Java Exam 2025

1. Классы языка Java и их синтаксис. Члены класса. Статические члены класса.Поля и методы класса. Гла приложения. Конструкторы класса. Цепочкиконструкторов. Блоки инициализации. Статические поля и мет классов.Инициализация статических полей класса

这是基于第一题 的详细拆解。

1.1 Классы языка Java и их синтаксис

Java 语言的类及其语法

```java
public class Student {
    // Тело класса
    // 类的主体
}
```

- Объяснение (解释): Класс — это шаблон для создания объектов. Основной синтаксис включает ключевое слово `class`, имя класса и фигурные скобки. 类是创建对象的模板。基本语法包括关键字 `class`、类名和花括号。

2.2 Члены класса

类的成员

```java
public class Student {
    String name;          // Поле (成员变量)
    void study() { ... }  // Метод (成员方法)
}
```

- Объяснение (解释): Члены класса — это основные элементы, из которых состоит класс. К ним относятся поля (переменные) и методы (функции). 类的成员是构成类的基本元素。它们包括字段（变量）和方法（函数）。

2.3 Статические члены класса

类的静态成员

```java
public class Student {
    static String university = "MSU"; // Статическое поле (静态字段)

    static void showUniversity() {    // Статический метод (静态方法)
        System.out.println(university);
    }
}
```

- Объяснение (解释): Статические члены объявляются с ключевым словом `static`. Они принадлежат самому классу, а не экземплярам класса. 静态成员使用 `static` 关键字声明。它们属于类本身，而不是类的实例。

2.4 Поля и методы класса

类的字段和方法

```java
public class Student {
    int age; // Поле: хранит состояние (字段：存储状态)

    void grow() { // Метод: определяет поведение (方法：定义行为)
        age++;
    }
}
```

- Объяснение (解释): Поля хранит данные или состояние объекта. Методы определяют поведение объекта и могут изменять его состояние. 字段存储对象的数据或状态。方法定义对象的行为，并可以修改其状态。

2.5 Главный метод приложения

应用程序的主方法

```java
public static void main(String[] args) {
    // Точка входа в программу
    // 程序的入口点
}
```

- Объяснение (解释): Это точка входа для любого приложения Java. Он всегда должен быть `public`, `static`, `void` и принимать массив строк `String[]`. 这是任何 Java 应用程序的入口点。它必须始终是 `public`、`static`、`void`，并接收一个字符串数组 `String[]`。

2.6 Конструкторы класса

类的构造器

```java
public class Student {
    public Student() {
        // Код конструктора
        // 构造器代码
    }
}
```

- Объяснение (解释): Конструктор — это специальный блок кода, который вызывается при создании нового объекта (с помощью `new`). Он используется для инициализации объекта. 构造器是一个特殊的代码块，在创建新对象（使用 `new`）时被调用。它用于初始化对象。

2.7 Цепочки конструкторов

构造器链

```java
public class Student {
    String name;

    public Student() {
        this("Unknown"); // Вызов другого конструктора (调用另一个构造器)
    }

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


class Person {
    String name;

    // Конструктор родителя
    // 父类构造器
    Person(String name) {
        this.name = name;
        System.out.println("1. Parent Constructor: " + name);
    }
}

class Student extends Person {
    int id;

    // Конструктор потомка
    // 子类构造器
    Student(String name, int id) {
        super(name); // Вызов конструктора родителя (обязательно первая строка!)
                     // 调用父类构造器（必须是第一行！）
        this.id = id;
        System.out.println("2. Student Constructor");
    }
}
```

- Объяснение (解释): Это процесс, когда один конструктор вызывает другой конструктор того же класса (используя `this()`) или конструктор родительского класса (используя `super()`). 这是一个构造器调用同一个类的另一个构造器（使用 `this()`）或父类构造器（使用 `super()`）的过程。

2.8 Блоки инициализации

初始化块

```
public class Student {
    {
        System.out.println("Instance Block"); // Блок инициализации экземпляра (实例初始化块)
    }

    static {
        System.out.println("Static Block"); // Статический блок инициализации (静态初始化块)
    }
}
```

- Объяснение (解释): Это блоки кода в фигурных скобках внутри класса. Обычные блоки выполняются при создании каждого объекта, а статические — только один раз при загрузке класса. 这是类内部花括号中的代码块。普通块在每次创建对象时执行，而静态块在类加载时只执行一次。

2.9 Статические поля и методы классов

类的静态字段和方法

(注：这部分内容在题目中与第3点有重叠，但在考试中可能要求再次强调具体用途)

```
Math.PI;         // Статическое поле (静态字段)
Math.abs(-5);    // Статический метод (静态方法)
```

- Объяснение (解释): Эти элементы можно использовать, не создавая экземпляр класса. Обычно они используются для утилит или констант. 这些元素可以在不创建类实例的情况下使用。通常用于工具类或常量。

2.10 Инициализация статических полей класса

静态类字段的初始化

```
public class Config {
    static int timeout;

    static {
        // Сложная логика инициализации
        // 复杂的初始化逻辑
        timeout = 1000 * 60;
    }
}
```

- Объяснение (解释): Статические поля инициализируются при загрузке класса. Это можно сделать напрямую при объявлении или внутри статического блока инициализации. 静态字段在类加载时初始化。这可以在声明时直接完成，也可以在静态初始化块内完成。—

2. Пакеты классов и интерфейсов. Импорт классов и интерфейсов из другихпакетов. Разновидности видимо и интерфейсов. Соглашения обименовании пакетов, классов, директорий и файлов при программировании н

2.1 Пакеты классов и интерфейсов

类和接口的包

```
// Объявляем, что этот класс живет в пакете "ru.msu.utils"
// 声明该类位于 "ru.msu.utils" 包中
package ru.msu.utils;

public class MathTools {
    public void sayHello() {
        System.out.println("Привет из пакета utils! (来自 utils 包的问候！)");
    }
}
```

```
package ru.msu.main;

// !!! Ключевой момент: Импорт (关键点：导入) !!!
// Мы говорим Java: "Найди класс MathTools в пакете ru.msu.utils"
// 我们告诉 Java："去 ru.msu.utils 包里找 MathTools 这个类"
import ru.msu.utils.MathTools;

public class Application {
    public static void main(String[] args) {
        // Теперь мы можем использовать MathTools как обычно
        // 现在我们可以像往常一样使用 MathTools
        MathTools tools = new MathTools();
        tools.sayHello();
    }
}
```

- Объяснение (解释): Пакет — это пространство имен, которое группирует связанные классы и интерфейсы для предотвращения конфликтов имен и управления доступом. 包是一个命名空间，它将相关的类和接口组合在一起，以防止命名冲突并控制访问权限。

2.2 Импорт классов и интерфейсов из других пакетов

从其他包导入类和接口

```
package com.msu.exam;

import java.util.List;        // Импорт конкретного класса (导入具体类)
import java.util.Scanner;

public class Main {
    List<String> list;        // Можно использовать короткое имя (可以使用短名称)
}
```

- Объяснение (解释): Ключевое слово `import` позволяет использовать классы из других пакетов по их коротким именам, не указывая полный путь каждый раз. `import` 关键字允许按短名称使用其他包中的类，而无需每次都指定完整路径。

2.3 Разновидности видимости классов и интерфейсов

类和接口的可见性类型

```
// 1. Public: Виден везде (随处可见)
public class PublicClass { ... }

// 2. Package-Private (default): Виден только внутри пакета (仅包内可见)
class DefaultClass { ... }
```

- Объяснение (解释): У классов верхнего уровня есть только два уровня доступа: `public` (доступен из любого пакета) и пакетный (доступен только внутри своего пакета). 顶层类只有两个访问级别：`public`（可从任何包访问）和包级私有（仅在自己的包内可访问）。

2.4 Соглашения об именовании пакетов

包的命名约定

```
// Правильно (正确):
package ru.msu.cmc.exam;

// Неправильно (错误):
package RU.MSU.CMC.Exam;
```

- Объяснение (解释): Имена пакетов всегда записываются строчными (маленькими) буквами, чтобы избежать конфликтов с именами классов или интерфейсов. 包名总是用小写字母书写，以避免与类名或接口名冲突。

2.5 Соглашения об именовании классов

类的命名约定

```
// UpperCamelCase (PascalCase)
public class ExamQuestion { ... }
public class StudentRecord { ... }
```

- Объяснение (解释): Имена классов должны быть существительными и записываться в стиле UpperCamelCase (каждое слово с большой буквы). 类名应该是名词，并采用 UpperCamelCase 风格（每个单词首字母大写）书写。

2.6 Соглашения об именовании директорий и файлов

目录和文件的命名约定

```
Файл (File): Student.java
|-- Класс (Class): public class Student { ... }
```

- Объяснение (解释): Имя файла с исходным кодом должно в точности совпадать с именем публичного класса, который в нем находится (включая регистр букв). 源代码文件的名称必须与其中包含的公共类的名称完全匹配（包括大小写）。

---

3. Наследование полей и методов классов в языке Java. Перекрытие наследуемых методов. Использование конструкторов наследуемых классов. Разновидности видимости членов классов. Полиморфизм. Абстрактн Конечные (final) классы.

3.1 Наследование полей и методов классов в языке Java

```java
class Parent {
    public void publicField = "Public Field"; // доступно везде
    protected void protectedField = "Protected
        Field"; // доступно в подклассах и в том же пакете
    void defaultField = "Default
        Field"; // package-private - только в этом пакете
    private void privateField = "Private Field"; // только в этом
        классе

    // Метод класса Parent
    public void PublicMethod() {}
    protected void ProtectedMethod() {}
    void DefaultMethod() {}
    private void PrivateMethod() {}
}

class Child extends Parent {
    public void test(){
        // Доступ к унаследованным полям и методам
        System.out.println(publicField); // Доступно
        System.out.println(protectedField); // Доступно

        System.out.println(defaultField); // Доступно (если в том же
        пакете)
        // System.out.println(privateField); // Ошибка: недоступно

        PublicMethod(); // Доступно
        ProtectedMethod(); // Доступно
        DefaultMethod(); // Доступно (если в том же пакете)
        // PrivateMethod(); // Ошибка: недоступно
    }
}
```

3.2 Перекрытие наследуемых методов

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

    public final void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog extends Aniaml {
    @Override
    public void makeSound() { // Перекрытие метода
        System.out.println("Dog Bark");
    }

    //можно вызвать родительский метод
    public void MakeSoundAndParentSound() {
        super.makeSound(); // Вызов метода родителя
        this.makeSound(); // Вызов перекрытого метода
    }

    // Ошибка: нельзя переопределить final метод
    // @Override
    // public void sleep() {
    //     System.out.println("Dog is sleeping");
    // }
}
```

3.3 Использование конструкторов наследуемых классов

```java
class Animal {
    private String name;
    private int age;

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

class Dog extends Animal {
    private String breed;

    public Dog() {
        this.breed = "Unknown";
    }

    public Dog(String name, int age) {
        super(name, age); // Вызов конструктора родителя
        this.breed = "Unknown";
    }
    public Dog(String name, int age, String breed) {
        super(name, age); // Вызов конструктора родителя - должен
        быть первой строкой
        this.breed = breed;
    }
}
```

3.4 Разновидности видимости членов классов.

```java
package ru.example.package1;

public class VisibilityDemo {
    public int publicVar = 1;          // Доступно везде
    protected int protectedVar = 2;    // Доступно в подклассах и в
        том же пакете
    int defaultVar =
            3;                  // package-private - только в этом пакете
    private int privateVar = 4;        // только в этом классе

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

```java
package ru.example.package2;

import ru.example.package1.VisibilityDemo;

// Класс в другом пакете, но подкласс
public class SubClass extends VisibilityDemo {
    publicVar = 10;          // Доступно
    protectedVar = 20;       // Доступно
    // defaultVar = 30;      // Ошибка: недоступно
    // privateVar = 40;      // Ошибка: недоступно

    publicMethod();          // Доступно
    protectedMethod();       // Доступно
    // defaultMethod();     // Ошибка: недоступно
    // privateMethod();     // Ошибка: недоступно

}

// Класс в другом пакете, не подкласс
public class OtherClassDiffPackage {
    public void test() {
        VisibilityDemo demo = new VisibilityDemo();
        demo.publicVar = 10;           // Доступно
        // demo.protectedVar = 20;     // Ошибка: недоступно
        // demo.defaultVar = 30;       // Ошибка: недоступно
        // demo.privateVar = 40;       // Ошибка: недоступно

        demo.publicMethod();          // Доступно
        // demo.protectedMethod();     // Ошибка: недоступно
        // demo.defaultMethod();       // Ошибка: недоступно
        // demo.privateMethod();       // Ошибка: недоступно
    }
}
```

Модификатор	Класс	Пакет	Подкласс (другой пакет)	Все остальные
public	Да	Да	Да	Да
protected	Да	Да	Да	Нет
default	Да	Да	Нет	Нет
private	Да	Нет	Нет	Нет

3.5 Полиморфизм

```java
// 1. Перекрытие методов (Overriding)
class Animal {
    public void makeSound() {}
}
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}

// 2. Перегрузка методов (Overloading)
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public int add (int a, int b, int c) {
        return a + b + c;
    }
    public double add(double a, double b) {
        return a + b;
```

```
    }
}

// 3. Полиморфизм через интерфейсы
interface CanFly {
    void fly();
}
class Bird implements CanFly {
    @Override
    public void fly() {
        System.out.println("Bird is flying");
    }
}
class Airplane implements CanFly {
    @Override
    public void fly() {
        System.out.println("Airplane is flying");
    }
}
```

3.6 Абстрактные классы

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

    // Обычный метод (с реализацией)
    public void sleep() {
        System.out.println("Animal is sleeping");
    }

    // Статический метод в абстрактном классе
    public static void staticMethod() {
        System.out.println("Static method in abstract class");
    }

    //конструктор абстрактного класса
    public Animal() {
        System.out.println("Constructor of Abstract Animal");
    }

    // Поля абстрактного класса
    protected String name;
}

class Dog extends Animal {
    // конкретная клфасс должна реализовать абстрактные методы
    @Override
    public void makeSound() {
        System.out.println("Dog Bark");
    }

    // можно иметь собственные поля и методы
    public void showName() {
        System.out.println("Dog's name is " + name);
    }
}
```

3.7 Конечные (final) классы

```
// 1. final класс нельзя наследовать
final class Animal {
    public void makeSound() {...}
}

// Ошибка: нельзя наследовать final класс
// class Dog extends Animal {

// 2. final класс можно наследовать
final class Dog extends Mammal {
    // можно иметь обычные методы
    public void bark() {...}

    // можно иметь final методы – их нельзя переопределять в
        подклассах
    public final void finalMethod() {...}
}
```

4. Вложенность классов. Статические вложенные и внутренние классы. Доступ к статическим вложенным Внутренние классы. Доступ к внутренним классам. Внутренние локальные классы. Внутренние анонимные

4.1 Статические вложенные классы

```
class Animal {
    private String name = "Animal";
    static String dogType = "Bulldog";

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

    // Статический вложенный класс
    static class DogToy {
        private String toyName;

        public DogToy(String toyName) {
            this.toyName = toyName;
        }

        public void play(){
            // Доступ к статическому полю внешнего класса
            System.out.println("Playing with " + toyName +
    " of type " + dogType);
            // Ошибка: нельзя обратиться к нестатическому полю
    внешнего класса
            // System.out.println("Animal name is " + name);
        }

        public static void showDogType() {
            System.out.println("Dog type is " + dogType);
        }
    }
}
```

4.2 Доступ к статическим вложенным классам.

```
public class Main {
    public static void main(String[] args) {
        // Создание экземпляра статического вложенного класса
        Animal.DogToy toy = new Animal.DogToy("Bone");
        toy.play();

        // Вызов статического метода вложенного класса
        Animal.DogToy.showDogType();
    }
}
```

4.3 Внутренние классы

```
class Dog {
    private String name;
    private String breed;

    public Dog(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }

    // Внутренний класс
    class DogCollor{
        private String color;
        private String size;

        public DogCollor(String color, String size) {
            this.color = color;
            this.size = size;
        }

        public void showInfo() {
            // Доступ к полям внешнего класса
            System.out.println("Dog Name: " + name + ", Breed: " +
    breed + ", Color: " + color + ", Size: " + size);
        }
    }
}
```

4.4 Доступ к внутренним классам.

```
public class Main {
    public static void main(String[] args) {
        // Сначала создаем экземпляр внешнего класса
        Dog dog = new Dog("Buddy", "Golden Retriever");
        // Затем создаем экземпляр внутреннего класса
        Dog.DogCollor collor = dog.new DogCollor("Golden", "Large");
        collor.showInfo();
    }
}
```

4.5 Внутренние локальные классы

```
class Dog {
    private String name;

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

    // Метод с локальным внутренним классом
    public void getForWalk(String parkName) {
        final int walkDuration = 60; // Локальная переменная

        // Локальный внутренний класс
        class Walk {
            public void startWalk() {
                // Доступ к полям внешнего класса и локальным
    переменным
                System.out.println(name + " is walking in " +
    parkName);
                // Доступ к final локальной переменной
                System.out.println("Walk duration: " + walkDuration
    + " minutes");
            }
        }

        // Создание экземпляра локального класса и вызов метода
        Walk walk = new Walk();
        walk.startWalk();
```

```
    }
}

// Другой пример использования локального класса(метод с возвратом
       локального класса)
public Runnable getRunnable() {
    final String message = "Hello from Runnable";

    // Локальный класс, реализующий интерфейс Runnable
    class MyRunnable implements Runnable {
        @Override
        public void run() {
            System.out.println(message);
        }
    }

    return new MyRunnable();
}
```

4.6 Внутренние анонимные классы

```
// 1. Анонимный класс от интерфейса / Anonymous class from
       interface / 基于接口的匿名类
interface Animal {
    void makeSound();
}

class Test1 {
    void test() {
        // Создание анонимного класса / Creating anonymous class /
           创建匿名类
        Animal dog = new Animal() {
            @Override
            public void makeSound() {
                System.out.println("Woof!");
            }
        };
        dog.makeSound(); // Woof!
    }
}

// 2. Анонимный класс от абстрактного класса / Anonymous class from
       abstract class / 基于抽象类的匿名类
abstract class Vehicle {
    abstract void move();
}

class Test2 {
    void test() {
        Vehicle car = new Vehicle() {
            @Override
            void move() {
                System.out.println("Car is driving");
            }
        };
        car.move(); // Car is driving
    }
}

// 3. Анонимный класс с конструктором / Anonymous class with
       constructor-like / 类似构造函数的匿名类
class Dog {
    String name;
    Dog(String name) { this.name = name; }
    void bark() { System.out.println("Woof!"); }
}

class Test3 {
    void test() {
        // Передача параметров в "конструктор" / Passing parameters
           to "constructor" / 传递参数给"构造函数"
        Dog dog = new Dog("Rex") {
            @Override
            void bark() {
                System.out.println(name + " says: WOOF WOOF!");
            }
        };
        dog.bark(); // Rex says: WOOF WOOF!
    }
}

// 4. Анонимный класс в аргументе метода / Anonymous class in method
       argument / 方法参数中的匿名类
interface OnClickListener {
    void onClick();
}

class Button {
    void setOnClickListener(OnClickListener listener) {
        listener.onClick();
    }
}

class Test4 {
    void test() {
        Button button = new Button();
        // Анонимный класс как аргумент / Anonymous class as
           argument / 匿名类作为参数
        button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick() {
                System.out.println("Button clicked!");
            }
        });
    }
}
```

5. Типы-перечисления. Поля и методы типов-перечислений.

这是 第 2 题 (Question 2) 的详细解析（根据常见的 Java 考纲顺序）。

这道题考察的是 Java 中一个非常强大的特性：枚举 (Enum)。很多初学者以为枚举只是简单的常量列表，但实际上在 Java 中，枚举是类 (Class)，这使得它比 C/C++ 中的枚举强大得多。

---

1. Типы-перечисления (Enums)

枚举类型

定义 (Definition): `enum` — 这是一种特殊的 Java 类，用于定义一组固定的常量（例如：季节、星期几、订单状态）。

```
// RU: Объявление простого перечисления.
// EN: Declaring a simple enum.
// CN: 声明一个简单的枚举。
public enum Season {
    WINTER, SPRING, SUMMER, AUTUMN
}

// RU: Использование.
// CN: 使用。
Season current = Season.SUMMER;
```

- Объяснение (解释): Внутри Java `enum Season` превращается в класс, который наследуется от `java.lang.Enum`. Каждый элемент (WINTER, SPRING…) — это статический финальный объект этого класса. 在 Java 内部，`enum Season` 会变成一个继承自 `java.lang.Enum` 的类。每个元素（WINTER, SPRING…）都是这个类的静态最终 (static final) 对象。

---

2. Поля типов-перечислений

枚举类型的字段

既然枚举是类，它就可以有字段 (Fields) 和 构造器 (Constructor)。这是 Java 枚举最酷的地方：你可以给常量附加数据。

```
public enum CoffeeSize {
    // RU: Вызов конструктора для каждой константы.
    // EN: Calling constructor for each constant.
    // CN: 为每个常量调用构造器。
    SMALL(100),
    MEDIUM(200),
    LARGE(300);

    // RU: Поле перечисления.
    // EN: Enum field.
    // CN: 枚举字段。
    private int milliliters;

    // RU: Конструктор (всегда private или package-private).
    // EN: Constructor (always private or package-private).
    // CN: 构造器（必须是 private 或包级私有）。
    CoffeeSize(int milliliters) {
        this.milliliters = milliliters;
    }

    // RU: Геттер для поля.
    // CN: 字段的 Getter 方法。
    public int getMl() {
        return milliliters;
    }
}
```

- Объяснение (解释): Вы можете хранить данные внутри каждого элемента перечисления. Конструктор вызывается автоматически при загрузке класса. Создавать объекты enum через **new** нельзя! 你可以在每个枚举元素内部存储数据。构造器在类加载时自动调用。不能通过 **new** 手动创建枚举对象！

---

3. Методы типов-перечислений (Пользовательские)

枚举类型的方法（自定义方法）

枚举也可以有普通的方法，甚至是抽象方法。

```
public enum Operation {
    PLUS, MINUS;

    // RU: Обычный метод.
    // CN: 普通方法。
    public double apply(double x, double y) {
        switch(this) {
            case PLUS: return x + y;
            case MINUS: return x - y;
            default: return 0;
        }
    }
}
```

```java
// Usage:
double result = Operation.PLUS.apply(10, 20); // 30.0
```

- Объяснение (解释): Перечисления могут содержать логику. Это позволяет избавиться от длинных `if-else` цепочек в основном коде, переместив логику внутрь самого перечисления. 枚举可以包含逻辑。这允许你将逻辑移动到枚举内部，从而消除主代码中冗长的 `if-else` 链。

---

4. Встроенные методы (Standard Methods)

内置方法

所有枚举都自动拥有几个重要的方法，这是编译器生成的。

```java
Season s = Season.WINTER;

// [1] values()
// RU: Возвращает массив всех констант (для перебора).
// CN: 返回所有常量的数组（用于遍历）。
for (Season season : Season.values()) {
    System.out.println(season);
}

// [2] valueOf(String name)
// RU: Превращает строку в enum.
// CN: 将字符串转换为枚举。
Season w = Season.valueOf("WINTER");

// [3] ordinal()
// RU: Возвращает порядковый номер (индекс), начиная с 0.
// CN: 返回序号（索引），从 0 开始。
int index = Season.SPRING.ordinal(); // 1

// [4] name()
// RU: Возвращает имя константы в виде строки.
// CN: 以字符串形式返回常量名称。
String n = Season.SUMMER.name(); // "SUMMER"
```

- Объяснение (解释): `values()` и `valueOf()` — это статические методы, добавляемые компилятором. `ordinal()` использовать не рекомендуется для логики программы, так как порядок констант может измениться. `values()` 和 `valueOf()` 是编译器添加的静态方法。不建议在程序逻辑中使用 `ordinal()`，因为常量的顺序可能会改变。

---

6. Стандартная библиотека коллекций языка Java. Интерфейсы, реализации иалгоритмы коллекций. Структ библиотеки коллекций. Коллекции, множества исписки. Использование реализаций интерфейсов коллекций. (maps) вбиблиотеке коллекций. Использование различных реализаций карт. Итераторыкарт и коллекций. С алгоритмы библиотеки для работы с коллекциями и массивами.

这是 第 3 题 (Question 3) 的详细解析。这道题是 Java 面试和考试中的“重中之重”。Java 集合框架 (JCF) 是日常开发中最常用的工具。

我们将按照你给出的每一个小句作为独立的小问来解答。

---

1. Стандартная библиотека коллекций языка Java

Java 标准集合库

概念： Java 集合框架 (Java Collections Framework, JCF) 是一组用于存储和操作对象组的类和接口的统一架构。 它位于 `java.util` 包中。

- RU: Это единая архитектура для представления и манипулирования группами объектов.
- EN: It is a unified architecture for representing and manipulating groups of objects.
- CN: 它是一个用于表示和操作对象组的统一架构。

---

2. Интерфейсы, реализации и алгоритмы коллекций

集合的接口、实现和算法

库由三个主要部分组成：

1. **接口 (Interfaces):** 抽象数据类型 (如 `List`, `Set`, `Map`)。它们定义了集合应该“做什么”。
2. **实现 (Implementations):** 具体的类 (如 `ArrayList`, `HashSet`)。它们定义了集合“怎么做”。
3. **算法 (Algorithms):** 执行有用计算的方法 (如 `Collections.sort`, `Collections.shuffle`)。

```java
// Interface (List) -> Implementation (ArrayList)
List<String> list = new ArrayList<>();

// Algorithm (Sort)
Collections.sort(list);
```

---

3. Структура библиотеки коллекций

集合库的结构

这是考试中需要画图或者描述清楚的部分。

- Collection (Root): 集合层次结构的根接口。
- List: 有序集合。
- Set: 不包含重复元素的集合。
- Queue: 队列。
- Map: 注意！ Map 不是 Collection 的子接口！ 它是一个独立的键值对 (Key-Value) 分支。

---

4. Коллекции, множества и списки

集合、集 (Sets) 和列表 (Lists)

这是最常用的两种集合类型的对比。

List (Список / 列表):

- 有序 (Ordered): 元素有索引 (0, 1, 2…)。
- 允许重复 (Duplicates OK): 可以有两个相同的对象。
- 示例: `ArrayList`, `LinkedList`.

Set (Множество / 集):

- 无序 (Unordered): 通常不保证顺序 (HashSet)。
- 唯一 (Unique): 不允许重复元素。
- 示例: `HashSet`, `TreeSet`.

```java
// List: Сохраняет порядок, разрешает дубликаты
// List: 保留顺序，允许重复
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Apple"); // OK

// Set: Уникальность, порядок не гарантирован
// Set: 唯一性，不保证顺序
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Apple"); // Игнорируется (Ignored)
```

---

5. Использование реализаций интерфейсов коллекций

使用集合接口的实现

核心原则： 面向接口编程。 声明变量时使用接口 (`List`)，实例化时使用具体类 (`ArrayList`)。这样以后想换实现（比如换成 `LinkedList`）非常容易。

```java
// RU: Правильно! Используем интерфейс слева.
// EN: Correct! Use interface on the left.
// CN: 正确！左边使用接口。
List<String> data = new ArrayList<>();

// RU: Если нужно часто вставлять в начало, просто меняем правую
        часть.
// EN: If we need frequent insertions at the beginning, just change
        the right side.
// CN: 如果需要频繁在头部插入，只需更改右边。
// List<String> data = new LinkedList<>();

// RU: Не рекомендуется (привязка к реализации).
// CN: 不推荐（绑定到了具体实现）。
// ArrayList<String> data = new ArrayList<>();
```

---

6. Карты (maps) в библиотеке коллекций

集合库中的映射 (Map)

`Map` 用于存储 **键值对 (Key-Value Pairs)**。

- Key (键): 必须唯一 (Unique)。
- Value (值): 可以重复。

```java
Map<String, Integer> phoneBook = new HashMap<>();
phoneBook.put("Alice", 123456);
phoneBook.put("Bob", 654321);
```

---

7. Использование различных реализаций карт

使用各种 Map 的实现

不同的 Map 实现有不同的特性：

1. **HashMap:**

- **特点:** 最快 (Fastest)。无序 (Unordered)。
- **用途:** 通用缓存、查找表。

1. **LinkedHashMap:**

- **特点:** 记住插入顺序 (Insertion Order)。稍慢一点。
- **用途:** 需要按添加顺序遍历时。

1. **TreeMap:**

- **特点:** 按 Key 排序 (Sorted by Key)。最慢 (Slowest, O(log n))。
- **用途:** 需要按字母顺序或数字大小显示数据时。

```
// RU: Сортировка по ключам (алфавит).
// CN: 按键排序（字母顺序）。
Map<String, Integer> treeMap = new TreeMap<>();
treeMap.put("Banana", 2);
treeMap.put("Apple", 1);
// Iteration: Apple, then Banana
```

---

8. Итераторы карт и коллекций

Map 和 Collection 的迭代器

Collection Iterator: 这是遍历列表的标准方式（`for-each` 循环底层就是它）。

```
List<String> list = Arrays.asList("A", "B");
Iterator<String> it = list.iterator();
while(it.hasNext()) {
    String s = it.next();
    // it.remove(); // RU: Безопасное удаление! (CN: 安全删除！)
}
```

Map Iteration (Map 没有直接的 iterator): Map 必须先转换成 Set 才能遍历。

1. `keySet()`: 遍历键。
2. `values()`: 遍历值。
3. `entrySet()`: 遍历键值对（最高效）。

```
Map<String, Integer> map = new HashMap<>();
// ... put data ...

// RU: Самый эффективный способ перебора Map.
// CN: 遍历 Map 最高效的方法。
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

---

9. Стандартные алгоритмы библиотеки для работы с коллекциями и массивами

用于集合和数组的标准库算法

Java 提供了两个工具类：`java.util.Collections`（针对集合）和 `java.util.Arrays`（针对数组）。

Collections 类 (工具类):

```
List<Integer> nums = new ArrayList<>();
nums.add(3); nums.add(1); nums.add(2);

Collections.sort(nums);    // 排序: [1, 2, 3]
Collections.reverse(nums); // 反转: [3, 2, 1]
Collections.shuffle(nums); // 打乱 (Shuffling)
Integer max = Collections.max(nums); // 最大值
```

Arrays 类 (工具类):

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

Arrays.sort(arr); // 排序
// RU: Двоичный поиск (работает только на отсортированном массиве).
// CN: 二分查找（仅适用于已排序的数组）。
int index = Arrays.binarySearch(arr, 3);

// RU: Преобразование массива в список (фиксированного размера).
// CN: 将数组转换为列表（固定大小）。
List<Integer> list = Arrays.asList(1, 2, 3);
```

7.Методы рефакторинга для преобразования структуры программы на языке Java. Рефакторинг типов. Реф иерархии наследования. Перемещение методов по иерархии наследования и между классами. Использова Eclipse для рефакторинга программы на языке Java.

7.1 Методы рефакторинга для преобразования структуры программы на языке Java.

这是 第 7 题 (Question 7) 的核心内容。 考题原文提到："Методы рефакторинга… Рефакторинг иерархии наследования. Перемещение методов"（重构方法…继承层次重构。移动方法）。

这里为你挑选了 3 个最经典、最适合初学者 的重构例子。

---

1. Extract Method (Выделение метода / 提取方法)

场景： 一个方法太长了，或者某段代码逻辑是独立的。我们把它"挖"出来变成一个新方法。 **Сценарий:** Метод слишком длинный, или часть кода логически независима. Мы "выделяем" её в новый метод. **Scenario:** A method is too long, or a piece of code is logically independent. We "extract" it into a new method.

🔴 **Before (До / 重构前)**

```
void printOwing() {
    printBanner();

    // RU: Печать деталей (дублирование или сложная логика)
    // EN: Print details (duplication or complex logic)
    // CN: 打印细节（重复代码或复杂逻辑）
    System.out.println("name: " + name);
    System.out.println("amount: " + amount);
}
```

🟢 **After (После / 重构后)**

```
void printOwing() {
    printBanner();
    printDetails(amount); // RU: Вызов нового метода
                          // EN: Call the new method
                          // CN: 调用新方法
}

// RU: Выделенный метод. Код стал чище.
// EN: Extracted method. Code became cleaner.
// CN: 被提取的方法。代码变得更整洁。
void printDetails(double amount) {
    System.out.println("name: " + name);
    System.out.println("amount: " + amount);
}
```

---

2. Rename Method (Переименование метода / 重命名方法)

场景： 方法名不能清楚地说明它在做什么。 **Сценарий:** Имя метода не объясняет ясно, что он делает. **Scenario:** The method name does not clearly explain what it does.

🔴 **Before (До / 重构前)**

```
// RU: Имя "doIt" ничего не значит
// EN: Name "doIt" means nothing
// CN: 名字 "doIt" 没有任何意义
public int doIt() {
    return days * 24;
}
```

🟢 **After (После / 重构后)**

```
// RU: Имя объясняет суть (часы в днях)
// EN: Name explains the essence (hours in days)
// CN: 名字解释了本质（天数转小时）
public int convertDaysToHours() {
    return days * 24;
}
```

---

3. Pull Up Method (Подъем метода / 上移方法)

场景： 两个子类有完全相同的代码。为了消除重复，我们把它移到父类里。(这是考题中"继承层次重构"的重点) **Сценарий:** Два подкласса имеют абсолютно одинаковый код. Чтобы убрать дублирование, мы переносим его в родительский класс. **Scenario:** Two subclasses have exactly the same code. To remove duplication, we move it to the parent class.

🔴 Before (До / 重构前)

```
class Dog extends Animal {
    // RU: Дублирующийся код
    // EN: Duplicated code
    // CN: 重复代码
    void sleep() { System.out.println("Zzz..."); }
}

class Cat extends Animal {
    // RU: Дублирующийся код
    // EN: Duplicated code
    // CN: 重复代码
    void sleep() { System.out.println("Zzz..."); }
}
```

🟢 After (После / 重构后)

```
class Animal {
    // RU: Метод перемещен в родителя (Рефакторинг иерархии)
    // EN: Method moved to parent (Hierarchy refactoring)
    // CN: 方法被移到父类（继承层次重构）
    void sleep() {
        System.out.println("Zzz...");
    }
}

class Dog extends Animal {
    // RU: Теперь пусто, наследует sleep()
    // EN: Now empty, inherits sleep()
    // CN: 现在是空的，继承了 sleep()
}

class Cat extends Animal {
    // ...
}
```

7.2 Рефакторинг типов.

这是 第 7 题 (Question 7) 中的 “Рефакторинг типов” (类型重构) 。

最经典、最常考的例子是 “Generalize Type” (泛化类型)，即用更通用的接口替换具体的实现类。

---

1. Generalize Type (Обобщение типа / 泛化类型)

场景： 我们把变量声明为具体类（如 `ArrayList`），这限制了灵活性。应该改为接口（如 `List`）。 Сценарий: Мы объявляем переменную как конкретный класс (`ArrayList`), что ограничивает гибкость. Лучше использовать интерфейс (`List`). **Scenario:** We declare a variable as a concrete class (`ArrayList`), limiting flexibility. It's better to use an interface (`List`).

🔴 Before (До / 重构前)

```
// RU: Жесткая привязка к ArrayList. Мы не можем легко заменить его
       на LinkedList.
// EN: Hard dependency on ArrayList. We cannot easily switch to
       LinkedList.
// CN: 硬编码依赖 ArrayList。我们无法轻松切换到 LinkedList。
public void processData() {
    ArrayList<String> names = new ArrayList<>();
    names.add("Alice");
    // ...
}
```

🟢 After (После / 重构后)

```
// RU: Использование интерфейса List. Теперь можно подставить любую
       реализацию списка.
// EN: Using List interface. Now we can swap in any list
       implementation.
// CN: 使用 List 接口。现在可以替换为任何列表实现。
public void processData() {
    List<String> names = new ArrayList<>();
    // List<String> names = new LinkedList<>(); // RU: Легко
         изменить! (CN: 容易更改！)
    names.add("Alice");
    // ...
}
```

---

2. Replace Primitive with Object (Замена примитива объектом / 以对象取代基本类型)

场景： 代码里用简单的 `String` 或 `int` 表示复杂概念（如电话号码、邮政编码）。应该创建一个专门的类。 Сценарий: Код использует простые `String` или `int` для сложных понятий (телефон, индекс). Стоит создать специальный класс. **Scenario:** The code uses simple `String` or `int` for complex concepts (phone, zip code). A special class should be created.

🔴 Before (До / 重构前)

```
class User {
    String name;

    // RU: Просто строка. Нет проверки формата, нет логики.
    // EN: Just a string. No format validation, no logic.
    // CN: 只是个字符串。没有格式验证，没有逻辑。
    String phoneNumber;
}
```

🟢 After (После / 重构后)

```
class User {
    String name;

    // RU: Теперь это тип! Внутри класса PhoneNumber может быть
         валидация.
    // EN: Now it's a type! Inside PhoneNumber class, there can be
         validation.
    // CN: 现在它是一个类型！PhoneNumber 类内部可以包含验证逻辑。
    PhoneNumber phoneNumber;
}

class PhoneNumber {
    private String number;
    // Constructor, format logic...
}
```

7.3 Рефакторинг иерархии наследования.

这是 第 7 题 (Question 7) 中的 “Рефакторинг иерархии наследования” (继承层次重构)。

这一部分的重点在于改变类之间的父子关系，而不仅仅是移动方法。最经典的两个例子是 “提炼超类” (Extract Superclass) 和 “折叠继承体系” (Collapse Hierarchy)。

---

1. Extract Superclass (Выделение суперкласса / 提炼超类)

场景： 两个类有相似的字段和方法，但它们没有共同的父类（或者父类太通用了）。我们需要创建一个新的父类来存放共性。 Сценарий: Два класса имеют похожие поля и методы, но у них нет общего родителя. Мы создаем новый родительский класс для общего кода. **Scenario:** Two classes have similar fields and methods, but they lack a common parent. We create a new parent class for the common code.

🔴 Before (До / 重构前)

```
// RU: Два независимых класса с дублированием (name, email)
// EN: Two independent classes with duplication (name, email)
// CN: 两个独立的类，存在重复代码 (name, email)

class Student {
    String name;
    String email;
    void study() { ... }
}

class Teacher {
    String name;
    String email;
    void teach() { ... }
}
```

🟢 After (После / 重构后)

```
// RU: Создаем общего родителя Person
// EN: Create a common parent Person
// CN: 创建一个共同的父类 Person
class Person {
    String name;
    String email;
}

// RU: Теперь классы наследуются от Person. Дублирование исчезло.
// EN: Now classes inherit from Person. Duplication is gone.
// CN: 现在这些类继承自 Person。重复代码消失了。
class Student extends Person {
    void study() { ... }
}

class Teacher extends Person {
    void teach() { ... }
}
```

---

2. Collapse Hierarchy (Свертывание иерархии / 折叠继承体系)

场景： 子类和父类太像了，子类几乎没有添加任何新功能（由于重构或其他原因）。这时应该把它们合并。 Сценарий: Подкласс и суперкласс слишком похожи, подкласс почти ничего не добавляет. Их следует объединить. **Scenario:** The subclass and superclass are too similar; the subclass adds almost nothing. They should be merged.

🔴 Before (До / 重构前)

```
class Employee {
    int salary;
}

// RU: Этот класс почти пустой. Он не нужен.
// EN: This class is almost empty. It is not needed.
// CN: 这个类几乎是空的。它是不必要的。
class Salesman extends Employee {
    // RU: Нет уникального поведения
    // EN: No unique behavior
    // CN: 没有独特的行为
}
```

🟢 After (После / 重构后)

```
// RU: Мы удалили класс Salesman и перенесли всё использование в
      Employee
// EN: We removed Salesman class and moved all usage to Employee
// CN: 我们删除了 Salesman 类，并将所有引用都移到了 Employee
class Employee {
    int salary;
    // ...
}
```

7.4 Перемещение методов по иерархии наследования и между классами.

这是 第 7 题 (Question 7) 的最后一部分："Перемещение методов по иерархии наследования и между классами"（在继承层次结构中移动方法以及在类之间移动方法）。

这部分主要考察两个核心操作：

1. **Push Down Method (Спуск метода / 方法下移):** 针对继承关系。
2. **Move Method (Перемещение метода / 移动方法):** 针对类与类之间的协作。

---

1. Push Down Method (Спуск метода / 方法下移)

场景： 父类中有一个方法，但只有部分子类需要它。对于其他子类来说，这个方法是多余甚至错误的。 Сценарий: В родительском классе есть метод, который нужен только некоторым подклассам. **Scenario:** The parent class has a method that is used by only **some** subclasses.

🔴 Before (До / 重构前)

```
class Animal {
    // RU: Ошибка: не все животные лают. Кошкам этот метод не нужен.
    // EN: Error: not all animals bark. Cats don't need this method.
    // CN: 错误：不是所有动物都会叫。猫不需要这个方法。
    void bark() {
        System.out.println("Woof!");
    }
}

class Dog extends Animal { }

class Cat extends Animal {
    // RU: Наследует bark(), что странно.
    // EN: Inherits bark(), which is weird.
    // CN: 继承了 bark()，这很奇怪。
}
```

🟢 After (После / 重构后)

```
class Animal {
    // RU: Метод убран из родителя.
    // EN: Method removed from parent.
    // CN: 方法从父类中移除。
}

class Dog extends Animal {
    // RU: Метод перемещен ("спущен") сюда, где он действительно
       нужен.
    // EN: Method moved ("pushed down") here, where it is actually
       needed.
    // CN: 方法被移动（"下移"）到这里，这里才是真正需要它的地方。
    void bark() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    // RU: Чисто. Нет лишних методов.
    // EN: Clean. No extra methods.
    // CN: 干净。没有多余的方法。
}
```

---

2. Move Method (Перемещение метода / 移动方法)

场景： 一个类（Class A）中的方法，使用另一个类（Class B）的数据比用自己类的数据还多。这种现象叫"特性依恋"(Feature Envy)。应该把这个方法移到 Class B 去。 Сценарий: Метод в классе A использует данные класса B больше, чем свои собственные. Это называется "Зависть к функциям". Метод нужно перенести в класс B. **Scenario:** A method in Class A uses more data from Class B than from its own class. This is called "Feature Envy". The method should be moved to Class B.

🔴 Before (До / 重构前)

目标： `Student` 类里有个打印 `Course` 详情的方法。

```
class Student {
    // ...

    // RU: Этот метод живет в Student, но использует только данные
       Course.
    // EN: This method lives in Student, but uses only Course data.
    // CN: 这个方法在 Student 里，但只使用了 Course 的数据。
    void printCourseInfo(Course c) {
        System.out.println("Course: " + c.getTitle() + ", Price: "
        + c.getPrice());
    }
}

class Course {
    private String title;
    private double price;
    // getters...
}
```

🟢 After (После / 重构后)

操作： 把方法剪切到 `Course` 类里。

```
class Student {
    // RU: Теперь Student просто вызывает метод у Course.
    // EN: Now Student just calls the method on Course.
    // CN: 现在 Student 只是调用 Course 的方法。
    void displayInfo(Course c) {
        c.printInfo();
    }
}

class Course {
    private String title;
    private double price;

    // RU: Метод перемещен сюда. Ему здесь самое место.
    // EN: Method moved here. It belongs here.
    // CN: 方法移到了这里。这才是它的归宿。
    void printInfo() {
        System.out.println("Course: " + this.title + ", Price: " + t
        his.price);
    }
}
```

7.5 Использование среды Eclipse для рефакторинга программы на языке Java.

这是 第 7 题 (Question 7) 的最后一个考点："Использование среды Eclipse для рефакторинга программы на языке Java"（使用 Eclipse 环境重构 Java 程序）。

在考试中，回答这道题的关键不仅是写代码，还要描述"操作步骤"(Action/Steps)。Eclipse 的强大之处在于它是自动化的。

这里有两个最常用的 Eclipse 重构功能演示。

---

1. Rename (Переименование / 重命名)

场景： 你想改一个变量名。如果你手动改，你需要查找整个项目里所有用到它的地方。用 Eclipse，一键搞定。 Сценарий: Вы хотите изменить имя переменной. Вручную это долго. Eclipse делает это автоматически во всем проекте. **Scenario:** You want to rename a variable. Doing it manually is slow. Eclipse does it automatically across the whole project.

操作步骤 (Steps):

1. **RU:** Выделите переменную `n` -> Нажмите Alt + Shift + R -> Введите `name` -> Enter.
2. **EN:** Select variable `n` -> Press Alt + Shift + R -> Type `name` -> Enter.
3. **CN:** 选中变量 `n` -> 按 Alt + Shift + R -> 输入 `name` -> 回车。

🔴 Before (До / 操作前)

```
public class Student {
    // RU: Плохое имя, непонятно что это
    // EN: Bad name, unclear meaning
    // CN: 名字不好，不清楚是什么
    private String n;
```

```
    public String getN() {
        return n;
    }
}
```

🟢 After (После / Eclipse 自动修改后)

```
public class Student {
    // RU: Eclipse переименовал поле И метод (getN -> getName)
    // EN: Eclipse renamed the field AND the method (getN ->
        getName)
    // CN: Eclipse 重命名了字段和方法 (getN -> getName)
    private String name;

    public String getName() {
        return name;
    }
}
```

---

2. Encapsulate Field (Инкапсуляция поля / 封装字段)

场景： 你有一些 `public` 字段，想改成标准的 `private` 字段加 Getters/Setters。 Сценарий: У вас есть `public` поля, нужно превратить их в `private` с геттерами и сеттерами. **Scenario:** You have `public` fields, need to convert them to `private` with getters and setters.

操作步骤 (Steps):

1. **RU:** Правый клик по полю -> **Refactor** -> **Encapsulate Field**…
2. **EN:** Right-click on field -> **Refactor** -> **Encapsulate Field**…
3. **CN:** 右键点击字段 -> **Refactor** -> **Encapsulate Field**…

🔴 Before (До / 操作前)

```
public class User {
    // RU: Публичный доступ (небезопасно)
    // EN: Public access (unsafe)
    // CN: 公有访问 (不安全)
    public int age;
}
```

🟢 After (После / Eclipse 自动修改后)

```
public class User {
    // RU: Eclipse сделал поле приватным
    // EN: Eclipse made the field private
    // CN: Eclipse 把字段变成了私有
    private int age;

    // RU: И автоматически сгенерировал методы
    // EN: And automatically generated methods
    // CN: 并且自动生成了方法
    public int getAge() {
        return age;
    }

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

---

3. 常用快捷键 (Полезные горячие клавиши / Useful Shortcuts)

考试时如果能写出这几个快捷键，会非常加分：

功能 (Function)	快捷键 (Windows)	俄语说明 (RU)
Rename	`Alt + Shift + R`	Переименование (самый важный!)
Extract Method	`Alt + Shift + M`	Выделение метода (код в новый метод)
Extract Local Variable	`Alt + Shift + L`	Выделение локальной переменной
Organize Imports	`Ctrl + Shift + O`	Организация импорта (удаление лишнего)

---

8. Родовые типы в языке Java. Назначение родовых типов. Не ковариантность родовых типов. Родовой тип wildcard. Родовые методы. Ограниченные родовые типы.

---

1. Родовые типы в языке Java

Java 语言中的泛型

```
// Класс Box может хранить объект любого типа T
// Box 类可以存储任何类型 T 的对象
public class Box<T> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

// Использование (使用):
Box<Integer> integerBox = new Box<>();
Box<String> stringBox = new Box<>();
```

- Объяснение (解释): Родовые типы (Generics) позволяют абстрагировать тип данных, используемый в классе или интерфейсе. `T` — это параметр типа, который заменяется реальным типом при создании объекта. 泛型允许我们在类或接口中抽象出数据类型。`T` 是一个类型参数，在创建对象时会被替换为具体的类型。

---

2. Назначение родовых типов

泛型的用途

```
// [1] Без Generics (No Type Safety)
// 无泛型 (无类型安全)
List list = new ArrayList();
list.add("Hello");
list.add(100); // Ошибка не видна при компиляции (编译时看不出错误)
// String s = (String) list.get(1); // Runtime Exception! (运行时异常)

// [2] С Generics (Type Safety)
// 有泛型 (类型安全)
List<String> listGen = new ArrayList<>();
listGen.add("Hello");
// listGen.add(100); // Ошибка компиляции! (编译错误！)
String s = listGen.get(0); // Приведение типов не нужно (不需要类型转
        换)
```

- Объяснение (解释): Главные цели: 1. Обеспечение строгой типизации на этапе компиляции (обнаружение ошибок раньше). 2. Устранение необходимости в явном приведении типов (casting). 主要目的：1. 在编译阶段提供严格的类型检查（更早发现错误）。2. 消除显式强制类型转换的需要。

---

3. Не ковариантность родовых типов

泛型的非协变性

```
public void test() {
    List<String> strings = new ArrayList<>();

    // ❌ ОШИБКА КОМПИЛЯЦИИ (COMPILATION ERROR)
    // List<Object> objects = strings;

    // Почему это запрещено? (为什么禁止这样做？)
    // Если бы это было можно, мы могли бы сделать так:
    // 如果允许这样做，我们就可以：
    // objects.add(new Integer(123)); // Добавили число в список
        строк! (把数字加进了字符串列表！)
    // String s = strings.get(0); // ClassCastException при чтении
}
```

- Объяснение (解释): В Java дженерики инвариантны: `List<String>` не является подтипом `List<Object>`, даже если `String` является подтипом `Object`. Это сделано для предотвращения порчи кучи (Heap Pollution). 在 Java 中，泛型是不变的：`List<String>` 不是 `List<Object>` 的子类型，即使 `String` 是 `Object` 的子类型。这是为了防止堆污染（Heap Pollution）。

---

4. Родовой тип wildcard

通配符泛型 (`?`)

```
// Метод принимает список ЛЮБОГО типа
// 该方法接受“任何”类型的列表
public static void printList(List<?> list) {
    for (Object elem : list) {
        System.out.print(elem + " ");
    }
    System.out.println();
}

// Использование:
List<Integer> li = Arrays.asList(1, 2, 3);
List<String> ls = Arrays.asList("one", "two");
printList(li); // Работает (Works)
printList(ls); // Работает (Works)
```

- Объяснение (解释): Символ `?` (wildcard) означает “неизвестный тип”. `List<?>` — это супертип для любых списков. Однако, в такой список нельзя ничего добавлять (кроме `null`), так как компилятор не знает конкретный тип элементов. 符号 `?`（通配符）表示“未知类

型”。`List<?>` 是所有列表的父类型。但是，你不能向这种列表中添加任何元素（`null` 除外），因为编译器不知道具体是哪种类型。

---

5. Родовые методы

泛型方法

```java
public class ArrayUtil {
    // <T> перед типом возврата объявляет метод родовым
    // 返回类型前的 <T> 声明这是一个泛型方法
    public static <T> T getMiddle(T[] a) {
        return a[a.length / 2];
    }
}

// Использование (T выводится автоматически)
// 使用（自动推断 T）
String[] names = {"John", "Q", "Public"};
String middle = ArrayUtil.getMiddle(names); // "Q"
```

- Объяснение (解释): Родовые методы позволяют использовать параметры типа в отдельных методах, не делая весь класс родовым. Тип `T` определяется компилятором на основе переданных аргументов. 泛型方法允许在单个方法中使用类型参数，而无需将整个类定义为泛型。编译器会根据传入的参数自动推断类型 `T`。

---

6. Ограниченные родовые типы

受限泛型 (Bounded Types)

这是这一题最难的部分，通常分为 Upper Bound（`extends`）和 Lower Bound（`super`）。考试常考 `extends`。

示例 A: Upper Bound (Верхняя граница) 我们只想接受 `Number` 或其子类（如 Integer, Double）。

```java
// T должен быть наследником Number
// T 必须是 Number 的子类
public class MathBox<T extends Number> {
    private T value;

    public MathBox(T value) { this.value = value; }

    public double doubleValue() {
        // Мы уверены, что у T есть метод doubleValue()
        // 我们确信 T 拥有 doubleValue() 方法
        return value.doubleValue();
    }
}

// MathBox<String> box = new MathBox<>("Hi"); // ❌ Ошибка компиляции! String не Number.
MathBox<Integer> iBox = new MathBox<>(10);    // ✅ OK
```

- Объяснение (解释): `T extends ClassName` ограничивает параметр типа так, что он должен быть либо указанным классом, либо его наследником. Это позволяет вызывать методы этого класса внутри дженерика. `T extends ClassName` 将类型参数限制为该类本身或其子类。这允许你在泛型代码内部调用该类的方法（例如 `Number` 的 `doubleValue()`）。

---

9. Потоки байтового вывода языка Java. Назначение и возможности классов OutputStream, ByteArrayOutputStream, FileOutputStream, PipedOutputStream, FilterOutputStream, BufferedOutputStream, DataOutputStream, PrintStream. Потоки символьного вывода языка Java.

1. Потоки байтового вывода языка Java (OutputStream)

Java 字节输出流 (OutputStream)

```java
// Абстрактный базовый класс для всех байтовых потоков вывода
// 所有字节输出流的抽象基类
public abstract class OutputStream implements Closeable, Flushable {
    public abstract void write(int b) throws IOException;
    // ...
}
```

- Объяснение (解释): `OutputStream` — это абстрактный родитель всех классов, которые выводят байты. Он определяет базовый метод `write()`, но сам по себе не используется для создания объектов. `OutputStream` 是所有输出字节类的抽象父类。它定义了基本的 `write()` 方法，但本身不能用于直接创建对象。

---

2. ByteArrayOutputStream

字节数组输出流

```java
public void testByteArray() throws IOException {
    // RU: Пишет данные в память (внутренний массив), а не на диск.
    // CN: 将数据写入内存（内部数组），而不是磁盘。
    ByteArrayOutputStream baos = new ByteArrayOutputStream();

    baos.write(65); // 'A'
    baos.write(66); // 'B'

    // RU: Преобразование накопленных байтов в массив.
    // CN: 将积累的字节转换为数组。
    byte[] result = baos.toByteArray(); // [65, 66]
}
```

- Объяснение (解释): Используется, когда нужно собрать данные в памяти (буфер) перед тем, как отправить их куда-то еще. Данные хранятся в динамически расширяемом массиве байтов. 用于在将数据发送到其他地方之前，先在内存（缓冲区）中收集数据。数据存储在动态扩展的字节数组中。

---

3. FileOutputStream

文件输出流

```java
public void testFile() throws IOException {
    // RU: true означает режим добавления (append), false – перезапись.
    // CN: true 表示追加模式 (append)，false 表示覆盖。
    FileOutputStream fos = new FileOutputStream("test.txt", true);

    String text = "Hello Java IO";
    // RU: Нужно явно преобразовать строку в байты.
    // CN: 需要显式地将字符串转换为字节。
    fos.write(text.getBytes());

    fos.close(); // Обязательно закрывать! (务必关闭！)
}
```

- Объяснение (解释): Предназначен для записи необработанных байтов (например, изображений или текста) в файл на диске. 用于将原始字节（例如图像或文本）写入磁盘文件。

---

4. PipedOutputStream

管道输出流

```java
public void testPipe() throws IOException {
    PipedOutputStream out = new PipedOutputStream();
    PipedInputStream in = new PipedInputStream();

    // RU: Обязательно соединить вход и выход!
    // CN: 必须连接输入和输出！
    out.connect(in);

    // RU: Обычно используется в двух разных потоках (Threads).
    // Один поток пишет в 'out', другой читает из 'in'.
    // CN: 通常在两个不同的线程中使用。一个线程写入 'out'，另一个从 'in' 读取。
    new Thread(() -> {
        try { out.write('X'); } catch (IOException e) {}
    }).start();
}
```

- Объяснение (解释): Создает канал связи между двумя потоками (threads). Все, что записано в `PipedOutputStream`, становится доступным для чтения в связанном `PipedInputStream`. 在两个线程之间建立通信通道。写入 `PipedOutputStream` 的所有内容都可以在连接的 `PipedInputStream` 中读取。

---

5. FilterOutputStream

过滤输出流

```java
// Это базовый класс для "оберток" (Decorators)
// 这是所有“包装器”（装饰器）的基类
public class FilterOutputStream extends OutputStream {
    protected OutputStream out; // Ссылка на реальный поток (引用真实的流)

    public FilterOutputStream(OutputStream out) {
        this.out = out;
    }
}
```

- Объяснение (解释): Сам по себе этот класс редко используется. Он служит базой для классов, которые добавляют функциональность к существующему потоку (например, буферизацию или шифрование). 这个类本身很少直接使用。它是那些为现有流添加功能（如缓冲或加密）的类的基类。

---

6. BufferedOutputStream

缓冲输出流

```
public void testBuffered() throws IOException {
    // RU: Оборачиваем FileOutputStream для скорости.
    // CN: 包装 FileOutputStream 以提高速度。
    FileOutputStream fos = new FileOutputStream("data.bin");
    BufferedOutputStream bos = new BufferedOutputStream(fos);

    // RU: Данные копятся в буфере и записываются на диск большим
        куском.
    // CN: 数据积攒在缓冲区中，然后一大块一大块地写入磁盘。
    bos.write(1);

    bos.flush(); // Принудительная запись (强制写入)
    bos.close();
}
```

- Объяснение (解释): Повышает производительность ввода-вывода. Вместо того чтобы обращаться к диску для каждого байта, он накапливает данные в памяти и пишет их блоками. 提高 I/O 性能。它不是为每个字节都访问磁盘，而是将数据在内存中积累，然后分块写入。

---

7. DataOutputStream

数据输出流

```
public void testData() throws IOException {
    DataOutputStream dos = new DataOutputStream(new
        FileOutputStream("file.dat"));

    // RU: Записывает примитивные типы Java переносимым способом.
    // CN: 以可移植的方式写入 Java 基本数据类型。
    dos.writeInt(123);       // 4 bytes
    dos.writeDouble(45.67);  // 8 bytes
    dos.writeBoolean(true);  // 1 byte

    dos.close();
}
```

- Объяснение (解释): Позволяет приложению записывать примитивные типы данных Java (int, double, boolean) в поток вывода переносимым способом. 允许应用程序以可移植的方式将 Java 基本数据类型（int, double, boolean）写入输出流。

---

8. PrintStream

打印流

```
public void testPrint() throws FileNotFoundException {
    // System.out - это объект типа PrintStream!
    // System.out 就是一个 PrintStream 类型的对象！
    PrintStream ps = new PrintStream(new FileOutputStream("log.txt")
        );

    // RU: Удобные методы для печати текста. Не выбрасывает
        IOException.
    // CN: 便捷的文本打印方法。不抛出 IOException。
    ps.println("Hello World");
    ps.printf("Age: %d", 25);

    ps.close();
}
```

- Объяснение (解释): Добавляет возможность удобно печатать представления различных значений данных. В отличие от других потоков, он никогда не выбрасывает IOException и поддерживает автоматический сброс (auto-flush). 添加了方便地打印各种数据值表示形式的功能。与其他流不同，它从不抛出 IOException，并且支持自动刷新 (auto-flush)。

---

9. Потоки символьного вывода языка Java

Java 字符输出流

题目最后提到了“字符输出流”。这是为了区分“字节 (Byte)”和“字符 (Char)”。

```
// Байтовый поток (Byte Stream) -> OutputStream
// Работает с raw binary data (изображения, видео).
// 处理原始二进制数据（图像、视频）。

// Символьный поток (Character Stream) -> Writer
// Работает с текстом и кодировками (UTF-8).
// 处理文本和编码（UTF-8）。

Writer writer = new FileWriter("text.txt");
writer.write("Привет!"); // Можно писать строки напрямую (可以直接写字
        符串)
writer.close();
```

- Объяснение (解释): Если `OutputStream` предназначен для байтов, то для работы с текстом в Java используется иерархия классов на основе абстрактного класса `Writer`. Они автоматически обрабатывают кодировки символов. 如果说 `OutputStream` 是用于字节的，那么 Java 中处理文本则使用基于抽象类 `Writer` 的类层次结构。它们会自动处理字符编码。

---

10. Потоки ввода языка Java. Назначение и возможности классов InputStream, ByteArrayInputStream, FileInp PipedInputStream, FilterInputStream, BufferedInputStream, DataInputStream. Потоки символьного ввода языка Чтение данных из потока с помощью класса Scanner.

---

1. Потоки ввода языка Java (InputStream)

Java 字节输入流 (InputStream)

```
// Абстрактный базовый класс для всех байтовых потоков ввода
// 所有字节输入流的抽象基类
public abstract class InputStream implements Closeable {

    // RU: Возвращает следующий байт данных (0-255) или -1, если
        конец потока.
    // EN: Returns the next byte of data (0-255) or -1 if the end of
        the stream is reached.
    // CN: 返回下一个字节的数据（0-255），如果到达流的末尾则返回 -1。
    public abstract int read() throws IOException;
}
```

- Объяснение (解释): `InputStream` — это родитель всех входных потоков. Его главный метод `read()` читает по одному байту. Это медленно, поэтому обычно используются наследники. `InputStream` 是所有输入流的父类。它的主要方法 `read()` 每次读取一个字节。这很慢，所以通常使用其子类。

---

2. ByteArrayInputStream

字节数组输入流

```
public void testByteArrayInput() {
    byte[] data = { 65, 66, 67 }; // A, B, C

    // RU: Читаем данные из массива в памяти, как будто это поток.
    // CN: 从内存数组中读取数据，就像读取流一样。
    ByteArrayInputStream bais = new ByteArrayInputStream(data);

    int b = bais.read(); // 65 ('A')
}
```

- Объяснение (解释): Позволяет использовать буфер в памяти (массив байтов) как источник данных для потока ввода. 允许将内存中的缓冲区（字节数组）用作输入流的数据源。

---

3. FileInputStream

文件输入流

```
public void testFileInput() throws IOException {
    // RU: Открывает соединение с реальным файлом.
    // CN: 打开与真实文件的连接。
    FileInputStream fis = new FileInputStream("config.txt");

    int i;
    // RU: Читаем побайтово, пока не встретим -1 (конец файла).
    // CN: 逐字节读取，直到遇到 -1（文件结束）。
    while ((i = fis.read()) != -1) {
        System.out.print((char) i);
    }

    fis.close();
}
```

- Объяснение (解释): Предназначен для чтения байтов из файла в файловой системе. Это один из самых часто используемых классов. 专用于从文件系统中的文件读取字节。这是最常用的类之一。

---

4. PipedInputStream

管道输入流

```
public void testPipedInput() throws IOException {
    PipedInputStream in = new PipedInputStream();
    PipedOutputStream out = new PipedOutputStream();

    // RU: Соединяем "трубу".
    // CN: 连接“管道”。
    in.connect(out);

    // RU: Читаем то, что другой поток записал в 'out'.
    // CN: 读取另一个线程写入 'out' 的内容。
    int data = in.read();
}
```

- Объяснение (解释): Принимает данные, записанные в связанный `PipedOutputStream`. Используется для передачи данных между потоками (threads). 接收写入到相关联的 `PipedOutputStream` 中的数据。用于在线程之间传输数据。

---

5. FilterInputStream

过滤输入流

```
// Базовый класс-декоратор
// 装饰器基类
public class FilterInputStream extends InputStream {
    protected volatile InputStream in; // Обернутый поток (被包装的流)

    protected FilterInputStream(InputStream in) {
        this.in = in;
    }
}
```

- Объяснение (解释): Базовый класс для декораторов, которые добавляют новую функциональность к существующему потоку ввода. 装饰器的基类，用于向现有的输入流添加新功能。

---

6. BufferedInputStream

缓冲输入流

```
public void testBufferedInput() throws IOException {
    FileInputStream fis = new FileInputStream("large_video.mp4");

    // RU: Читает большие куски данных в память, уменьшая количество
        обращений к диску.
    // CN: 将大块数据读取到内存中，减少磁盘访问次数。
    BufferedInputStream bis = new BufferedInputStream(fis);

    int data = bis.read(); // Быстро! (Fast!)
    bis.close();
}
```

- Объяснение (解释): Добавляет буферизацию. Когда вы запрашиваете 1 байт, он читает сразу блок (например, 8KB) в память. Это значительно ускоряет чтение файлов. 添加缓冲功能。当你请求 1 个字节时，它会一次性读取一个块（例如 8KB）到内存中。这极大地加快了文件读取速度。

---

7. DataInputStream

数据输入流

```
public void testDataInput() throws IOException {
    DataInputStream dis = new DataInputStream(new FileInputStream("
        data.bin"));

    // RU: Читаем примитивы в том же порядке, в котором писали
        (через DataOutputStream).
    // CN: 按照写入的顺序（通过 DataOutputStream）读取基本类型。
    int i = dis.readInt();       // 4 bytes
    double d = dis.readDouble(); // 8 bytes
    boolean b = dis.readBoolean();

    dis.close();
}
```

- Объяснение (解释): Позволяет читать примитивные типы данных Java (int, float, boolean) из потока. 允许从流中读取 Java 基本数据类型（int, float, boolean）。

---

8. Потоки символьного ввода языка Java

Java 字符输入流

这里再次强调 Byte vs Char 的区别。

```
// InputStream -> Reader
// FileInputStream -> FileReader

public void testReader() throws IOException {
    // RU: FileReader автоматически декодирует байты в символы
        (char).
    // CN: FileReader 自动将字节解码为字符 (char)。
    Reader reader = new FileReader("text.txt");

    int data = reader.read(); // Возвращает char (0-65535)
    reader.close();
}
```

- Объяснение (解释): Для чтения текста используется иерархия `Reader` (читатель). Она работает с 16-битными символами Unicode, в отличие от 8-битных байтов `InputStream`. 读取文本使用 `Reader`（读取器）层次结构。与使用 8 位字节的 `InputStream` 不同，它处理 16 位 Unicode 字符。

---

9. Чтение данных из потока с помощью класса Scanner

使用 Scanner 类从流中读取数据

这是考试中必考的实用工具，因为它比 `InputStream` 好用太多了。

```
public void testScanner() {
    // RU: Scanner может читать из InputStream, файла или строки.
    // CN: Scanner 可以从 InputStream、文件或字符串读取。
    Scanner scanner = new
        Scanner(System.in); // Чтение с клавиатуры (从键盘读取)

    System.out.print("Enter number: ");

    // RU: Удобные методы для парсинга токенов.
    // CN: 用于解析标记的便捷方法。
    if (scanner.hasNextInt()) {
        int number = scanner.nextInt();
        System.out.println("You entered: " + number);
    }

    String word = scanner.next(); // Читает слово до пробела (读取直到
        空格的一个单词)
    String line = scanner.nextLine(); // Читает всю строку (读取整行)

    scanner.close();
}
```

- Объяснение (解释): Класс `Scanner` — это высокоуровневый текстовый сканер. Он разбивает входные данные на токены с помощью разделителя (по умолчанию пробел) и может автоматически парсить числа и строки. `Scanner` 类是一个高级文本扫描器。它使用分隔符（默认为空格）将输入数据分解为标记 (tokens)，并且可以自动解析数字和字符串。

---

11. Интернационализация программ в языке Java. Хайлы текстовых ресурсов.Выбор языка пользователя для выдачи текстовых сообщений

1. Интернационализация программ в языке Java (I18n)

Java 程序的国际化

```
// I18n = Internationalization (между 'I' и 'n' 18 букв)
// I18n = Internationalization ('I' 和 'n' 之间有 18 个字母)

// Цель: Написать код один раз, а тексты хранить отдельно.
// 目标：代码只写一次，文本分开存储。
```

- Объяснение (解释): Интернационализация — это процесс проектирования приложения так, чтобы его можно было адаптировать к различным языкам и регионам без изменения исходного кода. 国际化是设计应用程序的过程，使其无需更改源代码即可适应不同的语言和地区。

---

2. Хайлы текстовых ресурсов (Resource Bundles)

文本资源文件

在 Java 中，翻译通常存储在 `.properties` 文件中。文件名必须遵循严格的命名规则：`BaseName_language_country.properties`。

Структура файлов (文件结构):

1. **messages.properties** (Default / 默认 - 英语)

```
greeting=Hello
farewell=Goodbye
```

1. **messages_ru.properties** (Russian / 俄语)

```
greeting=Привет
farewell=Пока
```

1. **messages_zh.properties** (Chinese / 中文)

```
greeting=你好
farewell=再见
```

- Объяснение (解释): Текстовые ресурсы хранятся в файлах `.properties` в виде пар “ключ=значение”. Ключи (например, `greeting`) одинаковы во всех файлах, а значения зависят от языка. 文本资源以“键=值”对的形式存储在 `.properties` 文件中。键（例如 `greeting`）在所有文件中都是相同的，但值取决于语言。

---

3. Выбор языка пользователя для выдачи текстовых сообщений

选择用户语言以显示文本消息

我们需要使用 `java.util.Locale` 来指定语言，用 `java.util.ResourceBundle` 来加载对应的文件。

```
import java.util.Locale;
import java.util.ResourceBundle;

public class I18nDemo {
    public static void main(String[] args) {
        // [1] Создаем локали для разных языков
        // [1] 为不同的语言创建 Locale 对象
        Locale localeEn = new Locale("en", "US"); // English
        Locale localeRu = new Locale("ru", "RU"); // Russian
        Locale localeZh = new Locale("zh", "CN"); // Chinese

        // [2] Выбираем язык пользователя (например, Русский)
        // [2] 选择用户语言（例如：俄语）
        Locale currentLocale = localeRu;

        // [3] Загружаем нужный файл ресурсов
        // Java ищет файл: messages_ru.properties
        // [3] 加载相应的资源文件
        // Java 会寻找：messages_ru.properties
        ResourceBundle bundle =
         ResourceBundle.getBundle("messages", currentLocale);

        // [4] Получаем текст по ключу
        // [4] 通过键获取文本
        String msg = bundle.getString("greeting");

        System.out.println(msg); // Вывод (Output): Привет
    }
}
```

- Объяснение (解释): Класс `Locale` определяет регион (язык + страна). Метод `ResourceBundle.getBundle()` автоматически ищет наиболее подходящий файл `.properties` для заданной локали. Если точное совпадение не найдено, используется файл по умолчанию. `Locale` 类定义了区域（语言 + 国家）。`ResourceBundle.getBundle()` 方法会自动为指定的区域查找最匹配的 `.properties` 文件。如果没有找到完全匹配的，则使用默认文件。

---

12. Лямбда выражения в языке Java8. Замена анонимных классов лямбда выражениями. Хункциональные и Синтаксис Lambda-выражений. Примеры функциональных интерфейсов из пакета java.util.function.

---

1. Замена анонимных классов лямбда выражениями

用 Lambda 表达式替换匿名类

在 Java 8 之前，为了把一段代码（比如线程逻辑）传递给方法，我们不得不写冗长的匿名内部类。Lambda 表达式解决了这个问题。

🔴 Before (Java 7 - Анонимный класс / 匿名类)

```
// RU: Мы хотим запустить код в отдельном потоке.
// EN: We want to run code in a separate thread.
// CN: 我们想在单独的线程中运行代码。
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello from Anonymous Class!");
    }
}).start();
```

🟢 After (Java 8 - Lambda Expression)

```
// RU: То же самое, но в одну строку. "Мусорный" код (new
      Runnable...) исчез.
// EN: The same, but in one line. "Garbage" code is gone.
// CN: 同样的功能，但只有一行。“垃圾”代码（new Runnable...）消失了。
new Thread(() -> System.out.println("Hello from Lambda!")).start();
```

- Объяснение (解释): Lambda-выражение — это, по сути, анонимная функция. Она позволяет реализовать метод интерфейса более лаконично, не создавая громоздкий анонимный класс. Lambda 表达式本质上是一个匿名函数。它允许你以更简洁的方式实现接口方法，而无需创建笨重的匿名类。

---

2. Хункциональные интерфейсы

函数式接口

这是 Lambda 表达式能工作的前提条件。

```
// RU: Интерфейс с ТОЛЬКО ОДНИМ абстрактным методом.
// CN: 只有一个抽象方法的接口。
@FunctionalInterface // (Optional annotation / 可选注解)
interface MathOperation {
    int operate(int a, int b);

    // RU: Default и static методы не мешают интерфейсу быть
         функциональным.
    // CN: Default 和 static 方法不影响接口成为函数式接口。
    default void sayHello() { System.out.println("Hi"); }
}
```

- Объяснение (解释): Хункциональный интерфейс — это интерфейс, который содержит ровно один абстрактный метод. Lambda-выражение всегда “подставляется” именно на место этого единственного метода. 函数式接口是恰好包含一个抽象方法的接口。Lambda 表达式总是被“填入”这个唯一方法的位置。

---

3. Синтаксис Lambda-выражений

Lambda 表达式的语法

基本格式：`(parameters) -> expression` 或 `(parameters) -> { statements; }`

```
// 1. Без параметров (No parameters)
() -> System.out.println("Hello");

// 2. Один параметр (One parameter)
// RU: Скобки вокруг x можно опустить.
// CN: x 周围的括号可以省略。
x -> x * x;

// 3. Несколько параметров (Multiple parameters)
(x, y) -> x + y;

// 4. Блок кода (Block of code)
// RU: Если есть {}, нужен return и ;
// CN: 如果有 {}，则需要 return 和 ;
(x, y) -> {
    int sum = x + y;
    return sum;
};
```

- Объяснение (解释): Синтаксис состоит из трех частей: список параметров, стрелка `->` и тело функции (выражение или блок кода). Типы параметров обычно выводятся компилятором автоматически. 语法由三部分组成：参数列表、箭头 `->` 和函数体（表达式或代码块）。参数类型通常由编译器自动推断。

---

4. Примеры функциональных интерфейсов из пакета java.util.function

java.util.function 包中的函数式接口示例

Java 8 提供了很多内置的标准接口，这样我们就不用每次都自己定义了。考试常考以下四个：

A. Predicate (Предикат / 断言)

用途： 接收一个参数，返回 `boolean`（用于检查条件）。

```
// RU: Проверяет, длина строки > 5?
// CN: 检查字符串长度是否 > 5？
Predicate<String> isLong = s -> s.length() > 5;

System.out.println(isLong.test("Hello World")); // true
```

B. Consumer (Потребитель / 消费者)

用途： 接收一个参数，不返回任何值（void）。（用于打印、写入数据库等）。

```
// RU: Печатает строку в верхнем регистре.
// CN: 以大写形式打印字符串。
Consumer<String> printer = s -> System.out.println(s.toUpperCase());

printer.accept("java"); // JAVA
```

C. Function<T, R> (Хункция / 函数)

用途： 接收类型 T，返回类型 R。（用于转换数据）。

```
// RU: Преобразует Строку в Число (длину).
// CN: 将字符串转换为数字（长度）。
Function<String, Integer> lengthFunc = s -> s.length();

int len = lengthFunc.apply("Code"); // 4
```

D. Supplier (Поставщик / 供给者)

用途： 不接收参数，返回一个 T。（用于工厂方法、延迟生成数据）。

```
// RU: Генерирует случайное число.
// CN: 生成随机数。
Supplier<Double> randomizer = () -> Math.random();

System.out.println(randomizer.get());
```

---

13. Потоки в языке Java8. Определение потоков. Отличие операций потоков от операций коллекций. Обработка потока в конвейере. Методы для порождения потоков. Методы для преобразования потоков в конвейере. Преобразования в конце конвейера.

好的，我们进入 第 13 题 (Question 13) 。这是 Java 8 中非常重要的一章：Stream API。

请注意，这里的“流” (Stream) 不是 IO 流 (InputStream)，而是数据流。

---

1. Потоки в языке Java8. Определение потоков.

Java 8 中的流。流的定义。

```
// RU: Stream — это последовательность элементов, поддерживающая
        последовательные и параллельные операции агрегирования.
// CN: Stream 是支持串行和并行聚合操作的元素序列。

List<String> list = Arrays.asList("a", "b", "c");

// RU: Создание потока из коллекции.
// CN: 从集合创建流。
Stream<String> stream = list.stream();
```

- Объяснение (解释): `Stream` (java.util.stream.Stream) — это не структура данных. Он не хранит данные. Это инструмент для обработки данных (как конвейер на заводе). `Stream` (java.util.stream.Stream) 不是数据结构。它不存储数据。它是用于处理数据的工具（就像工厂里的流水线）。

---

2. Отличие операций потоков от операций коллекций.

流操作与集合操作的区别。

这是一个理论重点，考试常问。

对比表 (Сравнение / Comparison):

特性 (Feature)	集合 (Collections)	流 (Streams)
主要目的 (Main Goal)	存储数据 (Storage)	计算数据 (Computation)
数据处理 (Processing)	外部迭代 (External Iteration - `for-each`)	内部迭代 (Internal Iteration)
数据量 (Size)	有限 (Finite)	可以是无限的 (Can be infinite)
修改 (Modification)	可以修改元素 (Can modify)	不修改源数据 (Does not modify source)
执行时机 (Execution)	立即执行 (Eager)	延迟执行 (Lazy)

- Объяснение (解释): Коллекции — это про то, как хранить данные (в памяти). Потоки — это про то, как вычислять данные (процессинг). Потоки ленивы: они ничего не делают, пока не вызвана терминальная операция. 集合关注的是如何存储数据（在内存中）。流关注的是如何计算数据（处理）。流是延迟的：在调用终端操作之前，它们什么也不做。

---

3. Обработка данных потока в конвейере.

在流水线中处理流数据。

流的操作分为三个阶段，就像工厂流水线： Source (原料) -> Intermediate Ops (加工) -> Terminal Op (成品)

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David
        ");

names.stream() // 1. Источник (Source / 源)
     .filter(s -> s.startsWith("C")) // 2. Промежуточная операция
        (Intermediate / 中间操作)
     .map(String::toUpperCase)       // 2. Промежуточная операция
        (Intermediate / 中间操作)
     .forEach(System.out::println);  // 3. Терминальная операция
        (Terminal / 终端操作)
```

- Объяснение (解释): Конвейер состоит из источника (коллекция, массив), нуля или более промежуточных операций (трансформация) и одной терминальной операции (результат). 流水线由源（集合、数组）、零个或多个中间操作（转换）和一个终端操作（结果）组成。

---

4. Методы для порождения потоков.

生成流的方法。

```
// 1. Из коллекции (From Collection)
List<String> list = Arrays.asList("A", "B");
Stream<String> s1 = list.stream();

// 2. Из массива (From Array)
String[] arr = {"A", "B"};
Stream<String> s2 = Arrays.stream(arr);

// 3. Статический метод of() (Static method of)
Stream<String> s3 = Stream.of("A", "B", "C");

// 4. Бесконечные потоки (Infinite Streams - для генерации)
Stream<Double> randoms = Stream.generate(Math::random);
Stream<Integer> evens = Stream.iterate(0, n -> n + 2);
```

- Объяснение (解释): Самый частый способ — вызов `.stream()` у коллекции. Но можно создавать потоки и из массивов, и даже генерировать бесконечные последовательности чисел. 最常用的方法是对集合调用 `.stream()`。但也可以从数组创建流，甚至生成无限的数字序列。

---

5. Методы для преобразования потоков в конвейере (Промежуточные).

在流水线中转换流的方法（中间操作）。

这些操作返回一个新的 Stream，并且是惰性 (Lazy) 的。

```
Stream<String> stream = Stream.of("apple", "banana", "cherry", "apr
        icot");

stream
    // [A] filter: Фильтрация (Отсеивает лишнее)
    // 过滤：保留以 "a" 开头的
    .filter(s -> s.startsWith("a"))

    // [B] map: Преобразование (Меняет каждый элемент)
    // 映射：把每个字符串变成它的长度 (String -> Integer)
    .map(s -> s.length())

    // [C] sorted: Сортировка
    // 排序
    .sorted()

    // [D] distinct: Удаление дубликатов
    // 去重
    .distinct()

    // [E] limit: Ограничение количества
    // 限制：只取前 3 个
    .limit(3);
```

- Объяснение (解释): Промежуточные операции (Intermediate Operations) всегда возвращают новый поток. Они не выполняются сразу, а лишь “настраивают” конвейер. 中间操作总是返回一个新的流。它们不会立即执行，而只是“配置”流水线。

---

6. Преобразования в конце конвейера (Терминальные).

流水线末端的转换（终端操作）。

这些操作会触发整个流水线的执行，并返回结果（不再是 Stream）。流一旦经过终端操作，就关闭了，不能再用了。

```
Stream<String> stream = Stream.of("one", "two", "three");

// [A] forEach: Действие для каждого элемента (void)
// 对每个元素执行操作
// stream.forEach(System.out::println);
```

```java
// [B] collect: Сбор результатов в коллекцию (List, Set)
// 将结果收集到集合中
List<String> resultList = stream.collect(Collectors.toList());

// [C] count: Подсчет количества элементов (long)
// 统计元素数量
// long count = stream.count();

// [D] reduce: Сведение всех элементов к одному значению (агрегация)
// 将所有元素归约为一个值（聚合）
// Optional<String> concatenated = stream.reduce((s1, s2) -> s1 +
        s2);
```

- Объяснение (解释): Терминальные операции (Terminal Operations) запускают процесс обработки и возвращают результат (коллекцию, число или void). После этого поток закрывается и его нельзя использовать повторно. 终端操作启动处理过程并返回结果（集合、数字或 void）。此后流关闭，不能再次使用。

---

14. Модули в Java 9. Назначение модулей. Синтаксис описания модулей. Зависимость от модулей. Экспорт пакетов модуля. Открытость модуля.

好的，我们进入 第 14 题 (Question 14) 。这是 Java 9 引入的“模块系统” (Project Jigsaw)，它的主要目的是解决“JAR 地狱” (JAR Hell) 和打破单体架构。

这一题的重点在于理解 `module-info.java` 文件的写法。

---

1. Назначение модулей

模块的用途

模块是比“包” (Package) 更高一级的封装单位。它是一组包和资源的集合。

- 解决 “Classpath Hell” (Решение проблемы Classpath): 在 Java 9 之前，所有类都混在一个大桶里 (Classpath)。如果两个库用了同一个类的不同版本，程序就会崩溃。模块明确了“我依赖谁，谁依赖我”。
- 强封装 (Сильная инкапсуляция): 在 Java 8 中，`public` 类在任何地方都能被访问。在 Java 9 模块中，即使是 `public` 类，如果没有被导出 (exported)，外部模块也是看不见的。

---

2. Синтаксис описания модулей

模块描述的语法

模块的定义必须放在一个名为 `module-info.java` 的特殊文件中，该文件位于源代码的根目录下。

```java
// 文件名: module-info.java

// RU: Ключевое слово module + имя модуля
// CN: 关键字 module + 模块名
module com.mycompany.mymodule {
    // Тело модуля
    // 模块体
}
```

- Объяснение (解释): Описание модуля находится в файле `module-info.java`. Это декларация того, что модуль требует для работы и что он предоставляет другим. 模块描述位于 `module-info.java` 文件中。它声明了该模块运行所需的内容以及它提供给其他模块的内容。

---

3. Зависимость от модулей (requires)

对模块的依赖

如果你的模块需要使用别人的代码，必须显式声明。

```java
module com.mycompany.app {
    // RU: Модуль требует модуль java.sql для работы.
    // CN: 该模块需要 java.sql 模块才能工作。
    requires java.sql;

    // RU: requires transitive означает, что тот, кто зависит от
        нас,
    // автоматически получит зависимость и от com.mylib.core.
    // CN: requires transitive 意味着依赖我们的模块也会自动依赖
        com.mylib.core。
    requires transitive com.mylib.core;
}
```

- Объяснение (解释): Директива `requires` указывает, от каких других модулей зависит данный модуль. Без этого вы не сможете использовать классы из этих модулей, даже если они public. `requires` 指令指明当前模块依赖于哪些其他模块。如果没有它，即使类是 public 的，你也无法使用这些模块中的类。

---

4. Экспорт пакетов модуля (exports)

导出模块的包

默认情况下，模块里的所有包都是隐藏的。你想让别人用的包，必须“暴露”出来。

```java
module com.mycompany.utils {
    // RU: Пакет com.mycompany.utils.api доступен всем.
    // CN: 包 com.mycompany.utils.api 对所有人可用。
    exports com.mycompany.utils.api;

    // RU: Пакет com.mycompany.utils.internal СКРЫТ.
    // Его нельзя использовать из других модулей!
    // CN: 包 com.mycompany.utils.internal 是隐藏的。其他模块无法使用它！
    // (Нет exports для internal)

    // RU: Экспорт только для конкретного друга.
    // CN: 仅对特定的朋友导出。
    exports com.mycompany.utils.debug to com.mycompany.test;
}
```

- Объяснение (解释): Директива `exports` делает public классы в указанном пакете видимыми для других модулей. Все, что не экспортировано, остается строго внутренним (инкапсулированным). `exports` 指令使指定包中的 public 类对其他模块可见。所有未导出的内容都严格保持内部私有（封装）。

---

5. Открытость модуля (Openness)

模块的开放性

这是一个难点。Java 9 默认禁止反射 (Reflection) 访问私有成员（这破坏了 Spring, Hibernate 等框架）。为了允许反射，我们需要“开放”模块。

场景 A: 开放整个模块 (Open Module)

```java
// RU: Разрешает рефлексию (reflection) ко ВСЕМ пакетам модуля.
// CN: 允许对模块内的“所有”包进行反射访问。
open module com.mycompany.entity {
    // ...
}
```

场景 B: 开放特定包 (Opens Package)

```java
module com.mycompany.service {
    // RU: Разрешает рефлексию только к этому пакету (например, для
        Hibernate).
    // CN: 仅允许对该包进行反射访问（例如为了给 Hibernate 使用）。
    opens com.mycompany.service.models;
}
```

- Объяснение (解释): `opens` разрешает доступ через рефлексию (Deep Reflection) к приватным полям и методам пакета. `open module` делает это для всего модуля сразу. Это критически важно для фреймворков. `opens` 允许通过反射（深度反射）访问包的私有字段和方法。`open module` 则是立刻对整个模块开放。这对框架来说至关重要。

---

15. Возможности версий Java9-Java17.

这是 第 15 题 (Question 15) 的详细解答。虽然考纲中只写了标题，但这是面试和现代 Java 开发中非常热门的话题。

这道题考察的是你对 “现代 Java” (Modern Java) 语法的掌握程度。从 Java 9 到 Java 17 (LTS)，语法变得更简洁、更像脚本语言。

以下是考试中最可能问到的 6 个关键特性：

---

1. Вывод типов локальных переменных (var) - Java 10

局部变量类型推断 (var)

Java 终于可以像 JavaScript 或 Kotlin 一样，让编译器自己猜类型了。

```java
// Java 9 (Old)
// String message = "Hello";
```

```java
// ArrayList<String> list = new ArrayList<String>();

// Java 10+ (New)
public void testVar() {
    // RU: Компилятор сам понимает, что это String.
    // CN: 编译器自己明白这是 String。
    var message = "Hello";

    // RU: Очень удобно для длинных типов.
    // CN: 对于很长的类型名非常方便。
    var list = new ArrayList<String>();
}
```

- Объяснение (解释): Ключевое слово `var` позволяет не указывать тип переменной явно, если компилятор может вывести его из правой части выражения. Работает только внутри методов (локальные переменные). 关键字 `var` 允许你不显式指定变量类型，只要编译器能从表达式右侧推断出来。仅适用于方法内部（局部变量）。

---

2. Switch выражения (Switch Expressions) - Java 14

Switch 表达式

旧的 `switch` 很笨重，容易漏写 `break`。新的 `switch` 可以返回值，而且语法更漂亮。

```java
// Java 14+
public int getDays(String day) {
    // RU: Используем стрелку ->. break не нужен! Возвращаем значение.
    // CN: 使用箭头 ->。不需要 break！直接返回值。
    int numLetters = switch (day) {
        case "MONDAY", "FRIDAY", "SUNDAY" -> 6;
        case "TUESDAY"                    -> 7;
        default                           -> day.length();
    };
    return numLetters;
}
```

- Объяснение (解释): Новый синтаксис `switch` позволяет использовать его как выражение (возвращать значение). Стрелочный синтаксис `->` предотвращает случайное выполнение следующих кейсов (no fall-through). 新的 `switch` 语法允许将其用作表达式（返回值）。箭头语法 `->` 防止了意外执行后续的 case（消除了穿透效应）。

---

3. Текстовые блоки (Text Blocks) - Java 15

文本块

以前写 JSON 或 SQL 字符串简直是噩梦（全是 `\"` 和 `\n`）。现在可以用三个引号 `"""`。

```java
// Java 15+
public void testTextBlocks() {
    // RU: Многострочная строка без экранирования.
    // CN: 无需转义的多行字符串。
    String json = """
        {
            "name": "Java",
            "version": 17
        }
        """;
}
```

- Объяснение (解释): Текстовые блоки (`"""`) позволяют писать многострочные строки, сохраняя форматирование и избегая лишнего экранирования кавычек. 文本块 (`"""`) 允许编写多行字符串，保留格式并避免不必要的引号转义。

---

4. Сопоставление с образцом для instanceof (Pattern Matching) - Java 16

instanceof 的模式匹配

以前判断完类型还得强转，现在一步到位。

```java
// Java 16+
public void printLength(Object obj) {
    // RU: Если obj это String, то сразу создается переменная 's'.
    // CN: 如果 obj 是 String，则立即创建变量 's'。
    if (obj instanceof String s) {
        // RU: Не нужно делать (String) obj
        // CN: 不需要写 (String) obj
        System.out.println(s.length());
    }
}
```

- Объяснение (解释): Эта фича устраняет необходимость явного приведения типов (casting) после проверки `instanceof`. Переменная создается автоматически. 此特性消除了在 `instanceof` 检查后进行显式类型转换 (casting) 的需要。变量会自动创建。

---

5. Записи (Records) - Java 16

记录类

这是为了消灭样板代码（Boilerplate Code）。如果你只是想存数据（DTO），不要再写 Getter/Setter/Equals/HashCode 了。

```java
// Java 16+

// RU: Одной строкой мы создаем класс с полями, конструктором, геттерами, equals и toString.
// CN: 只需一行，我们就创建了一个包含字段、构造器、Getters、equals 和 toString 的类。
public record Point(int x, int y) { }

// Использование:
Point p = new Point(10, 20);
int x = p.x(); // Не getX(), а просто x()
System.out.println(p); // Point[x=10, y=20]
```

- Объяснение (解释): `record` — это компактный способ объявления неизменяемых (immutable) классов-носителей данных. Компилятор автоматически генерирует конструктор, геттеры, `equals`, `hashCode` и `toString`. `record` 是一种声明不可变 (immutable) 数据载体类的紧凑方式。编译器会自动生成构造器、访问器、`equals`、`hashCode` 和 `toString`。

---

6. Запечатанные классы (Sealed Classes) - Java 17

密封类

这让你能严格控制谁可以继承你。

```java
// Java 17+ (LTS)

// RU: Мы разрешаем наследоваться ТОЛЬКО Circle и Square.
// CN: 我们只允许 Circle 和 Square 继承此类。
public sealed class Shape permits Circle, Square { }

final class Circle extends Shape { }
final class Square extends Shape { }

// ❌ Ошибка компиляции: Triangle не в списке permits.
// CN: 编译错误：Triangle 不在 permits 列表中。
// final class Triangle extends Shape { }
```

- Объяснение (解释): Запечатанные классы (`sealed`) позволяют разработчику точно контролировать иерархию наследования, указывая список разрешенных подклассов (`permits`). 密封类 (`sealed`) 允许开发者通过指定允许的子类列表 (`permits`) 来精确控制继承层次结构。

---

16. Назначение библиотеки Standard Widget Toolkit. Основные классы библиотеки. Класс Display. Класс Shell. Composite. Класс Control. Цикл событий (event loop).

这是 第 16 题 (Question 16) 的完整解析。

这标志着考试内容从纯 Java 语言转到了 GUI（图形用户界面）开发。SWT (Standard Widget Toolkit) 是 Eclipse IDE 能够运行的基础，它的核心理念是“调用操作系统的原生 API”，这使得 SWT 程序看起来和原生程序一模一样，而且速度很快。

---

1. Назначение библиотеки Standard Widget Toolkit (SWT)

Standard Widget Toolkit 库的用途

```java
// Swing (Old Java GUI): Рисует кнопки сам (выглядят одинаково везде).
// Swing (旧 Java GUI): 自己绘制按钮（随处看起来都一样）。

// SWT: Вызывает операционную систему для создания кнопки.
// SWT: 调用操作系统来创建按钮。
// Windows -> WinAPI
// Linux -> GTK
// macOS -> Cocoa
```

- Объяснение (解释): SWT — это библиотека для создания графического интерфейса пользователя (GUI). Её главная цель — обеспечить “родной” (native) вид и производительность приложений, используя прямые вызовы API операционной системы.

SWT 是一个用于构建图形用户界面 (GUI) 的库。它的主要目标是通过直接调用操作系统的 API，提供“原生”的外观和性能。

---

2. Основные классы библиотеки

库的主要类

SWT 的类层次结构非常严格，你需要记住以下继承关系： Widget -> Control -> (Composite, Button, Label…)

- **Widget (Виджет):** Базовый абстрактный класс для всех объектов UI.
- **Control (Управляющий элемент):** Виджет, имеющий окно в ОС (handle).
- **Composite (Композит):** Контейнер, который может содержать другие контролы.
- **Shell (Окно):** Главное окно приложения.
- **Объяснение (解释):** Вся иерархия начинается с класса `Widget`. Все, что мы видим на экране (кнопки, метки, окна), является наследниками `Control`. 整个层次结构从 `Widget` 类开始。我们在屏幕上看到的所有内容（按钮、标签、窗口）都是 `Control` 的子类。

---

3. Класс Display

Display 类

```
// Создается один раз на все приложение
// 整个应用程序只创建一个
Display display = new Display();

// Метод для связи с ОС
// 与操作系统通信的方法
display.readAndDispatch();
```

- **Объяснение (解释):** Класс `Display` представляет собой связь между SWT и операционной системой. Он управляет циклом событий и передает команды от мыши/клавиатуры в ваше приложение. `Display` 类代表 SWT 与操作系统之间的连接。它管理事件循环，并将来自鼠标/键盘的命令传递给你的应用程序。

---

4. Класс Shell

Shell 类

```
// Shell — это "окно", которое видит пользователь
// Shell 是用户看到的“窗口”
Shell shell = new Shell(display);

shell.setText("Exam Window"); // Заголовок окна (窗口标题)
shell.setSize(400, 300);      // Размер (尺寸)

// Обязательно нужно открыть!
// 必须打开！
shell.open();
```

- **Объяснение (解释):** `Shell` — это окно верхнего уровня (окно приложения или диалоговое окно). Он служит корнем для всех остальных элементов интерфейса. `Shell` 是顶层窗口（应用程序窗口或对话框）。它是所有其他界面元素的根节点。

---

5. Класс Composite

Composite 类

```
// Composite — это контейнер (панель)
// Composite 是一个容器（面板）
Composite panel = new Composite(shell, SWT.BORDER);

// Мы кладем кнопку ВНУТРЬ панели (родитель — panel)
// 我们把按钮放进面板“里面”（父级是 panel）
Button btn = new Button(panel, SWT.PUSH);
```

- **Объяснение (解释):** `Composite` — это класс, который может содержать другие элементы управления (`Controls`). Он используется для группировки виджетов и создания сложной разметки. `Composite` 是一个可以包含其他控件（`Controls`）的类。它用于对小部件进行分组并创建复杂的布局。

---

6. Класс Control

Control 类

```
// Control — это родитель для Button, Label, Text и т.д.
// Control 是 Button, Label, Text 等的父类
Control myControl = new Button(shell, SWT.PUSH);

// Методы класса Control (общие для всех):
// Control 类的方法（所有控件通用）：
myControl.setVisible(true);   // Видимость (可见性)
myControl.setEnabled(false);  // Активность (启用/禁用)
myControl.setBackground(Color.RED); // Цвет фона (背景色)
```

- **Объяснение (解释):** `Control` — это абстрактный класс для всех элементов GUI, которые имеют графическое представление (окно) в операционной системе. У него есть размеры, позиция, цвет и шрифт. `Control` 是所有在操作系统中具有图形表示（窗口句柄）的 GUI 元素的抽象类。它具有大小、位置、颜色和字体等属性。

---

7. Цикл событий (Event Loop)

事件循环

这是 SWT 程序最核心的“样板代码” (Boilerplate Code)。如果考试让你写一个“最小的 SWT 程序”，你必须写出这段代码。

```
public static void main(String[] args) {
    Display display = new Display(); // 1. Связь с ОС
    Shell shell = new Shell(display); // 2. Окно
    shell.open(); // 3. Показать окно

    // === EVENT LOOP (ЦИКЛ СОБЫТИЙ) ===
    // Пока окно не уничтожено...
    // 只要窗口没有被销毁……
    while (!shell.isDisposed()) {

        // ...читай события (клики, нажатия) и обрабатывай их.
        // ...读取事件（点击、按键）并处理它们。
        // Если событий нет, спи (sleep), чтобы не грузить процессор.
        // 如果没有事件，就休眠 (sleep)，以免占用 CPU。
        if (!display.readAndDispatch()) {
            display.sleep();
        }
    }

    // Освобождение ресурсов (Обязательно!)
    // 释放资源（必须！）
    display.dispose();
}
```

- **Объяснение (解释):** Цикл событий — это бесконечный цикл, который ждет действий пользователя (событий) от операционной системы и передает их соответствующим виджетам. Без этого цикла программа откроется и сразу закроется. 事件循环是一个无限循环，它等待来自操作系统的用户操作（事件）并将它们传递给相应的小部件。没有这个循环，程序打开后会立即关闭。

---

17. Пакеты библиотеки Standard Widget Toolkit. Класс SWT. Иерархия управляющих элементов. Конструкт управляющих элементов.

这是 第 17 题 (Question 17) 的详细解析。

这道题考察的是 SWT 开发的“语法基础”。如果你不懂这一题，连最基本的按钮都写不出来。核心在于理解 SWT 类（那个装满常量的类）以及 构造器 的独特写法。

---

1. Пакеты библиотеки Standard Widget Toolkit

SWT 库的包结构

SWT 的包组织得非常扁平化，没有 Swing 那么深。

```
// [1] Главный пакет с константами (Самый важный!)
// [1] 包含常量的核心包（最重要的！）
import org.eclipse.swt.SWT;

// [2] Пакет с виджетами (Кнопки, Окна, Метки)
// [2] 包含小部件的包（按钮、窗口、标签）
import org.eclipse.swt.widgets.*;

// [3] Пакет с менеджерами компоновки (Расположение элементов)
// [3] 布局管理器包（元素排列）
import org.eclipse.swt.layout.*;

// [4] Пакет для обработки событий (Клики мыши, клавиатура)
// [4] 事件处理包（鼠标点击、键盘）
import org.eclipse.swt.events.*;
```

- **Объяснение (解释):** В отличие от AWT/Swing, где классы разбросаны, в SWT почти все визуальные компоненты живут в `org.eclipse.swt.widgets`. 与 AWT/Swing 类分散不同，在 SWT 中，几乎所有的可视化组件都居住在 `org.eclipse.swt.widgets` 中。

2. Класс SWT

SWT 类 (常量类)

这是一个非常特殊的类。它不是用来创建对象的，它是用来存放常量 (Constants) 的。

```
// RU: Класс SWT содержит тысячи констант для настройки стиля.
// CN: SWT 类包含数千个用于配置样式的常量。

// Пример использования (使用示例):
int style = SWT.BORDER | SWT.READ_ONLY | SWT.CENTER;
```

- Объяснение (解释): Класс `org.eclipse.swt.SWT` — это просто “мешок” с константами (`static final int`). Вы используете их, чтобы задать внешний вид и поведение виджетов (например, `SWT.PUSH` для кнопки или `SWT.BOLD` для шрифта). `org.eclipse.swt.SWT` 类只是一个装满常量 (`static final int`) 的“袋子”。你使用它们来定义小部件的外观和行为（例如，按钮用 `SWT.PUSH`，字体用 `SWT.BOLD`）。

3. Иерархия управляющих элементов

控件的层次结构

这是考试中需要背诵的继承树：

```
java.lang.Object
  |
  +-- org.eclipse.swt.widgets.Widget (Базовый класс / 基类)
        |
        +-- org.eclipse.swt.widgets.Control (Есть окно ОС / 拥有系统窗口)
              |
              +-- org.eclipse.swt.widgets.Scrollable (Есть полосы прокрутки / 有滚动条)
                    |
                    +-- org.eclipse.swt.widgets.Composite (Контейнер / 容器)
                    |     |
                    |     +-- Shell (Окно / 窗口)
                    |     +-- Canvas (Холст / 画布)
                    |
                    +-- Button, Label, Text, List (Листовые элементы / 叶子节点元素)
```

- Объяснение (解释):
- Widget: База для всего. Отвечает за освобождение ресурсов (`dispose`).
- Control: Элемент, у которого есть “ручка” (handle) в операционной системе. Его можно двигать и красить.
- Composite: Элемент, который может содержать другие элементы (родитель).
- Widget: 一切的基础。负责资源释放 (`dispose`)。
- Control: 在操作系统中拥有“句柄” (handle) 的元素。可以移动和着色。
- Composite: 可以包含其他元素的元素（父容器）。

4. Конструкторы и стили управляющих элементов

控件的构造器和样式

这是 SWT 最独特的规则：“父级在构造时确定，且不可更改”。

构造器语法 (Constructor Syntax)

```
// Общий вид конструктора:
// 构造器的通用形式：
public ClassName(Composite parent, int style) { ... }
```

代码示例 (Code Example)

```
public void createWidgets(Shell shell) {
    // ------------------------------------------------------
    // 1. Кнопка (Button)
    // ------------------------------------------------------
    // Родитель: shell
    // Стиль: SWT.PUSH (обычная кнопка)
    Button btn = new Button(shell, SWT.PUSH);
    btn.setText("Click Me");

    // ------------------------------------------------------
    // 2. Текстовое поле (Text)
    // ------------------------------------------------------
    // Стиль: SWT.BORDER (рамка) | SWT.PASSWORD (звездочки вместо букв)
    // RU: Мы комбинируем стили с помощью побитового ИЛИ (|)
    // CN: 我们使用按位或 (|) 来组合样式
    Text password = new Text(shell, SWT.BORDER | SWT.PASSWORD);

    // ------------------------------------------------------
    // 3. Метка (Label)
    // ------------------------------------------------------
    // Стиль: SWT.NONE (стиль по умолчанию)
    Label label = new Label(shell, SWT.NONE);
    label.setText("Enter Password:");
}
```

- Объяснение (解释):

1. Parent (Родитель): В отличие от Swing (где мы делаем `add()`), в SWT мы передаем родителя сразу в конструктор. Ребенок “приклеивается” к родителю навсегда.
2. Style (Стиль): Это битовая маска (int). Если нужно несколько стилей, их объединяют через `|`.
3. Parent (父级): 与 Swing（我们需要调用 `add()`）不同，在 SWT 中，我们立即将父级传递给构造器。子元素永远“粘”在父元素上。
4. Style (样式): 这是一个位掩码 (int)。如果需要多个样式，用 `|` 将它们合并。

18. Класс Widget библиотеки Standard Widget Toolkit. Освобождение ресурсов графики и управляющих эле Класс Control. Конструктор, стили, события, характерные методы класса Control.

这是 第 18 题 (Question 18) 的详细解析 。

这道题考察的是 SWT 最底层的机制：资源管理。如果不理解这一点，写的程序运行一会儿就会因为内存泄漏（准确说是操作系统句柄耗尽）而崩溃。

1. Класс Widget библиотеки Standard Widget Toolkit

Standard Widget Toolkit 库的 Widget 类

`Widget` 是所有 SWT UI 组件的老祖宗（抽象基类）。

```
public abstract class Widget {
    // ...
    // RU: Главный метод. Возвращает true, если виджет уже уничтожен.
    // CN: 核心方法。如果小部件已被销毁，则返回 true。
    public boolean isDisposed();

    // RU: Освобождает ресурсы ОС, связанные с этим виджетом.
    // CN: 释放与此小部件关联的操作系统资源。
    public void dispose();
    // ...
}
```

- Объяснение (解释): `Widget` — это корень иерархии. Он не обязательно имеет “тело” на экране (например, `TreeItem` — это Widget, но не Control). Его главная задача — связь с операционной системой. `Widget` 是层次结构的根。它不一定在屏幕上有“实体”（例如 `TreeItem` 是 Widget，但不是 Control）。它的主要任务是与操作系统通信。

2. Освобождение ресурсов графики и управляющих элементов

释放图形和控件资源

这是 SWT 与 Swing/JavaFX 最大的不同点。Java 的垃圾回收器 (GC) 管不了操作系统的资源！

两大原则 (Два правила):

1. Rule 1: 如果你 `new` 了一个 SWT 对象（特别是 `Color`, `Font`, `Image`），你必须负责 `dispose()` 它。
2. Rule 2: 对于控件（Controls），如果你销毁了父级（Parent），子级会自动被销毁。

```
public void resourceManagementDemo(Shell shell) {
    // [1] Создаем ресурс (Цвет)
    // [1] 创建资源（颜色）
    // RU: Мы используем "new", значит мы обязаны вызвать dispose().
    // CN: 我们使用了 "new"，这意味着我们必须负责调用 dispose()。
    Color redColor = new Color(shell.getDisplay(), 255, 0, 0);

    Button btn = new Button(shell, SWT.PUSH);
    btn.setBackground(redColor); // Используем ресурс (使用资源)

    // [2] Правильное освобождение (Correct Disposal)
    // RU: Когда окно закрывается, мы должны удалить цвет.
    // CN: 当窗口关闭时，我们要手动删除颜色。
    shell.addDisposeListener(e -> {
        // Проверяем, не удален ли он уже (检查是否已被删除)
        if (!redColor.isDisposed()) {
            redColor.dispose(); // Обязательно! (必须！)
        }
```

```
    });
}
```

- Объяснение (解释): В Java есть Garbage Collector, но он управляет только памятью Java. SWT использует ресурсы ОС (дескрипторы/handles). Если их не освободить методом `dispose()`, приложение "потечет" и может зависнуть. Java 有垃圾回收器，但它只管理 Java 内存。SWT 使用操作系统资源（句柄）。如果不通过 `dispose()` 释放它们，应用程序会发生"泄漏"并可能卡死。

---

3. Класс Control

Control 类

`Control` 继承自 `Widget`。它是所有可见元素的父类（有窗口句柄）。

- 继承链: `Widget` -> `Control` -> (`Button`, `Label`, `Composite`…)
- 本质: 凡是能在屏幕上画出来、能接受鼠标键盘事件的东西，都是 `Control`。

---

4. Конструктор, стили, события класса Control

Control 类的构造器、样式和事件

A. Конструктор (Constructor)

SWT 强制要求在构造时指定父级。

```
// public Control(Composite parent, int style)
// RU: Родитель обязателен. Стиль обязателен (можно передать
        SWT.NONE).
// CN: 父级是必须的。样式是必须的（可以传 SWT.NONE）。
Control button = new Button(shell, SWT.PUSH);
```

B. Стили (Styles)

样式定义了控件的初始外观。

```
// RU: Битовая маска стилей.
// CN: 样式的位掩码。
int style = SWT.BORDER      // Рамка (边框)
          | SWT.LEFT        // Выравнивание по левому краю (左对齐)
          | SWT.READ_ONLY;  // Только для чтения (只读)

Text text = new Text(shell, style);
```

C. События (Events)

`Control` 处理通用的交互事件（鼠标、键盘、焦点、绘制）。

```
Button btn = new Button(shell, SWT.PUSH);

// [1] Типизированный слушатель (Typed Listener) - проще
       использовать
// [1] 类型化监听器 - 使用更简单
btn.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseDown(MouseEvent e) {
        System.out.println("Click!");
    }
});

// [2] Универсальный слушатель (Untyped Listener) - более гибкий
// [2] 通用监听器 - 更灵活
btn.addListener(SWT.MouseDown, event -> {
    System.out.println("Low level click handling");
});
```

---

5. Фарактерные методы класса Control

Control 类的特征方法

这些是所有控件（按钮、文本框、列表）通用的方法。

```
public void manipulateControl(Control ctrl) {
    // 1. Геометрия (Geometry / 几何属性)
    ctrl.setSize(100, 50);      // Размер (尺寸)
    ctrl.setLocation(10, 10);   // Позиция (位置)
    // или (or)
    ctrl.setBounds(10, 10, 100, 50);

    // 2. Состояние (State / 状态)
    ctrl.setVisible(true);      // Показать/Скрыть (显示/隐藏)
    ctrl.setEnabled(false);     // Активен/Неактивен (Серя кнопка)
        (启用/禁用)

    // 3. Внешний вид (Appearance / 外观)
    ctrl.setBackground(new Color(null, 0, 255, 0)); // Фон (背景)
    ctrl.setForeground(new Color(null, 0, 0, 0));   // Текст (前景色/
        文字颜色)
    ctrl.setToolTipText("Это подсказка");           // Подсказка (工
        具提示)

    // 4. Фокус (Focus / 焦点)
    ctrl.setFocus(); // Запросить фокус ввода (请求输入焦点)

    // 5. Данные (Data / 数据)
    // RU: Можно прикрепить любой объект к контролу (как тег).
    // CN: 可以把任何对象绑定到控件上（像标签一样）。
    ctrl.setData("MyID", 12345);
}
```

---

19. Класс Shell. Конструктор, стили, события, характерные методы класса Shell. Класс Composite. Конструктор, стили, события, характерные методы класса Composite. Класс Canvas. Конструктор, стили, события, характерные методы класса Canvas.

这是 第 19 题 (Question 19) 的详细解析。

这道题涵盖了构建 GUI 界面最核心的三个"容器级"组件：窗口 (Shell)、容器 (Composite) 和 画布 (Canvas)。

---

1. Класс Shell (Окно)

Shell 类（窗口）

`Shell` 代表应用程序的一个窗口。它是所有其他控件的"根"。

A. Конструктор и создание (构造器与创建)

```
// [1] Главное окно приложения (Main Window)
// [1] 应用程序的主窗口
Display display = new Display();
Shell mainShell = new Shell(display, SWT.SHELL_TRIM);

// [2] Диалоговое окно (Dialog Window)
// [2] 对话框窗口（依赖于主窗口）
// RU: Модальное окно (блокирует родителя)
// CN: 模态窗口（阻塞父窗口）
Shell dialog = new Shell(mainShell, SWT.DIALOG_TRIM | SWT.APPLICATI
        ON_MODAL);
```

B. Стили (Styles)

`Shell` 的样式决定了窗口的行为和外观。

样式 (Style)	说明 (Description)
`SWT.SHELL_TRIM`	标准窗口 (标题栏 + 最小化 + 最大化 + 关闭 + 可调整大小)。
`SWT.DIALOG_TRIM`	对话框标准 (标题栏 + 关闭按钮 + 边框)。
`SWT.NO_TRIM`	无边框窗口 (没有任何装饰，通常用于启动画面 Splash Screen)。
`SWT.ON_TOP`	总在最前 (Always on top)。
`SWT.RESIZE`	可调整大小。
`SWT.APPLICATION_MODAL`	模态 (用户必须先关闭此窗口才能操作其他窗口)。

C. Фарактерные методы (特征方法)

```
mainShell.setText("My Application"); // Заголовок (标题)
mainShell.setImage(new Image(display, "icon.png")); // Иконка (图标)

mainShell.pack(); // Подогнать размер под содержимое (根据内容自动调整大
        小)
// ИЛИ (OR)
mainShell.setSize(800, 600); // Явный размер (显式设置大小)

mainShell.open(); // Сделать видимым и активным (显示并激活)
mainShell.close(); // Закрыть (запускает событие dispose) (关闭)
```

D. События (Events)

主要通过 `ShellListener` 处理。

- `shellClosed`: 当用户点击"X"关闭按钮时触发（可以在这里拦截关闭，比如弹出"是否保存？"）。
- `shellActivated`: 窗口获得焦点。

---

2. Класс Composite (Композит / Контейнер)

Composite 类（复合容器）

`Composite` 是所有可以包含其他控件的控件的父类。你可以把它想象成 HTML 中的 `<div>` 或者 Swing 中的 `JPanel`。

A. Конструктор (Constructor)

```
// Всегда нужен родитель!
// 永远需要父级！
Composite panel = new Composite(shell, SWT.BORDER);
```

B. Стили (Styles)

样式 (Style)	说明 (Description)
`SWT.BORDER`	边框 (在面板周围画一条线)。
`SWT.NO_BACKGROUND`	无背景 (不绘制背景，用于防止闪烁或自定义绘制)。
`SWT.NO_FOCUS`	无焦点 (此面板不接受键盘焦点)。
`SWT.DOUBLE_BUFFERED`	双重缓冲 (用于消除绘图闪烁)。

C. Фарактерные методы (特征方法)

`Composite` 的核心功能是布局管理 (Layout Management)。

```
// 1. Установка Layout (布局)
// RU: Говорим, как располагать детей внутри.
// CN: 告诉它如何排列内部的子元素。
panel.setLayout(new GridLayout(2, false));

// 2. Управление детьми (Children Management)
Control[] children = panel.getChildren(); // Получить все элементы
        внутри (获取内部所有元素)

// 3. Обновление (Refresh)
// RU: Если вы добавили кнопку динамически, нужно пересчитать
       размеры.
// CN: 如果你动态添加了按钮，需要重新计算尺寸。
panel.layout();
```

---

3. Класс Canvas (Фолст)

Canvas 类（画布）

`Canvas` 是 `Composite` 的子类，专门用于自定义绘图。如果你想画线条、圆圈、图表，或者显示复杂的图片，就用它。

A. Конструктор (Constructor)

```
Canvas canvas = new Canvas(shell, SWT.NONE);
```

B. Стили (Styles)

通常与 `Composite` 相同，但常结合以下样式使用：

- `SWT.NO_BACKGROUND`: 不要在每次重绘时擦除背景（防止闪烁）。
- `SWT.NO_REDRAW_RESIZE`: 改变大小时不重绘（性能优化）。

C. События (Events) - PaintListener

这是 `Canvas` 的灵魂。你必须监听 `Paint` 事件来绘图。

```
canvas.addPaintListener(new PaintListener() {
    @Override
    public void paintControl(PaintEvent e) {
        // GC = Graphic Context (Графический контекст)
        // GC = 图形上下文 (画笔)
        GC gc = e.gc;

        gc.setBackground(display.getSystemColor(SWT.COLOR_BLUE));
        gc.fillOval(10, 10, 100, 100); // Рисуем синий круг (画个蓝圆)
        gc.drawText("Hello Canvas", 20, 50);
    }
});
```

D. Фарактерные методы (特征方法)

- `scroll(int destX, int destY, ...)`: 高效滚动画布内容。
- `redraw()`: 非常重要！ 告诉系统“这里脏了，请触发 Paint 事件重绘”。不要直接调用 paint 方法，要调用 `redraw()`。

---

20. Структурирование интерфейса пользователя с помощью классов TabFolder и TabItem. Конструктор, ст события, характерные методы классов TabFolder и TabItem.

这是 第 20 题 (Question 20) 的完整详细解析 。

这道题考察的是如何构建 选项卡式界面 (Tabbed Interface)。这是现代 GUI 应用程序（如浏览器、设置面板、IDE 编辑器）中最常见的布局模式之一。

在 SWT 中，结构化界面的核心在于理解三个组件的关系：

1. TabFolder (容器/文件夹)
2. TabItem (标签头)
3. Control (标签对应的内容，通常是 `Composite`)

---

1. Структурирование интерфейса пользователя

用户界面的结构化

```
// Структура (结构):
// TabFolder (Папка)
//   |-- TabItem 1 (Вкладка 1) -> Control 1 (Содержимое 1)
//   |-- TabItem 2 (Вкладка 2) -> Control 2 (Содержимое 2)
```

- Объяснение (解释): `TabItem` сам по себе — это просто заголовок. Чтобы отобразить содержимое (кнопки, текст), нужно создать отдельный контейнер (например, `Composite`) и привязать его к вкладке методом `setControl()`. `TabItem` 本身只是一个标题。为了显示内容（按钮、文本），你需要创建一个独立的容器（例如 `Composite`），并通过 `setControl()` 方法将其绑定到选项卡上。

---

2. Класс TabFolder

TabFolder 类（选项卡容器）

它是所有选项卡的“父亲”。

A. Конструктор и Стили (Constructor & Styles)

```
// RU: Создаем контейнер вкладок.
// CN: 创建选项卡容器。
TabFolder folder = new TabFolder(shell, SWT.TOP | SWT.BORDER);
```

Стили (Styles):

- `SWT.TOP`: Вкладки сверху (标签在顶部 - 标准样式)。
- `SWT.BOTTOM`: Вкладки снизу (标签在底部)。
- `SWT.BORDER`: Рамка (边框)。

B. События (Events)

Самое важное событие — выбор вкладки.

```
folder.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        // RU: Получаем выбранную вкладку.
        // CN: 获取被选中的选项卡。
        TabItem selectedItem = (TabItem) e.item;
        System.out.println("Выбрано: " + selectedItem.getText());
    }
});
```

C. Фарактерные методы (Methods)

- `getSelectionIndex()`: Возвращает индекс выбранной вкладки (返回选中标签的索引)。
- `setSelection(int index)`: Программно переключает вкладку (代码切换标签)。

---

3. Класс TabItem

TabItem 类（选项卡项）

它是具体的“标签页耳朵”。

A. Конструктор (Constructor)

```
// RU: Прикрепляем вкладку к папке.
// CN: 将选项卡绑定到文件夹上。
TabItem item = new TabItem(folder, SWT.NONE);
```

B. Фарактерные методы (Methods)

- `setText(String text)`: Заголовок вкладки (设置标题).
- `setImage(Image image)`: Иконка (设置图标).
- `setControl(Control control)`: Ключевой метод! Привязывает содержимое к вкладке (关键方法！绑定内容到选项卡).

---

4. Полный пример кода (Complete Example)

完整代码示例

这个例子展示了如何正确地把 `Composite`、`TabItem` 和 `TabFolder` 组装在一起。

```
public void createTabbedUI(Shell shell) {
    shell.setLayout(new FillLayout()); // Растянуть на все окно (填满窗口)

    // 1. Создаем папку (创建文件夹)
    TabFolder folder = new TabFolder(shell, SWT.TOP);

    // --- Вкладка 1: Общие настройки (Tab 1: General Settings) ---
    TabItem tab1 = new TabItem(folder, SWT.NONE);
    tab1.setText("General");

    // Создаем содержимое для первой вкладки (Composite)
    // 为第一个标签创建内容容器
    Composite comp1 = new Composite(folder, SWT.NONE);
    comp1.setLayout(new RowLayout());
    new Button(comp1, SWT.PUSH).setText("Button 1");
    new Button(comp1, SWT.CHECK).setText("Option A");

    // !!! Связываем вкладку и содержимое !!!
    // !!! 将标签与内容绑定 !!!
    tab1.setControl(comp1);

    // --- Вкладка 2: Редактор (Tab 2: Editor) ---
    TabItem tab2 = new TabItem(folder, SWT.NONE);
    tab2.setText("Editor");

    Text text = new Text(folder, SWT.BORDER | SWT.MULTI);
    text.setText("Type here...");

    // Связываем (绑定)
    tab2.setControl(text);
}
```

- Объяснение (解释): Обратите внимание: родитель для `comp1` и `text` — это **folder**, а не shell. Но `TabItem` управляет их видимостью (setVisible), когда пользователь переключает вкладки. 注意：`comp1` 和 `text` 的父级是 **folder**，而不是 shell。但是当用户切换标签时，`TabItem` 负责管理它们的可见性 (setVisible)。

---

21. Рисование графических элементов с помощью класса Graphics Context (GC) библиотеки Standard Widget Toolkit (SWT). Рисование линий, фигур, изображений графических файлов, курсоров. Задание их атрибутов представления

这是 第 21 题 (Question 21) 的详细解析 。

这道题是关于 SWT 图形绘制 (GDI) 的核心。所有的自定义绘图（画图板、图表、游戏画面）都依赖于 `GC` (Graphics Context) 类。

我们将题目拆解为 5 个小问来回答。

---

1. Рисование графических элементов с помощью класса Graphics Context (GC)

使用 Graphics Context (GC) 类绘制图形元素

`GC` (Graphics Context) 是你的“画笔”。你不能凭空画画，通常需要在 `Canvas` 的 `PaintListener` 中获取它。

```
Canvas canvas = new Canvas(shell, SWT.DOUBLE_BUFFERED);

canvas.addPaintListener(new PaintListener() {
    @Override
    public void paintControl(PaintEvent e) {
        // RU: Получаем объект GC из события. Это наша кисть.
        // EN: Get the GC object from the event. This is our brush.
        // CN: 从事件中获取 GC 对象。这就是我们的画笔。
        GC gc = e.gc;

        // Теперь можно рисовать... (Now we can draw...)
        gc.drawString("Hello Graphics", 10, 10);
    }
});
```

- Объяснение (解释): `GC` — это класс, предоставляющий методы для рисования на `Drawable` поверхностях (Shell, Composite, Canvas, Image). Самый правильный способ рисования — внутри события `PaintEvent`. `GC` 是一个类，提供了在 `Drawable` 表面（Shell, Composite, Canvas, Image）上绘图的方法。最正确的绘图方式是在 `PaintEvent` 事件内部进行。

---

2. Рисование линий

绘制线条

```
// RU: Рисует линию от точки (x1, y1) до (x2, y2).
// CN: 绘制一条从点 (x1, y1) 到 (x2, y2) 的线。
gc.drawLine(0, 0, 100, 100);

// RU: Рисует ломаную линию (соединяет массив точек).
// CN: 绘制折线（连接点数组）。
int[] points = {10,10, 50,50, 100,10};
gc.drawPolyline(points);
```

- Объяснение (解释): Метод `drawLine` соединяет две точки. Координаты начинаются с верхнего левого угла (0,0). Ось Y направлена вниз. `drawLine` 方法连接两个点。坐标系从左上角 (0,0) 开始。Y 轴向下延伸。

---

3. Рисование фигур (прямоугольники, овалы)

绘制图形（矩形、椭圆）

这里要区分 Draw (轮廓) 和 Fill (填充)。

```
// 1. Прямоугольник (Rectangle)
// RU: Только рамка (контур). Использует Foreground color.
// CN: 只有边框（轮廓）。使用前景色。
gc.drawRectangle(10, 10, 200, 100);

// RU: Залитая фигура. Использует Background color.
// CN: 实心图形。使用背景色。
gc.fillRectangle(10, 10, 200, 100);

// 2. Овал / Круг (Oval / Circle)
// RU: Вписывается в прямоугольник.
// CN: 内切于矩形。
gc.drawOval(50, 50, 100, 100); // Круг (Circle)
gc.fillOval(50, 50, 100, 100);
```

- Объяснение (解释): Методы, начинающиеся с `draw` (например, `drawRectangle`), рисуют контур цветом переднего плана. Методы `fill` (например, `fillRectangle`) закрашивают фигуру цветом фона. 以 `draw` 开头的方法（如 `drawRectangle`）用前景色绘制轮廓。以 `fill` 开头的方法（如 `fillRectangle`）用背景色填充图形。

---

4. Рисование изображений графических файлов

绘制图形文件的图像

注意：图像是昂贵的资源，用完记得 `dispose()` ！

```
// [1] Загрузка (Loading)
// RU: Загружаем картинку из файла.
// CN: 从文件加载图片。
Image image = new Image(display, "logo.png");

// [2] Рисование (Drawing)
// RU: Рисуем картинку в точке (0, 0).
// CN: 在点 (0, 0) 处绘制图片。
gc.drawImage(image, 0, 0);

// RU: Масштабирование (рисуем с изменением размера).
// CN: 缩放（改变大小绘制）。
// gc.drawImage(image, srcX, srcY, srcW, srcH, destX, destY, destW, destH);

// [3] Освобождение (Disposal) - ОЧЕНЬ ВАЖНО!
// CN: 释放资源 - 非常重要！
// image.dispose(); (Делать это нужно при закрытии окна)
```

- Объяснение (解释): Класс `Image` используется для загрузки графики (PNG, JPG, GIF). Метод `gc.drawImage` копирует пиксели изображения на холст. `Image` 类用于加载图形（PNG, JPG, GIF）。`gc.drawImage` 方法将图像像素复制到画布上。

---

5. Рисование курсоров (и установка курсоров)

绘制光标（及设置光标）

通常我们不“画”光标，而是“设置”光标。但如果要自定义光标，也是通过图像来实现的。

```
// [1] Системный курсор (System Cursor)
// RU: Стандартная "рука".
// CN: 标准的“手型”光标。
Cursor handCursor = display.getSystemCursor(SWT.CURSOR_HAND);
shell.setCursor(handCursor);

// [2] Пользовательский курсор (Custom Cursor)
// RU: Создаем курсор из картинки.
// CN: 从图片创建光标。
// Cursor custom = new Cursor(display, sourceData, maskData, hotX, hotY);
// shell.setCursor(custom);
```

- Объяснение (解释): Курсор устанавливается для конкретного контрола методом `setCursor`. Можно использовать системные курсоры (стрелка, рука, ожидание) или создавать свои из изображений. 光标是通过 `setCursor` 方法为特定控件设置的。可以使用系统光标（箭头、手型、等待）或从图像创建自定义光标。

---

6. Задание их атрибутов представления на экране

设置它们在屏幕上的显示属性

这些设置必须在调用 `draw` 或 `fill` 之前完成。

```
// 1. Цвет (Color)
// RU: Цвет линий/текста.
// CN: 线条/文本的颜色。
gc.setForeground(display.getSystemColor(SWT.COLOR_RED));

// RU: Цвет заливки.
// CN: 填充颜色。
gc.setBackground(display.getSystemColor(SWT.COLOR_YELLOW));

// 2. Толщина линии (Line Width)
// RU: Ширина линии в пикселях.
// CN: 线条宽度（像素）。
gc.setLineWidth(5);

// 3. Стиль линии (Line Style)
// RU: Сплошная, пунктирная, точечная...
// CN: 实线、虚线、点线……
gc.setLineStyle(SWT.LINE_DASH); // - - - -
// gc.setLineStyle(SWT.LINE_SOLID); // _______
```

- Объяснение (解释): `GC` имеет состояние. Если вы установите красный цвет, все последующие операции рисования будут красными, пока вы не измените цвет снова. `GC` 是有状态的。如果你设置了红色，后续所有的绘图操作都会是红色的，直到你再次更改颜色。

---

22. Обработка событий в библиотеке Standard Widget Toolkit. Интерфейс обработчика события PaintListener перерисовки изображений на экране.

这是 第 22 题 (Question 22) 的详细解析。

这道题考察的是 SWT 中最核心的交互机制——事件处理，特别是绘图事件。如果不理解 `PaintListener`，你就无法在屏幕上绘制动态的图像或自定义控件。

---

1. Обработка событий в библиотеке SWT

SWT 库中的事件处理

SWT 使用标准的 观察者模式 (Observer Pattern)。

- Event (Событие): 用户做的事情（点击、移动鼠标、改变窗口大小）。
- Listener (Слушатель): 一个接口，包含处理事件的方法。
- Widget (Виджет): 事件源（按钮、窗口、画布）。

```
// Общая схема (General Scheme / 通用模式):
widget.addSomeListener(new SomeListener() {
    public void handleEvent(Event e) {
        // Code...
    }
});
```

- Объяснение (解释): В SWT обработка событий происходит через добавление слушателей (Listeners) к виджетам. Когда происходит событие, ОС сообщает об этом SWT, а SWT вызывает метод вашего слушателя. 在 SWT 中，事件处理是通过向小部件添加监听器 (Listeners) 来完成的。当事件发生时，操作系统通知 SWT，然后 SWT 调用你的监听器方法。

---

2. Интерфейс обработчика события PaintListener

PaintListener 事件处理接口

这是专门用于绘图的接口。当操作系统认为窗口的一部分需要“重绘”（比如窗口被遮挡后又显示出来，或者大小改变了）时，就会触发这个事件。

定义 (Definition):

```
public interface PaintListener extends SWTEventListener {
    // RU: Вызывается, когда нужно перерисовать виджет.
    // CN: 当需要重绘小部件时被调用。
    public void paintControl(PaintEvent e);
}
```

---

3. Перерисовка изображений на экране (Пример кода)

在屏幕上重绘图像（代码示例）

要在屏幕上画图，通常使用 `Canvas` 组件加上 `PaintListener`。

关键点 (Key Points):

1. PaintEvent e: 事件对象包含一个 `GC` (Graphics Context)。
2. e.gc: 只能在 `paintControl` 方法内部使用这个 GC。不要 dispose 它！（它是系统传给你的，系统负责释放）。

```
import org.eclipse.swt.events.PaintEvent;
import org.eclipse.swt.events.PaintListener;
// ... imports ...

public void setupPainting(Shell shell) {
    // [1] Создаем Canvas (Холст)
    // CN: 创建画布，开启双重缓冲以防闪烁
    Canvas canvas = new Canvas(shell, SWT.DOUBLE_BUFFERED);

    // Загружаем изображение (Load Image)
    Image image = new Image(shell.getDisplay(), "photo.png");

    // [2] Добавляем PaintListener
    // CN: 添加绘图监听器
    canvas.addPaintListener(new PaintListener() {
        @Override
        public void paintControl(PaintEvent e) {
            // RU: Получаем GC из события. НЕ удаляйте (dispose) этот GC!
            // EN: Get GC from event. DO NOT dispose this GC!
            // CN: 从事件中获取 GC。不要 dispose 这个 GC！
            GC gc = e.gc;

            // RU: Рисуем изображение
            // CN: 绘制图像
            if (image != null) {
                // (image, x, y)
                gc.drawImage(image, 10, 10);
            }

            // RU: Рисуем рамку поверх изображения
            // CN: 在图像上方画一个边框
            gc.setForeground(shell.getDisplay().getSystemColor(SWT.COLOR_RED));
            gc.drawRectangle(10, 10, 100, 100);
        }
    });

    // [3] Важно! Освобождаем картинку при закрытии окна
    // CN: 重要！窗口关闭时释放图片资源
    shell.addDisposeListener(event -> {
        if (image != null && !image.isDisposed()) image.dispose();
    });
}
```

---

4. Метод redraw() — Как запустить перерисовку вручную?

redraw() 方法 — 如何手动触发重绘?

`paintControl` 是被动调用的。如果你更改了数据（比如图片坐标变了），你需要通知系统“嘿，这里脏了，请重绘”。

```
// RU: Этот метод говорит системе: "Вызови PaintListener как можно скорее".
// EN: This method tells the system: "Call PaintListener as soon as possible".
// CN: 这个方法告诉系统：“尽快调用 PaintListener”。
canvas.redraw();
```

- Объяснение (解释): Никогда не вызывайте `paintControl()` напрямую! Если вы хотите обновить экран, вызовите метод `redraw()` у виджета. SWT поставит задачу перерисовки в очередь событий. 永远不要直接调用 `paintControl()`！如果你想更新屏幕，请调用小部件的 `redraw()` 方法。SWT 会将重绘任务放入事件队列中。

---

23. Интерфейс обработчика клавиатуры мыши MouseListener, вращения колеса MouseWheelListener, переме мыши MouseMoveListener.

这是 第 23 题 (Question 23) 的详细解析。

这道题考察的是用户与程序最基本的交互方式：鼠标操作。在 SWT 中，鼠标事件被细分为三个不同的接口，分别处理点击、移动和滚动。

1. Интерфейс MouseListener (Обработка кликов)

MouseListener 接口（处理点击）

用于处理鼠标按键的状态变化（按下、松开、双击）。

Методы (Methods):

- `mouseDown(MouseEvent e)`: 按下鼠标键时触发。
- `mouseUp(MouseEvent e)`: 松开鼠标键时触发。
- `mouseDoubleClick(MouseEvent e)`: 双击时触发。

```
Button btn = new Button(shell, SWT.PUSH);
btn.setText("Click Me");

// RU: Используем MouseAdapter, чтобы не реализовывать все 3 метода.
// CN: 使用 MouseAdapter，这样就不需要实现所有 3 个方法。
btn.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseDown(MouseEvent e) {
        // e.button: 1 = Left, 2 = Middle, 3 = Right
        System.out.println("Mouse Down: Button " + e.button);
    }

    @Override
    public void mouseDoubleClick(MouseEvent e) {
        System.out.println("Double Click!");
    }
});
```

- Объяснение (解释): `MouseListener` отслеживает именно нажатия кнопок. Объект `MouseEvent` содержит информацию о том, какая кнопка была нажата (`e.button`) и где (`e.x`, `e.y`). `MouseListener` 专门跟踪按键动作。`MouseEvent` 对象包含按下了哪个键 (`e.button`) 以及位置 (`e.x`, `e.y`) 的信息。

---

2. Интерфейс MouseMoveListener (Перемещение мыши)

MouseMoveListener 接口（鼠标移动）

用于追踪鼠标指针的移动轨迹。注意：这个事件触发频率极高！

Методы (Methods):

- `mouseMove(MouseEvent e)`: 当鼠标在控件上方移动时不断触发。

```
Canvas canvas = new Canvas(shell, SWT.BORDER);

canvas.addMouseMoveListener(new MouseMoveListener() {
    @Override
    public void mouseMove(MouseEvent e) {
        // RU: Получаем текущие координаты курсора.
        // CN: 获取光标的当前坐标。
        String position = "X: " + e.x + ", Y: " + e.y;

        // RU: Обновляем заголовок окна (не делайте тяжелых
        вычислений здесь!)
        // CN: 更新窗口标题（不要在这里做繁重的计算！）
        shell.setText(position);
    }
});
```

- Объяснение (解释): Используется для рисования (drag-and-drop), игр или подсветки элементов. Будьте осторожны: тяжелый код внутри `mouseMove` замедлит работу программы. 用于绘图（拖放）、游戏或元素高亮。请注意：`mouseMove` 中的繁重代码会拖慢程序运行速度。

---

3. Интерфейс MouseWheelListener (Вращение колеса)

MouseWheelListener 接口（滚轮旋转）

用于处理鼠标滚轮的滚动。

Методы (Methods):

- `mouseScrolled(MouseEvent e)`: 当滚轮滚动时触发。

```
shell.addMouseWheelListener(new MouseWheelListener() {
    @Override
    public void mouseScrolled(MouseEvent e) {
        // RU: e.count показывает направление и силу прокрутки.
        // > 0 : Вверх (Up)
        // < 0 : Вниз (Down)
        // CN: e.count 显示滚动的方向和力度。
        // > 0 : 向上
        // < 0 : 向下

        if (e.count > 0) {
            System.out.println("Scroll UP");
        } else {
            System.out.println("Scroll DOWN");
        }
    }
});
```

- Объяснение (解释): Главное поле здесь — `e.count`. В Windows обычно один "щелчок" колеса равен +/- 3, но это зависит от настроек ОС. 这里的主要字段是 `e.count`。在 Windows 中，通常滚轮的一"格"等于 +/- 3，但这取决于操作系统的设置。

---

4. Объект MouseEvent (Событие мыши)

MouseEvent 对象

所有上述监听器都接收同一个参数 `MouseEvent`。它的核心字段如下：

字段 (Field)	说明 (Description)
`x`, `y`	鼠标相对于控件的坐标 (Coordinates relative to control)。
`button`	按下的键 (1=左, 2=中, 3=右)。仅在 `MouseListener` 中有效。
`count`	滚动量。仅在 `MouseWheelListener` 中有效。
`stateMask`	键盘修饰键状态 (Shift, Ctrl, Alt)。用于检测 "Ctrl+Click"。

---

24. Списки библиотеки SWT в интерфейсе пользователя. Инициализация списков. Слушатели событий. Опре выбранного элемента списка.

这是 第 24 题 (Question 24) 的详细解析。

这道题考察的是 SWT 中用于显示简单文本列表的控件：List。注意不要把它和 Java 集合框架中的 `java.util.List` 搞混了，这里的全名是 `org.eclipse.swt.widgets.List`。

---

1. Списки библиотеки SWT в интерфейсе пользователя

用户界面中的 SWT 列表

`List` 控件用于显示一列字符串，用户可以从中选择一项或多项。

---

2. Инициализация списков

列表的初始化

在创建列表时，我们需要决定它是单选 (Single) 还是多选 (Multi)。

```
// Импорт: org.eclipse.swt.widgets.List

public void createList(Shell shell) {
    // [1] Создание (Creation)
    // Стиль SWT.MULTI: Можно выбрать несколько элементов
        (Ctrl+Click).
    // Стиль SWT.V_SCROLL: Вертикальная прокрутка.
    // CN: SWT.MULTI 样式：可以选择多个元素 (Ctrl+Click)。
    // CN: SWT.V_SCROLL 样式：垂直滚动条。
    List list = new List(shell, SWT.BORDER | SWT.MULTI | SWT.V_SCRO
        LL);

    // [2] Заполнение данными (Populating Data)
    // RU: Добавление по одному элементу.
    // CN: 逐个添加元素。
    list.add("Java");
    list.add("Python");
    list.add("C++");

    // RU: Добавление массива строк (заменяет старые данные!).
    // CN: 添加字符串数组（会覆盖旧数据！）。
    String[] items = { "Spring", "Hibernate", "SWT" };
    list.setItems(items);

    // RU: Выделение элемента по индексу (например, 0-й).
    // CN: 根据索引选中元素（例如第 0 个）。
    list.select(0);
}
```

- Объяснение (解释): Основные методы инициализации: `add()` добавляет строку в конец, а `setItems()` полностью перезаписывает содержимое списка. 初始化的主要方法：`add()` 在末尾添加字符串，而 `setItems()` 会完全覆盖列表内容。

---

3. Слушатели событий

事件监听器

要响应用户的点击操作，我们使用 `SelectionListener`。

```java
list.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        // RU: Этот метод вызывается при клике мышью или нажатии
        Enter.
        // CN: 当鼠标点击或按下回车键时调用此方法。
        System.out.println("User clicked on the list");

        // Внимание: объект 'e' здесь не содержит текст выбранного
        элемента.
        // Нужно спрашивать у самого списка (см. пункт 4).
        // 注意：这里的对象 'e' 不包含选中的文本。需要去问列表本身（见第 4 点）。
    }

    @Override
    public void widgetDefaultSelected(SelectionEvent e) {
        // RU: Вызывается при двойном клике (Double Click).
        // CN: 双击时调用。
        System.out.println("Double click detected!");
    }
});
```

- Объяснение (解释): `widgetSelected` срабатывает при одиночном выборе. `widgetDefaultSelected` срабатывает при двойном щелчке (или Enter). `widgetSelected` 在单击选择时触发。`widgetDefaultSelected` 在双击（或回车）时触发。

---

4. Определение выбранного элемента списка

确定列表中被选中的元素

因为列表可能是多选的，所以 SWT 总是返回数组。

```java
// [1] Получение текста (Getting Text)
// RU: Возвращает массив строк, которые выбрал пользователь.
// CN: 返回用户选中的字符串数组。
String[] selectedItems = list.getSelection();

if (selectedItems.length > 0) {
    for (String item : selectedItems) {
        System.out.println("Selected: " + item);
    }
}

// [2] Получение индексов (Getting Indices)
// RU: Возвращает индекс (номер) выбранного элемента.
// CN: 返回选中元素的索引（编号）。
int index = list.getSelectionIndex(); // Для одиночного выбора (单选)
int[] indices = list.getSelectionIndices(); // Для множественного (多
    选)

if (index != -1) {
    System.out.println("Selected index: " + index);

    // RU: Можно удалить выбранный элемент.
    // CN: 可以删除选中的元素。
    // list.remove(index);
}
```

- Объяснение (解释): Метод `getSelection()` возвращает массив строк (`String[]`). Если ничего не выбрано, массив будет пустым (но не null). Метод `getSelectionIndex()` возвращает `-1`, если ничего не выбрано. `getSelection()` 方法返回字符串数组（`String[]`）。如果没有选中任何内容，数组为空（但不是 null）。如果没有选中任何内容，`getSelectionIndex()` 返回 `-1`。

---

25. Кнопки библиотеки SWT в интерфейсе пользователя. Радио-кнопки. Инициализация кнопок. Слушатели с Определение выбора пользователя.

这是 第 25 题 (Question 25) 的详细解析。

这道题考察的是 SWT 中最基础的交互控件——按钮 (Button)。 在 SWT 中，普通按钮、复选框 (Checkbox) 和 单选按钮 (Radio Button) 实际上都是同一个类 (`org.eclipse.swt.widgets.Button`)，只是构造时的样式 (Style) 不同。

---

1. Кнопки библиотеки SWT и Инициализация кнопок

SWT 库的按钮与初始化

要创建不同类型的按钮，我们在构造器中传入不同的常量。

```java
// [1] Обычная кнопка (Push Button)
// RU: Стиль SWT.PUSH. Используется для выполнения действия.
// CN: 样式 SWT.PUSH。用于执行动作（如“确定”、“取消”）。
Button pushBtn = new Button(shell, SWT.PUSH);
pushBtn.setText("Save");

// [2] Флажок (Checkbox)
// RU: Стиль SWT.CHECK. Имеет состояние (галочка есть/нет).
// CN: 样式 SWT.CHECK。具有状态（勾选/未勾选）。
Button checkBtn = new Button(shell, SWT.CHECK);
checkBtn.setText("I agree to terms");

// [3] Радио-кнопка (Radio Button)
// RU: Стиль SWT.RADIO. Только одна кнопка в группе может быть
    выбрана.
// CN: 样式 SWT.RADIO。组内只能选中一个按钮。
Button radioBtn = new Button(shell, SWT.RADIO);
radioBtn.setText("Option A");
```

- Объяснение (解释): Класс всегда один — `Button`. Поведение определяется стилем. 类始终只有一个——`Button`。行为由样式决定。

---

2. Радио-кнопки (Radio Buttons)

单选按钮

单选按钮有一个特殊的“互斥逻辑” (Exclusion Logic)。

```java
// RU: Чтобы сгруппировать радио-кнопки, их нужно поместить в один
    Composite (или Group).
// CN: 为了将单选按钮分组，必须将它们放在同一个 Composite（或 Group）中。

Group group = new Group(shell, SWT.SHADOW_ETCHED_IN);
group.setText("Choose Gender");
group.setLayout(new RowLayout(SWT.VERTICAL));

// Эти две кнопки теперь связаны (внутри group):
// 这两个按钮现在是关联的（在 group 内部）：
Button male = new Button(group, SWT.RADIO);
male.setText("Male");

Button female = new Button(group, SWT.RADIO);
female.setText("Female");
```

- Объяснение (解释): SWT автоматически обеспечивает логику “только один выбран”. Все кнопки `SWT.RADIO`, имеющие одного и того же родителя, становятся одной группой. SWT 自动提供“单选”逻辑。所有具有**相同父级的** `SWT.RADIO` 按钮会自动组成一个组。

---

3. Слушатели событий

事件监听器

所有按钮都使用 `SelectionListener`。

```java
pushBtn.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        // RU: Код выполняется при клике.
        // CN: 点击时执行代码。
        System.out.println("Button Clicked!");

        // RU: Источник события (какая именно кнопка нажата).
        // CN: 事件源（具体是哪个按钮被按下了）。
        Button source = (Button) e.getSource();
    }
});
```

- Объяснение (解释): Мы используем `SelectionAdapter` (класс-заглушку), чтобы не реализовывать метод `widgetDefaultSelected` (он редко нужен для кнопок). 我们使用 `SelectionAdapter`（适配器类），这样就不需要实现 `widgetDefaultSelected` 方法（按钮通常不需要它）。

---

4. Определение выбора пользователя

确定用户的选择

对于 `CHECK` 和 `RADIO` 类型的按钮，我们需要知道它们是否被选中。

```java
// Метод getSelection() возвращает boolean
// getSelection() 方法返回 boolean

// Пример проверки (Check Example):
if (checkBtn.getSelection()) {
    System.out.println("User agreed.");
} else {
    System.out.println("User disagreed.");
}

// Пример для радио-кнопок (Radio Example):
if (male.getSelection()) {
    System.out.println("Selected: Male");
}
```

- Объяснение (解释): Метод `getSelection()` работает только для кнопок с состоянием (`CHECK`, `RADIO`, `TOGGLE`). Для обычной кнопки (`PUSH`) он не имеет смысла (она не хранит состояние). `getSelection`

() 方法仅适用于有状态的按钮（CHECK, RADIO, TOGGLE）。对于普通按钮（PUSH），它没有意义（因为普通按钮不保存状态）。