Конспект по предмету "Введение в программирование"

Грехов Александр

Содержание

1 Информация о курсе

1.1 Основная информация

Поток — y2024.
Группы M3136-M3142.
Преподаватель — Корнеев Георгий Александрович.

Курс фокусируется на языке программирования Java.
Для данного курса подойдет JDK 17 и выше.

1.2 Соглашение о коде

- Максимальная ширина строки — 120 символов
- Отступы обязательны
  - Размер отступа — 4 пробела
- Фигурные скобочки
  - Открывающая — на той же строке
  - Ракрывающая — на отдельной строке

1.3 Соглашение об именовании

- `TypeNamingConvention`
- `methodNamingConvention()`
- `fieldNamingConvention`
- `variableNamingConvention`
- `CONSTANT_NAMING_CONVENTION`

2 Основные концепции Java

2.1 Компилятор Java (javac)

Компилятор Java (javac) — это инструмент, который преобразует исходный код Java в байт-код.

```
javac HelloWorld.java
```

2.2 Байт-код и виртуальная машина Java (java)

Байт-код — это промежуточный код, который может быть выполнен на любой платформе, где установлена виртуальная машина Java (JVM).

Виртуальная машина Java (JVM) — это интерпретатор байт-кода, который позволяет выполнять Java-программы на любой платформе.

```
java HelloWorld
```

2.3 Java Runtime Environment (JRE)

Java Runtime Environment (JRE) — это набор библиотек и других компонентов, необходимых для выполнения Java-программ. JRE включает JVM, библиотеки классов и другие ресурсы.

2.4 JIT-компиляция

JIT-компиляция (Just-In-Time) — это метод оптимизации, при котором байт-код компилируется в машинный код непосредственно перед выполнением. Это позволяет улучшить производительность Java-программ.

2.5 Сборка мусора

Сборка мусора (Garbage Collection) — это процесс автоматического управления памятью, при котором неиспользуемые объекты удаляются из памяти, освобождая ресурсы для новых объектов.

2.6 Редакции Java-платформы

Java-платформа имеет несколько редакций, каждая из которых предназначена для различных типов приложений:

- **Micro Edition (Java ME)**
  Java ME предназначена для мобильных устройств и встроенных систем. Она включает подмножество библиотек Java SE и дополнительные API для работы с ограниченными ресурсами.
- **Standard Edition (Java SE))**
  Java SE — это основная редакция Java, предназначенная для разработки настольных и серверных приложений. Она включает полный набор библиотек и инструментов для разработки.
- **Enterprise Edition (Java EE)**
  Java EE предназначена для разработки корпоративных приложений. Она включает дополнительные API для работы с веб-сервисами, компонентами и другими корпоративными технологиями.

3 Начало работы с Java

3.1 Класс и основная функция

Код на Java начинается с создания класса.
Название класса файла должно совпадать с названием файла.

Используем `public class FileName`, чтобы создать класс.

```java
// В файле "HelloWorld.java" создадим класс "HelloWorld"
public class HelloWorld {}
```

Чтобы можно было запускать программу напрямую (как основной файл), в классе расположим функцию `main()`. Она задается как `public static void`.

```java
// В классе "HellowWorld" расположим функцию "main"
public class HelloWorld {
    public static void main() {}
}
```

Принятно передавать в `main()` аргументы, введенные при запуске. Это делается через массив строк, названный `args`.

```java
public class HelloWorld {
    public static void main(String[] args) {}
}
```

3.2 Переменные

- **Основные типы данных в Java**
  - `boolean`: хранит значение true или false.
  - `byte`: целое число от -2^7 до $2^7 - 1$ (1 байт).
  - `short`: целое число от -2^{15} до $2^{15} - 1$ (2 байт).
  - `int`: целое число от -2^{31} до $2^{31} - 1$ (4 байт).
  - `long`: целое число от -2^{63} до $2^{63} - 1$ (8 байт).
  - `float`: число с плавающей точкой (4 байта).
  - `double`: число с плавающей точкой двойной точности (8 байт).

- `char`: символ Unicode (2 байта).

- **Ссылочные типы данных**
  - `variable[]`: массив, где `variable` — тип данных
  - `String`: строка

Переменные обявляются в помощью конструкции `variable: variableName;`. Переменную можно сразу инициализировать, придав ей значение.

```
public class HelloWorld {
    public static void main(String[] args) {
        boolean firstBoolVariable;
        boolean secondBoolVariable = true;
        int intVariable = 123;
        String stringVariable = "HelloWorld";
    }
}
```

3.3 Стандартный поток ввода и вывода

Стандартный поток осуществляется через класс `System`.

- Через статические поле `out` производится поток вывода.
- Через статические поле `err` производится поток ввода ошибок.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Это стандартное сообщение.");
        System.err.println("Это сообщение об ошибке.");
    }
}
```

- Через статические поле `in` производится поток ввода.

  Однако чтобы им воспользоваться, необходимо использовать специальный класс `Scanner`.

```
import java.util.Scanner;
public class HelloWorld {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Введите имя: ");
```

```
        String name = scanner.nextLine();
        System.out.println("Здравствуйте, " + name);
    }
}
```

3.4 Условные конструкции

В Java условные конструкции работают вполне стандартно.

Условная конструкция начинается с `if`, потом круглых скобочках передается условие — некоторое выражение, возвращающее булевое значение (`true` или `false`).
В фигурных скобочках же указывается тело условной конструкции — то, что будет выполнено при значении true.

С помощью конструкции `else if` можно прописать дополнительное условие, если в прошлом булевое выражение вернуло `false`.
Так же есть возможность прописать отдельное тело, если ни одно условие не вернуло true, это делается с помощью `else`.

```
// Получим количество введенных при запуске агрументов с помощью метода .length

public class HelloWorld {
    public static void main(String[] args) {
        int numOfArgs = args.length

        if (args.length == 0) {
            System.out.println("Hello, World!");
        } else if (args.length == 1) {
            System.out.println("Hello, " + args[0]);
        } else {
            System.err.println("Expected zero or one argument, got: " +
            ↪  args.length);
        }
    }
}
```

3.5 Циклы

В Java существует несколько видов циклов. Они вполне стандартны и взаимозаменяемы, обычно есть в любом языке.

- Цикл `for`: Этот цикл используется, когда известно количество итераций заранее.

  В круглых скобочках настроивается работа цикла.

  – Сначала инициализируется переменная, которая будет выступать в качестве счетчика.

  – Далее указывается условие, при котором цикл продолжает запускать итерации.

  – И в конце указывается то, как счетчик будет меняться с каждой итерацией.

  В фигурных скобочках размещается тело цикла. Код внутри будет запускаться каждую итерацию.

```java
// Данный цикл будет работать 10 раз, выводя номер итерации.

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

// Пример бесконечного цикла if

if (;;) {
    System.out.println("Беспокнечный цикл");
}
```

- Цикл `while`: Этот цикл выполняется до тех пор, пока условие, указанное в круглых скобочках, истинно.
  Он проверяет условие перед каждой итерацией.

  Как и у цикла `if`, в фигурных скобочках размещается тело цикла. Код внутри будет запускаться каждую итерацию.

```java
// Данный цикл будет работать пока i меньше 64, выводя степени двойки до 64.

int i = 1;
while (i < 64) {
    System.out.println(i);
    i = i * 2;
}

// Пример бесконечного цикла while

```

```java
while (true) {
    System.out.println("Беспокнечный цикл");
}
```

- Цикл `do...while`: Этот цикл похож на `while`, но условие проверяется после выполнения тела цикла, что гарантирует хотя бы одно выполнение действий.

```java
// Данный цикл продолжает итерации пока i меньше 2
// Несмотря на то, что i изначально задана как 2, цикл произведет первую
 ↪  итерацию.

int i = 2
do {
    System.out.println(i);
    i = i * 2;
} while (i < 2);
```

- Цикл `for-each`: Этот цикл используется для перебора элементов коллекции или массива.

  В круглых скобочках указывается тип элементов и сам объект, который нужно перебрать.

```java
// Этот цикл переберет все числа в массиве numbers.

int[] numbers = {1, 2, 3, 4, 5};
for (int number : numbers) {
    System.out.println(number);
}
```

3.6 Функции

В Java функции называются методами. Метод — это блок кода, который выполняет определенную задачу и может быть вызван по имени. Методы позволяют структурировать и переиспользовать код.

1. Объявление метода: Методы объявляются внутри классов. Если метод ничего не возвращает, используется ключевое слово `void`.

```java
public static void printMessage() {
    System.out.println("Hello, World!");
}
```

2. Параметры метода: Методы могут принимать параметры, которые передаются при вызове метода.

```java
public void greet(String name) {
    System.out.println("Hello, " + name);
}
```

3. Возвращаемый тип: Если метод возвращает значение, необходимо обозначить тип возвращаемых данных во время объявления метода.

```java
public int sum(int a, int b) {
    return a + b;
}
```

Метод возвращает `int`, поэтому и объявлен как `public int sum()`.

4 Массивы и ссылки

4.1 Одномерные массивы

- **Объявление**

  Одномерный массив объявляется с использованием квадратных скобок `[]` после типа данных.

```
int[] array;
```

  Переменная `array` может хранить ссылку на массив целых чисел.

  Объявление массива создаёт ссылку на массив, но сам массив ещё не создан. Если попытаться использовать массив до его создания, будет выброшено исключение `NullPointerException`.

- **Создание**

  Создание массива включает в себя выделение памяти для элементов массива с использованием ключевого слова `new`.

```
array = new int[10];
```

  Создание массива выделяет память для хранения указанного количества элементов указанного типа (в данном случае 10-ти элементов `int`).

  Размер массива фиксирован и не может быть изменён после создания. Если нужно изменить размер, придётся создать новый массив и скопировать данные.

- **Вывод данных**

  Можно выводить элементы массива по-отдельности, указывая в `[]` индекс нужного элемента.

```
    int[] array = {1, 2};

    System.out.println(array[0]);
    System.out.println(array[1]);
```

  Чтобы вывести весь массив, можно использовать метод `Arrays.toString()`, который преобразует массив в строку.

```
import java.util.Arrays;

public class Main {
```

```
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};
        System.out.println(Arrays.toString(array));
    }
}
```

- **Длина массива**

  Доступ к длине массива осуществляется через свойство `length`.

```
int length = array.length;
```

  `length` — это свойство, а не метод, поэтому не нужно использовать круглые скобки.

- **Инициализаторы**

  Массивы могут быть инициализированы при создании с использованием фигурных скобок `{}`.

```
int[] array = {1, 2, 3, 4, 5};
```

  При инициализации массива таким образом, размер массива определяется автоматически на основе количества элементов.
  Нельзя указать и длинну массива и элементы одновременно.

- **Инициализация по-умолчанию**

  При создании массива все его элементы инициализируются значениями по умолчанию (для `int` это 0).

```
int[] array = new int[5];
// Все элементы инициализируются нулями
```

  Для других типов данных значения по умолчанию обычно эквивалентны нулю (например, для `boolean` это `false`, для объектов — `null`).

- **Итерация**

  Итерация по массиву может быть выполнена с использованием цикла `for` или `for-each`.

```
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}

for (int element : array) {
```

```
        System.out.println(element);
    }
```

В начале каждой итерации в `element` загружается копия елемента массива. Поэтому цикл `for-each` удобен для чтения элементов массива, но не позволяет изменять элементы.

- **Сравнение**

  Метод `.equals()` в Java используется для сравнения содержимого объектов на равенство. В отличие от оператора `==`, который сравнивает ссылки на объекты, метод `.equals()` сравнивает значения объектов.

```
public class Main {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "Hello";
        String str3 = new String("Hello");

        // Сравнение с использованием оператора ==
        System.out.println(str1 == str2); // true
        System.out.println(str1 == str3); // false

        // Сравнение с использованием метода .equals()
        System.out.println(str1.equals(str2)); // true
        System.out.println(str1.equals(str3)); // true
    }
}
```

  В этом примере:

  – `str1 == str2` возвращает `true`, потому что обе переменные ссылаются на один и тот же объект в пуле строк.

  – `str1 == str3` возвращает `false`, потому что `str3` создается с использованием оператора `new`, что создает новый объект в памяти.

  – `str1.equals(str2)` и `str1.equals(str3)` возвращают `true`, потому что метод `.equals()` сравнивает содержимое строк, а не их ссылки.

4.2 Многомерные массивы

- **Объявление**

Многомерные массивы объявляются с использованием нескольких пар квадратных скобок.

```
int[][] matrix;
```

Многомерные массивы могут иметь любое количество измерений.

- **Полное и частичное создание**

  Полное создание массива:

```
matrix = new int[3][3];
```

  Частичное создание массива:

```
matrix = new int[3][];
matrix[0] = new int[2];
matrix[1] = new int[3];
matrix[2] = new int[4];
```

- **Непрямоугольные массивы**

  Массивы могут быть непрямоугольными, как показано в примере выше. Это означает, что каждая строка может иметь разное количество столбцов.

- **Вывод двумерного массива**

  Чтобы вывести весь массив, можно использовать метод `Arrays.deepToString()`, который преобразует многомерный массив в строку.

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };
        System.out.println(Arrays.deepToString(matrix));
    }
}
```

4.3 Ссылки на массивы

- **Ссылки**

Массивы в Java являются объектами, и переменные, содержащие массивы, являются ссылками на эти объекты.

```
int[] array1 = {1, 2, 3};
int[] array2 = array1;
```

Теперь `array2` ссылается на тот же массив, что и `array1`.
Изменения, внесенные через одну ссылку, будут видны через другую.

- **Передача ссылок**

  При передаче массива в метод передаётся ссылка на массив, а не его копия.

  Изменения, внесённые в массив внутри метода, будут видны снаружи метода.

- **Возврат ссылок**

  Метод может возвращать ссылку на массив.

```
public int[] createArray() {
    return new int[]{1, 2, 3};
}
```

- **Копирование массива**

  Чтобы скопировать массив можно использовать метод `System.arraycopy()`.

```
int[] original = {1, 2, 3, 4, 5};
int[] copy = new int[original.length];
System.arraycopy(original, 0, copy, 0, original.length);
```

  С помощью этого метода можно создать новый массив, который будет являться копией оригинального массива.

4.4 Классы

- **Полные имена классов**

  Полное имя класса включает пакет, в котором он находится.

```
java.util.Scanner scanner;
```

  Полные имена классов используются для избежания конфликтов имен.

- **Импорт классов**

  Импорт позволяет использовать классы без указания полного имени.

```java
import java.util.Scanner;
```

Можно импортировать все классы из пакета, но это может привести к конфликтам имён.

```java
import java.util.*;
```

- **Создание объектов**

  Объекты создаются с помощью ключевого слова `new`, которое вызывает конструктор класса.

```java
Scanner scanner = new Scanner(System.in);
```

  Создание объекта класса Scanner, который будет читать данные из стандартного ввода.

- **Сборка мусора и уничтожение объектов**

  Сборка мусора освобождает память, занятую объектами, которые больше не используются. Сборка мусора происходит автоматически.

  Важно закрывать ресурсы (например, файлы или сетевые соединения), чтобы избежать утечек.

4.5 Scanner

- **Источники данных**
  - **Строка**

    `Scanner` может читать данные из строки.

```java
Scanner scanner = new Scanner(System.in);
```

  - **Стандартный ввод**

    `Scanner` может читать данные из стандартного ввода (клавиатуры).

```java
Scanner scanner = new Scanner(System.in);
```

    Используется для интерактивного ввода данных пользователем.

- **Получение данных**
  - **Строки**

    Метод `nextLine` читает всю строку до символа новой строки.

```java
String input = scanner.nextLine();
```

Может использоваться для чтения строк, содержащих пробелы.

– **Числа**

Метод `nextInt` читает целое число из ввода.

```java
int number = scanner.nextInt();
```

Если ввод не является числом, будет выброшено исключение `InputMismatchException`.
Чтобы избежать этого, можно использовать метод `hasNextInt` для проверки наличия следующего целого числа.

5 Ввод-вывод и исключения

5.1 Исключения

- **try-catch**

  Исключение — это событие, которое происходит во время выполнения программы и нарушает нормальный поток её инструкций.
  Когда в методе возникает ошибка, метод создает объект исключения и передает его системе выполнения.

  Конструкция `try-catch` используется для обработки исключений в Java.

  Код, который может вызвать исключение, помещается в блок `try`. Если возникает исключение, оно перехватывается блоком `catch`, где можно обработать ошибку.

```java
try {
    // Код, который может вызвать исключение
    int result = 10 / 0;
} catch (ArithmeticException e) {
    // Обработка исключения
    System.out.println("Деление на ноль запрещено!");
}
```

  Важно правильно определять тип исключения в блоке catch.
  Если исключение не будет поймано, программа завершится с ошибкой.

- **Проверяемые исключения, throws**

  Проверяемые исключения (`checked exceptions`) должны быть либо обработаны в блоке `try-catch`, либо объявлены в сигнатуре метода с помощью ключевого слова `throws`.

  Ключевое слово `throws` в Java используется для указания того, что метод может выбросить одно или несколько исключений. Это позволяет информировать вызывающий метод о возможных исключениях, чтобы они могли быть обработаны соответствующим образом.

```java
public void readFile(String fileName) throws IOException {
    FileReader file = new FileReader(fileName);
    BufferedReader fileInput = new BufferedReader(file);
    fileInput.close();
}
```

В данном случае метод readFile может выбросить IOException, это ошибку нужно обработать при вызове readFile.

Все исключения, кроме наследников `RuntimeException`, являются проверяемыми.

- **Обработка исключений**

  Обработка исключений позволяет программе продолжать выполнение после возникновения ошибки. Это достигается путем перехвата и обработки исключений, что предотвращает завершение программы.

  – **Несколько catch-блоков**

  Можно использовать несколько блоков `catch` для обработки различных типов исключений.

```java
try {
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[10]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Ошибка: индекс вне границ массива");
} catch (Exception e) {
    System.out.println("Общая ошибка");
}
```

  Блоки `catch` проверяются сверху вниз, поэтому более специфичные исключения должны быть выше.

  – **Сообщения об ошибках**

  Сообщения об ошибках помогают понять причину исключения.

```java
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.err.println("Ошибка: " + e.getMessage());
}
```

  Можно вручную писать текст ошибки, и выводить через стандартный поток вывода ошибок или ипользовать `e.getMessage()` для получения подробного сообщения об ошибке.

  – **Стек исполнения**

  Стек исполнения (stack trace) показывает последовательность вызовов мето-

дов, которые привели к исключению. Используйте метод `e.printStackTrace();`.

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    e.printStackTrace();
}
```

Полезно для отладки и понимания, где произошло исключение.

5.2 Ресурсы

Ресурсы в Java — это объекты, которые должны быть закрыты после использования. Примеры включают файлы, сетевые соединения и базы данных. Закрытие ресурсов важно для предотвращения утечек ресурсов.

- **Закрытие и утечка ресурсов**

  Если ресурсы не закрываются, это может привести к утечке ресурсов.

  Утечка ресурсов происходит, когда программа не освобождает ресурсы после их использования, что может привести к исчерпанию доступных ресурсов, снижению производительности и ошибкам.

- **try-catch-finally**

  Конструкция `try-catch-finally` используется для обработки исключений и гарантирует выполнение определенного кода независимо от того, произошло исключение или нет.

  – `try`: Блок, в котором выполняется код, который может вызвать исключение.

  – `catch`: Блок, который обрабатывает исключения, возникшие в блоке `try`.

  – `finally`: Блок, который выполняется в любом случае, независимо от того, было ли исключение или нет.

```
try {
    // Код, который может вызвать исключение
} catch (ExceptionType e) {
    // Обработка исключения
} finally {
    // Код, который выполняется в любом случае (например закрытие ресурсов)
}
```

Обычно используется для освобождения ресурсов.

- **Блок использования ресурса**

  С Java 7 был введен `try-with-resources` — это специальная конструкция `try`, которая автоматически закрывает ресурсы по завершении блока.

```java
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    // Использование ресурса
} catch (IOException e) {
    // Обработка исключения
}
```

  Ресурс должен реализовывать интерфейс `AutoCloseable`.

- **Использование нескольких ресурсов одновременно**

  Можно использовать несколько ресурсов в одном блоке try-with-resources.

```java
try (FileReader file1 = new FileReader("file1.txt");
     FileReader file2 = new FileReader("file2.txt")) {
    // работа с файлами
} catch (IOException e) {
    e.printStackTrace();
}
```

  Все ресурсы будут закрыты в обратном порядке их объявления.

5.3 Кодировки

Кодировки играют важную роль при работе с текстовыми данными в Java, особенно при чтении и записи файлов. Давайте рассмотрим основные аспекты работы с кодировками.

- **Кодировка по-умолчанию**

  При работе с файлами в Java, кодировка по умолчанию зависит от операционной системы. Например, на Windows это может быть `Cp1251` для русского языка, а на Unix-подобных системах — `UTF-8`. Однако, начиная с Java 18, кодировка по умолчанию для всех платформ была изменена на `UTF-8`.

- **Явное указание кодировки** Чтобы явно указать кодировку при работе с файлами, можно использовать соответствующие конструкторы и методы классов ввода-вывода.

Указание кодировки помогает избежать проблем с несовместимостью.

5.4 Readers

- **Reader**

  `Reader` — это абстрактный класс, который используется для чтения потоков символов.

  Он является родительским классом для многих других классов, таких как `FileReader`, `BufferedReader` и `InputStreamReader`.

  Основные методы включают `read()`, `close()`, `mark()`, `reset()` и `skip()`.

```java
import java.io.Reader;
import java.io.StringReader;

public class ReaderExample {
    public static void main(String[] args) {
        String data = "Hello, World!";
        Reader reader = new StringReader(data);
        try {
            int character;
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

  `Reader` сам по себе не может быть использован напрямую, так как он абстрактный. Необходимо использовать его подклассы.

  Важно закрывать Reader после использования, чтобы освободить ресурсы.

- **FileReader**

`FileReader` используется для чтения текстовых файлов. Он является подклассом `InputStreamReader` и наследует все его методы. Предназначен для упрощения чтения файлов.

```java
import java.io.FileReader;
import java.io.IOException;

public class FileReaderExample {
    public static void main(String[] args) {
        try (FileReader reader = new FileReader("example.txt")) {
            int character;
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Использует кодировку по умолчанию для вашей системы.
Если вам нужно использовать другую кодировку,
лучше использовать `InputStreamReader` с указанием кодировки.

`FileReader` бросает `FileNotFoundException`, если файл не существует,
и `IOException` для других ошибок ввода-вывода.

- **BufferedReader**

Используется для более эффективного чтения текста из входного потока символов, буферизуя символы, чтобы обеспечить эффективное считывание строк и массивов символов.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(
            new FileReader("example.txt")
        )) {
            String line;
            while ((line = reader.readLine()) != null) {
```

```
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Значительно улучшает производительность при чтении больших файлов, так как уменьшает количество обращений к диску.

По умолчанию размер буфера составляет 8192 символа, но его можно изменить, указав второй параметр в конструкторе.

- **InputStreamReader**

  `InputStreamReader` преобразует байтовый поток в поток символов, что позволяет читать текстовые данные из байтовых потоков.

  Он является мостом между байтовыми и символьными потоками.

```
import java.io.InputStreamReader;
import java.io.FileInputStream;
import java.io.IOException;

public class InputStreamReaderExample {
    public static void main(String[] args) {
        try (InputStreamReader reader = new InputStreamReader(
            new FileInputStream("example.txt"), "UTF-8"
        )) {
            int character;
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

`InputStreamReader` позволяет указать кодировку, что делает его более гибким по сравнению с `FileReader`.

Важно правильно указывать кодировку

- **InputStream**

  `InputStream` — это абстрактный класс для чтения байтов из потока. Он является базовым классом для всех классов, которые читают байты из потока.

  `InputStream` сам по себе не может быть использован напрямую, так как он абстрактный. Необходимо использовать его подклассы.

5.5 Writers

- **Writer**

  `Writer` — это абстрактный класс в пакете `java.io`, который представляет поток символов. Он является родительским классом для всех классов, которые пишут символы в выходной поток.

  Основные методы включают:

  – `write(int c)`: записывает один символ.

  – `write(char[] cbuf, int off, int len)`: записывает часть массива символов.

  – `write(String str, int off, int len)`: записывает часть строки.

  – `flush()`: очищает поток.

  – `close()`: закрывает поток.

- **FileWriter**

  `FileWriter` — это подкласс `Writer`, который используется для записи текстовых данных в файл. Он использует кодировку символов по умолчанию для платформы, если не указано иное.

  Основные методы:

  1. `FileWriter(String fileName)`: создает объект `FileWriter`, связанный с файлом.
  2. `FileWriter(String fileName, boolean append)`: создает объект FileWriter, связанный с файлом, с возможностью добавления данных в конец файла.
  3. Методы записи аналогичны методам `Writer`.

```
1 FileWriter writer = new FileWriter("output.txt");
```

```
writer.write("Hello, World!");
writer.close();
```

- **BufferedWriter**

  `BufferedWriter` — это подкласс `Writer`, который буферизует символы для повышения эффективности записи. Он оборачивает другой объект `Writer` и добавляет буферизацию.

  Основные методы.

  1. `BufferedWriter(Writer out)`: создает буферизированный поток.
  2. `BufferedWriter(Writer out, int sz)`: создает буферизированный поток с указанным размером буфера.
  3. Методы записи аналогичны методам `Writer`.

```
BufferedWriter bufferedWriter = new BufferedWriter(new
↪  FileWriter("output.txt"));
bufferedWriter.write("Hello, Buffered World!");
bufferedWriter.close();
```

- **OutputStreamWriter**

  `OutputStreamWriter` — это мост между потоками символов и байтов. Он преобразует символы в байты с использованием указанной кодировки.

  Основные методы:

  1. `OutputStreamWriter(OutputStream out)`: создает объект `OutputStreamWriter`, связанный с выходным потоком байтов.
  2. `OutputStreamWriter(OutputStream out, String charsetName)`: создает объект `OutputStreamWriter` с указанной кодировкой.
  3. Методы записи аналогичны методам `Writer`.

```
OutputStreamWriter outputStreamWriter = new OutputStreamWriter(
    new FileOutputStream("output.txt"), "UTF-8"
);
outputStreamWriter.write("Hello, UTF-8 World!");
outputStreamWriter.close();
```

- **OutputStream**

  `OutputStream` — это абстрактный класс, представляющий поток байтов. Он яв-

ляется суперклассом для всех классов, которые пишут байты в выходной поток.

- **PrintWriter**

  PrintWriter — это подкласс Writer, который предоставляет удобные методы для форматированной записи данных в текстовый поток. Он поддерживает автоматическую очистку буфера и может быть обернут вокруг других объектов Writer.

  Основные методы:

  1. PrintWriter(Writer out): создает объект PrintWriter, связанный с другим объектом Writer.
  2. `PrintWriter(OutputStream out)`: создает объект `PrintWriter`, связанный с выходным потоком байтов.
  3. `print(...)`: методы для записи различных типов данных.
  4. `printf(...)`: методы для форматированной записи данных.
  5. `println(...)`: методы для записи данных с новой строки.

```
1 PrintWriter printWriter = new PrintWriter(new FileWriter("output.txt"));
2 printWriter.println("Hello, PrintWriter!");
3 printWriter.close();
```

6 Классы и объекты

6.1 Неизменяемые объекты

Неизменяемый объект — это объект, состояние которого не может измениться после его создания. Такие объекты особенно полезны в многопоточных приложениях, так как они безопасны для использования несколькими потоками без необходимости синхронизации.

- **Как создать неизменяемый объект**

  1. Сделайте все поля `final` и `private`.
  2. Не предоставляйте методы для изменения состояния объекта.
  3. Убедитесь, что класс не может быть расширен (объявите его `final`). Если объект содержит изменяемые поля, убедитесь, что они не могут быть изменены после создания объекта.

```java
public final class ImmutablePerson {
    private final String name;
    private final int age;

    public ImmutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

- **Конструкторы (Constructors)**

  Конструкторы — это специальные методы, которые используются для инициализации объектов класса. Они вызываются автоматически при создании объекта и имеют то же имя, что и класс.

  1. Простой конструктор

```java
public class MyClass {
    public MyClass() {
        // Инициализация
    }
}
```

2. Конструктор с параметрами

```java
public class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }
}
```

Есть возможность сделать несколько конструкторов, принимающих различные типы данных.

- **Методы (Methods)**

Методы — это блоки кода, которые выполняют определенные действия и могут возвращать результат. Они позволяют повторно использовать код и структурировать проет.

1. Объявление метода

```java
public class MyClass {
    public void myMethod() {
        // Действия
    }
}
```

2. Вызов метода

```java
MyClass obj = new MyClass();
obj.myMethod();
```

- **Статические методы (Static Methods)**

Статические методы принадлежат классу, а не экземпляру класса. Они могут быть вызваны без создания объекта. Отличается от обычных методов тем, что статические методы могут обращаться только к статическим полям и методам.

```java
public class MyClass {
    public static void myStaticMethod() {
        // Действия
```

```
    }
}
```

- **@Override**

  В Java есть аннотация `@Override`, необходимая для указания того, что метод в классе переопределяет метод родительского класса.

  Она указывается перед функций, переназначающей другую.

```java
class SuperClass {
    void display() {
        System.out.println("Метод суперкласса");
    }
}

class SubClass extends SuperClass {
    @Override
    void display() {
        System.out.println("Переопределенный метод подкласса");
    }
}
```

6.2 Изменяемые объекты

Изменяемые объекты — это объекты, состояние которых можно изменять после их создания. В отличие от неизменяемых объектов, изменяемые объекты позволяют изменять их внутренние данные в течение их жизненного цикла.

```java
public class MutablePerson {
    private String name;
    private int age;

    public MutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

- **Основные отличия от неизменяемых объектов**

  1. Состояние изменяемых объектов можно изменять после создания, тогда как состояние неизменяемых объектов остается постоянным.
  2. Неизменяемые объекты безопасны для использования в многопоточных приложениях без дополнительной синхронизации, в то время как изменяемые объекты требуют синхронизации для обеспечения потокобезопасности.
  3. Изменяемые объекты могут быть более эффективными в плане производительности, так как они не требуют создания новых объектов при каждом изменении состояния.

- **Инкапсуляция (Encapsulation)**

  Инкапсуляция — это принцип объектно-ориентированного программирования, который заключается в сокрытии внутреннего состояния объекта и предоставлении доступа к нему только через публичные методы. Это помогает защитить данные от некорректного использования и упрощает управление сложностью программы.

  Как реализовать инкапсуляцию:

  1. Сделайте поля класса `private`
  2. Предоставьте публичные методы для доступа к этим полям (геттеры и сеттеры)

```
public class EncapsulatedPerson {
    private String name;
    private int age;

    // Конструктор
```

```
6       public EncapsulatedPerson(String name, int age) {
7           this.name = name;
8           this.age = age;
9       }
10
11      // Геттер для имени
12      public String getName() {
13          return name;
14      }
15
16      // Сеттер для имени
17      public void setName(String name) {
18          this.name = name;
19      }
20
21      // Геттер для возраста
22      public int getAge() {
23          return age;
24      }
25
26      // Сеттер для возраста
27      public void setAge(int age) {
28          this.age = age;
29      }
30  }
```

7 Коллекции

7.1 Общие понятия

- **Интерфейсы**
  Коллекции в `Java` представлены набором интерфейсов в пакете `java.util`, которые определяют основные методы работы с коллекциями данных. Основные интерфейсы:

  1. `Collection<E>`: базовый интерфейс, от которого наследуются другие коллекции, такие как `List`, `Set`, и `Queue`. Определяет методы добавления, удаления элементов, проверки на наличие элемента, получения размера и т.д.
  2. `List<E>`: расширяет `Collection`, представляя упорядоченные коллекции, поддерживающие доступ к элементам по индексу.
  3. `Set<E>`: расширяет `Collection`, представляя коллекции, которые не могут содержать дублирующихся элементов.
  4. `Map<K, V>`: интерфейс, который не наследует `Collection`, но представляет коллекции в виде пар "ключ-значение". Ассоциативные массивы, такие как HashMap, реализуют этот интерфейс.
  5. `Queue<E>`: интерфейс, представляющий коллекции для обработки элементов в порядке очереди.

- **Параметры типов**
  В `Java` коллекции обычно параметризованы, чтобы работать с объектами определенного типа, что реализуется с помощью обобщений (`generics`). Это обеспечивает безопасность типов на этапе компиляции:

```java
List<String> list = new ArrayList<>();
list.add("Hello"); // В list мы можем добавить только объект String
```

  Использование обобщений позволяет избежать ошибок типа и устраняет необходимость приведения типов.

- **Равенство и equals()**
  Метод `equals()` проверяет равенство объектов. Важно переопределить этот метод для объектов, которые будут использоваться в коллекциях, таких как `Set`, для правильного определения уникальности объектов.

- **Хеширование и hashCode()**

Метод `hashCode()` возвращает целочисленное значение, которое используется для хранения объектов в структурах данных, таких как `HashMap` и `HashSet`. Если два объекта равны по методу `equals()`, они должны иметь одинаковый хеш-код. Это необходимо для правильного функционирования коллекций, основанных на хешировании.

7.2 Списки

- **Интерфейс List**
  Определяет методы для работы с упорядоченными коллекциями: доступ к элементам по индексу, добавление и удаление элементов.

- **Класс ArrayList**
  `ArrayList` реализует интерфейс `List` и представляет собой динамически расширяющийся массив. Обеспечивает быстрый доступ к элементам по индексу (за `O(1)`), но вставка и удаление в середине списка требуют сдвига элементов (за `O(n)`).

```java
List<String> arrayList = new ArrayList<>();
arrayList.add("One");
arrayList.add("Two");
arrayList.remove(0);
System.out.println(arrayList.get(0)); // Output: Two
```

- **Класс LinkedList**
  `LinkedList` также реализует интерфейс `List`, но основан на двусвязном списке. Вставка и удаление элементов быстрее, чем у `ArrayList` (за `O(1)`, но доступ по индексу занимает больше времени (за `O(n)`).

- **Добавление и удаление элементов**
  Методы `add()`, `remove()`, `clear()`, `addAll()` позволяют добавлять и удалять элементы, а также работать с подмножествами.

- **Индексированный доступ**
  Метод `get(int index)` возвращает элемент по индексу.
  Метод `set(int index, E element)` изменяет элемент по заданному индексу.

- **Итерация**
  Для перебора элементов можно использовать циклы `for`, `foreach`, или итератор:

```java
for (String s : arrayList) {
    System.out.println(s);
```

```
}

Iterator<String> iterator = linkedList.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

7.3 Множества

- **Интерфейс Set**
  `Set` – это интерфейс, который расширяет `Collection`. Он определяет поведение коллекций, которые не могут содержать дублирующиеся элементы. Основные реализации включают `HashSet`, `LinkedHashSet`, и `TreeSet`. Основные методы интерфейса:
  - `add(E e)`: добавляет элемент в множество. Если такой элемент уже существует, то метод возвращает `false`, иначе — `true`.
  - `remove(Object o)`: удаляет элемент из множества. Возвращает `true`, если элемент был успешно удален.
  - `contains(Object o)`: проверяет, существует ли элемент в множестве. Возвращает `true`, если элемент найден.
  - `size()`: возвращает количество элементов в множестве.
  - `isEmpty()`: проверяет, пусто ли множество.
  - `clear()`: удаляет все элементы из множества.
- **Класс HashSet**
  `HashSet` является наиболее часто используемой реализацией интерфейса `Set`. Этот класс основан на хеш-таблице, что обеспечивает быструю вставку, удаление и проверку на наличие элемента (в среднем за `O(1)`).

  Особенности `HashSet`:
  - Не гарантирует порядок хранения элементов.
  - Поддерживает `null` в качестве элемента.
- **Класс LinkedHashSet**
  `LinkedHashSet` расширяет `HashSet` и сохраняет порядок добавления элементов.

Это означает, что при итерации элементы будут возвращаться в том же порядке, в котором они были добавлены.

Особенности `LinkedHashSet`:

- Основан на хеш-таблице и двусвязном списке.
- Сохраняет порядок вставки элементов.
- Немного медленнее, чем `HashSet`, из-за необходимости поддерживать порядок элементов.

- **Класс TreeSet**
  `TreeSet` реализует интерфейс `NavigableSet`, который, в свою очередь, расширяет `SortedSet`. `TreeSet` поддерживает упорядоченное хранение элементов в соответствии с их естественным порядком или порядком, заданным компаратором.

  Особенности `TreeSet`:

  - Поддерживает упорядоченный порядок элементов.
  - Операции вставки, удаления и поиска выполняются за `O(log n)`.
  - Не допускает `null` в качестве элемента.

- **Добавление и удаление элементов**
  Множества поддерживают основные операции добавления и удаления элементов, но, в отличие от списков, они не имеют индексированного доступа к элементам. Операции добавления и удаления работают следующим образом:

  - Метод `add(E e)` добавляет элемент в множество, если он еще не присутствует.
  - Метод `remove(Object o)` удаляет элемент, если он присутствует.
  - Метод `clear()` очищает множество, удаляя все элементы.

- **Итерация**
  Перебор элементов можно осуществлять с помощью циклов и итераторов, как в случае с `List`.

7.4 Отображения

Отображения представляют собой структуры данных, где каждый элемент хранится в виде пары "ключ-значение". Интерфейс `Map<K, V>` и его реализации обеспечивают возможности хранения данных, поиска, вставки и удаления на основе ключей.

- **Интерфейс Map** Интерфейс `Map<K, V>` определяет основные методы для работы с отображениями:
  - `put(K key, V value)`: добавляет пару "ключ-значение"в отображение. Если такой ключ уже существует, то старое значение будет заменено на новое.
  - `get(Object key)`: возвращает значение, связанное с ключом, или `null`, если ключ не найден.
  - `remove(Object key)`: удаляет пару по ключу.
  - `containsKey(Object key)` и `containsValue(Object value)`: проверяют наличие ключа или значения в отображении.
  - `keySet()`: возвращает `Set` всех ключей.
  - `values()`: возвращает коллекцию всех значений.
  - `entrySet()`: возвращает набор всех пар "ключ-значение"`(Map.Entry<K, V>)`, который можно использовать для итерации по отображению.
- **Класс HashMap**
  `HashMap` — это реализация интерфейса `Map`, основанная на хеш-таблице. Она обеспечивает быструю вставку, удаление и доступ к элементам (в среднем `O(1)`).
  - Порядок хранения элементов не гарантируется.
  - Может содержать `null` как ключи, так и значения.
- **Класс LinkedHashMap**
  `LinkedHashMap` расширяет HashMap и сохраняет порядок добавления элементов. Это полезно, когда важен порядок перебора.
- **Класс TreeMap**
  TreeMap реализует интерфейс `NavigableMap`, что обеспечивает хранение элементов в отсортированном порядке по ключу.
  - Поддерживает естественный порядок ключей или порядок, заданный компаратором.

– Не допускает `null` в качестве ключей (значения могут быть `null`).

- **Отображения как ассоциативные массивы**
  Отображения в Java часто сравнивают с ассоциативными массивами, так как они позволяют хранить данные в виде пар "ключ-значение"и быстро получать доступ к значению по ключу. Ассоциативные массивы особенно полезны для задач поиска, индексации и хранения уникальных данных.

7.5 Упорядоченные колекции

Упорядоченные коллекции в `Java` — это коллекции, в которых элементы могут быть отсортированы или доступны в определенном порядке. Это достигается с помощью интерфейсов `Comparable` и `Comparator`, а также реализаций `NavigableSet` и `NavigableMap`.

- **Сравнение и compareTo()**
  Интерфейс `Comparable<T>` используется для задания естественного порядка объектов. Метод `compareTo(T o)` сравнивает текущий объект с указанным объектом и возвращает:

  – Отрицательное число, если текущий объект меньше.

  – Ноль, если объекты равны.

  – Положительное число, если текущий объект больше.

  Пример реализации:

```java
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }
}
```

- **Компараторы**
  Интерфейс `Comparator<T>` позволяет определить порядок объектов, отличный от естественного. Для этого реализуется метод `compare(T o1, T o2)`.

  Пример использования:

```
Comparator<Person> nameComparator = (p1, p2) ->
 ↪  p1.getName().compareTo(p2.getName());
Comparator<Person> ageComparator = (p1, p2) -> Integer.compare(p1.getAge(),
 ↪  p2.getAge());
```

8 Наследование

Наследование (Inheritance) — один из ключевых принципов объектно-ориентированного программирования, который позволяет создать новые классы на основе существующих. Это способствует повторному использованию кода, расширению функциональности и упрощению сопровождения программного обеспечения.

8.1 Интерфейсы

- **Определение**
  Интерфейс в `Java` — это контракт, который определяет набор методов, которые класс должен реализовать. В интерфейсах методы не имеют реализации, а только сигнатуры. Интерфейсы позволяют определить общие функциональные возможности для разных классов, даже если они не связаны друг с другом через классическое наследование.

- **Применение**
  Интерфейсы применяются, когда нужно обеспечить некоторый общий функционал для различных классов. Например, в случае, когда классы не связаны иерархией наследования, но имеют общую функциональность. Интерфейсы широко используются в ситуациях, когда необходимо поддерживать полиморфизм.

  Объявление интерфейса:

```java
public interface Drawable {
    void draw();
}
```

  Здесь интерфейс `Drawable` определяет один метод `draw()`, который должен быть реализован любым классом, реализующим этот интерфейс.

- **Реализация**
  Класс реализует интерфейс с помощью ключевого слова `implements`, и он может реализовывать сразу несколько интерфейсов. Это позволяет создавать классы с более гибкими и многообразными функциональными возможностями.

  Пример:

```java
public interface Flyable {
    void fly();
}

```

```java
public interface Swimmable {
    void swim();
}

public class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }

    @Override
    public void swim() {
        System.out.println("Duck is swimming");
    }
}
```

В этом примере класс `Duck` реализует два интерфейса — `Flyable` и `Swimmable`, предоставляя собственные реализации для методов `fly()` и `swim()`.

- **Наследование интерфейсов**
  Интерфейсы в `Java` могут наследовать друг друга с помощью ключевого слова `extends`. При этом один интерфейс может расширять сразу несколько интерфейсов (в отличие от классов, которые могут наследовать только один класс). Это позволяет создавать более сложные иерархии интерфейсов и задавать различные уровни абстракции.

  Пример:

```java
public interface Movable {
    void move();
}

public interface Flyable extends Movable {
    void fly();
}

public interface Swimmable extends Movable {
    void swim();
}
```

В данном примере интерфейс `Flyable` расширяет интерфейс `Movable`, добавляя метод `fly()`, а интерфейс `Swimmable также расширяет Movable`, добавляя метод `swim()`. Таким образом, любой класс, реализующий `Flyable` или `Swimmable`,

также должен реализовать метод `move()`, так как это требование было определено в базовом интерфейсе `Movable`.

8.2 Наследование

- **Синтаксис**
  В `Java` наследование реализуется с помощью ключевого слова `extends`. Новый класс, называемый подклассом (или производным), расширяет функциональность существующего класса, называемого суперклассом (или базовым). Подкласс наследует все публичные и защищённые члены суперкласса, но может переопределять методы для предоставления своей реализации.

- **Применение**
  Наследование используется для создания иерархии классов, где более общие классы находятся ближе к вершине, а более специфичные — ниже. Это позволяет создавать гибкие и масштабируемые архитектуры программного обеспечения.

  Пример:

```java
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
```

  В данном примере класс `Dog` наследует класс `Animal` и получает его метод `eat()`, а также добавляет свой метод `bark()`.

- **Модификатор protected**
  Модификатор доступа `protected` предоставляет доступ к членам класса в пределах того же пакета и в подклассах, даже если они находятся в другом пакете. Это полезно при необходимости делиться внутренними деталями реализации с подклассами, но не делать их доступными из других частей программы.

8.3 Пакеты

- **Синтаксис**
  Пакеты — это механизм группировки классов, интерфейсов и других пакетов в логически связанные единицы. Они помогают структурировать код, избегать конфликтов имён и управлять доступом к классам. Для создания пакета используется ключевое слово `package`.

  Пример объявления пакета:

```
1 package com.example.myapp;
```

  Эта строка должна быть первой в файле, чтобы указать, что класс находится в пакете `com.example.myapp`.

- **Применение**
  Пакеты используются для организации кода и предотвращения конфликтов имён, так как классы с одинаковыми именами могут находиться в разных пакетах. Кроме того, пакеты управляют доступом.

  Для использования классов из других пакетов необходимо импортировать их с помощью ключевого слова `import`:

```
1 import com.example.myapp.MyClass;
```

  Также можно использовать `import com.example.myapp.*;`, чтобы импортировать все классы из пакета.

- **Файл package-info.java**
  Файл `package-info.java` — это специальный файл, который используется для документирования пакетов в `Java`. Он может содержать аннотации и комментарии, относящиеся ко всему пакету. Этот файл служит для улучшения организации кода и его документирования, помогая лучше понимать назначение пакета.

9 Синтаксис Java

СинтаксисJava — это набор правил, которые определяют, как должны быть структурированы программы на Java. Синтаксис включает в себя различные элементы, такие как типы данных, операции, операторы, структуру исходного кода и многое другое.

9.1 Типы данных

Типы данных в Java можно разделить на две основные категории: примитивные типы и ссылочные (объектные) типы.

9.1.1 Примитивные типы данных

Примитивные типы данных являются базовыми элементами, используемыми для хранения простых значений, таких как числа и символы. В Java существует 8 примитивных типов:

- **Целочисленные типы**:
  - `byte` (8 бит): хранит значения от -128 до 127.
  - `short` (16 бит): хранит значения от $-32,768$ до $32,767$.
  - `int` (32 бита): хранит значения от -2^{31} до 2^{31-1}.
  - `long` (64 бита): хранит значения от -2^{63} до 2^{63-1}.
- **Числа с плавающей точкой**:
  - `float` (32 бита): используется для хранения чисел с плавающей точкой одинарной точности.
  - `double` (64 бита): используется для хранения чисел с плавающей точкой двойной точности.
- **Символьный тип**:
  - `char` (16 бит): хранит один символ в кодировке `UTF-16`.
- **Логический тип**:
  - `boolean`: может принимать значения `true` или `false`.

- **Обёртки примитивных типов**:
  `Java` предоставляет классы-обёртки для каждого примитивного типа, которые позволяют работать с примитивами как с объектами. Например, для типа `int` существует класс `Integer`, для типа `double` — класс `Double` и т. д. Это полезно, когда нужно использовать примитивные типы в коллекциях, так как коллекции работают только с объектами.

9.1.2 Массивы

Массивы представляют собой структуру данных, которая хранит элементы одного типа в фиксированном размере.

- **Массивы как объекты**
  Массивы в Java являются объектами, и для них можно вызвать методы класса `Object`. Например, можно использовать `array.length` для получения длины массива.

- **Ковариантность**
  Массивы в `Java` ковариантны, то есть массив типа `SuperType[]` может ссылаться на массив типа `SubType[]`, если `SubType` является подклассом `SuperType`.

- **Reification (Восстановление типов)**
  В `Java` массивы сохраняют информацию о типе своих элементов во время выполнения (`reification`). Это означает, что `Java` может проверять тип элементов массива в рантайме.

9.2 Операции

Операции в `Java` — это действия, которые могут выполняться над операндами. Операторы, используемые в этих операциях, имеют определённые приоритеты и ассоциативность.

9.2.1 Приоритеты

Приоритет операторов определяет порядок их выполнения в выражении. Операторы с более высоким приоритетом выполняются раньше операторов с более низким приоритетом.

9.2.2 Ассоциативность

Ассоциативность определяет порядок выполнения операторов с одинаковым приоритетом: слева направо или справа налево.

9.3 Структура исходного кода

Программа на Java состоит из классов и методов, которые могут содержать поля, конструкторы и блоки инициализации.

9.3.1 Заголовок

Каждый файл начинается с заголовка, включающего объявление пакета и импорт необходимых классов.

```
1 package com.example;
2
3 import java.util.List;
```

9.3.2 Классы

Классы — это основные строительные блоки в `Java`. Они могут содержать поля, методы, конструкторы и внутренние классы.

```
1 public class MyClass {
2     // Поля, методы и конструкторы
3 }
```

9.3.3 Интерфейсы

Интерфейсы определяют контракты для реализации другими классами.

```
1 public interface MyInterface {
2     void doSomething();
3 }
```

9.3.4 Поля

Поля (переменные класса) используются для хранения состояния объекта.

```
1 public class MyClass {
2     private int value;
3 }
```

9.3.5 Конструкторы

Конструкторы — это методы, которые вызываются при создании объекта.

```
public MyClass(int value) {
    this.value = value;
}
```

9.3.6 Методы

Методы определяют поведение объекта.

```
public void display() {
    System.out.println("Value: " + value);
}
```

10 Дизайн ООП

10.1 SOLID

Принципы SOLID — это набор из пяти основных принципов объектно-ориентированного программирования и проектирования, которые помогают разработчикам создавать более гибкие, расширяемые и поддерживаемые системы.

10.1.1 SRP: Принцип единственной ответственности (Single Responsibility Principle)

Определение: Каждый класс должен иметь одну и только одну причину для изменения, то есть класс должен иметь только одну ответственность.

Пояснение: Если класс выполняет несколько задач, изменения в одной части могут повлиять на другие. Это усложняет тестирование и поддержку кода. Принцип SRP призывает разделять функциональность по классам таким образом, чтобы каждый класс отвечал за свою конкретную часть логики.

Пример:

Нарушение SRP:

```
public class Employee {
    private String name;
    private double salary;

    public void calculateSalary() {
        // Логика расчета зарплаты
    }

    public void saveToDatabase() {
        // Логика сохранения в базу данных
    }
}
```

В этом примере класс Employee отвечает и за бизнес-логику (расчет зарплаты), и за сохранение данных, что нарушает SRP.

Следование SRP:

```
public class Employee {
    private String name;
    private double salary;
```

```
4
5       public void calculateSalary() {
6           // Логика расчета зарплаты
7       }
8   }
9
10  public class EmployeeRepository {
11      public void save(Employee employee) {
12          // Логика сохранения сотрудника в базу данных
13      }
14  }
```

Теперь каждый класс имеет свою единственную ответственность: `Employee` — бизнес-логика сотрудника, `EmployeeRepository` — сохранение данных.

10.1.2 OCP: Принцип открытости/закрытости (Open/Closed Principle)

Определение: Программные сущности должны быть **открыты для расширения**, но **закрыты для модификации**.

Пояснение: Новый функционал должен добавляться путем создания нового кода, а не изменения существующего. Это снижает риск внесения ошибок в уже проверенный код.

Пример:

Нарушение OCP:

```
1   public class ShapeDrawer {
2       public void drawShape(Object shape) {
3           if (shape instanceof Circle) {
4               drawCircle((Circle) shape);
5           } else if (shape instanceof Square) {
6               drawSquare((Square) shape);
7           }
8           // При добавлении новой фигуры потребуется изменить этот метод
9       }
10
11      private void drawCircle(Circle circle) {
12          // Логика рисования круга
13      }
14
15      private void drawSquare(Square square) {
16          // Логика рисования квадрата
```

```
    }
}
```

Следование OCP:

```
public interface Shape {
    void draw();
}

public class Circle implements Shape {
    @Override
    public void draw() {
        // Логика рисования круга
    }
}

public class Square implements Shape {
    @Override
    public void draw() {
        // Логика рисования квадрата
    }
}

public class ShapeDrawer {
    public void drawShape(Shape shape) {
        shape.draw();
    }
}
```

Теперь для добавления новой фигуры достаточно создать новый класс, реализующий `Shape`, не изменяя существующий код.

10.1.3 LSP: Принцип подстановки Лисков (Liskov Substitution Principle)

Определение: Объекты базового класса должны быть заменяемы объектами производных классов без нарушения работы программы.

Пояснение: Это означает, что подкласс должен дополнять, а не изменять поведение базового класса. Нарушение LSP приводит к неожиданным ошибкам при использовании полиморфизма.

Пример:

Проблема с прямоугольником и квадратом:

```java
public class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

public class Square extends Rectangle {
    @Override
    public void setWidth(int size) {
        this.width = size;
        this.height = size;
    }

    @Override
    public void setHeight(int size) {
        this.width = size;
        this.height = size;
    }
}
```

Используя полиморфизм, ожидаем, что `Rectangle` и `Square` будут работать одинаково, но изменение ширины или высоты `Square` изменяет обе стороны, что может привести к непредвиденным последствиям.

10.1.4 ISP: Принцип разделения интерфейса (Interface Segregation Principle)

Определение: Клиенты не должны зависеть от методов, которые они не используют.

Пояснение: Большие интерфейсы следует разделять на более маленькие и специфичные, чтобы клиенты зависели только от тех методов, которые им действительно нужны.

Пример:

Нарушение ISP:

```java
public interface Worker {
    void work();
    void eat();
}

public class Robot implements Worker {
    @Override
    public void work() {
        // Робот работает
    }

    @Override
    public void eat() {
        // Роботу не нужно есть, но он обязан реализовать этот метод
    }
}
```

Следование ISP:

```java
public interface Workable {
    void work();
}

public interface Eatable {
    void eat();
}

public class Human implements Workable, Eatable {
    @Override
    public void work() {
        // Человек работает
    }

    @Override
    public void eat() {
        // Человек ест
    }
}

public class Robot implements Workable {
    @Override
```

```
    public void work() {
        // Робот работает
    }
}
```

Теперь `Robot` не обязан реализовывать метод `eat()`.

10.1.5 DIP: Принцип инверсии зависимостей (Dependency Inversion Principle)

Определение: Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций.

Пояснение: Код должен зависеть от абстракций (интерфейсов), а не от конкретных реализаций. Это повышает модульность и упрощает тестирование.

Пример:

Нарушение DIP:

```
public class MySQLDatabase {
    public void connect() {
        // Подключение к MySQL
    }
}

public class Application {
    private MySQLDatabase database;

    public Application() {
        database = new MySQLDatabase();
    }

    public void start() {
        database.connect();
    }
}
```

Следование DIP:

```
public interface Database {
    void connect();
}

public class MySQLDatabase implements Database {
    @Override
```

```
    public void connect() {
        // Подключение к MySQL
    }
}

public class PostgreSQLDatabase implements Database {
    @Override
    public void connect() {
        // Подключение к PostgreSQL
    }
}

public class Application {
    private Database database;

    public Application(Database database) {
        this.database = database;
    }

    public void start() {
        database.connect();
    }
}
```

Теперь `Application` зависит от абстракции `Database`, и мы можем подставить любую реализацию.

10.2 Квадрат и прямоугольник

10.2.1 Постановка задачи

Рассмотреть, как правильно спроектировать отношения между классами `Rectangle` (прямоугольник) и `Square` (квадрат) в соответствии с принципами ООП, особенно с учетом принципа подстановки Лисков.

10.2.2 Источник проблем

1. **Наследование:** С математической точки зрения, квадрат является частным случаем прямоугольника.
2. **Принцип подстановки Лисков:** Подкласс должен полностью соответствовать поведению базового класса.

3. **Изменение свойств:** У квадрата ширина и высота всегда равны, что нарушает логику изменения отдельных свойств `width` и `height` в `Rectangle`.

10.2.3 Возможные решения

1. **Отказ от изменений**

   (a) **Возврат нового значения**

   Вместо изменения состояния объекта, методы возвращают новый объект с измененным состоянием.

```java
public class ImmutableRectangle {
    private final int width;
    private final int height;

    public ImmutableRectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public ImmutableRectangle setWidth(int width) {
        return new ImmutableRectangle(width, this.height);
    }

    public ImmutableRectangle setHeight(int height) {
        return new ImmutableRectangle(this.width, height);
    }
}
```

   (b) **Возврат флага**

   Методы изменения возвращают булевый флаг, указывающий на успешность операции.

```java
public class Square extends Rectangle {
    @Override
    public boolean setWidth(int width) {
        if (width == this.height) {
            this.width = width;
            return true;
        }
        return false;
    }
}
```

(c) Исключения

Бросать исключение при попытке установить некорректное значение.

```
public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        if (width != this.height) {
            throw new IllegalArgumentException("Width and height must be equal");
        }
        this.width = width;
    }
}
```

2. Отказ от наследования

(a) Полный отказ от наследования

`Square` и `Rectangle` не связаны отношением наследования.

```
public class Rectangle {
    private int width;
    private int height;
    // Геттеры и сеттеры
}

public class Square {
    private int side;
    // Геттеры и сеттеры
}
```

(b) Выделение общего базового класса

Создать абстрактный класс или интерфейс `Quadrilateral` (четырехугольник).

```
public abstract class Quadrilateral {
    public abstract int getArea();
}

public class Rectangle extends Quadrilateral {
    private int width;
    private int height;
    // Реализация методов
}

```

```
        public class Square extends Quadrilateral {
            private int side;
            // Реализация методов
        }
```

3. **Дополнительные действия**

   Изменение логики методов базового класса для учета особенностей подклассов.

4. **Выделение модифицируемых сущностей**

   Разделить объекты на изменяемые и неизменяемые, применяя соответствующие подходы к каждому.

5. **Отказ от квадратов**

   Не создавать отдельный класс `Square`, а управлять этим внутри класса `Rectangle`.

```
    public class Rectangle {
        private int width;
        private int height;

        public static Rectangle createSquare(int side) {
            return new Rectangle(side, side);
        }
    }
```

10.3 Равенство

10.3.1 Свойства равенства

Метод `equals` должен удовлетворять следующим свойствам:

1. **Рефлексивность:** Для любого ненулевого значения `x`, `x.equals(x)` должно возвращать `true`.
2. **Симметричность:** Для любых ненулевых значений `x` и `y`, `x.equals(y)` должно возвращать `true` тогда и только тогда, когда `y.equals(x)` возвращает `true`.
3. **Транзитивность:** Если `x.equals(y)` и `y.equals(z)` возвращают `true`, то `x.equals(z)` должно возвращать `true`.
4. **Согласованность:** Многократные вызовы `x.equals(y)` должны возвращать одинаковый результат, если объекты не изменились.
5. **Неравенство с `null`:** Для любого ненулевого значения `x`, `x.equals(null)`

должно возвращать `false`.

10.3.2 Метод equals

Переопределение `equals`:

```java
public class Person {
    private String name;
    private int age;

    // Конструктор, геттеры и сеттеры

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return age == person.age &&
               Objects.equals(name, person.name);
    }
}
```

Пояснение:

1. Проверяем, ссылаются ли объекты на одну область памяти.
2. Проверяем, не является ли объект `null` и совпадают ли классы.
3. Приводим объект и сравниваем поля.

10.3.3 Метод hashCode

Переопределение `hashCode`:

```java
@Override
public int hashCode() {
    return Objects.hash(name, age);
}
```

Пояснение:

1. Метод `hashCode` должен возвращать одинаковое значение для объектов, которые равны по `equals`.
2. Это важно для корректной работы хеш-таблиц, таких как `HashMap` и `HashSet`.

10.3.4 Взаимодействие с наследованием

1. **Наивная реализация**

   При переопределении `equals` в подклассе могут возникнуть проблемы с симметричностью и транзитивностью.

   Пример проблемы:

```
    public class Point {
        protected int x, y;

        // Переопределение equals и hashCode
    }

    public class ColorPoint extends Point {
        private String color;

        // Переопределение equals и hashCode
    }
```

   Если `ColorPoint` сравнивается с `Point`, могут возникнуть несоответствия.

2. **Использование сравнения предка**

   При переопределении `equals` в подклассе следует использовать `super.equals(obj)` для сравнения полей суперкласса.

   Пример:

```
    @Override
    public boolean equals(Object obj) {
        if (!super.equals(obj)) return false;
        if (getClass() != obj.getClass()) return false;
        ColorPoint that = (ColorPoint) obj;
        return Objects.equals(color, that.color);
    }
```

3. **Сегрегация сравнения**

   Использование метода `canEqual` для проверки совместимости классов.

   Пример:

```
    public class Point {
        // Поля, конструкторы

```

```
4           @Override
5           public boolean equals(Object obj) {
6               if (this == obj) return true;
7               if (!(obj instanceof Point)) return false;
8               Point point = (Point) obj;
9               return point.canEqual(this) && x == point.x && y == point.y;
10          }
11
12          public boolean canEqual(Object obj) {
13              return obj instanceof Point;
14          }
15      }
16
17      public class ColorPoint extends Point {
18          private String color;
19
20          // Конструкторы
21
22          @Override
23          public boolean equals(Object obj) {
24              if (this == obj) return true;
25              if (!(obj instanceof ColorPoint)) return false;
26              if (!super.equals(obj)) return false;
27              ColorPoint that = (ColorPoint) obj;
28              return that.canEqual(this) && Objects.equals(color, that.color);
29          }
30
31          @Override
32          public boolean canEqual(Object obj) {
33              return obj instanceof ColorPoint;
34          }
35      }
```

Пояснение:

(a) Метод `canEqual` используется для проверки, является ли объект экземпляром текущего класса.

(b) Это помогает сохранить симметричность и транзитивность при наследовании.