Вопросы к экзамену по Java База

Samuel Burns

17 января 2025 г.

1 Язык программирования Java. Общая характеристика

Java — это объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (ныне принадлежащей Oracle Corporation). Он был впервые представлен в 1995 году.

Основные характеристики Java

- **Платформенная независимость**. Программы, написанные на Java, компилируются в байт-код, который выполняется на виртуальной машине Java (JVM). Это позволяет запускать Java-приложения на любой платформе, поддерживающей JVM, без необходимости перекомпиляции кода.
- Java является **строго объектно-ориентированным** языком программирования. Все данные и функции в Java организованы в виде объектов и классов.
- Java предоставляет **автоматическое управление памятью** с помощью сборщика мусора (Garbage Collector).
- Java поддерживает **многопоточность на уровне языка**.

2 Консольный ввод и вывод (классы, методы, особенности)

В Java консольный ввод и вывод осуществляется с помощью классов **System**, **Scanner** и **PrintStream**.

Консольный вывод

Для вывода данных в консоль используется класс System и его статическое поле out, которое является экземпляром класса PrintStream.

```
[]
public class Main {
    public static void main(String[] args) {
        System.out.print("Hello, ");
        System.out.println("World!");
        System.out.printf("Formatted output: %d %s\n", 42, "answer");
    }
}
```

Консольный ввод

Для ввода данных из консоли используется класс Scanner, который находится в пакете java.util. Основные методы для ввода данных:

- nextInt(): Считывает целое число.
- nextDouble(): Считывает число с плавающей точкой.
- nextLine(): Считывает строку.
- next(): Считывает слово (до пробела).

Особенности

- Буферизация ввода: Методы nextInt(), nextDouble() и другие не потребляют символ новой строки, оставленный после ввода. Поэтому перед использованием nextLine() после этих методов рекомендуется вызвать scanner.nextLine() для потребления этого символа.
- Закрытие Scanner: После завершения работы с Scanner рекомендуется закрыть его с помощью метода close(), чтобы освободить ресурсы.
- Форматированный вывод: Метод printf позволяет использовать форматирование, аналогичное форматированию в языке C.

3 Примитивные типы данных

- `byte`: 8-битный целочисленный тип, диапазон значений от -128 до 127.
- `short`: 16-битный целочисленный тип, диапазон значений от $-32,768$ до $32,767$.
- `int`: 32-битный целочисленный тип, диапазон значений от -2^{16} до 2^{16}.
- `long`: 64-битный целочисленный тип, диапазон значений от -2^{32} до 2^{32}.
- `float`: 32-битный тип с плавающей точкой, диапазон значений примерно от 1.4×10^{-45} до 3.4×10^{38}.
- `double`: 64-битный тип с плавающей точкой, диапазон значений примерно от 4.9×10^{-324} до 1.7×10^{308}.
- `char`: 16-битный символьный тип, представляет одиночный символ в кодировке Unicode, диапазон значений от 0 до $65,535$.
- `boolean`: представляет логическое значение и может принимать одно из двух значений: `true` или `false`.

4 Числовые типы данных, операции над ними

- `byte`: 8-битный целочисленный тип, диапазон значений от -128 до 127.
- `short`: 16-битный целочисленный тип, диапазон значений от $-32,768$ до $32,767$.
- `int`: 32-битный целочисленный тип, диапазон значений от -2^{16} до 2^{16}.
- `long`: 64-битный целочисленный тип, диапазон значений от -2^{32} до 2^{32}.
- `float`: 32-битный тип с плавающей точкой, диапазон значений примерно от 1.4×10^{-45} до 3.4×10^{38}.

- `double`: 64-битный тип с плавающей точкой, диапазон значений примерно от 4.9×10^{-324} до 1.7×10^{308}.

Операции над числовыми типами данных

Арифметические операции

- Сложение (+)
- Вычитание (-)
- Умножение (*)
- Деление (/)
- Остаток от деления (%)

Инкремент и декремент

- Инкремент (++)
- Декремент (==)

Операции сравнения

- Равно (==)
- Не равно (!=)
- Меньше (<)
- Больше (>)
- Меньше или равно (<=)
- Больше или равно (>=)

5 Логический тип данных, операции над ним

`boolean`: представляет логическое значение и может принимать одно из двух значений: `true` или `false`.

Логические операции

- Логическое И (&&)
- Логическое ИЛИ (||)
- Логическое НЕ (!)

6 Строковой и символьный типы данных, операции над ними

`char`: 16-битный символьный тип, представляет одиночный символ в кодировке Unicode, диапазон значений от 0 до $65,535$.

`String`: Представляет последовательность символов. Ссылочный тип данных.

Операции над символьными типами данных

- **Сравнение символов**:
  - Равно: `char c1 = 'A'; char c2 = 'A'; boolean isEqual = (c1 == c2);`
  - Не равно: `boolean isNotEqual = (c1 != c2);`
  - Меньше: `boolean isLess = (c1 < c2);`
  - Больше: `boolean isGreater = (c1 > c2);`
  - Меньше или равно: `boolean isLessOrEqual = (c1 <= c2);`
  - Больше или равно: `boolean isGreaterOrEqual = (c1 >= c2);`

7 Класс String. Особенности реализации. Неизменяемость строк

В Java строки являются неизменяемыми (immutable). Это означает, что после создания объекта String его содержимое не может быть изменено. Любая операция, которая, кажется, изменяет строку, на самом деле создает новый объект String.

Операции над строковыми типами данных

- **Создание строк**:
  - Литералы строк: `String greeting = "Hello";`
  - Конструкторы класса String: `String str = new String("Hello");`
- **Конкатенация строк**:
  - Оператор +: `String result = "Hello"+ "World";`

- Метод concat: `String result = "Hello".concat("World");`

- **Сравнение строк**:
  - Метод equals: `boolean isEqual = "Hello".equals("Hello");`
  - Метод equalsIgnoreCase: `boolean isEqualIgnoreCase = "Hello".equalsIgnoreC`
  - Метод compareTo: `int comparison = "Hello".compareTo("World");`

- **Получение длины строки**:
  - Метод length: `int length = "Hello".length();`

- **Получение символов из строки**:
  - Метод charAt: `char firstChar = "Hello".charAt(0);`

- **Получение подстроки**:
  - Метод substring: `String substring = "Hello".substring(1, 4);`

- **Преобразование строки**:
  - Метод toUpperCase: `String upperCase = "Hello".toUpperCase();`
  - Метод toLowerCase: `String lowerCase = "Hello".toLowerCase();`

- **Поиск в строке**:
  - Метод indexOf: `int index = "Hello".indexOf('e');`
  - Метод lastIndexOf: `int lastIndex = "Hello".lastIndexOf('l');`

- **Замена в строке**:
  - Метод replace: `String replaced = "Hello".replace('e', 'a');`

8 Условный оператор. Варианты синтаксиса

Синтаксис условного оператора

Простой if

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

Оператор `if-else`

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

Оператор `if-else if-else`

```
if (условие1) {
    // код, который выполняется, если условие1 истинно
} else if (условие2) {
    // код, который выполняется, если условие2 истинно
} else {
    // код, который выполняется, если ни одно из условий не истинно
}
```

Примеры использования

Простой `if`

```
int x = 10;
if (x > 5) {
    System.out.println("x больше 5");
}
```

Оператор `if-else`

```
int y = 3;
if (y > 5) {
    System.out.println("y больше 5");
} else {
    System.out.println("y меньше или равно 5");
}
```

Оператор `if-else if-else`

```
int z = 7;
if (z > 10) {
    System.out.println("z больше 10");
} else if (z > 5) {
    System.out.println("z больше 5, но меньше или равно 10");
```

```
} else {
    System.out.println("z меньше или равно 5");
}
```

9 Цикл for. Синтаксис, особенности, применение

Синтаксис цикла `for`

```
for (инициализация; условие; обновление) {
    // код, который выполняется на каждой итерации
}
```

- **Инициализация**: Выполняется один раз перед началом цикла. Обычно используется для инициализации счетчика.
- **Условие**: Проверяется перед каждой итерацией. Если условие истинно, выполняется тело цикла. Если ложно, цикл завершается.
- **Обновление**: Выполняется после каждой итерации. Обычно используется для обновления счетчика.

Примеры использования

Простой цикл `for`

```
for (int i = 0; i < 5; i++) {
    System.out.println("Итерация: " + i);
}
```

Цикл `for` с использованием массива

```
int[] numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < numbers.length; i++) {
    System.out.println("Элемент: " + numbers[i]);
}
```

Цикл `for` с использованием улучшенного синтаксиса (enhanced for loop)

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

```
    System.out.println("Элемент: " + number);
}
```

Особенности цикла `for`

- **Многократная инициализация**: Можно инициализировать несколько переменных.

```
for (int i = 0, j = 10; i < 5; i++, j--) {
    System.out.println("i: " + i + ", j: " + j);
}
```

- **Пустая инициализация, условие или обновление**: Любая из частей цикла `for` может быть опущена.

```
int i = 0;
for (; i < 5;) {
    System.out.println("Итерация: " + i);
    i++;
}
```

- **Бесконечный цикл**: Можно создать бесконечный цикл, опустив условие.

```
for (;;) {
    System.out.println("Бесконечный цикл");
    // Для выхода из цикла можно использовать break
    break;
}
```

10 Цикл while. Синтаксис, особенности, применение

Цикл `while` используется для повторения блока кода до тех пор, пока условие истинно. Он проверяет условие **перед** каждой итерацией и выполняет тело цикла, если условие истинно.

Синтаксис цикла `while`

```
while (условие) {
    // код, который выполняется на каждой итерации
}
```

- **Условие**: Проверяется перед каждой итерацией. Если условие истинно, выполняется тело цикла. Если ложно, цикл завершается.

11 Массивы. Объявление, операции над массивом, особенности реализации, применение

Массивы в Java представляют собой структуры данных, которые хранят **фиксированное** количество элементов одного типа.

Объявление и инициализация массива

```
int[] numbers = new int[5]; // Массив из 5 элементов типа int
```

Инициализация массива с элементами

```
int[] numbers = {1, 2, 3, 4, 5}; // Массив с элементами 1, 2, 3, 4, 5
```

Операции над массивами

Доступ к элементам массива

```
int[] numbers = {1, 2, 3, 4, 5};
int firstElement = numbers[0]; // Доступ к первому элементу
int lastElement = numbers[numbers.length - 1]; // Доступ к последнему элементу
```

Изменение элементов массива

```
numbers[0] = 10; // Изменение первого элемента на 10
```

Итерация по элементам массива

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

Или

```
for (int number : numbers) {
    System.out.println("Элемент: " + number);
}
```

Особенности реализации массивов

- **Фиксированный размер**: Размер массива фиксируется при его создании и не может быть изменен.
- **Типизация**: Все элементы массива должны быть одного типа.
- **Инициализация по умолчанию**: Элементы массива инициализируются значениями по умолчанию для их типа (например, 0 для `int`, `null` для объектов).

12 Массивы. Объявление многомерных массивов и операции над ними

Объявление и инициализация двумерного массива

```
int[][] matrix = new int[3][3]; // Двумерный массив 3x3
```

Инициализация двумерного массива с элементами

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

Объявление и инициализация трехмерного массива

```
int[][][] cube = new int[2][2][2]; // Трехмерный массив 2x2x2
```

Операции над многомерными массивами

Доступ к элементам двумерного массива

```
int[][] matrix = {
    {1, 2, 3},
```

```
    {4, 5, 6},
    {7, 8, 9}
};
int element = matrix[0][0]; // Доступ к элементу в первой строке и первом столбц
```

Изменение элементов двумерного массива

```
matrix[0][0] = 10; // Изменение элемента в первой строке и первом столбце на 10
```

Итерация по элементам двумерного массива

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

Использование улучшенного цикла `for` для двумерного массива

```
for (int[] row : matrix) {
    for (int element : row) {
        System.out.println("Элемент: " + element);
    }
}
```

13 Массивы. Объявление, операции над массивом. Линейный и бинарный поиск

Линейный поиск

Линейный поиск (или последовательный поиск) — это простой алгоритм поиска, который проверяет каждый элемент массива последовательно до тех пор, пока не найдет искомый элемент или не достигнет конца массива.

Алгоритм линейного поиска

```
public int linearSearch(int[] array, int target) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == target) {
            return i; // Элемент найден, возвращаем индекс
```

```
        }
    }
    return -1; // Элемент не найден
}
```

Бинарный поиск

Бинарный поиск — это более эффективный алгоритм поиска, который работает только с отсортированными массивами. Он делит массив пополам и проверяет средний элемент. Если средний элемент равен искомому, поиск завершается. Если искомый элемент меньше среднего, поиск продолжается в левой половине массива, иначе — в правой половине.

Алгоритм бинарного поиска

```
public int binarySearch(int[] array, int target) {
    int left = 0;
    int right = array.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (array[mid] == target) {
            return mid; // Элемент найден, возвращаем индекс
        }
        if (array[mid] < target) {
            left = mid + 1; // Ищем в правой половине
        } else {
            right = mid - 1; // Ищем в левой половине
        }
    }
    return -1; // Элемент не найден
}
```

14 Массивы. Объявление, операции над массивом. Передача массив в метод

В Java массивы передаются в методы по ссылке. Это означает, что метод получает ссылку на массив, а не его копию. Изменения, внесённые в массив внутри метода, отражаются на оригинальном массиве.

15 Классы. Объявление классов. Конструктор. Неизменяемый класс

Классы в Java являются основными строительными блоками для создания объектов. Они определяют структуру и поведение объектов, включая их свойства (поля) и методы.

Объявление классов

Класс объявляется с помощью ключевого слова `class`, за которым следует имя класса и тело класса, заключённое в фигурные скобки.

```
public class ClassName {
    // Поля (переменные)
    private int field1;
    private String field2;

    // Конструктор
    public ClassName(int field1, String field2) {
        this.field1 = field1;
        this.field2 = field2;
    }

    // Методы
    public void display() {
        System.out.println("Field1: " + field1 + ", Field2: " + field2);
    }
}
```

Конструктор

Конструктор — это специальный метод, который вызывается при создании объекта класса. Он имеет то же имя, что и класс, и не имеет возвращаемого типа. Конструктор используется для инициализации объектов.

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

    // Конструктор
    public Person(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }

    // Метод для отображения информации
    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

Неизменяемый класс

Неизменяемый класс — это класс, объекты которого не могут быть изменены после их создания. Для создания неизменяемого класса необходимо:

1. Сделать все поля приватными и финальными.
2. Предоставить публичный конструктор для инициализации всех полей.
3. Не предоставлять методов, которые изменяют состояние объекта.
4. Предоставить публичные методы для доступа к значениям полей.

```
public final class ImmutableClass {
    private final int field1;
    private final String field2;

    // Конструктор
    public ImmutableClass(int field1, String field2) {
        this.field1 = field1;
        this.field2 = field2;
    }

    // Методы доступа
    public int getField1() {
        return field1;
    }

    public String getField2() {
        return field2;
```

```
    }
}
```

Принципы ООП классов в Java

- **Инкапсуляция**: Классы позволяют скрывать внутреннюю реализацию и предоставлять интерфейс для взаимодействия с объектами.
- **Наследование**: Классы могут наследовать свойства и методы от других классов.
- **Полиморфизм**: Объекты могут быть использованы в различных контекстах, что даёт гибкость в программировании.

16 Отличие класса и объекта. Создание экземпляров класса. Особенности перегруженных конструкторов

- **Класс** — это шаблон или чертеж для создания объектов. Класс не занимает память для хранения данных, он только описывает, как должны выглядеть объекты.
- **Объект** — это экземпляр класса. Он представляет собой конкретную реализацию класса и занимает память для хранения данных.

Создание экземпляров класса

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

Особенности перегруженных конструкторов

Перегрузка конструкторов позволяет создавать несколько конструкторов в одном классе с разными параметрами.

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

```
    // Конструктор без параметров
    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    // Конструктор с одним параметром
    public Person(String name) {
        this.name = name;
        this.age = 0;
    }

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

    // Метод для отображения информации
    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

17 Статические методы. Перегрузка методов. Объявление констант

Статические методы принадлежат классу, а не экземпляру класса. Они могут быть вызваны без создания объекта класса.

Перегрузка методов

Перегрузка методов (method overloading) позволяет создавать несколько методов с одним и тем же именем, но с разными параметрами. Это полезно для предоставления различных способов выполнения одной и той же операции.

Объявление констант

Константы — это переменные, значения которых не могут быть изменены после их инициализации. В Java константы объявляются с помощью ключевого слова `final`.

Особенности статических методов, перегрузки методов и констант

- **Статические методы**:
  - Принадлежат классу, а не экземпляру класса.
  - Могут быть вызваны без создания объекта класса.
  - Часто используются для утилитарных функций и операций, которые не зависят от состояния объекта.
- **Перегрузка методов**:
  - Позволяет создавать несколько методов с одним и тем же именем, но с разными параметрами.
  - Удобна для предоставления различных способов выполнения одной и той же операции.
- **Константы**:
  - Объявляются с помощью ключевого слова `final`.
  - Значения констант не могут быть изменены после их инициализации.

18 Классы. Объявление классов. Ссылка this

```
public class Person {
    // Поля (переменные)
    private String name;
    private int age;

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

```
    // Метод для отображения информации
    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

Использование ссылки this

Ссылка this используется для ссылки на текущий объект внутри метода или конструктора. Она полезна для разрешения конфликтов имен между полями класса и параметрами методов или конструкторов.

19 Инкапсуляция. Модификаторы доступа. Понятие геттеров и сеттеров

Инкапсуляция — это один из ключевых принципов объектно-ориентированного программирования (ООП), который позволяет скрывать внутреннюю реализацию объекта и предоставлять доступ к его данным только через определенные методы. Это помогает защитить данные от некорректного использования и обеспечивает контроль над доступом к данным.

Пример инкапсуляции

```
public class Person {
    // Приватные поля
    private String name;
    private int age;

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

    // Геттер для поля name
    public String getName() {
        return name;
    }
```

```
    // Сеттер для поля name
    public void setName(String name) {
        this.name = name;
    }

    // Геттер для поля age
    public int getAge() {
        return age;
    }

    // Сеттер для поля age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}
```

Модификаторы доступа

Модификаторы доступа определяют уровень доступа к членам класса (полям и методам). В Java существует четыре уровня доступа:

- **private**: Доступ только внутри класса.
- **default** (без модификатора): Доступ внутри пакета.
- **protected**: Доступ внутри пакета и в подклассах.
- **public**: Доступ отовсюду.

Геттеры и сеттеры

Геттеры и сеттеры — это методы, которые предоставляют доступ к приватным полям класса. Геттеры используются для получения значений полей, а сеттеры — для установки значений полей.

20 Наследование. Особенности реализации в Java. Переопределение методов

Наследование в Java реализуется с помощью ключевого слова `extends`.

Переопределение методов (method overriding) позволяет подклассу предоставить специфическую реализацию метода, который уже определен в суперклассе. Это полезно для изменения или расширения поведения метода в подклассе.

21 Наследование. Особенности реализации в Java. Ссылка super

Ссылка `super` в Java используется для обращения к членам суперкласса (базового класса) из подкласса (производного класса). Она позволяет вызывать конструкторы, методы и доступ к полям суперкласса. Это особенно полезно для переопределения методов и конструкторов, а также для доступа к скрытым полям суперкласса.

```
public class Animal {
    private String name;

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

    // Метод суперкласса
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

public class Dog extends Animal {
    // Конструктор подкласса
    public Dog(String name) {
        super(name); // Вызов конструктора суперкласса
    }

    // Переопределение метода
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

22 Полиморфизм. Ссылочный полиморфизм. Объявление массивов ссылочного типа

Полиморфизм — это один из ключевых принципов объектно-ориентированного программирования (ООП), который позволяет объектам разных классов быть обработанными через единый интерфейс. В Java полиморфизм достигается с помощью наследования и интерфейсов.

Ссылочный полиморфизм

Ссылочный полиморфизм (или полиморфизм времени выполнения) позволяет ссылке суперкласса указывать на объект подкласса. Это означает, что методы подкласса могут быть вызваны через ссылку суперкласса.

Объявление массивов ссылочного типа

Массивы ссылочного типа в Java позволяют хранить ссылки на объекты. Это полезно для работы с коллекциями объектов одного типа или его подтипов.

```
public class Animal {
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

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

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

public class Main {
```

```
    public static void main(String[] args) {
        Animal[] animals = new Animal[3];
        animals[0] = new Animal();
        animals[1] = new Dog();
        animals[2] = new Cat();

        for (Animal animal : animals) {
            animal.makeSound();
        }
    }
}
```

23 Полиморфизм. Полиморфизм по методам. Переопределение методов

Полиморфизм по методам позволяет объектам разных классов быть обработанными через единый интерфейс. Это достигается с помощью переопределения методов в подклассах. Методовый полиморфизм позволяет вызывать переопределенные методы подклассов через ссылки суперкласса.

Переопределение методов

Переопределение методов (method overriding) позволяет подклассу предоставить специфическую реализацию метода, который уже определен в суперклассе. Это полезно для изменения или расширения поведения метода в подклассе.

```
public class Animal {
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

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

```
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.makeSound(); // Вывод: Bark
        myCat.makeSound(); // Вывод: Meow
    }
}
```

24 Абстрактные классы. Абстрактные метод

Абстрактный класс — это класс, который не может быть инстанцирован напрямую. Он может содержать как абстрактные методы (методы без реализации), так и конкретные методы (методы с реализацией). Абстрактные классы используются для определения общих характеристик и поведения, которые должны быть реализованы в подклассах.

Абстрактный метод — это метод, который объявлен в абстрактном классе, но не имеет реализации. Абстрактные методы должны быть реализованы в подклассах.

```
    abstract class Animal {
    // Абстрактный метод
    public abstract void makeSound();

    // Конкретный метод
    public void sleep() {
        System.out.println("This animal is sleeping.");
    }
}

class Dog extends Animal {
```

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

class Cat extends Animal {
    // Реализация абстрактного метода
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

25 Интерфейс. Интерфейсные ссылки. Стандартные интерфейсы

Интерфейс в Java — это ссылочный тип, подобный классу, который используется для определения набора методов, которые класс должен реализовать. Интерфейсы не могут содержать реализации методов (до Java 8), но могут содержать константы и методы по умолчанию (начиная с Java 8).

Определение интерфейса

```
public interface Animal {
    void makeSound();
    void sleep();
}
```

Реализация интерфейса

Класс, реализующий интерфейс, должен предоставить реализации всех методов, объявленных в интерфейсе.

```
public class Dog implements Animal {
    @Override
    public void makeSound() {
```

```
        System.out.println("Woof!");
    }

    @Override
    public void sleep() {
        System.out.println("Dog is sleeping.");
    }
}
```

Интерфейсные ссылки

Интерфейсные ссылки позволяют создавать переменные типа интерфейса, которые могут ссылаться на любой объект, реализующий этот интерфейс.

```
Animal myDog = new Dog();
myDog.makeSound(); // Вывод: Woof!
myDog.sleep();     // Вывод: Dog is sleeping.
```

Стандартные интерфейсы

Java предоставляет множество стандартных интерфейсов, которые часто используются в разработке программного обеспечения. Некоторые из них включают:

- `java.lang.Comparable`: используется для сравнения объектов.
- `java.util.Iterator`: используется для итерации по коллекциям.
- `java.util.List`: интерфейс для списков.
- `java.util.Set`: интерфейс для множеств.
- `java.util.Map`: интерфейс для отображений (ключ-значение).

26 Исключения. Виды исключений. Объявление и выброс исключений

Исключения в Java используются для обработки ошибок и необычных ситуаций, которые могут возникнуть во время выполнения программы.

Исключения в Java делятся на две основные категории: проверяемые (checked) и непроверяемые (unchecked).

Проверяемые исключения — это исключения, которые должны быть обработаны или объявлены в методе. Они наследуются от класса `Exception`.

- `IOException`
- `SQLException`
- `FileNotFoundException`

Непроверяемые исключения (Unchecked Exceptions)

Непроверяемые исключения — это исключения, которые не обязательно обрабатывать или объявлять. Они наследуются от класса `RuntimeException`.

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`

Метод может объявлять, что он выбрасывает исключения, используя ключевое слово `throws`.

```
public void readFile(String fileName) throws IOException {
    // Код для чтения файла
}
```

Исключения могут быть выброшены с помощью ключевого слова `throw`.

```
public void checkAge(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Age must be 18 or older");
    }
}
```

Создание пользовательских исключений

Пользовательские исключения могут быть созданы путем наследования от класса `Exception` или `RuntimeException`.

```
public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

public void customMethod() throws CustomException {
    throw new CustomException("This is a custom exception");
}
```

27 Исключения. Виды исключений. Перехват исключений

Проверяемые исключения (Checked Exceptions)

Проверяемые исключения должны быть обработаны или объявлены в методе.

- `IOException` — базовый класс для исключений ввода-вывода.
- `FileNotFoundException` — выбрасывается, когда файл не найден.
- `SQLException` — выбрасывается при ошибках в работе с базой данных.
- `ClassNotFoundException` — выбрасывается, когда класс не найден.
- `InstantiationException` — выбрасывается, когда не удается создать экземпляр класса.
- `InterruptedException` — выбрасывается, когда поток прерван.
- `NoSuchMethodException` — выбрасывается, когда метод не найден.
- `NoSuchFieldException` — выбрасывается, когда поле не найдено.
- `MalformedURLException` — выбрасывается при некорректном URL.
- `ParseException` — выбрасывается при ошибках парсинга.

Непроверяемые исключения (Unchecked Exceptions)

Непроверяемые исключения не обязательно обрабатывать или объявлять.

- `NullPointerException` — выбрасывается при попытке доступа к объекту через null-ссылку.
- `ArrayIndexOutOfBoundsException` — выбрасывается при попытке доступа к несуществующему индексу массива.
- `ArithmeticException` — выбрасывается при арифметических ошибках, таких как деление на ноль.
- `IllegalArgumentException` — выбрасывается при передаче недопустимого аргумента методу.
- `IllegalStateException` — выбрасывается, когда метод вызывается в недопустимом состоянии.
- `IndexOutOfBoundsException` — базовый класс для исключений, связанных с индексами.
- `NumberFormatException` — выбрасывается при некорректном преобразовании строки в число.
- `UnsupportedOperationException` — выбрасывается, когда метод не поддерживается.
- `ClassCastException` — выбрасывается при некорректном приведении типов.
- `ConcurrentModificationException` — выбрасывается при изменении коллекции во время итерации.

Исключения могут быть обработаны с помощью блока `try-catch`.

```
public void readFile(String fileName) {
    try {
        // Код для чтения файла
    } catch (IOException e) {
        System.out.println("An error occurred: " + e.getMessage());
    }
}
```

28 Обобщения. Обобщенный класс. Обобщенный метод

Обобщения в Java позволяют создавать классы, интерфейсы и методы, которые работают с любыми типами данных, обеспечивая при этом типовую безопасность и устраняя необходимость в приведении типов.

Обобщенный класс позволяет использовать параметры типа для определения типов данных, с которыми класс будет работать.

Пример обобщенного класса

```
public class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

Обобщенный метод

Обобщенный метод позволяет использовать параметры типа для определения типов данных, с которыми метод будет работать.

```
public class Util {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Обобщенные интерфейсы

Обобщённые интерфейсы позволяют использовать параметры типа для определения типов данных, с которыми интерфейс будет работать.

```
public interface Pair<K, V> {
    void setKey(K key);
    K getKey();
    void setValue(V value);
    V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    @Override
    public void setKey(K key) {
        this.key = key;
    }

    @Override
    public K getKey() {
        return key;
    }

    @Override
    public void setValue(V value) {
        this.value = value;
    }

    @Override
    public V getValue() {
        return value;
    }
}
```

29 Класс ArrayList. Описание, основные методы. Примеры использования

Класс `ArrayList` в Java является частью Java Collections Framework и реализует интерфейс `List`. Он представляет собой динамический массив, который может изменять свой размер по мере добавления или удаления элементов. `ArrayList` позволяет хранить дубликаты и поддерживает до-

ступ к элементам по индексу.

Основные методы

- `add(E e)`: Добавляет элемент в конец списка.
- `get(int index)`: Возвращает элемент по указанному индексу.
- `set(int index, E element)`: Заменяет элемент по указанному индексу новым элементом.
- `remove(int index)`: Удаляет элемент по указанному индексу.
- `size()`: Возвращает количество элементов в списке.
- `isEmpty()`: Проверяет, пуст ли список.
- `contains(Object o)`: Проверяет, содержится ли указанный элемент в списке.

Примеры использования

Создание и добавление элементов

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        System.out.println(list); // Вывод: [Apple, Banana, Cherry]
    }
}
```

Доступ к элементам

```
public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
```

```
        list.add("Cherry");

        String firstElement = list.get(0);
        System.out.println(firstElement); // Вывод: Apple
    }
}
```

Удаление элементов

```
public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        list.remove(1);
        System.out.println(list); // Вывод: [Apple, Cherry]
    }
}
```

30 Класс HashMap. Описание, основные методы. Примеры использования

Класс `HashMap` в Java является частью Java Collections Framework и реализует интерфейс `Map`. Он представляет собой структуру данных, которая хранит пары "ключ-значение". `HashMap` не гарантирует порядок элементов и допускает наличие одного `null` ключа и множества `null` значений.

Основные методы

- `put(K key, V value)`: Добавляет пару "ключ-значение"в карту.
- `get(Object key)`: Возвращает значение, связанное с указанным ключом.
- `remove(Object key)`: Удаляет пару "ключ-значение"по указанному ключу.

- `containsKey(Object key)`: Проверяет, содержится ли указанный ключ в карте.
- `containsValue(Object value)`: Проверяет, содержится ли указанное значение в карте.
- `size()`: Возвращает количество элементов в карте.
- `isEmpty()`: Проверяет, пуста ли карта.
- `keySet()`: Возвращает набор всех ключей в карте.
- `values()`: Возвращает коллекцию всех значений в карте.
- `entrySet()`: Возвращает набор всех пар "ключ-значение"в карте.

Создание и добавление элементов

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Apple", 1);
        map.put("Banana", 2);
        map.put("Cherry", 3);

        System.out.println(map); // Вывод: {Apple=1, Banana=2, Cherry=3}
    }
}
```

Доступ к элементам

```
public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Apple", 1);
        map.put("Banana", 2);
        map.put("Cherry", 3);

        int value = map.get("Banana");
        System.out.println(value); // Вывод: 2
    }
}
```

31 Stream API. Описание. Возможности. Примеры использования

Stream API была введена в Java 8 и предоставляет мощный способ обработки последовательностей элементов. Stream API позволяет выполнять операции над коллекциями данных в функциональном стиле, что делает код более читаемым и лаконичным. Stream API поддерживает как последовательную, так и параллельную обработку данных.

Возможности

- **Фильтрация**: Отбор элементов, удовлетворяющих определенному условию.
- **Преобразование**: Применение функции к каждому элементу потока.
- **Сортировка**: Упорядочивание элементов потока.
- **Агрегация**: Сведение элементов потока к одному значению (например, сумма, среднее значение).
- **Параллельная обработка**: Выполнение операций над потоком в нескольких потоках для улучшения производительности.

Примеры использования

Фильтрация и преобразование

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamAPIExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "cherry", "date");

        List<String> filteredWords = words.stream()
            .filter(word -> word.startsWith("b"))
            .map(String::toUpperCase)
            .collect(Collectors.toList());
```

```
        System.out.println(filteredWords); // Вывод: [BANANA]
    }
}
```

Сортировка

```
public class StreamAPIExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "cherry", "date");

        List<String> sortedWords = words.stream()
            .sorted()
            .collect(Collectors.toList());

        System.out.println(sortedWords); // Вывод: [apple, banana, cherry, date]
    }
}
```

Агрегация

```
import java.util.Arrays;
import java.util.List;

public class StreamAPIExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        int sum = numbers.stream()
            .reduce(0, Integer::sum);

        System.out.println(sum); // Вывод: 15
    }
}
```

Параллельная обработка

```
public class StreamAPIExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        int sum = numbers.parallelStream()
            .reduce(0, Integer::sum);
```

```
        System.out.println(sum); // Вывод: 15
    }
}
```

32 Ключевое слово final. Применение в различных контекстах

Final переменные

Переменные, объявленные как `final`, не могут быть изменены после их инициализации.

```
public class FinalVariableExample {
    public static void main(String[] args) {
        final int CONSTANT = 10;
        // CONSTANT = 20; // Ошибка: нельзя изменить значение final переменной
        System.out.println(CONSTANT); // Вывод: 10
    }
}
```

Final методы

Методы, объявленные как `final`, не могут быть переопределены в подклассах.

```
class Parent {
    final void display() {
        System.out.println("Display method in Parent class");
    }
}

class Child extends Parent {
    // void display() { // Ошибка: нельзя переопределить final метод
    //     System.out.println("Display method in Child class");
    // }
}
```

Final классы

Классы, объявленные как `final`, не могут быть унаследованы.

```
final class FinalClass {
    void display() {
        System.out.println("Display method in FinalClass");
    }
}

// class SubClass extends FinalClass { // Ошибка: нельзя наследовать final класс
// }
```

Final параметры

Параметры методов, объявленные как `final`, не могут быть изменены внутри метода.

```
public class FinalParameterExample {
    void display(final int value) {
        // value = 20; // Ошибка: нельзя изменить значение final параметра
        System.out.println(value); // Вывод: 10
    }

    public static void main(String[] args) {
        FinalParameterExample example = new FinalParameterExample();
        example.display(10);
    }
}
```

33 Особенности сравнения примитивных типов данных и объектов

Примитивные типы данных в Java (например, `int`, `double`, `char`) хранят свои значения непосредственно в стеке. Сравнение примитивных типов данных выполняется с использованием операторов ==, !=, <, >, <=, >=.

Объекты в Java хранят ссылки на данные в куче. Сравнение объектов с использованием операторов == и != проверяет равенство ссылок, а не содержимого объектов. Для сравнения содержимого объектов используется метод `equals`.

Сравнение ссылок

```
public class ObjectComparisonExample {
```

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

        System.out.println(str1 == str2); // Вывод: false (разные ссылки)
    }
}
```

Сравнение содержимого

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

        System.out.println(str1.equals(str2)); // Вывод: true (одинаковое содерж
    }
}
```

Переопределение метода equals

Для корректного сравнения содержимого объектов необходимо переопределить метод `equals` в классе.

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

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

    @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 && name.equals(person.name);
    }
}
```

```
public class EqualsOverrideExample {
    public static void main(String[] args) {
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Alice", 30);

        System.out.println(person1.equals(person2)); // Вывод: true
    }
}
```

34 Сравнительная характеристика статических и нестатических методов

Статические методы	**Нестатические методы**
Принадлежат классу, а не экземпляру класса.	Принадлежат экземпляру класса.
Могут быть вызваны без создания экземпляра класса.	Требуют создания экземпляра класса для вызова.
Не могут напрямую обращаться к нестатическим полям или методам.	Могут обращаться к статическим и нестатическим полям и методам.
Используются для выполнения задач, которые не зависят от состояния объекта.	Используются для выполнения задач, которые зависят от состояния объекта.

35 Сравнительная характеристика массивов и ArrayList

Массивы	**ArrayList**
Фиксированный размер, определяется при создании.	Динамический размер, может изменяться по мере добавления или удаления элементов.
Тип элементов должен быть указан при создании.	Может хранить элементы любого типа, включая `null`.
Доступ к элементам по индексу.	Доступ к элементам по индексу.
Не предоставляет встроенных методов для добавления или удаления элементов.	Предоставляет встроенные методы для добавления (`add`), удаления (`remove`) и других операций.
Быстрее в доступе к элементам по индексу.	Медленнее в доступе к элементам по индексу из-за дополнительных проверок и методов.
Не является частью Java Collections Framework.	Является частью Java Collections Framework и реализует интерфейс `List`.
Пример: `int[] array = new int[10];`	Пример: `ArrayList<Integer> list = new ArrayList<>();`

36 Сравнительная характеристика примитивных и ссылочных типов данных. Особенности передачи параметров в методах

Примитивные типы данных	**Ссылочные типы данных**
Хранят значения непосредственно в стеке.	Хранят ссылки на объекты в куче.
Примеры: `int`, `double`, `char`, `boolean`.	Примеры: `String`, `ArrayList`, `Integer`, `Object`.
Имеют фиксированный размер.	Размер может варьироваться в зависимости от содержимого объекта.
Быстрее в доступе и операциях.	Медленнее в доступе и операциях из-за необходимости работы с кучей.
Не могут быть `null`.	Могут быть `null`.
Передача параметров в методы по значению.	Передача параметров в методы по ссылке.

Особенности передачи параметров в методах

Передача примитивных типов

Примитивные типы данных передаются в методы по значению. Это означает, что **копия** значения передается в метод, и изменения внутри метода не влияют на оригинальное значение.

Передача ссылочных типов

Ссылочные типы данных передаются в методы по ссылке. Это означает, что передается копия ссылки на объект, и изменения внутри метода могут влиять на оригинальный объект, если изменяются его поля или состояние.

Передача неизменяемых объектов

Некоторые ссылочные типы, такие как `String`, являются неизменяемыми. Это означает, что их содержимое не может быть изменено после создания. Передача таких объектов в методы также происходит по ссылке, но изменения внутри метода не влияют на оригинальный объект.

37 Классификация циклов. Ключевые слова для перехода на следующую итерацию и досрочное завершение цикла

Цикл `for` используется для выполнения блока кода определенное количество раз. Он состоит из трех частей: инициализации, условия и обновления.

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

Цикл `while` выполняет блок кода до тех пор, пока условие истинно. Условие проверяется перед каждой итерацией.

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

Цикл `do-while` аналогичен циклу `while`, но условие проверяется после выполнения блока кода, что гарантирует выполнение блока кода хотя бы один раз.

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

Цикл `for-each` используется для итерации по элементам массива или коллекции. Он не требует явного управления индексом.

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

Ключевые слова для управления циклами

`break`

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

`continue`

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue;
    }
    System.out.println(i);
}
```

38 Функциональные интерфейсы. Лямбда-функции. Варианты объявлений

Функциональный интерфейс в Java — это интерфейс, который содержит **ровно один абстрактный метод**. Функциональные интерфейсы могут иметь множество методов по умолчанию или статических методов, но только один абстрактный метод. Они используются как основа для лямбда-выражений и методов ссылок.

```
@FunctionalInterface
interface MyFunctionalInterface {
    void myMethod();
}
```

Лямбда-функции

Лямбда-функции (или лямбда-выражения) позволяют передавать функциональность в виде аргументов методов или хранить их в переменных. Лямбда-выражения предоставляют способ определения анонимных функций прямо в месте их использования.

Синтаксис лямбда-выражений

Лямбда-выражения имеют следующий синтаксис:

```
(parameters) -> expression
```

или

```
(parameters) -> { statements; }
```

Примеры лямбда-выражений

Без параметров

```
MyFunctionalInterface func = () -> System.out.println("Hello, World!");
func.myMethod(); // Вывод: Hello, World!
```

С одним параметром

```
@FunctionalInterface
interface StringProcessor {
    String process(String str);
}

StringProcessor processor = (str) -> str.toUpperCase();
System.out.println(processor.process("hello")); // Вывод: HELLO
```

С несколькими параметрами

```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

MathOperation addition = (a, b) -> a + b;
System.out.println(addition.operate(5, 3)); // Вывод: 8
```

С блоком кода

```
MathOperation multiplication = (a, b) -> {
    int result = a * b;
    return result;
};
System.out.println(multiplication.operate(5, 3)); // Вывод: 15
```

Варианты объявлений

Анонимный класс

```
MyFunctionalInterface func = new MyFunctionalInterface() {
    @Override
    public void myMethod() {
        System.out.println("Hello, World!");
    }
};
func.myMethod(); // Вывод: Hello, World!
```

Лямбда-выражение

```
MyFunctionalInterface func = () -> System.out.println("Hello, World!");
func.myMethod(); // Вывод: Hello, World!
```

Метод ссылки

Метод ссылки позволяет ссылаться на существующий метод или конструктор.

```
@FunctionalInterface
interface Printer {
    void print(String message);
}

Printer printer = System.out::println;
printer.print("Hello, World!"); // Вывод: Hello, World!
```

39 Функциональные интерфейсы. Лямбда-функции. Стандартные функциональные интерфейсы

`Predicate<T>`

Интерфейс `Predicate<T>` представляет предикат (условие) с одним аргументом.

```
import java.util.function.Predicate;
```

```
Predicate<String> isEmpty = String::isEmpty;
System.out.println(isEmpty.test("")); // Вывод: true
System.out.println(isEmpty.test("Hello")); // Вывод: false
```

Function<T, R>

Интерфейс `Function<T, R>` представляет функцию, которая принимает один аргумент и возвращает результат.

```
import java.util.function.Function;

Function<String, Integer> length = String::length;
System.out.println(length.apply("Hello")); // Вывод: 5
```

Consumer<T>

Интерфейс `Consumer<T>` представляет операцию, которая принимает один аргумент и не возвращает результата.

```
import java.util.function.Consumer;

Consumer<String> printer = System.out::println;
printer.accept("Hello, World!"); // Вывод: Hello, World!
```

Supplier<T>

Интерфейс `Supplier<T>` представляет поставщика результатов без аргументов.

```
import java.util.function.Supplier;

Supplier<String> greeting = () -> "Hello, World!";
System.out.println(greeting.get()); // Вывод: Hello, World!
```

BinaryOperator<T>

Интерфейс `BinaryOperator<T>` представляет операцию над двумя аргументами одного типа, возвращающую результат того же типа.

```
import java.util.function.BinaryOperator;

BinaryOperator<Integer> multiply = (a, b) -> a * b;
System.out.println(multiply.apply(3, 4)); // Вывод: 12
```

UnaryOperator<T>

Интерфейс `UnaryOperator<T>` представляет операцию над одним аргументом одного типа, возвращающую результат того же типа.

```
import java.util.function.UnaryOperator;

UnaryOperator<Integer> square = x -> x * x;
System.out.println(square.apply(4)); // Вывод: 16
```

BiFunction<T, U, R>

Интерфейс `BiFunction<T, U, R>` представляет функцию, которая принимает два аргумента и возвращает результат.

```
import java.util.function.BiFunction;

BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
System.out.println(add.apply(2, 3)); // Вывод: 5
```

BiConsumer<T, U>

Интерфейс `BiConsumer<T, U>` представляет операцию, которая принимает два аргумента и не возвращает результата.

```
import java.util.function.BiConsumer;

BiConsumer<String, String> concat = (a, b) -> System.out.println(a + b);
concat.accept("Hello, ", "World!"); // Вывод: Hello, World!
```

BiPredicate<T, U>

Интерфейс `BiPredicate<T, U>` представляет предикат (условие) с двумя аргументами.

```
import java.util.function.BiPredicate;

BiPredicate<String, String> equals = String::equals;
System.out.println(equals.test("Hello", "Hello")); // Вывод: true
System.out.println(equals.test("Hello", "World")); // Вывод: false
```

40 План реализации собственного исключения

Определение класса исключения

Создайте новый класс, который наследует `Exception` или один из его подклассов. Обычно собственные исключения наследуют `Exception` для проверяемых исключений или `RuntimeException` для непроверяемых исключений.

```
public class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }

    public MyCustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Конструкторы

Определите конструкторы для вашего исключения. Обычно это включает конструктор с сообщением об ошибке и конструктор с сообщением и причиной.

```
public class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }

    public MyCustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Дополнительные поля и методы

Если необходимо, добавьте дополнительные поля и методы для хранения информации, специфичной для вашего исключения.

```
public class MyCustomException extends Exception {
    private final int errorCode;

    public MyCustomException(String message, int errorCode) {
        super(message);
        this.errorCode = errorCode;
    }

    public MyCustomException(String message, Throwable cause, int errorCode) {
        super(message, cause);
        this.errorCode = errorCode;
    }

    public int getErrorCode() {
        return errorCode;
    }
}
```

Генерация исключения

Создайте условия, при которых будет генерироваться ваше исключение, и используйте ключевое слово `throw` для его выброса.

```
public class Example {
    public void performOperation(int value) throws MyCustomException {
        if (value < 0) {
            throw new MyCustomException("Value cannot be negative", 1001);
        }
        // Другие операции
    }
}
```

Обработка исключения

Используйте блок `try-catch` для обработки вашего исключения.

```
public class Main {
    public static void main(String[] args) {
        Example example = new Example();
        try {
            example.performOperation(-1);
```

```
        } catch (MyCustomException e) {
            System.out.println("Caught exception: " + e.getMessage());
            System.out.println("Error code: " + e.getErrorCode());
        }
    }
}
```