Java и его приложения

1. Характеристики простых типов данных. Операции, выражения, правила приведения типов.

Простые типы данных в Java

Java предоставляет восемь примитивных типов данных:

Тип данных	Размер	Диапазон значений
`byte`	8 бит	от -128 до 127
`short`	16 бит	от -32,768 до 32,767
`int`	32 бит	от -2^31 до 2^31-1
`long`	64 бит	от -2^63 до 2^63-1
`float`	32 бит	от ±1.4E-45 до ±3.4028235E+38
`double`	64 бит	от ±4.9E-324 до ±1.7976931348623157E+308
`char`	16 бит	от '\u0000' (0) до '\uffff' (65,535)
`boolean`	1 бит	`true` или `false`

Операции в Java

Арифметические операции

- `+` (сложение)
- `-` (вычитание)
- `*` (умножение)
- `/` (деление)
- `%` (остаток от деления)
- `++` (инкремент)
- `--` (декремент)

Операции сравнения

- `==` (равно)
- `!=` (не равно)
- `>` (больше)
- `<` (меньше)
- `>=` (больше или равно)
- `<=` (меньше или равно)

Логические операции

- `&&` (логическое И)
- `||` (логическое ИЛИ)
- `!` (логическое НЕ)
- `&` (побитовое И)
- `|` (побитовое ИЛИ)
- `^` (побитовое исключающее ИЛИ)
- `~` (побитовое НЕ)
- `<<` (сдвиг влево)
- `>>` (сдвиг вправо с сохранением знака)
- `>>>` (сдвиг вправо с заполнением нулями)

Операции присваивания

- `=` (простое присваивание)
- `+=`, `-=`, `*=`, `/=`, `%=` (составное присваивание)
- `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=` (составное битовое присваивание)

Выражения в Java

Выражение — это комбинация операндов (переменных, констант, вызовов методов) и операторов, которая вычисляется для получения значения определенного типа.

Типы выражений:

- Арифметические выражения
- Логические выражения
- Строковые выражения (конкатенация)
- Выражения присваивания

Правила приведения типов

Автоматическое приведение (расширение типа)

Выполняется компилятором автоматически, когда меньший тип преобразуется в больший:

```
byte → short → int → long → float → double
       char → int
```

Пример:

```
int i = 100;
long l = i;     // Автоматическое приведение int к long
float f = l;    // Автоматическое приведение long к float
double d = f;   // Автоматическое приведение float к double
```

Явное приведение (сужение типа)

Требует явного указания с помощью операции приведения типа (casting), когда больший тип преобразуется в меньший:

```
double → float → long → int → short → byte
                        int → char
```

Пример:

```
double d = 100.04;
long l = (long)d;     // Явное приведение double к long
int i = (int)l;       // Явное приведение long к int
char c = (char)i;     // Явное приведение int к char
byte b = (byte)i;     // Явное приведение int к byte
```

Особые случаи приведения типов:

1. **Приведение для числовых литералов**:

```
byte b = 100;  // Компилятор знает, что 100 помещается в byte
byte c = 200;  // Ошибка компиляции - 200 не помещается в byte
```

2. **Приведение в выражениях**:
   - Все операнды типа `byte`, `short` или `char` автоматически повышаются до `int`
   - Если один из операндов имеет тип `long`, весь результат будет типа `long`
   - Если один из операндов имеет тип `float`, весь результат будет типа `float`
   - Если один из операндов имеет тип `double`, весь результат будет типа `double`
3. **Приведение между примитивными и ссылочными типами**:
   - Автобоксинг — автоматическое преобразование примитивных типов в соответствующие им объекты-обертки
   - Автораспаковка — автоматическое преобразование объектов-оберток в примитивные типы

```
Integer boxedInt = 42;  // Автобоксинг
int primitiveInt = boxedInt;  // Автораспаковка
```

4. **Приведение строк к примитивным типам**:

```
int i = Integer.parseInt("123");
double d = Double.parseDouble("123.45");
```

2. Операторы. Блок операторов. Управляющие операторы. Операторы перехода.

Операторы в Java

Оператор — это наименьшая независимая единица в языке Java, которая указывает компьютеру выполнить конкретное действие. Операторы в Java заканчиваются точкой с запятой (`;`).

Основные типы операторов:

- Операторы объявления
- Операторы присваивания
- Операторы управления потоком
- Операторы перехода
- Вызов метода

Блок операторов

Блок операторов — это группа операторов, заключенных в фигурные скобки `{}`. Блоки определяют область видимости переменных и могут содержать вложенные блоки.

```
{
    int x = 10;
    {
        int y = 20;
        System.out.println(x + y);
    }
    // y здесь недоступен
    System.out.println(x);
}
// x здесь недоступен
```

Управляющие операторы

Управляющие операторы предназначены для управления потоком выполнения программы на основе различных условий.

1. Условные операторы

Оператор if-else

```
if (условие) {
    // код выполняется, если условие истинно
} else {
    // код выполняется, если условие ложно
}
```

Оператор if-else if-else

```
if (условие1) {
    // код выполняется, если условие1 истинно
} else if (условие2) {
    // код выполняется, если условие1 ложно и условие2 истинно
} else {
    // код выполняется, если все условия ложны
}
```

Тернарный оператор

```
результат = условие ? значение1 : значение2;
```

Оператор switch

```
switch (выражение) {
    case значение1:
        // код, если выражение == значение1
        break;
    case значение2:
        // код, если выражение == значение2
        break;
    default:
        // код, если нет совпадений
}
```

С Java 12 появился улучшенный switch с возвращаемым значением:

```
String result = switch (day) {
    case "MONDAY", "FRIDAY" -> "Work day";
    case "SATURDAY", "SUNDAY" -> "Weekend";
    default -> "Mid-week";
};
```

2. Операторы цикла

Цикл for

```
for (инициализация; условие; инкремент/декремент) {
    // тело цикла
}
```

Цикл for-each (расширенный for)

```
for (тип переменная : коллекция) {
    // тело цикла
}
```

Цикл while

```
while (условие) {
    // тело цикла
}
```

Цикл do-while

```
do {
    // тело цикла
} while (условие);
```

Операторы перехода

Операторы перехода изменяют стандартный порядок выполнения операторов.

1. Оператор break

Используется для выхода из циклов (`for`, `while`, `do-while`) или оператора `switch`.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // выход из цикла при i == 5
    }
    System.out.println(i);
}
```

Оператор `break` с меткой позволяет выйти из вложенных циклов:

```
outerLoop: for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        if (i * j > 10) {
            break outerLoop; // выход из обоих циклов
        }
        System.out.println(i + " * " + j + " = " + (i * j));
    }
}
```

2. Оператор continue

Используется для перехода к следующей итерации цикла, пропуская оставшиеся операторы в текущей итерации.

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue; // пропустить четные числа
    }
    System.out.println(i); // выводятся только нечетные числа
}
```

Оператор `continue` с меткой позволяет перейти к следующей итерации внешнего цикла:

```
outerLoop: for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (j > i) {
            continue outerLoop; // переход к следующей итерации
внешнего цикла
        }
        System.out.println(i + " " + j);
    }
}
```

3. Оператор return

Используется для выхода из метода и опционально возвращает значение вызывающему коду.

```
public int sum(int a, int b) {
    return a + b; // возвращает сумму и выходит из метода
}

public void printMessage() {
    System.out.println("Hello");
    return; // выход из метода без возврата значения
    // код после return недостижим
}
```

3. Массивы в языке Java. Массив как параметр и тип возвращаемого значения метода. Аргументы метода main().

Массивы в языке Java

Массив — это структура данных, которая хранит фиксированное количество элементов одного типа. В Java массивы являются объектами, которые неявно наследуются от класса `Object` и реализуют интерфейсы `Cloneable` и `Serializable`.

Создание массивов

1. Объявление массива:

```
тип[] имяМассива;     // предпочтительный стиль
тип имяМассива[];     // допустимый стиль
```

2. Выделение памяти:

```
имяМассива = new тип[размер];
```

3. Объявление и выделение памяти в одном операторе:

```
тип[] имяМассива = new тип[размер];
```

4. Инициализация массива при создании:

```
тип[] имяМассива = {значение1, значение2, ..., значениеN};
тип[] имяМассива = new тип[]{значение1, значение2, ...,
значениеN};
```

Доступ к элементам массива

Доступ к элементам массива осуществляется через индекс в квадратных скобках. Индексы начинаются с 0:

```
int[] numbers = {10, 20, 30, 40, 50};
int firstElement = numbers[0];    // 10
int thirdElement = numbers[2];    // 30
numbers[1] = 25;                  // изменение значения второго
элемента
```

Длина массива

Длина массива доступна через поле `length`:

```
int length = numbers.length;    // получение длины массива
```

Многомерные массивы

Java поддерживает многомерные массивы, которые реализуются как массивы массивов:

```
// Двумерный массив
int[][] matrix = new int[3][4];     // 3 строки, 4 столбца

// Инициализация двумерного массива
int[][] matrix = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

// Доступ к элементам
int element = matrix[1][2];     // 7

// Массивы переменной длины
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[4];
jagged[2] = new int[3];
```

Массив как параметр метода

Массив можно передавать как параметр метода:

```
public void processArray(int[] array) {
    for (int element : array) {
        System.out.println(element);
    }
}

// Вызов метода
int[] numbers = {1, 2, 3, 4, 5};
processArray(numbers);
```

В Java массивы передаются по ссылке, а не по значению. Это означает, что изменения, внесенные в массив внутри метода, будут видны и за пределами метода:

```
public void modifyArray(int[] array) {
    array[0] = 100;     // изменение первого элемента
}

int[] numbers = {1, 2, 3, 4, 5};
System.out.println(numbers[0]);     // 1
modifyArray(numbers);
System.out.println(numbers[0]);     // 100 - значение изменилось
```

Массив как тип возвращаемого значения метода

Метод может возвращать массив:

```java
public int[] createArray(int size) {
    int[] newArray = new int[size];
    for (int i = 0; i < size; i++) {
        newArray[i] = i * 10;
    }
    return newArray;
}

// Вызов метода
int[] myArray = createArray(5);    // [0, 10, 20, 30, 40]
```

Аргументы метода main()

Метод `main()` является точкой входа в Java-приложение и имеет следующую сигнатуру:

```java
public static void main(String[] args)
```

Параметр `args` — это массив строк, который содержит аргументы командной строки, переданные при запуске программы:

```java
public class CommandLineArguments {
    public static void main(String[] args) {
        System.out.println("Количество аргументов: " + args.length);

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

При запуске программы аргументы передаются через пробел:

```
java CommandLineArguments arg1 arg2 arg3
```

Вывод:

```
Количество аргументов: 3
Аргумент 0: arg1
Аргумент 1: arg2
Аргумент 2: arg3
```

Особенности аргументов `main()`:

1. Если аргументы не переданы, массив `args` будет пустым (длина 0), но не `null`
2. Аргументы всегда имеют тип `String`, при необходимости их нужно преобразовать в другие типы

3. Аргументы, содержащие пробелы, необходимо заключать в кавычки
4. Для доступа к именованным опциям (например, `-file filename.txt`) нужно самостоятельно парсить массив `args`

4. Классы в языке Java. Компоненты класса: данные и методы. Конструкторы. Ссылка this. Перегрузка методов. Final-компоненты. Статические компоненты класса. Операция «сборка мусора».

Компоненты класса: данные и методы

Класс в Java – это шаблон для создания объектов, который определяет структуру данных и поведение, которое эти данные будут иметь.

Основные компоненты класса:

Поля (данные)

- Переменные, определенные в классе
- Представляют состояние объекта
- Могут быть примитивного или ссылочного типа
- Имеют модификаторы доступа (public, private, protected, default)

```
public class Person {
    private String name;        // поле с модификатором private
    private int age;            // поле с модификатором private
    public String country;      // поле с модификатором public
}
```

Методы

- Функции, определенные в классе
- Определяют поведение объекта
- Могут получать параметры и возвращать значения
- Имеют модификаторы доступа

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

    // Метод для получения имени
    public String getName() {
        return name;
    }

    // Метод для установки имени
    public void setName(String name) {
        this.name = name;
    }

    // Метод для получения возраста
    public int getAge() {
        return age;
    }

    // Метод для установки возраста
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}
```

Конструкторы

Конструктор – это специальный метод, который вызывается при создании объекта класса. Он используется для инициализации полей объекта.

Особенности конструкторов:

- Имя конструктора совпадает с именем класса
- Конструктор не имеет возвращаемого типа (даже void)
- Если в классе не определен ни один конструктор, компилятор создает конструктор по умолчанию без параметров
- Класс может иметь несколько конструкторов с разными параметрами (перегрузка)

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

    // Конструктор по умолчанию
    public Person() {
        name = "Unknown";
        age = 0;
    }

    // Конструктор с параметрами
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Конструктор с частичными параметрами
    public Person(String name) {
        this.name = name;
        this.age = 0;
    }
}
```

Ссылка this

Ключевое слово `this` ссылается на текущий объект. Оно используется для:

1. Разрешения конфликта имен между параметрами метода и полями класса:

```
public void setName(String name) {
    this.name = name;  // this.name ссылается на поле класса
}
```

2. Вызова другого конструктора того же класса:

```
public Person(String name) {
    this(name, 0);  // Вызов конструктора Person(String, int)
}
```

3. Передачи ссылки на текущий объект другим методам:

```
public void register(Registry registry) {
    registry.add(this);  // Передача текущего объекта
}
```

Перегрузка методов

Перегрузка методов – это механизм, позволяющий определять несколько методов с одинаковым именем, но разными параметрами.

Правила перегрузки:

- Методы должны иметь одинаковое имя
- Методы должны отличаться количеством и/или типами параметров
- Только возвращаемого типа недостаточно для перегрузки

```
public class Calculator {
    // Перегруженные методы sum
    public int sum(int a, int b) {
        return a + b;
    }

    public double sum(double a, double b) {
        return a + b;
    }

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

    // Эта перегрузка недопустима (отличается только возвращаемым
типом)
    // public double sum(int a, int b) {
    //     return (double)(a + b);
    // }
}
```

Final-компоненты

Ключевое слово `final` может применяться к различным компонентам:

Final-переменные

- Значение может быть присвоено только один раз
- Могут быть проинициализированы при объявлении или в конструкторе
- Константы обычно объявляются как `static final`

```
public class Constants {
    // Константа, инициализированная при объявлении
    public static final double PI = 3.14159;

    // Final-переменная экземпляра
    private final String id;

    public Constants(String id) {
        this.id = id;  // Инициализация в конструкторе
    }

    public void method() {
        final int localVar = 10;  // Final локальная переменная
        // localVar = 20;  // Ошибка компиляции
    }
}
```

Final-методы

- Не могут быть переопределены в подклассах
- Обеспечивают неизменное поведение при наследовании

```
public class Parent {
    // Final-метод не может быть переопределен
    public final void finalMethod() {
        System.out.println("This is a final method");
    }
}

public class Child extends Parent {
    // Ошибка компиляции - нельзя переопределить final-метод
    // @Override
    // public void finalMethod() {
    //     System.out.println("Trying to override final method");
    // }
}
```

Final-классы

- Не могут быть расширены (наследованы)
- Обеспечивают неизменную структуру класса

```
public final class FinalClass {
    // Этот класс не может быть расширен
}

// Ошибка компиляции - нельзя наследовать от final-класса
// public class Subclass extends FinalClass {
// }
```

Статические компоненты класса

Ключевое слово `static` указывает, что компонент принадлежит классу, а не конкретному экземпляру.

Статические поля

- Общие для всех экземпляров класса
- Инициализируются при загрузке класса
- Доступны через имя класса без создания объекта

```
public class Counter {
    private static int count = 0;  // Статическое поле

    public Counter() {
        count++;  // Увеличение при создании объекта
    }

    public static int getCount() {
        return count;  // Доступ к статическому полю
    }
}

// Использование
Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println(Counter.getCount());  // Выведет 2
```

Статические методы

- Не зависят от состояния объекта
- Могут обращаться только к статическим полям и методам
- Не могут использовать `this` или `super`
- Вызываются через имя класса

```
public class MathUtils {
    public static int max(int a, int b) {
        return (a > b) ? a : b;
    }

    public static double square(double x) {
        return x * x;
    }
}

// Использование
int maximum = MathUtils.max(10, 20);
double squared = MathUtils.square(2.5);
```

Статические блоки инициализации

- Выполняются при загрузке класса
- Используются для инициализации статических полей
- Могут быть несколько статических блоков, выполняются в порядке объявления

```
public class Database {
    private static Connection connection;

    // Статический блок инициализации
    static {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection =
DriverManager.getConnection("jdbc:mysql://localhost/mydb", "user",
"password");
            System.out.println("Database connection established");
        } catch (Exception e) {
            System.err.println("Error connecting to database: " +
e.getMessage());
        }
    }

    public static Connection getConnection() {
        return connection;
    }
}
```

Статические вложенные классы

- Вложенные классы, объявленные как static
- Не имеют доступа к нестатическим членам внешнего класса
- Могут быть использованы без создания экземпляра внешнего класса

```
public class Outer {
    private static int staticField = 10;
    private int instanceField = 20;

    // Статический вложенный класс
    public static class StaticNested {
        public void display() {
            System.out.println("Static field: " + staticField);
            // System.out.println(instanceField);  // Ошибка
компиляции
        }
    }
}

// Использование
Outer.StaticNested nested = new Outer.StaticNested();
nested.display();
```

Операция «сборка мусора»

Сборка мусора (Garbage Collection) – это автоматический процесс управления памятью в Java.

Основные принципы:

- Java автоматически освобождает память, занятую объектами, которые больше не используются
- Объект считается недостижимым, когда на него нет ссылок или все ссылки находятся в недостижимых объектах
- Сборщик мусора запускается JVM в разные моменты времени
- Программист не может напрямую управлять сборкой мусора

Метод finalize()

- Вызывается перед уничтожением объекта сборщиком мусора
- Может быть переопределен для освобождения ресурсов
- Не гарантируется, что метод будет вызван
- В современной Java рекомендуется использовать try-with-resources вместо finalize()

```
public class Resource {
    private FileHandle file;

    public Resource(String path) {
        file = openFile(path);
    }

    @Override
    protected void finalize() throws Throwable {
        try {
            if (file != null) {
                file.close();  // Освобождение ресурса
            }
        } finally {
            super.finalize();
        }
    }
}
```

Явный вызов сборки мусора

- `System.gc()` и `Runtime.getRuntime().gc()` предлагают JVM выполнить сборку мусора
- Нет гарантии, что сборка мусора будет выполнена немедленно
- В большинстве случаев не рекомендуется явно вызывать сборку мусора

```
public void cleanUp() {
    bigObject = null;  // Удаление ссылки
    System.gc();  // Предложение выполнить сборку мусора
}
```

5. Наследование в Java. Суперкласс и подклассы. Конструкторы подкласса. Доступ к компонентам при наследовании. Переопределение методов.

Наследование в Java

Наследование – один из фундаментальных принципов объектно-ориентированного программирования, который позволяет создать новый класс на основе существующего. В Java наследование реализуется с помощью ключевого слова `extends`.

```
public class Animal {
    // Содержимое суперкласса
}

public class Dog extends Animal {
    // Содержимое подкласса
}
```

Java поддерживает только одиночное наследование классов (один класс может наследоваться только от одного класса), но множественное наследование интерфейсов.

Суперкласс и подклассы

Суперкласс (базовый класс, родительский класс) – класс, от которого наследуются другие классы.

Подкласс (производный класс, дочерний класс) – класс, который наследуется от другого класса.

При наследовании:

- Подкласс получает все поля и методы суперкласса (кроме конструкторов)
- Подкласс может добавлять новые поля и методы
- Подкласс может переопределять методы суперкласса
- Объект подкласса является экземпляром как своего класса, так и суперкласса

```
public class Vehicle {
    protected String make;
    protected String model;
    protected int year;

    public void start() {
        System.out.println("Vehicle started");
    }

    public void stop() {
        System.out.println("Vehicle stopped");
    }
}

public class Car extends Vehicle {
    private int numberOfDoors;

    public void honk() {
        System.out.println("Beep beep!");
    }

    @Override
    public void start() {
        System.out.println("Car engine started");
    }
}
```

Конструкторы подкласса

При создании объекта подкласса сначала вызывается конструктор суперкласса, а затем конструктор подкласса.

1. **Неявный вызов конструктора суперкласса**: Если в конструкторе подкласса не указан явный вызов конструктора суперкласса, Java автоматически вставляет вызов конструктора суперкласса без параметров `super()` в начало конструктора подкласса.

```
public class Animal {
    public Animal() {
        System.out.println("Animal constructor");
    }
}

public class Dog extends Animal {
    public Dog() {
        // Неявный вызов super() здесь
        System.out.println("Dog constructor");
    }
}
```

2. **Явный вызов конструктора суперкласса**: Для вызова конкретного конструктора суперкласса используется ключевое слово `super()` с соответствующими параметрами.

```
public class Animal {
    protected String name;

    public Animal() {
        this.name = "Unknown";
    }

    public Animal(String name) {
        this.name = name;
    }
}

public class Dog extends Animal {
    private String breed;

    public Dog() {
        super();  // Вызов конструктора Animal()
        this.breed = "Unknown";
    }

    public Dog(String name, String breed) {
        super(name);  // Вызов конструктора Animal(String)
        this.breed = breed;
    }
}
```

Важные правила:

- Вызов `super()` должен быть первой инструкцией в конструкторе подкласса
- Если суперкласс не имеет конструктора без параметров, подкласс должен явно вызывать один из доступных конструкторов суперкласса

Доступ к компонентам при наследовании

Доступность компонентов суперкласса в подклассе зависит от модификаторов доступа:

Модификатор	В том же классе	В том же пакете	В подклассе (другой пакет)	Везде
private	Да	Нет	Нет	Нет
default	Да	Да	Нет	Нет
protected	Да	Да	Да	Нет
public	Да	Да	Да	Да

Примеры доступа к компонентам:

```java
// В пакете animals
package animals;

public class Animal {
    private String privateField = "Private";
    String defaultField = "Default";
    protected String protectedField = "Protected";
    public String publicField = "Public";

    private void privateMethod() { }
    void defaultMethod() { }
    protected void protectedMethod() { }
    public void publicMethod() { }
}
```

```java
// В том же пакете
package animals;

public class Mammal extends Animal {
    public void access() {
        // System.out.println(privateField);  // Ошибка - недоступно
        System.out.println(defaultField);     // Доступно в том же
пакете
        System.out.println(protectedField);   // Доступно в подклассе
        System.out.println(publicField);      // Доступно везде

        // privateMethod();  // Ошибка - недоступно
        defaultMethod();     // Доступно в том же пакете
        protectedMethod();   // Доступно в подклассе
        publicMethod();      // Доступно везде
    }
}
```

```java
// В другом пакете
package pets;

import animals.Animal;

public class Dog extends Animal {
    public void access() {
        // System.out.println(privateField);  // Ошибка - недоступно
        // System.out.println(defaultField);  // Ошибка - недоступно
в другом пакете
        System.out.println(protectedField);   // Доступно в подклассе
        System.out.println(publicField);      // Доступно везде

        // privateMethod();  // Ошибка - недоступно
        // defaultMethod();  // Ошибка - недоступно в другом пакете
        protectedMethod();   // Доступно в подклассе
        publicMethod();      // Доступно везде
    }
}
```

Переопределение методов

Переопределение метода (override) – это реализация метода в подклассе, который уже определен в суперклассе, с той же сигнатурой (имя, параметры, возвращаемый тип).

Правила переопределения методов:

- Метод в подклассе должен иметь то же имя
- Метод в подклассе должен иметь те же типы параметров в том же порядке
- Метод в подклассе должен иметь тот же возвращаемый тип или его подтип (ковариантный возврат)
- Метод в подклассе не может иметь более строгий модификатор доступа
- Метод в подклассе не может выбрасывать новые или более широкие исключения
- Статические методы не могут быть переопределены (они скрываются)
- Final методы не могут быть переопределены
- Конструкторы не наследуются и не могут быть переопределены
- Private методы не наследуются и не могут быть переопределены

Аннотация `@Override` рекомендуется (но не обязательна) для явного указания, что метод переопределяет метод суперкласса.

```
public class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }

    public Animal reproduce() {
        return new Animal();
    }
}

public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }

    @Override
    public Dog reproduce() {  // Ковариантный возврат
        return new Dog();
    }
}
```

Вызов методов суперкласса

Для вызова переопределенного метода суперкласса из подкласса используется ключевое слово `super`.

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

public class Dog extends Animal {
    @Override
    public void eat() {
        super.eat();  // Вызов метода суперкласса
        System.out.println("Dog is eating bones");
    }
}
```

6. Абстрактные методы. Абстрактные классы и интерфейсы и их реализация.

Абстрактные методы

Абстрактный метод – это метод, который объявлен, но не имеет реализации. Он определяет только сигнатуру (имя, параметры, возвращаемый тип), а реализация должна быть предоставлена в подклассах.

Синтаксис абстрактного метода:

```
abstract тип_возврата имя_метода(параметры);
```

Особенности абстрактных методов:

- Не имеют тела (фигурных скобок и кода внутри)
- Должны быть объявлены с ключевым словом `abstract`
- Могут быть объявлены только в абстрактных классах или интерфейсах
- Не могут быть объявлены как `private`, `static` или `final`
- Должны быть реализованы (переопределены) в первом конкретном подклассе

```
public abstract class Shape {
    // Абстрактный метод
    public abstract double calculateArea();

    // Обычный метод
    public void display() {
        System.out.println("Area: " + calculateArea());
    }
}
```

Абстрактные классы

Абстрактный класс – это класс, который не может быть инстанцирован (нельзя создать его экземпляр) и предназначен для наследования.

Синтаксис абстрактного класса:

```
abstract class ИмяКласса {
    // Содержимое класса
}
```

Особенности абстрактных классов:

- Объявляются с ключевым словом `abstract`
- Могут содержать как абстрактные, так и обычные методы
- Могут содержать поля, конструкторы, статические методы
- Могут наследовать другие классы (абстрактные или конкретные)
- Могут реализовывать интерфейсы
- Подклассы должны реализовать все абстрактные методы или также быть абстрактными
- Не могут быть инстанцированы, но могут быть использованы как ссылочные типы

```
public abstract class Animal {
    private String name;

    // Конструктор абстрактного класса
    public Animal(String name) {
        this.name = name;
    }

    // Обычный метод
    public String getName() {
        return name;
    }

    // Абстрактный метод
    public abstract void makeSound();
}

// Конкретный подкласс
public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    // Реализация абстрактного метода
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

Интерфейсы

Интерфейс – это абстрактный тип, который определяет набор методов, которые класс должен реализовать. Интерфейсы позволяют достичь полного абстрагирования и множественного наследования функциональности.

Синтаксис интерфейса:

```
interface ИмяИнтерфейса {
    // Содержимое интерфейса
}
```

Особенности интерфейсов:

- Все методы в интерфейсе неявно являются `public` и `abstract` (кроме default и static методов в Java 8+)
- Все поля в интерфейсе неявно являются `public`, `static` и `final` (константы)
- Интерфейс может наследовать другие интерфейсы (множественное наследование)
- Класс может реализовывать несколько интерфейсов

- Интерфейс не может быть инстанцирован, но может быть использован как ссылочный тип
- С Java 8 интерфейсы могут иметь default и static методы с реализацией
- С Java 9 интерфейсы могут иметь private методы

```
public interface Drawable {
    // Константа
    int MAX_SIZE = 100;  // неявно public static final

    // Абстрактный метод
    void draw();  // неявно public abstract

    // Default метод (Java 8+)
    default void resize() {
        System.out.println("Resizing to default size");
    }

    // Static метод (Java 8+)
    static void printInfo() {
        System.out.println("Drawable interface");
    }

    // Private метод (Java 9+)
    private void helperMethod() {
        System.out.println("Helper method");
    }
}
```

Реализация интерфейсов

Класс реализует интерфейс с помощью ключевого слова `implements`. При этом класс должен предоставить реализацию для всех абстрактных методов интерфейса.

```
public class Circle implements Drawable {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    // Реализация метода интерфейса
    @Override
    public void draw() {
        System.out.println("Drawing a circle with radius " + radius);
    }

    // Можно переопределить default метод
    @Override
    public void resize() {
        System.out.println("Resizing circle");
    }
}
```

Класс может реализовывать несколько интерфейсов:

```
public interface Movable {
    void move(int x, int y);
}

public class Rectangle implements Drawable, Movable {
    private int width;
    private int height;
    private int x;
    private int y;

    // Конструктор
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    // Реализация метода Drawable
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle at (" + x + "," + y +
")");
    }

    // Реализация метода Movable
    @Override
    public void move(int x, int y) {
        this.x = x;
        this.y = y;
        System.out.println("Moved to (" + x + "," + y + ")");
    }
}
```

Сравнение абстрактных классов и интерфейсов

Характеристика	Абстрактный класс	Интерфейс
Инстанцирование	Нельзя создать экземпляр	Нельзя создать экземпляр
Методы	Могут быть абстрактными и конкретными	Абстрактные (и default/static с Java 8+)
Поля	Любые типы и модификаторы	Только константы (`public static final`)
Конструкторы	Может иметь конструкторы	Не может иметь конструкторов
Наследование	Одиночное (может расширять один класс)	Множественное (может расширять несколько интерфейсов)

Характеристика	Абстрактный класс	Интерфейс
Реализация	Класс может расширять только один абстрактный класс	Класс может реализовывать несколько интерфейсов
Модификаторы доступа	Любые	Методы неявно `public`, поля неявно `public static final`
Использование	Когда классы имеют общие атрибуты и поведение	Когда несвязанные классы должны реализовать общее поведение

7. Оболочки простых типов. Обзор пакета java.lang.

Оболочки простых типов

Оболочки примитивных типов (wrapper classes) – это классы, которые предоставляют объектное представление примитивных типов данных в Java. Они позволяют использовать примитивы там, где требуются объекты, например, в коллекциях.

Примитивный тип	Класс-оболочка	Расположение
boolean	Boolean	java.lang
byte	Byte	java.lang
short	Short	java.lang
int	Integer	java.lang
long	Long	java.lang
float	Float	java.lang
double	Double	java.lang
char	Character	java.lang

Создание объектов-оболочек

```
// Явное создание (устаревший способ)
Integer intObj1 = new Integer(42);  // Устарело с Java 9

// Использование статических методов valueOf (предпочтительный
способ)
Integer intObj2 = Integer.valueOf(42);
Double doubleObj = Double.valueOf(3.14);
Boolean boolObj = Boolean.valueOf(true);

// Автобоксинг (с Java 5)
Integer intObj3 = 42;  // Автоматическое преобразование int в Integer
Double doubleObj2 = 3.14;  // Автоматическое преобразование double в
Double
```

Распаковка (получение примитивного значения)

```
// Явная распаковка через методы xxxValue()
int primitive1 = intObj1.intValue();
double primitive2 = doubleObj.doubleValue();

// Автораспаковка (с Java 5)
int primitive3 = intObj3;  // Автоматическое преобразование Integer в
int
```

Преобразование строк в примитивы

```
// Преобразование строки в примитив
int i = Integer.parseInt("123");
double d = Double.parseDouble("3.14");
boolean b = Boolean.parseBoolean("true");

// Преобразование строки в объект-оболочку
Integer intObj = Integer.valueOf("123");
Double doubleObj = Double.valueOf("3.14");
```

Константы и полезные методы

```java
// Константы
int maxInt = Integer.MAX_VALUE;  // 2147483647
int minInt = Integer.MIN_VALUE;  // -2147483648
double posInf = Double.POSITIVE_INFINITY;
double negInf = Double.NEGATIVE_INFINITY;
double nan = Double.NaN;

// Проверка NaN
if (Double.isNaN(value)) {
    System.out.println("Value is not a number");
}

// Сравнение
int result = Integer.compare(5, 10);  // -1

// Преобразование типов
String binaryString = Integer.toBinaryString(42);  // "101010"
String hexString = Integer.toHexString(42);       // "2a"
```

Обзор пакета java.lang

Пакет `java.lang` – это основной пакет Java, который автоматически импортируется во все Java-программы. Он содержит классы, которые являются фундаментальными для языка Java.

Основные классы пакета java.lang:

1. **Object** – базовый класс для всех классов в Java
    - `equals(Object obj)` – сравнение объектов
    - `hashCode()` – получение хеш-кода объекта
    - `toString()` – получение строкового представления объекта
    - `getClass()` – получение объекта Class
    - `clone()` – создание копии объекта
    - `finalize()` – метод, вызываемый перед сборкой мусора (устарел с Java 9)
2. **Class** – представляет классы и интерфейсы в выполняющейся Java-программе
    - `getName()` – получение имени класса
    - `getSuperclass()` – получение суперкласса
    - `getInterfaces()` – получение интерфейсов, реализуемых классом
    - `getDeclaredMethods()` – получение методов класса
    - `newInstance()` – создание нового экземпляра класса (устарел с Java 9)
3. **String** – представляет строки в Java
    - `length()` – получение длины строки
    - `charAt(int index)` – получение символа по индексу
    - `substring(int beginIndex, int endIndex)` – получение подстроки
    - `equals(Object obj)` – сравнение строк
    - `equalsIgnoreCase(String str)` – сравнение строк без учета регистра

- `concat(String str)` – конкатенация строк
- `replace(char oldChar, char newChar)` – замена символов
- `split(String regex)` – разделение строки по регулярному выражению
- `toLowerCase()` / `toUpperCase()` – преобразование регистра

4. **StringBuilder** и **StringBuffer** – изменяемые строки

- `StringBuilder` – не синхронизирован, быстрее
- `StringBuffer` – синхронизирован, потокобезопасен
- `append(...)` – добавление в конец
- `insert(int offset, ...)` – вставка на позицию
- `delete(int start, int end)` – удаление части
- `reverse()` – переворачивание строки

5. **Math** – математические функции и константы

- `PI`, `E` – константы
- `abs(...)` – модуль числа
- `min(...)`, `max(...)` – нахождение минимума/максимума
- `sqrt(double a)` – квадратный корень
- `pow(double a, double b)` – возведение в степень
- `sin(double a)`, `cos(double a)`, `tan(double a)` – тригонометрические функции
- `random()` – случайное число от 0.0 до 1.0

6. **System** – системные операции

- `out` – стандартный вывод (PrintStream)
- `in` – стандартный ввод (InputStream)
- `err` – стандартный вывод ошибок (PrintStream)
- `currentTimeMillis()` – текущее время в миллисекундах
- `arraycopy(...)` – копирование массивов
- `exit(int status)` – завершение программы

7. **Thread** и **Runnable** – многопоточное программирование

- `Thread` – класс для создания потоков
- `Runnable` – интерфейс для реализации выполняемого кода
- `start()` – запуск потока
- `sleep(long millis)` – приостановка потока
- `join()` – ожидание завершения потока

8. **Exception** и его подклассы – обработка исключений

- `Throwable` – базовый класс для всех исключений
- `Exception` – базовый класс для проверяемых исключений
- `RuntimeException` – базовый класс для непроверяемых исключений
- `Error` – серьезные проблемы, обычно не обрабатываются

9. **Enum** – базовый класс для всех перечислений (с Java 5)

- `ordinal()` – порядковый номер константы
- `name()` – имя константы

- `valueOf(String name)` – получение константы по имени
- `values()` – получение всех констант

10. **Классы-оболочки** примитивных типов (описаны выше)
    - `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`
11. **Аннотации** (с Java 5)
    - `@Override` – метод переопределяет метод суперкласса
    - `@Deprecated` – метод/класс устарел
    - `@SuppressWarnings` – подавление предупреждений компилятора
    - `@FunctionalInterface` – функциональный интерфейс (с Java 8)

8. Обработка исключительных ситуаций. Иерархия классов исключений. Создание собственных классов исключений.

Обработка исключительных ситуаций

Исключение (exception) – это событие, которое происходит во время выполнения программы и нарушает нормальный поток инструкций. Механизм обработки исключений в Java позволяет обнаруживать и обрабатывать ошибки программно.

Блок try-catch-finally

Основной механизм обработки исключений – блок `try-catch-finally`:

```
try {
    // Код, который может вызвать исключение
    int result = 10 / 0;  // Вызовет ArithmeticException
} catch (ArithmeticException e) {
    // Обработка конкретного исключения
    System.out.println("Деление на ноль: " + e.getMessage());
} catch (Exception e) {
    // Обработка других исключений
    System.out.println("Произошла ошибка: " + e.getMessage());
} finally {
    // Код, который выполняется всегда, независимо от исключения
    System.out.println("Блок finally выполнен");
}
```

Особенности блока `try-catch-finally`:

- Блок `try` содержит код, который может вызвать исключение
- Блоки `catch` обрабатывают исключения определенных типов
- Блоки `catch` проверяются в порядке их объявления, поэтому более конкретные исключения должны обрабатываться перед более общими
- Блок `finally` выполняется всегда, независимо от того, было ли исключение или нет (кроме случая вызова `System.exit()`)
- Блок `finally` обычно используется для освобождения ресурсов

Try-with-resources (с Java 7)

Конструкция `try-with-resources` автоматически закрывает ресурсы, реализующие интерфейс `AutoCloseable`:

```
try (FileReader fr = new FileReader("file.txt");
     BufferedReader br = new BufferedReader(fr)) {
    // Использование ресурсов
    String line = br.readLine();
    System.out.println(line);
} catch (IOException e) {
    // Обработка исключений
    System.out.println("Ошибка ввода-вывода: " + e.getMessage());
}
// Ресурсы fr и br автоматически закрываются
```

Множественный catch (с Java 7)

Java 7 позволяет обрабатывать несколько типов исключений в одном блоке `catch`:

```
try {
    // Код, который может вызвать исключения
} catch (IOException | SQLException e) {
    // Обработка нескольких типов исключений
    System.out.println("Ошибка: " + e.getMessage());
}
```

Иерархия классов исключений

В Java все исключения являются подклассами класса `Throwable`. Основные ветви иерархии:

1. **Throwable** – корень иерархии исключений
    - **Error** – серьезные ошибки, обычно не обрабатываются программно
        - `OutOfMemoryError` – нехватка памяти
        - `StackOverflowError` – переполнение стека
        - `VirtualMachineError` – ошибки виртуальной машины
    - **Exception** – исключения, которые должны быть обработаны
        - **RuntimeException** – непроверяемые исключения (не требуют объявления в сигнатуре метода)
            - `ArithmeticException` – арифметические ошибки (например, деление на ноль)
            - `NullPointerException` – обращение к методу/полю объекта, который равен null
            - `IndexOutOfBoundsException` – индекс за пределами массива/коллекции
            - `IllegalArgumentException` – недопустимый аргумент метода

    - `ClassCastException` – недопустимое приведение типов
- **Другие Exception** – проверяемые исключения (должны быть объявлены или обработаны)
    - `IOException` – ошибки ввода-вывода
    - `SQLException` – ошибки в работе с базами данных
    - `ClassNotFoundException` – класс не найден
    - `InterruptedException` – поток был прерван

Проверяемые и непроверяемые исключения

1. **Проверяемые исключения (checked exceptions)**:
    - Подклассы `Exception` (кроме `RuntimeException` и его подклассов)
    - Должны быть объявлены в сигнатуре метода с помощью `throws` или обработаны в блоке `try-catch`
    - Обычно представляют условия, которые программа должна обработать (например, ошибки ввода-вывода)
2. **Непроверяемые исключения (unchecked exceptions)**:
    - `RuntimeException` и его подклассы
    - `Error` и его подклассы
    - Не требуют объявления в сигнатуре метода или обработки
    - Обычно представляют программные ошибки, которые сложно предсказать (например, обращение к нулевой ссылке)

Объявление исключений в методах

Метод, который может вызвать проверяемое исключение, должен объявить его в своей сигнатуре с помощью ключевого слова `throws`:

```
public void readFile(String fileName) throws IOException {
    FileReader fileReader = new FileReader(fileName);
    // Код чтения файла
}
```

Если метод может вызвать несколько типов исключений, их можно перечислить через запятую:

```
public void readAndProcessFile(String fileName) throws IOException,
SQLException {
    // Код работы с файлом и базой данных
}
```

Создание собственных классов исключений

Для создания собственного исключения необходимо создать класс, который наследуется от `Exception` (для проверяемых исключений) или `RuntimeException` (для

непроверяемых исключений).

Пример проверяемого исключения:

```
public class InsufficientFundsException extends Exception {
    private double amount;

    // Конструктор с сообщением
    public InsufficientFundsException(String message) {
        super(message);
    }

    // Конструктор с сообщением и параметром
    public InsufficientFundsException(String message, double amount)
{
        super(message);
        this.amount = amount;
    }

    // Конструктор с вложенным исключением
    public InsufficientFundsException(String message, Throwable
cause) {
        super(message, cause);
    }

    // Геттер для дополнительной информации
    public double getAmount() {
        return amount;
    }
}
```

Пример непроверяемого исключения:

```java
public class InvalidInputException extends RuntimeException {
    // Конструктор без параметров
    public InvalidInputException() {
        super();
    }

    // Конструктор с сообщением
    public InvalidInputException(String message) {
        super(message);
    }

    // Конструктор с сообщением и вложенным исключением
    public InvalidInputException(String message, Throwable cause) {
        super(message, cause);
    }

    // Конструктор с вложенным исключением
    public InvalidInputException(Throwable cause) {
        super(cause);
    }
}
```

Использование собственных исключений:

```
public class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    // Метод, который может вызвать проверяемое исключение
    public void withdraw(double amount) throws
InsufficientFundsException {
        if (amount <= 0) {
            throw new InvalidInputException("Сумма должна быть
положительной");
        }

        if (amount > balance) {
            throw new InsufficientFundsException("Недостаточно
средств", amount);
        }

        balance -= amount;
    }

    // Обработка исключений
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);

        try {
            account.withdraw(1500);  // Вызовет
InsufficientFundsException
        } catch (InsufficientFundsException e) {
            System.out.println(e.getMessage() + ": " +
e.getAmount());
        } catch (InvalidInputException e) {
            System.out.println("Ошибка ввода: " + e.getMessage());
        }
    }
}
```

Рекомендации по созданию исключений:

1. **Наследование**:
   - Наследуйте от `Exception` для проверяемых исключений
   - Наследуйте от `RuntimeException` для непроверяемых исключений
   - Наследуйте от существующих исключений, если ваше исключение логически связано с ними
2. **Конструкторы**:
   - Реализуйте несколько конструкторов для разных случаев
   - Всегда вызывайте конструктор суперкласса

3. **Дополнительная информация**:
    - Добавляйте поля для хранения дополнительной информации об ошибке
    - Предоставляйте геттеры для этих полей
4. **Именование**:
    - Используйте суффикс "Exception" в имени класса
    - Имя должно отражать проблему (например, `FileNotFoundException`)
5. **Сериализация**:
    - Добавьте `serialVersionUID` для сериализации
    - Реализуйте `Serializable`, чтобы исключение можно было сериализовать