://www.springframework.org/schema/beans/
7.          spring-beans.xsd">
8.
9.      <bean id="Bean"
10.         class="org.example.Bean">
11.             <constructor-arg value="This is a Bean"/>
12.         </bean>
13.
14.     </beans>
```

Здесь мы объявили так называемый Bean, который представляет из себя по сути объект класса org.example.Bean, в конструктор которому передана строка «This is a Bean». По умолчанию все бины создаются с помощью паттерна Singleton, то есть при получении данного бина из любого места в коде мы получаем один и тот же объект.

Класс Bean представлен на листинге 11.4.

#### Листинг 11.4. Класс Bean

```
1. package org.example;
2.
3. public class Bean {
4.     private String name;
5.
6.     public Bean(String newName) {
7.         name = newName;
8.     }
9.
10.    public String getName() {
11.        return name;
12.    }
13. }
```

Теперь рассмотрим вопрос, как получить данный бин из другого класса. Получить бин можно, используя класс ClassPathXmlApplicationContext. Получение бина и его использование представлены на листинге 11.5.

#### Листинг 11.5. Получение и использование бина

```
1. package org.example;
2.
3. import org.springframework.context.support
4.         .ClassPathXmlApplicationContext;
5. public class Main {
6.     public static void main(String[] args) {
```

```

7.     ClassPathXmlApplicationContext context =
8.         new ClassPathXmlApplicationContext(
9.             "applicationContext.xml");
10.
11.    Bean testBean = context.getBean("Bean", Bean.class);
12.
13.    System.out.println(testBean.getName());
14.
15.    context.close();
16. }
17. }
```

После этого можно запустить приложение с помощью конфигурации запуска, как в разделе 9 или 10. Запуск представлен на рис. 11.2.

```

✓ pr: build finished At 4/1/25, 2:3 sec, 991 ms
Copying resources... [pr]
Dependency analysis found 0 affected files
Updating dependency information... [pr]
Parsing java... [pr]
Writing classes... [pr]
Adding nullability assertions... [pr]
Adding pattern assertions... [pr]
Dependency analysis found 0 affected files
Running 'after' tasks
javac 23.0.2 was used to compile java sources
Finished, saving caches...
Executing post-compile tasks...
Synchronizing output directories...
4/1/25, 2:48 PM - Build completed successfully in 3 sec, 991 ms
```

Рис. 11.2. Запуск приложения

Для создания конфигурации через Java-аннотации рекомендуется не создавать директорию resources с файлом конфигурации XML, а создать отдельный класс. Создадим класс и назовем его SpringConfig, что представлено на рис. 11.3.

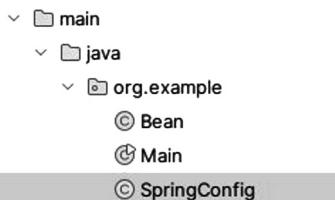


Рис. 11.3. Класс для конфигурирования через аннотации

Далее создадим конфигурацию, которая представлена на листинге 11.6.

**Листинг 11.6.** Конфигурация через аннотации

```
1. package org.example;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. @Configuration
7. public class SpringConfig {
8.     @Bean(name = "Bean")
9.     public org.example.Bean getBean() {
10.         return new org.example.Bean("This is a Bean");
11.     }
12. }
```

Затем пропишем получение бина и его использование, что представлено на листинге 11.7.

**Листинг 11.7.** Получение и использование бина

```
1. package org.example;
2.
3. import org.springframework.context.annotation
4.         .AnnotationConfigApplicationContext;
5. import org.springframework.context.support
6.         .ClassPathXmlApplicationContext;
7.
8. public class Main {
9.     public static void main(String[] args) {
10.         AnnotationConfigApplicationContext context =
11.             new AnnotationConfigApplicationContext(
12.                 SpringConfig.class);
13.
14.         Bean testBean = context.getBean("Bean", Bean.class);
15.
16.         System.out.println(testBean.getName());
17.
18.         context.close();
19.     }
20. }
```

Теперь можем запустить приложение, результат запуска представлен на рис. 11.4.

```
/home/      .jdks/corretto-23.0.2/bin/java ..
Hello Spring!

Process finished with exit code 0
```

Рис. 11.4. Запуск приложения

## Задание для выполнения лабораторной работы

Все классы сделать бинами, убрать наследование. В main вызвать каждый бин и вывести какую-либо информацию о нем.

1. Базовый класс: Животные. Дочерние классы: Кошка, Попугай, Рыбка.

2. Базовый класс: Сотрудник. Дочерние классы: Администратор, Программист, Менеджер.

3. Базовый класс: Человек. Дочерние классы: Студент, Преподаватель, Ассистент преподавателя.

4. Базовый класс: Транспортное средство. Дочерние классы: Легковой автомобиль, Грузовой автомобиль, Мотоцикл.

5. Базовый класс: Велосипед. Дочерние классы: Горный велосипед, Детский велосипед, BMX.

6. Базовый класс: Геометрическая фигура. Дочерние классы: Шар, Параллелепипед, Цилиндр.

7. Базовый класс: Книга. Дочерние классы: Аудиокнига, Фильм, Мюзикл.

8. Базовый класс: Мебель. Дочерние классы: Стол, Стул, Кровать.

9. Базовый класс: Монстр. Дочерние классы: Гоблин, Русалка, Дракон.

10. Базовый класс: Гаджеты. Дочерние классы: Часы, Смартфон, Ноутбук.

11. Базовый класс: Бытовая техника. Дочерние классы: Холодильник, Посудомоечная машина, Пылесос.

12. Базовый класс: Приложение. Дочерние классы: Социальная сеть, Игра, Погода.

13. Базовый класс: Оружие. Дочерние классы: Меч, Лук, Волшебная палочка.

14. Базовый класс: Заведение. Дочерние классы: Кафе, Магазин, Библиотека.

15. Базовый класс: Компьютерная периферия. Дочерние классы: Клавиатура, Наушники, Графический планшет.

## Контрольные вопросы

1. Что такое Spring Framework?
2. Какие проблемы решает Spring Framework?
3. Какие существуют способы конфигурирования Spring-приложения?
4. Когда следует использовать Spring Framework?
5. Как использовать Spring Framework в своем приложении, какие для этого нужны зависимости?
6. Что из себя представляет бин в приложении, написанном с использованием Spring Framework?
7. Как определить бин?
8. Какая роль ApplicationContext в приложении Spring?
9. Какими характеристиками по умолчанию обладает бин в приложении Spring?
10. Что происходит при вызове бина?

## 12 Inversion of Control (IoC) в Spring

---

IoC (Inversion of Control), или инверсия управления, представляет собой принцип разработки программного обеспечения, при котором контроль над потоком выполнения и созданием объектов переходит от приложения к фреймворку или контейнеру. Вместо того чтобы явно создавать и управлять объектами, разработчик определяет зависимости и описывает, как они должны быть созданы и внедрены в приложение. Это нужно для создания более гибких и модульных приложений.

Создаём новый проект в любой удобной среде разработки. После этого важно включить основные зависимости для корректной работы Spring.

Рассмотрим пример реализации инверсии управления. Создаем интерфейс Car (листинг 12.1) и классы, которые его реализуют: BMW (листинг 12.2), Mercedes (листинг 12.3). Также создадим «композицию» Dealer (листинг 12.4), которая будет содержать объект Car.

### Листинг 12.1. Интерфейс Car

```
1. package org.example;
2.
3. public interface Car {
4.     String getModel();
5. }
```

### Листинг 12.2. Класс BMW

```
1. package org.example;
2.
3. public class BMW implements Car {
4.     @Override
5.     public String getModel() {
6.         return "BMW";
7.     }
8. }
9.
```

**Листинг 12.3.** Класс Mercedes

```
1. package org.example;
2.
3. public class Mercedes implements Car {
4.     @Override
5.     public String getModel() {
6.         return "Mercedes";
7.     }
8. }
```

**Листинг 12.4.** Класс Dealer

```
1. package org.example;
2.
3. public class Dealer {
4.     private Car car;
5.
6.     // IoC
7.     public Dealer(Car music) {
8.         this.car = music;
9.     }
10.
11.    public void printModel() {
12.        System.out.println("Car model: " + car.getModel());
13.    }
14. }
```

Также создадим конфигурацию Spring-приложения или в виде XML-файла, что представлено на листинге 12.5, или с помощью аннотаций, что представлено на листинге 12.6.

**Листинг 12.5.** XML-конфигурация

```
1. <?xml version= "1.0" encoding= "UTF-8"?>
2. <beans xmlns= "http://www.springframework.org/schema/beans"
3.     xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation=
5.         "http://www.springframework.org/schema/beans
6.          http://www.springframework.org/schema/beans/
7.              spring-beans.xsd">
8.
9.     <bean id= "carBean"
10.        class= "org.example.BMW">
11.        </bean>
12.    </beans>
```

**Листинг 12.6.** Конфигурация с помощью аннотаций

```
1. package org.example;
2.
```

```
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. @Configuration
7. public class SpringConfig {
8.     @Bean(name = "carBean")
9.     public BMW getBean() {
10.         return new BMW();
11.     }
12. }
```

Теперь нужно создать класс, который реализует получение бина и инверсию управления с ним. Создаём класс Main, где это и реализуем. На листинге 12.7 представлена реализация посредством конфигурирования с помощью XML-файла, на листинге 12.8 — реализация посредством конфигурирования с помощью аннотаций.

**Листинг 12.7.** Класс Main для случая с XML-файлом

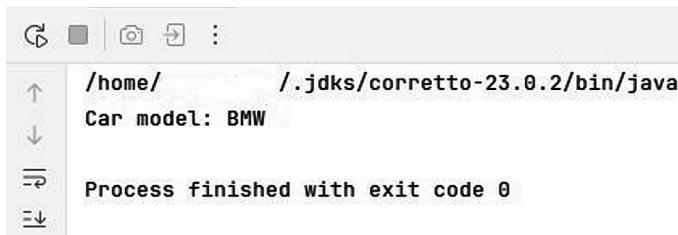
```
1. package org.example;
2.
3. import org.springframework.context.support
4.         .ClassPathXmlApplicationContext;
5.
6. public class Main {
7.     public static void main(String[] args) {
8.         ClassPathXmlApplicationContext context =
9.             new ClassPathXmlApplicationContext(
10.                 "applicationContext.xml");
11.
12.         Car car = context.getBean("carBean", Car.class);
13.         Dealer dealer = new Dealer(car);
14.         dealer.printModel();
15.         context.close();
16.     }
17. }
```

**Листинг 12.8.** Класс Main для случая с аннотациями

```
1. package org.example;
2.
3. import org.springframework.context.annotation
4.         .AnnotationConfigApplicationContext;
5.
6. public class Main {
7.     public static void main(String[] args) {
8.         AnnotationConfigApplicationContext context =
```

```
9.         new AnnotationConfigApplicationContext(
10.            SpringConfig.class);
11.
12.        Car car = context.getBean("carBean", Car.class);
13.        Dealer dealer = new Dealer(car);
14.        dealer.printModel();
15.        context.close();
16.    }
17. }
```

Теперь запускаем приложение и получаем результат, показанный на рис. 12.1.



```
Terminal /home/.jdks/corretto-23.0.2/bin/java
Car model: BMW
Process finished with exit code 0
```

Рис. 12.1. Работа приложения

Объяснение происходящего. Согласно принципу IoC, класс Dealer для работы должен получать объект из вне. Благодаря интерфейсу Car класс Dealer может работать со всеми созданными марками автомобилей, которые ему будут переданы в качестве параметра в конструктор. С помощью Spring Framework мы также вынесли создание объектов в конфигурационный файл. А потом получили данные объекты и использовали в классе Dealer. В итоге было создано приложение, которое полностью удовлетворяет принципам IoC.

### Задания для выполнения лабораторной работы

Сделать интерфейс для классов данных в задании согласно варианту. Все классы сделать бинами. Создать класс-композицию над объектом типа интерфейса. В main создать бины и передать их в класс-композицию согласно принципам IoC.

1. Базовый класс: Животные. Дочерние классы: Кошка, Попугай, Рыбка.
2. Базовый класс: Сотрудник. Дочерние классы: Администратор, Программист, Менеджер.

3. Базовый класс: Человек. Дочерние классы: Студент, Преподаватель, Ассистент преподавателя.
4. Базовый класс: Транспортное средство. Дочерние классы: Легковой автомобиль, Грузовой автомобиль, Мотоцикл.
5. Базовый класс: Велосипед. Дочерние классы: Горный велосипед, Детский велосипед, BMX.
6. Базовый класс: Геометрическая фигура. Дочерние классы: Шар, Параллелепипед, Цилиндр.
7. Базовый класс: Книга. Дочерние классы: Аудиокнига, Фильм, Мюзикл.
8. Базовый класс: Мебель. Дочерние классы: Стол, Стул, Кровать.
9. Базовый класс: Монстр. Дочерние классы: Гоблин, Русланка, Дракон.
10. Базовый класс: Гаджеты. Дочерние классы: Часы, Смартфон, Ноутбук.
11. Базовый класс: Бытовая техника. Дочерние классы: Холодильник, Посудомоечная машина, Пылесос.
12. Базовый класс: Приложение. Дочерние классы: Социальная сеть, Игра, Погода.
13. Базовый класс: Оружие. Дочерние классы: Меч, Лук, Волшебная палочка.
14. Базовый класс: Заведение. Дочерние классы: Кафе, Магазин, Библиотека.
15. Базовый класс: Компьютерная периферия. Дочерние классы: Клавиатура, Наушники, Графический планшет.

## Контрольные вопросы

1. Что такое инверсия управления?
2. Каковы основные принципы инверсии управления?
3. Для чего используется Spring Framework при реализации инверсии управления?
4. Зачем нужны принципы инверсии управления?
5. Роль интерфейсов при реализации инверсии управления?
6. Что имеет контроль над созданием объектов при реализации инверсии управления с помощью Spring Framework?
7. Почему создание объектов вручную имеет отрицательную оценку?
8. Каких негативных последствий можно избежать, если исключить создание вручную объектов?

## 13 Dependency injection (DI) в Spring

---

DI (Dependency Injection, внедрение зависимости) — процесс, при котором построение одного объекта предоставляется внешнему объекту. Или, точнее, это то место, где зависимость будет внедрена другим объектом. Понятнее будет на примере. Пусть у нас есть класс «Автомобиль», у него есть поле класса «Двигатель». Место, где «Двигатель» будет инициализирован в «Автомобиле», и будет внедрением зависимости.

В Spring Boot существует четыре типа DI:

- как поле класса;
- как приватное поле класса (не является хорошим способом);
- с помощью setter;
- с помощью конструктора.

DI через сеттер остаётся спорным из-за особенностей Spring Boot и его механизма IoC. Обычно программист сам управляет созданием объектов и их зависимостей, но в Spring Boot это делает IoC во время выполнения программы (Runtime). При использовании DI через сеттер невозможно точно определить, в какой момент будет внедрена зависимость, что может привести к неожиданному поведению.

DI как поле класса используется редко по причине нарушения инкапсуляции, ведь внедряемое поле должно быть помечено как public.

DI с помощью конструктора — наиболее предпочтительный способ осуществления DI. Связывание будет осуществлено в момент создания объекта, и вы точно будете знать когда именно IoC осуществит вызов.

Создаём новый проект в любой удобной среде разработки. После этого важно включить основные зависимости для корректной работы Spring.

Рассмотрим пример реализации внедрения зависимости. За основу возьмем пример из раздела 12. Там в классе Dealer есть конструктор, с помощью которого как раз можно осуществить

**внедрение зависимости.** Наша работа с классом Dealer представлена на листинге 13.1 для случая конфигурации с аннотациями и на листинге 13.2 для случая с XML-конфигурацией.

**Листинг 13.1.** Класс Main при работе с аннотациями

```
1. package org.example;
2.
3. import org.springframework.context.annotation
4.      .AnnotationConfigApplicationContext;
5.
6. public class Main {
7.     public static void main(String[] args) {
8.         AnnotationConfigApplicationContext context =
9.             new AnnotationConfigApplicationContext(
10.                SpringConfig.class);
11.
12.        Car car = context.getBean("carBean", Car.class);
13.        Dealer dealer = new Dealer(car);
14.        dealer.printModel();
15.        context.close();
16.    }
17. }
```

**Листинг 13.2.** Класс Main при работе XML-конфигурацией

```
1. package org.example;
2.
3. import org.springframework.context.support
4.      .ClassPathXmlApplicationContext;
5.
6. public class Main {
7.     public static void main(String[] args) {
8.         ClassPathXmlApplicationContext context =
9.             new ClassPathXmlApplicationContext(
10.                "applicationContext.xml");
11.
12.        Car car = context.getBean("carBean", Car.class);
13.        Dealer dealer = new Dealer(car);
14.        dealer.printModel();
15.        context.close();
16.    }
17. }
```

Чтобы сделать внедрение зависимости с помощью данного конструктора, нужно поменять Spring-конфигурацию. В конфигурации нужно создать бин класса, который хотим внедрить, а также создать сам класс, в который внедряем. Внедрение нужно

произвести в конфигурации. Итоговый файл XML-конфигурации представлен на листинге 13.3, конфигурации с помощью аннотаций — на листинге 13.4.

#### Листинг 13.3. XML-конфигурация с DI

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.   xsi:schemaLocation=
5.     "http://www.springframework.org/schema/beans
6.      http://www.springframework.org/schema/beans/
7.          spring-beans.xsd">
8.
9. <bean id="carBean"
10.    class="org.example.Mercedes">
11. </bean>
12.
13. <bean id="dealerBean"
14.    class="org.example.Dealer">
15.    <constructor-arg ref="carBean"/>
16. </bean>
17.
18. </beans>
```

#### Листинг 13.4. Конфигурация аннотациями с DI

```
1. package org.example;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. @Configuration
7. public class SpringConfig {
8.   @Bean(name = "carBean")
9.   public Mercedes getCarBean() {
10.     return new Mercedes();
11.   }
12.
13.   @Bean(name = "dealerBean")
14.   public Dealer getDealerBean() {
15.     return new Dealer(getCarBean());
16.   }
17. }
```

Итоговый класс Main представлен на листинге 13.5 для случая конфигурации с аннотациями и на листинге 13.6 для случая с XML-конфигурацией.

**Листинг 13.5.** Класс Main при работе с аннотациями

```
1. package org.example;
2. import org.springframework.context.annotation
3.         .AnnotationConfigApplicationContext;
4.
5. public class Main {
6.     public static void main(String[] args) {
7.         AnnotationConfigApplicationContext context =
8.             new AnnotationConfigApplicationContext(
9.                 SpringConfig.class);
10.
11.        Dealer dealer = context.getBean("dealerBean", Dealer.class);
12.        dealer.printModel();
13.        context.close();
14.    }
15. }
```

**Листинг 13.6.** Класс Main при работе XML-конфигураций

```
1. package org.example;
2.
3. import org.springframework.context.support
4.         .ClassPathXmlApplicationContext;
5.
6. public class Main {
7.     public static void main(String[] args) {
8.         ClassPathXmlApplicationContext context =
9.             new ClassPathXmlApplicationContext(
10.                "applicationContext.xml");
11.
12.        Dealer dealer = context.getBean("dealerBean", Dealer.class);
13.        dealer.printModel();
14.        context.close();
15.    }
16. }
```

В Spring Framework существует другой способ внедрения зависимостей с использованием аннотации Autowired. Для того чтобы объект стал потенциальным бином, его необходимо пометить аннотацией Component, что представлено на листингах 13.7 и 13.8.

**Листинг 13.7.** Класс BMW с аннотацией

```
1. package org.example;
2.
3. import org.springframework.stereotype.Component;
4.
```

```
5. @Component("bmwBean")
6. public class BMW implements Car {
7.     @Override
8.     public String getModel() {
9.         return "BMW";
10.    }
11. }
```

**Листинг 13.8.** Класс Mercedes с аннотацией

```
1. package org.example;
2.
3. import org.springframework.stereotype.Component;
4.
5. @Component("mercedesBean")
6. public class Mercedes implements Car {
7.     @Override
8.     public String getModel() {
9.         return "Mercedes";
10.    }
11. }
```

После обозначения компонентов следует внедрить их в класс. Внедрять можно даже в приватные поля класса. Реализуем внедрение в Dealer, что представлено на листинге 13.9.

**Листинг 13.9.** Внедрение в Dealer с помощью Autowired

```
1. package org.example;
2.
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.beans.factory.annotation.Qualifier;
5.
6. public class Dealer {
7.     @Autowired
8.     @Qualifier("mercedesBean")
9.     private Car car;
10.
11.    public Dealer() {}
12.
13.    public void printModel() {
14.        System.out.println("Car model: " + car.getModel());
15.    }
16. }
```

Для правильности внедрения таким способом в конфигурации важно указать сканирование компонентов. Конфигурация

с помощью Java-кода представлена на листинге 13.10, конфигурация с помощью XML-файла представлена на листинге 13.11. Класс Main остаётся неизмененным.

**Листинг 13.10.** Конфигурация с помощью Java-кода

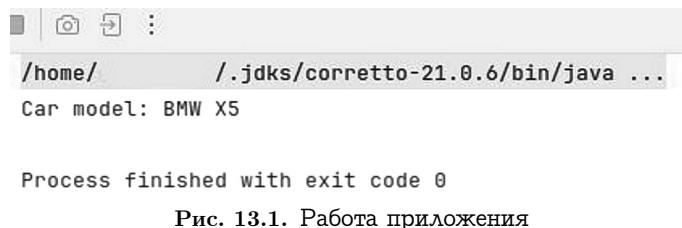
```
1. package org.example;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.ComponentScan;
5. import org.springframework.context.annotation.Configuration;
6.
7. @Configuration
8. @ComponentScan("org.example")
9. public class SpringConfig {
10.     @Bean(name = "dealerBean")
11.     public Dealer getDealerBean() {
12.         return new Dealer();
13.     }
14. }
```

**Листинг 13.11.** Конфигурация с помощью XML-файла

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.   xmlns:context="http://www.springframework.org/schema/
5.     context"
6.   xsi:schemaLocation="http://www.springframework.org/
7.     schema/beans
8.     http://www.springframework.org/schema/beans/spring-beans.xsd
9.     http://www.springframework.org/schema/context
10.    http://www.springframework.org/schema/context/
11.      spring-context.xsd">
12.
13. <context:component-scan base-package="org.example" />
14.
15. <bean id="dealerBean"
16.       class="org.example.Dealer">
17.   </bean>
18.
19. </beans>
```

При запуске всех вариантов приложения результат представлен на рис. 13.1.

В результате показаны примеры реализации DI с помощью Spring Framework.



```
[home/.jdks/corretto-21.0.6/bin/java ...  
Car model: BMW X5  
  
Process finished with exit code 0
```

Рис. 13.1. Работа приложения

## Задания для выполнения лабораторной работы

Сделать интерфейс для классов данных в задании согласно варианту. Все классы сделать бинами. Создать класс-композицию над объектом типа интерфейса. Сделать внедрение зависимости в класс-композицию с помощью XML-конфигурации, Java-конфигурации, аннотации Autowired.

1. Базовый класс: Животные. Дочерние классы: Кошка, Попугай, Рыбка.
2. Базовый класс: Сотрудник. Дочерние классы: Администратор, Программист, Менеджер.
3. Базовый класс: Человек. Дочерние классы: Студент, Преподаватель, Ассистент преподавателя.
4. Базовый класс: Транспортное средство. Дочерние классы: Легковой автомобиль, Грузовой автомобиль, Мотоцикл.
5. Базовый класс: Велосипед. Дочерние классы: Горный велосипед, Детский велосипед, ВМХ.
6. Базовый класс: Геометрическая фигура. Дочерние классы: Шар, Параллелепипед, Цилиндр.
7. Базовый класс: Книга. Дочерние классы: Аудиокнига, Фильм, Мюзикл.
8. Базовый класс: Мебель. Дочерние классы: Стол, Стул, Кровать.
9. Базовый класс: Монстр. Дочерние классы: Гоблин, Руслака, Дракон.
10. Базовый класс: Гаджеты. Дочерние классы: Часы, Смартфон, Ноутбук.
11. Базовый класс: Бытовая техника. Дочерние классы: Холодильник, Посудомоечная машина, Пылесос.
12. Базовый класс: Приложение. Дочерние классы: Социальная сеть, Игра, Погода.
13. Базовый класс: Оружие. Дочерние классы: Меч, Лук, Волшебная палочка.

14. Базовый класс: Заведение. Дочерние классы: Кафе, Магазин, Библиотека.

15. Базовый класс: Компьютерная периферия. Дочерние классы: Клавиатура, Наушники, Графический планшет.

## Контрольные вопросы

1. Что такое внедрение зависимостей?
2. Для чего используется Spring Framework при реализации внедрения зависимостей?
3. Как можно реализовать в Spring Framework внедрение зависимостей?
4. Какова роль интерфейсов при реализации внедрения зависимостей?
5. Какие существуют типы внедрения зависимостей?
6. Какие преимущества даёт нам внедрение зависимостей?
7. Как происходит построение объекта при правильном внедрении зависимостей?
8. Как реализуется внедрение зависимостей при помощи аннотации Autowired?

## Литература

1. Блох Дж. Эффективная Java. — СПб.: Питер, 2019.
2. Хорстманн К.С. Java. Библиотека профессионала (Том 1, Том 2). — СПб.: Питер, 2022.
3. Эккель Б. Философия Java. — СПб.: Питер, 2021.
4. Блинов И., Романчик В. Java. Методы программирования. — СПб.: БХВ-Петербург, 2023.
5. Шилдт Г. Java. Полное руководство. — М.: Диалектика, 2022.
6. Уоллс К. Spring в действии. — М.: ДМК Пресс, 2022. — 544 с.
7. Раджпут Д. Spring. Все паттерны проектирования. — СПб.: Питер, 2019. — 320 с.
8. Докука О., Лозинский И. Практика реактивного программирования в Spring. — М.: ДМК Пресс, 2019. — 320 с.

# Оглавление

<b>1. История языка программирования Java. Типы данных</b> .....	3
Задания для выполнения лабораторной работы.....	4
Контрольные вопросы .....	11
<b>2. Объектно-ориентированное программирование...</b>	12
Задания для выполнения лабораторной работы.....	14
Контрольные вопросы .....	23
<b>3. Класс Object. работа с хэш-таблицами .....</b>	24
Задания для выполнения лабораторной работы.....	25
Контрольные вопросы .....	28
<b>4. Обработка исключений .....</b>	30
Задания для выполнения лабораторной работы.....	31
Контрольные вопросы .....	33
<b>5. Строки и регулярные выражения.....</b>	35
Задания для выполнения лабораторной работы.....	37
Контрольные вопросы .....	41
<b>6. Работа с коллекциями .....</b>	42
Задания для выполнения лабораторной работы.....	43
Контрольные вопросы .....	48
<b>7. Многопоточность .....</b>	51
Задания для выполнения лабораторной работы.....	52
Контрольные вопросы .....	56
<b>8. Аннотации. Stream API.....</b>	57
Задания для выполнения лабораторной работы.....	58
Контрольные вопросы .....	62
<b>9. Инструмент сборки Maven .....</b>	63
Задания для выполнения лабораторной работы.....	64
Контрольные вопросы .....	75
<b>10. Инструмент сборки Gradle.....</b>	76
Задания для выполнения лабораторной работы.....	77

Контрольные вопросы .....	85
<b>11. Введение в работу с фреймворком Spring .....</b>	<b>85</b>
Задания для выполнения лабораторной работы.....	87
Контрольные вопросы .....	94
<b>12. Inversion of control (IoC) в Spring .....</b>	<b>95</b>
Задания для выполнения лабораторной работы.....	96
Контрольные вопросы .....	99
<b>13. Dependency injection (DI) в Spring.....</b>	<b>100</b>
Задания для выполнения лабораторной работы.....	102
Контрольные вопросы .....	108
<b>Литература .....</b>	<b>109</b>

Адрес издательства в Интернет [www.techbook.ru](http://www.techbook.ru)

Учебное издание

**Городничев** Михаил Геннадьевич, **Мосева** Марина Сергеевна,  
**Харрасов** Камиль Раисович, **Водиченков** Антон Дмитриевич

## **Информационные технологии и программирование на языке Java**

Учебное пособие для вузов

Редактор Ю. Н. Чернышов  
Компьютерная верстка Ю. Н. Чернышова  
Обложка художника В. В. Казюлина

Подписано в печать 30.06.2025. Печать цифровая. Формат 60×90/16. Уч. изд. л. 7.

Тираж 500 экз. (1-й завод – 50 экз.) Изд. № 251157

ООО «Научно-техническое издательство «Горячая линия – Телеком»