

// 易错陷阱:
 - 向 c.localCount? -> 还是 1 (因为它是 int, 赋值后就不变了)。
 - 向 c.globalCount? -> 是 3 (因为它是 static, c3 把它改成了 3, c1 也能看到 3)。

```
System.out.println(c1.localCount); // 输出 1
System.out.println(counter.globalCount); // 输出 3
```

找错题源

主题: Part C Q16 - 综合找错题模版 (Debugging Class)

题目描述 (中文): 下面的代码定义了一个 `GymMember` (健身房会员) 类, 它的功能包括: 记录会员名字、余额, 并在消费时扣款。代码中至少包含 6 处严重错误 (编译错误或逻辑错误)。请你: 1. 找出错误所在的行。2. 解释错误原因。3. 说明如何修正。

```
import java.util.ArrayList;
```

```
public class GymMember {
```

// [1] 所有的会员共享了同一个余额? 这合理吗?

```
public static double balance;
```

// [2] 名字直接暴露给外部修改, 破坏了封装性

```
public String name;
```

// [3] 初始化初始 List 就直接使用

```
private ArrayList<String> classesAttended;
```

// [4] 构造函数写了 void, 变成了普通方法

```
public void gymMember(String initialName, double initialBalance);
```

```
    // 参数初值:
```

```
    balance = initialBalance;
```

```
    // classesAttended 没有在这里 new ArrayList<>(), 后面会报空指针
```

// [5] 字符串比较使用了 ==

```
public boolean isMember(String inputName) {
```

```
    if (name == inputName) {
```

```
        return true;
```

```
    } else {
```

```
        return false;
```

```
    }
```

public void attendClass(String className, double cost) {

// [6] 逻辑错误: 没钱也能上课 (余额小于花费时应该拒绝)

```
    if (balance < cost) {
```

```
        System.out.println("Success! Class attended.");
```

```
        balance = cost;
```

 } else {

```
        System.out.println("Insufficient funds.");
```

```
    }
```

// [7] 集合语法错误: List 不能像数组那样用 [] 访问或赋值

```
// classesAttended[0] = className;
```

```
classesAttended.add(className);
```

```
System.out.println("Member: " + this.name);
```

* 错误 1: 静态量滥用 (Static Variable Misuse)

* 问题: static 意味着“全局共享”。如果我是会员 A, 你是会员 B, 我存了 100 块, 你的余额也会变 100 块。

* 修正: 去掉 static。

```
* private double balance;
```

* 错误 2: 封装性破坏 (Breaking Encapsulation)

* 问题: public String name; 使用了 this 关键字区分变量和参数变量 (这是一个好习惯, 虽然不是必须)

* 修正: 去掉 this。

```
* public GymMember(...)
```

* 错误 3: 字符串比较 (String Comparison)

* 位置: if (name == inputName)

* 问题: == 比较的是内存地址。如果两个 String 内容一样但地址不同, 会返回 false。

* 修正: 使用 equals()。

```
* if (name.equals(inputName))
```

* 错误 4: 逻辑反转 (Logic Error)

* 位置: if (balance < cost) {Success...}

* 问题: 额度 < 花费 竟然表示了? 逻辑写反了。

* 修正: 应该是余额 > 花费才允许消费。

```
* if (balance >= cost)
```

* 错误 5: 集合语法错误 (Collection Syntax)

* 位置: classesAttended[0] = ... (如果要实现的话)

* 问题: ArrayList 必须用 add(), get(), set(), 不能用 []。

* 修正: classesAttended.add(className);

* 错误 6: 逻辑陷阱 (NullPointer Exception)

* 位置: classesAttended.add(...)

* 问题: 在构造函数中忘记 classesAttended = new

```
ArrayList<String>();
```

* 修正: 去掉多余引号, 但是是 null, 调用 add() 会崩。

* 修正: 在构造函数里初始化它。

考前必背: 找错题源! 抓错! 清单

1. 知识点: Class & Objects (类与对象) Static 隐阱: 看到 static 修饰了 balance, name, id 这种明显属于类的属性, 必错。构造函数陷阱: 看到 public void className(...), 必错 (多了 void)。

Shadowing 陷阱: 构造函数参数名和属性名一样, 但没写

this. 一定要找 equals() 知识点: Arrays & Collections (数组与集合) 长度混淆数组 (length 无括号) String 使用 length () 有括号) List 使用 size() (有括号)。看到 s1.length 和 s1.length() 要区别

2. 知识点: Strings (字符串) 比较陷阱: 看到 s1 == s2, 必错。

一定要找 equals() 知识点: Arrays & Collections (数组与集合) 长度混淆数组 (length 无括号) String 使用 length () 有括号) List 使用 size() (有括号)。看到 s1.length 和 s1.length() 要区别

array.length, 必错泛型陷阱: List<T> list 必错。泛型不能用基本类型 (int, double), 必须用包装类 (Integer, Double)。

4. 知识点: Inheritance (继承) Super 调用: 如果子类构造函数里有 super(...), 它必须是第一行。如果写在第二行, 必错。Interface 实例化: Runnable (不是 new Runnable()); 必错 (接口不能 new, 除非重写匿名内部类方法) 5. 知识点:

Exceptions (异常捕获顺序): catch (Exception e) 写在了 catch (IOException e) 前面。必错! e 类型把类名完全截断了, 后面的 IOException 不知道。所以如果让你找 bug, 有很长一段语法错误 (Syntax) 和 逻辑错误 (Logic) 会斗对干。别忘了看注释 (注释错误) 找。也要看看是不是算我弄反了 (逻辑错误) 找。

面试题源 UML 代码生成

* 题目主题: Part C Q17 - 类的设计与实现 (Class Design & Implementation)

* 中文描述: 根据需求和 UML 图, 设计一个 GiftCard (礼品卡) 类, 实现扣款和查询余额功能。

[1] UML 图案解析 (UML Diagram to Code Mapping)

* 考题给出的图示转化为 Java 逻辑如下:

```
* |+ GiftCard |--> (类名 (Class Name))
```

```
* |+ amount: double |--> 属性 (Attributes/Fields)
```

```
* |+ GiftCard(amount: double) |--> 构造函数 (Constructor)
```

```
* |+ buyBook(totalPrice: double); void |--> 方法 (Methods)
```

```
* |+ amountRemaining(): double |--> 方法 (Methods)
```

* [UM] 知识点: 调用 CheatSheet: 1. 符号含义: -(Minus):

私有 (私) 只能在类内部访问, 封装性的体现 (+Plus):

公共 (公) 可以被外部访问 # (Hash): protected (受保护, 子类可用) 2. 格式语义: 变量命名: type -> Java

变量 type variableName; methodName(param: type);

returnType type returnType methodName(type)

* [2] 提供的测试代码 (Original Test Code)

* 你的目标是写出 GiftCard 类, 让下面的 main 函数能跑通, 并输出预期结果。

public class Q17_Test {

```
* public static void main(String[] args) {
```

```
*     // 初始化一张 100 元的卡
```

```
*     GiftCard g = new GiftCard(100);
```

* // 第一次购买 50.5 (剩余 49.5) -> 应输出: Books bought

* successfully g.buyBook(50.5);

* // 第二次购买 40.5 (剩余 9.0) -> 应输出: Books bought

* successfully g.buyBook(40.5);

* // 第三次购买 60.0 (余额 9.0 < 60) -> 应输出: Not

* enough money on the card g.buyBook(60);

* // 打印余额 -> 应输出: Remaining balance: 9.0

```
*     System.out.println("Remaining balance: " +
```

```
*         g.amountRemaining());
```

* } }

* [3] 标准答案代码 (Solution Code)

* 重点: 文件读取的模版 + 排序的 Lambda 写法。

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

```
import java.io.*;
```

public class Q17_Test {

```
* public static void main(String[] args) {
```

```
*     // 创建列表存储对象
```

```
*     List<Covid19> list = new ArrayList<>();
```

* // 1. 创建 GiftCard 对象

```
*     class GiftCard {
```

* [考点] 属性定义。根据 UML, amount 前面是 ‘-’, 所以必须

* private double amount;

* [考点] 构造函数。名字与类名相同, 没有返回值类型。

* 用于初始化对象的状态。

```
*             public GiftCard(double amount) {
```

* 使用 this 关键字区分变量和参数变量 (这是一个好习惯, 虽然不是必须)

```
*                 this.amount = amount;
```

* }

* [考点] 核心业务逻辑方法。

```
*                 // void 表示不返回任何值, 只做动作 (扣款 + 打印)。
```

```
*                 public void buyBook(double totalPrice) {
```

* 逻辑判断: 余额是否充足

```
*                     if (this.amount >= totalPrice) {
```

* 扣款逻辑

```
*                         this.amount = this.amount - totalPrice; // 或者
```

```
*                         this.amount -= totalPrice;
```

* } }

* [考点] Getter 方法。

* // 这是一个标准的 Getter, 用于返回私有变量的值。

```
*                     public double amountRemaining() {
```

* return this.amount;

* }

* [4] 题型考点总结与预测 (Exam Analysis & Predictions)

* 核心考点 (Core Concepts):

* 1. 封装 (Encapsulation): 哪怕题目没给 UML, 如果你看到 “private attribute” 字眼, 一定要用 private 关键字, 并配合 public 的 Constructor/Method.

* 2. 状态维护 (State Maintenance): 比如这里的 amount, 它会随着 buyBook 的调用而改变。这是 Object-Oriented 的核心—对象是有“记忆”的。

* 3. 构造函数 (Constructor): 记得构造函数没有 return type (连 void 都没有)。

* 变体: 增加 static 变量 (ID Generator)

* 逻辑: 如果是静态变量, 那么是类的属性, 不是对象的属性。

* 修正: 从 “Each GiftCard should have a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正: 从 “Each GiftCard has a unique ID, starting from 1.”

* 修正:

// 知识点: Main Method

```
public static void main(String[] args) {  
    System.out.println("Hello World");  
}  
  
// 知识点: Array Declaration & Length [Lecture]
```

```
int[] arr2 = new int[5]; // 动态初始化(默认0)  
int len = arr.length; // 没有括号!
```

// 知识点: Arrays | Utility Methods [Lecture 2]

```
import java.util.Arrays;  
Arrays.sort(arr); // 排序(修改原数组)
```

```
String s = Arrays.toString(arr); // 转字符串 [1, 2, 3]
```

```
String s1 = "Hello";
String s2 = "hello";
```

```
if (s1.equals(s2)) { ... } // 比较内容
```

// 知识点: String Operations [Lecture 2]

```
int len = str.length(); // 有括号!
String sub = str.substring(0, 2); // 截取 [0, 2)
```

```
String trim = str.trim();           // 去空格  
char c = str.charAt(1);           // 获取字符
```

// 知识点: StringBuilder [Lecture 2]
StringBuilder sb = new StringBuilder();

```
sb.append(" ");  
sb.append(" World");  
sb.append(" "); //反转
```

```
String res = sb.toString(); // 必须转回 String
```

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

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt(); // 读整数
```

```
String l = sc.nextLine(); // 读整行
```

```
import java.util.ArrayList;
```

```
ArrayList<String> list = new ArrayList<>();  
list.add("A"); // 添加
```

```
String s = list.get(0); // 获取(不能用[])
int size = list.size(); // 大小(有括号)
```

// 键值与 HashMap 对应

```
import java.util.HashMap;
```

```
map.put("Key", 100); // 存入  
Integer v = map.get("Key"); // 取出(没找到返回null)
```

```
boolean has = map.containsKey( key ); // 检查key
```

知识点：多态（Polymorphism）- 编译看左边，运行看右边
重点预测：方法会被重写（Override），但属性（Field）永远不变！

```
class Parent {  
    int x = 10;  
    void run() { System.out.println("Parent Run"); }  
  
class Child extends Parent {  
    int x = 20;  
    void run() { System.out.println("Child Run"); }  
  
public class PolyTest {  
    public static void main(String[] args) {  
        Parent p = new Child();  
  
        // [真题考法 1]: 方法调用  
        // 问：p.run() 输出什么？  
        // 答案：Child Run  
        // 解析：方法是多态的，运行的是 new Child() 里的方法。  
        p.run();  
  
        // [真题考法 2]: 属性访问 (最容易错！)  
        // 问：p.x 输出什么？  
        // 答案：10 (Parent 的 x)  
        // 解析：属性（Field）没有多态！引用类型是 Parent，就 Parent 的值。  
        System.out.println(p.x);  
  
        // [真题考法 3]: 类型转换异常 (ClassCastException)  
        // 问：Child c = (Child) new Parent(); 运行结果？  
        // 答案：Runtime Error (ClassCastException)  
        // 解析：父类对象不能强制转成子类引用，只有本来就是子类的对象才能转回去。  
    }  
  
知识点：Static 关键字 - 所有实例共享  
重点预测：修改一个对象的 static 变量，所有对象的该变量都会变。
```

```
class Counter {  
    static int count = 0; // 静态：共享  
    int id = 0; // 实例：独立  
  
    public Counter() {  
        count++;  
        id = count;  
    }  
  
public class StaticTest {  
    public static void main(String[] args) {  
        Counter c1 = new Counter();  
        Counter c2 = new Counter();  
  
        // [真题考法 1]: 追踪变量值  
        // 问：c1.id 和 c1.count 分别是多少？  
        // 答案：c1.id = 1, c1.count = 2  
        // 解析：  
        // 1. new c1 > counter 1 -> c1.id=1  
        // 2. new c2 > counter 2 -> c2.id=2  
        // 3. 此时问 c1.count，因为 count 是共享的，所以 c1 看到了最新的 2。  
        System.out.println(c1.id + " " + c1.count);  
    }  
  
知识点：Try-Catch-Finally 执行顺序  
重点预测：Finally 的绝对执行权，以及 return 的覆盖。
```

```
public class ExceptionTest {  
    public static int test() {  
        try {  
            // [真题考法 1]: 异常未被捕获  
            int x = 1 / 0; // 抛出 ArithmeticException  
            return 1;  
        } catch (NullPointerException e) {  
            // 问：会进这里吗？  
            // 答案：不会。因为类型不匹配。  
            return 2;  
        } catch (Exception e) {  
            // 问：会进这里吗？  
            // 答案：会。Exception 是所有异常的父类。  
            return 3;  
        } finally {  
            // [真题考法 2]: Finally 里的 return  
            // 问：最终返回几？  
            // 答案：4  
            // 解析：Finally 最后执行。如果 Finally 里有 return，// 它会覆盖掉 catch 块里的 return 3。  
            return 4;  
        }  
    }  
  
知识点：String 方法返回值  
重点预测：调用方法时不赋值，原字符串不变。
```

```
public class StringTest {  
    public static void main(String[] args) {  
        String s = "Java";  
  
        // [真题考法 1]: 拼接但不赋值  
        s.concat("Exam");  
        // 问：s 现在是多少？  
        // 答案："Java"  
        // 解析：String 是不可变的。s.concat() 生成了一个新字符串，但没人接收它。
```

```

// [真题考法 2]: 强制式调用
String s2 = s.toUpperCase().substring(0,1);
// 问：s2 是多少？
// 答案：'J'
// 解析：先大写 "JAVA"，再截取第0个字符。

// [真题考法 3]: equals vs ==
String a = "A";
String b = new String("A");
// a == b > False (地址不同)
// a.equals(b) > True (内容相同)
}

/*
 * 知识点：List(有序可重复) vs Set(无序不可重复)
 * 重点预测：往 Set 里加重复元素，往 List 里删元素。
 */
import java.util.*;

public class CollectionTest {
    public static void main(String[] args) {
        // [真题考法 1]: Set 去重
        Set<String> set = new HashSet<>();
        set.add("A");
        set.add("B");
        set.add("A"); // 重复添加
        // 问：set.size() 是多少？
        // 答案：2

        // [真题考法 2]: List 删除
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.remove(0);
        // 问：list.get(0) 现在是谁？
        // 答案："B"
        // 解析：删掉索引0的 "A" 后，"B" 自动补位变成索引0

        // [真题考法 3]: 泛型擦除 (Generic Erasure)
        List<Object> list = new ArrayList<String>();
        // 答案：编译错误 (Compile Error)。泛型必须左右一致。
    }
}

/*
 * 知识点：接口与抽象类的规则
 * 重点预测：实例化限制与继承规则。
 */
// 假设有 interface I 和 abstract class A

// [真题考法 1]: 实例化
I obj = new I(); -> 错 (Interface 不能 new)
I obj = new AbstractClass(); -> 错 (Abstract Class 不能 new)
I obj = new ConcreteChild(); -> 对 (多态)

// [真题考法 2]: 继承关系
class C implements I1, I2 -> 对 (多实现)
class C extends A1, A2 -> 错 (单继承)

// [真题考法 3]: 构造函数
// 抽象类有构造函数吗？-> 没有 (给子类 super() 用)
// 接口有构造函数吗？-> 没有 (绝对没有)
/*
 * 知识点：方法签名 (Signature)
 * 重点预测：改了什么才算重载？改了什么算错？
 */
class Demo {
    // 原始方法
    public int add(int a, int b) { return a + b; }

    // [真题考法 1]: 重载 (Overload)
    // 改变参数个数或类型 -> 对
    public int add(int a, int b, int c) { return a + b + c; }
    public double add(double a, double b) { return a + b; }

    // [真题考法 2]: 编译错误 (Duplicate)
    // 只改变返回值类型 -> 错！！！
    // public double add(int a, int b) { return a + b; }
    // 解析：编译器区分不了这一行和第一行，因为参数完全一样

    // [真题考法 3]: 访问修饰符 (Overrite)
    // 子类重写父类方法，权限只能“更大”，不能“更小”。
    // 父类是 protected，子类必须是 protected 或 public，能是 private。
}

/*
 * 知识点：递归追溯
 * 重点预测：给一段代码，手动模拟输出。
 */
public class RecursionTest {
    public static void print(int n) {
        if (n <= 0) return;
        System.out.print(n + " "); // [位置 A]
        print(n - 2);
        System.out.print(n + " "); // [位置 B]
    }

    // [真题考法 1]: 问 print(4) 输出什么？
    // 步骤：
    // 1. print(4)：输出 "4"，调 print(2)
    // 2. print(2)：输出 "2"，调 print(0)
    // 3. print(0)：return
    // 4. print(2) 回到 [位置 B]：输出 "2"
    // 5. print(4)：回到 [位置 B]，输出 "4"
    // 最终答案：4 2 2 4
}

/*
 * 知识点：Stream 惰性求值 & Lambda 语法
 * 重点预测：不调用终端操作就不执行；Lambda 变量作用域。
*/
import java.util.stream.Stream;
import java.util.List;

public class StreamTest {
    public static void main(String[] args) {
        List<String> list = List.of("a", "b", "c");

```

```

/* [真题考法 1]: 惰性求值 (Lazy Evaluation)
Stream<String> s = list.stream().filter(x -> {
    System.out.print("Filter ");
    return true;
});
// [问]: 输出什么?
// 答案: 什么都没有 (Nothing)。
// 解析: 没调用终端操作 (如 .collect, .count, .forEach),
// 中间操作 (filter) 根本不会运行。
}

// [真题考法 2]: Lambda 变量限制 (Effectively Final)
int num = 10;
list.forEach(x -> System.out.println(x + num)); -> 正确
// num = 20;
// [问]: 如果在 lambda 后面修改 num, 上面代码还能跑吗?
// 答案: 编译错误 (Compile Error)。
// 解析: Lambda 内部引用的外部变量必须是 final 或 effectively final (不可修改)。
}
*/
/* 知识点: 常见编译错误 (Compile Time Errors)
* 重点预测: 区分“运行报错”和“编译报错”。
*/
public class ErrorCheck {
    public void check() {
        // [错误 1]: 泛型不匹配
        // List<Object> l = new ArrayList<String>(); -> 错! (泛型不接受)
        // [错误 2]: 抽象类实例化
        // AbstractClass a = new AbstractClass(); -> 错!

        // [错误 3]: 访问私有成员
        // ob.privateField; -> 错! (在外部)

        // [错误 4]: 异常未处理
        // throw new IOException(); -> 错! (必须 try-catch 或 throws)
        // [错误 5]: 接口方法权限变小
        // class C implements I { void method() {} } -> 错!
        // 接口默认是 public, 实现类必须显式写 public, 缺省是 package-private (更小)。
    }
}

代码分析追踪题目
* 考点 1: 递归堆栈追踪 (Recursion Stack Trace)
* 难度: ★★★★★ (最容易错)
* 核心逻辑:
  1. 递归调用前的代码 (Pre-order) 会按顺序执行。
  2. 递归调用后的代码 (Post-order) 会被压入栈, 等递归到底层返回时, 倒序执行。
}

public class RecursionAnalysis {
    // 模拟题: 问 execution(3) 的输出结果是什么?
    public static void execution(int n) {
        if (n <= 0) {
            System.out.print("0 "); // [Base Case]
            return;
        }

        System.out.print(n + " "); // [位置 A: 前程]
        // 递归点: 注意参数变化 (n-1)
        execution(n - 1);

        System.out.print(n + " "); // [位置 B: 回程]
    }

    /*
     * [分析过程: 必须要画图]:
     * 1. execution(3) 被调用:
     *   - 打印 "3" (位置A)
     *   - 调用 execution(2) -> 等待它返回...
     *   - execution(2) 被调用:
     *     - 打印 "2" (位置A)
     *     - 调用 execution(1) -> 等待它返回...
     *     - execution(1) 被调用:
     *       - 打印 "1" (位置A)
     *       - 调用 execution(0) -> 等待它返回...
     *     - execution(0) 被调用:
     *       - 满足 if (n<=0), 打印 "0"
     *       - return (结束 execution(0))
     *     - 回到 execution(1): 继续执行位置B, 打印 "1"
     *     - 回到 execution(2): 继续执行位置B, 打印 "2"
     *     - 回到 execution(3): 继续执行位置B, 打印 "3"
     *
     * [最终结果]: 3 2 1 0 1 2 3
     * [易错点]: 很多人会漏掉回程的 1 2 3, 以为打印到 0 就结束了。
    */

    * 考点 2: 多态绑定规则 (Polymorphism Binding) 1. 方法 (Methods) 看左边 (实际对象是谁) 2. 属性 (Fields) 看左边 (类型是谁) 3. 静态 (Static) 看左边 (类是谁)。
    class Alpha {
        int x = 10;
        void show() { System.out.println("Alpha-Show"); }
    }

    class Beta extends Alpha {
        int x = 20; // 属性隐藏 (Hiding), 不是重写

        // 重写父类方法
        @Override
        void show() {
            System.out.println("Beta-Show: " + this.x + " " + super.x);
        }
    }

    public class PolyAnalysis {
        /*
         * [模拟题]: 下面代码输出什么?
        */
        public static void analyze() {
            Alpha obj = new Beta(); // 左边是 Alpha, 右边是 Beta
            // [分析步骤 1]: 属性访问
            // 规则: 属性有多态! 运行时看右边 (Beta)。
            // 执行 show(), 属性访问。
            // Beta.show(): 内部逻辑:
            //   - this: 当前对象(Beta)的 x > 20
            //   - super.x: 父类(Alpha)的 x > 10
            // 结果: Beta-Show: 20 10
            obj.show();
        }
    }

    * 考点 3: 异常与 Finally 的优先级 (Exception Priority)
    * 核心逻辑: 1. finally 最后执行。2. 如果 catch 里有 return, finally 里的代码会在 return 之前插队执行。3. 如果 finally 里也有 return, 它会直接覆盖掉前面的 return。
    */
    public class ExceptionAnalysis {
        /*
         * [模拟题]: 这个方法返回什么?
        */
        public static int testException() {
            try {
                int[] arr = new int[2];
                arr[5] = 10; // 这里抛出
                ArrayIndexOutOfBoundsException e = new ArrayIndexOutOfBoundsException("Index 5越界");
                return 1; // 这行永远不会执行
            } catch (NullPointerException | ArithmeticException e) {
                return 2; // 类型不匹配, 不进这里
            } catch (Exception e) {
                // 捕获成功 (ArrayIndexOutOfBoundsException 是 Exception 子类)
                // 准备返回 3, 但发现有 finally, 先挂起 return 3
                return 3;
            } finally {
                // 必执行区域
                // 这里有一个 return 4, 它太霸道了, 直接覆盖掉 catch 里的 3
                return 4;
            }
        }

        /*
         * [最终结果]: 4
         * [变形]: 如果 finally 里没有 return 4, 而是 System.out.print("F");
         * 那么结果是: 先打印 "F", 然后返回 3。
        */
    }

    * 考点 4: 集合操作与引用传递 (Collection Modifications) 核心逻辑: 1. Map.put(key, value): 如果 key 存在, 覆盖旧值。2. List 是对象, 方向间传递的是引用 (Reference)。import java.util.*;

    public class CollectionAnalysis {
        /*
         * [模拟题]: 最终 map 的 size 是多少? list 的内容是什么?
        */
        public static void analyze() {
            // 部分 1: Map 覆盖
            HashMap<String, Integer> map = new HashMap<>();
            map.put("A", 1);
            map.put("A", 2);
            map.put("A", 3); // Key "A" 已经存在, Value 1 被 3 覆盖
            // 此时 map: {A=3, B=2}, Size = 2
            // [易错点]: 很多人以为 size 是 3。

            // 部分 2: List 引用修改
            ArrayList<String> list = new ArrayList<>();
            list.add("X");
            modifyList(list); // 传入 list 的地址
            // [问]: 现在 list 里有什么?
            // 结果: [X, Y]
            // [解析]: modifyList 方法拿到了 list 的遥控器, 它加了 "Y", // 所以主函数里的 list 也变了。
        }

        public static void modifyList(ArrayList<String> l) {
            l.add("Y");
            l = new ArrayList<>(); // [陷阱]: l 指向了新对象, 但这影响外面的 list
            l.add("Z"); // 这个 "Z" 加到了新对象里, 外面看不见
        }
    }

    * 考点 5: 静态变量共享 (Static Shared State) 核心逻辑:
      1. static 变量属于类, 所有对象共用一份。一个对象改了, 大家都变。
      2. instance 变量属于对象, 互不干扰。
    class Counter {
        static int globalCount = 0; // 静态: 全服共享
        int localCount = 0; // 实例: 个人独享
    }

    public Counter() {
        globalCount++; // 每次 new, 全局 +1
        localCount = globalCount; // 把当前的全局值赋给个人
    }

    public class StaticAnalysis {
        /*
         * [模拟题]: c1.localCount 和 c2.localCount 分别是多少?
        */
        public static void run() {
            Counter c1 = new Counter();
            // 过程: globalCount 变 1。c1.localCount = 1。

            Counter c2 = new Counter();
            // 过程: globalCount 变 2。c2.localCount = 2。

            Counter c3 = new Counter();
            // 过程: globalCount 变 3。c3.localCount = 3。
        }
    }
}

```