

# Java笔记整理

## 1. 001

### 1.1 访问控制符的区别

描述Java中的访问控制符public、protected、private和默认（无修饰符）之间的主要区别，并解释它们在类成员访问权限上的作用。

- **public**：表示该成员可以被任何类访问，不论是在同一个包内还是不同包中的类，**任何地方**都可以访问这个成员。
- **protected**：表示该成员可以被同一个包中的类访问，或者可以被不同包中的子类访问。
- **private**：表示该成员只能在当前类内部访问，不能在类外部直接访问，即使是子类也**不能直接访问**父类的 private 成员。
- **默认修饰符(无修饰符)**：默认的访问级别是包内访问（Package-Private）。只有同一个包内的类能够访问该成员，其他包中的类无法访问。

### 1.2 final关键字的用途

解释Java中final关键字的三种主要用途，并给出每种用途的简单代码示例。

#### 1.2.1 **final** 用于修饰类

当 **final** 修饰一个类时，表示该类不能被继承。也就是说，不能有其他类从这个 **final** 类继承。

- **作用**：防止类被继承，可以确保类的行为不被改变。

### 1.2.1.1 示例代码：

```
● ● ●  
1 final class MyClass {  
2     public void display() {  
3         System.out.println(`This` is a final class.);  
4     }  
5 }  
6  
7 // 下面的代码会编译错误，因为不能继承一个 final 类  
8 // class AnotherClass extends MyClass {  
9 // }
```

Fence 1

在上面的代码中，`MyClass` 类被声明为 `final`，因此不能被继承。尝试从 `MyClass` 继承会导致编译错误。

### 1.2.2 `final` 用于修饰方法

当 `final` 修饰一个方法时，表示该方法不能被子类重写（Override）。即使该方法是从父类继承来的，也无法在子类中修改其实现。

- **作用**：防止方法被重写，保证方法的行为不会被改变。

### 1.2.2.1 示例代码：

```
● ● ●  
1 class Animal {  
2     public final void sound() {  
3         System.out.println("Animal makes a sound.");  
4     }  
5 }  
6  
7 class Dog extends Animal {  
8     // 下面的代码会编译错误，因为无法重写一个 final 方法  
9     // public void sound() {  
10     //     System.out.println("Dog barks.");  
11     // }  
12 }
```

Fence 2

## 1.2.3 final` 用于修饰变量

当 `final` 修饰一个变量时，表示该变量一旦赋值后就不能再修改。`final` 可以修饰局部变量、实例变量（成员变量）和类变量（静态变量）。

- **作用**：确保变量的值不可变。对于引用类型的变量，`final` 确保引用的 **地址不可改变**（即指向的对象不能重新指向其他对象），但对象 **本身的内容可以修改**。

### 1.2.3.1 示例代码：

- 修饰局部变量：



```
1 public class MyClass {  
2     public void display() {  
3         final int number = 10;  
4         // number = 20; // 这行会导致编译错误，因为 number 是 final 变量  
5         System.out.println(number);  
6     }  
7 }
```

Fence 3

- 修饰实例变量：



```
1 class MyClass {  
2     final int number;  
3  
4     MyClass(int num) {  
5         `this`.number = num; // final 变量可以在构造方法中赋值一次  
6     }  
7  
8     public void display() {  
9         // `this`.number = 20; // 这行会导致编译错误，因为 number 是  
10        final 变量  
11        System.out.println(number);  
12    }  
13 }
```

Fence 4

`number` 是实例变量，使用 `final` 修饰后，可以在**构造方法中赋值一次**，但在其他地方不能改变它的值。

- 修饰类变量（静态变量）：



```

1 class MyClass {
2     public static final double PI = 3.14159; // 类变量被声明为 final
3
4     public void display() {
5         System.out.println("Value of PI: " + PI);
6     }
7 }
```

Fence 5

## 1.2.4 总结

用法	描述	示例
<code>final</code> 修饰类	类不能被继承	<code>final class MyClass {}</code>
<code>final</code> 修饰方法	方法不能被子类重写	<code>public final void sound() {}</code>
<code>final</code> 修饰变量	变量值不能修改，适用于局部变量、实例变量、类变量	<code>final int x = 10;</code>

Table 1

## 1.3 static关键字的含义

阐述static关键字在Java中的含义，并讨论为什么一个静态方法不能访问类的实例变量或实例方法。

(1) `static` 关键字用于声明类的成员为静态成员。static作为静态成员变量和成员函数的修饰符，意味着它为该类的所有实例所共享，也就是说当某个类的实例修改了该静态成员变量，其修改值为该类的其它所有实例所见。

- `static` 修饰变量：即静态变量（类变量），静态变量属于类本身，而不是某个对象实例。所有的类实例共享同一个静态变量。



```

1 class Counter {
2     static int count = 0; // 静态变量
3
4     Counter() {
5         count++; // 每创建一个对象，静态变量count自增
6     }
}
```

```
7 }
8
9 public class Test {
10    public static void main(String[] args) {
11        new Counter();
12        new Counter();
13        System.out.println(Counter.count); // 输出 2
14    }
15 }
```

Fence 6

- **static 修饰方法**：静态方法属于类本身，可以在**没有创建**类实例的情况下调用。静态方法**无法直接访问**类的实例变量和实例方法。



```
1 class MyClass {
2     static int staticVar = 10; // 静态变量
3     int instanceVar = 20;      // 实例变量
4
5     static void staticMethod() {
6         System.out.println(staticVar); // 可以访问静态变量
7         // System.out.println(instanceVar); // 编译错误，不能访问实例变
8         // 量
9     }
10
11    void instanceMethod() {
12        System.out.println(staticVar); // 可以访问静态变量
13        System.out.println(instanceVar); // 可以访问实例变量
14    }
15
16 public class Test {
17     public static void main(String[] args) {
18         MyClass.staticMethod(); // 通过类名调用静态方法
19
20         MyClass obj = new MyClass();
21         obj.instanceMethod(); // 通过对对象调用实例方法
22     }
23 }
```

Fence 7

- 静态代码块：静态代码块在**类加载时执行一次**，用于初始化类的静态成员。当不同类之间具有继承关系时，将先调用父类的代码块，再调用子类的代码块。



```
1 public class StaticBlockExample {
2
3     // 静态变量
```

```
4 static int staticVariable;  
5  
6 // 静态代码块  
7 static {  
8     // 初始化静态变量  
9     staticVariable = 100;  
10    System.out.println("静态代码块被执行。staticVariable = " +  
11        staticVariable);  
12}  
13  
14 public static void main(String[] args) {  
15     // 访问静态变量  
16     System.out.println("主函数中 staticVariable = " +  
17         staticVariable);  
18}  
19}
```

Fence 8

输出



```
1 | 静态代码块被执行。staticVariable = 100  
2 | 主函数中 staticVariable = 100
```

Fence 9

(2) 当调用一个静态方法时，我们并不需要创建对象，它**通过类名直接调用**。由于静态方法不与任何特定对象实例关联，它没有隐含的 `this` 引用。不含 `this` 引用，静态方法自然无法访问任何实例变量或实例方法。

## 1.4 `this` 和 `super` 关键字的区别

描述Java中 `this` 和 `super` 关键字的区别，并解释它们各自在什么场景下使用。

- `this` : 用来指代这个类的这个实例对象的引用。可在如下场景使用。
  - **区分实例变量和局部变量** : 当方法或构造函数的参数名与实例变量（类成员变量）相同，`this` 可以用来区分它们。
  - **调用当前对象的其他构造函数** : `this()` 可以用来调用当前类的另一个构造函数。
  - **引用当前对象** : `this` 也可以用来传递 **当前对象** 的引用给其他方法或构造函数。



```
1 | class Person {
```

```

2   String name;
3   int age;
4
5   // 一个构造函数
6   Person(String name) {
7       this(name, 30); // 调用另一个构造函数
8   }
9
10  // 另一个构造函数
11  Person(String name, int age) {
12      this.name = name;
13      this.age = age;
14  }
15
16  void display() {
17      System.out.println("Name: " + name + ", Age: " + age);
18  }
19 }
20
21 public class Main {
22     public static void main(String[] args) {
23         Person person = new Person("Alice");
24         person.display(); // 输出: Name: Alice, Age: 30
25     }
26 }
```

Fence 10

- **super** : 代指父类的对象。可在如下场景使用。
  - **访问父类的成员（方法或变量）** : 当子类继承父类时，可以使用 super 来访问父类的成员，尤其是当子类和父类的成员有**相同名称**时。
  - **调用父类的构造函数** : **super()** 用来调用**父类的构造函数**，通常是子类构造函数的第一行代码。
  - **调用父类的方法** : 如果子类重写了父类的方法，子类可以使用 **super** 调用父类的版本。



```

1 class Animal {
2     String name = "Animal";
3
4     void speak() {
5         System.out.println("Animal is speaking");
6     }
7 }
8
9 class Dog extends Animal {
10    String name = "Dog";
11 }
```

```

12 void display() {
13     System.out.println("Dog's name: " + this.name); // 使用
this 访问当前类的name
14     System.out.println("Animal's name: " + super.name); // 使用
super 访问父类的name
15     super.speak(); // 调用父类的speak方法
16 }
17 }
18
19 public class Main {
20     public static void main(String[] args) {
21         Dog dog = new Dog();
22         dog.display();
23         // 输出:
24         // Dog's name: Dog
25         // Animal's name: Animal
26         // Animal is speaking
27     }
28 }

```

Fence 11

## 1.5 Java中的abstract和interface关键字

解释Java中的**abstract**关键字和**interface**关键字之间的主要区别，并讨论它们在面向对象编程中的作用。

### 1.5.1 **abstract** 类

**abstract** 类是一个不能被实例化的类，它用于定义一些共性功能，并且可以包含抽象方法和已实现的方法。**abstract** 类的主要目的是提供一个类的模板，供子类继承和扩展。

- **定义方式**：用 **abstract** 关键字声明一个类。
- **抽象方法**：**abstract** 类可以包含抽象方法，抽象方法没有方法体，子类必须实现这些抽象方法（除非子类也是抽象类）。
- **实例方法**：**abstract** 类也可以包含具体方法（已实现的方法），子类可以继承这些方法，也可以覆盖它们。
- **构造函数**：**abstract** 类可以有构造函数，子类可以调用父类的构造函数。

### 1.5.1.1 特点：

- 一个类只能继承一个 `abstract` 类 (Java 不支持多重继承)。
- `abstract` 类可以包含字段 (成员变量)。
- `abstract` 类支持构造方法、实例方法和静态方法。
- 继承 `abstract` 类的子类必须实现所有的抽象方法 (除非子类本身也是抽象类)。

### 1.5.1.2 示例：



```
1 abstract class Animal {
2     String name;
3
4     // 抽象方法，子类必须实现
5     abstract void sound();
6
7     // 已实现的方法
8     void eat() {
9         System.out.println(name + " is eating.");
10    }
11 }
12
13 class Dog extends Animal {
14     Dog(String name) {
15         this.name = name;
16     }
17
18     // 实现抽象方法
19     void sound() {
20         System.out.println(name + " barks.");
21     }
22 }
23
24 public class Main {
25     public static void main(String[] args) {
26         Dog dog = new Dog("Buddy");
27         dog.eat(); // 输出: Buddy is eating.
28         dog.sound(); // 输出: Buddy barks.
29     }
30 }
```

## 1.5.2 interface 接口

**interface** 接口是用于定义一组行为规范的结构，它不能包含具体的实现（Java 8 引入了默认方法和静态方法，但接口中的方法一般没有实现）。接口主要用于实现**多重继承**的效果，并用于定义对象之间的合同。

- **定义方式**：用 **interface** 关键字声明一个接口。
- **方法声明**：接口中的方法默认是 **public abstract** 的，即使不显式声明为 **abstract**，方法也是抽象的。Java 8 之后，接口可以包含默认方法（**default**）和静态方法（**static**）。
- **字段**：接口中的字段默认是 **public static final** 的，必须初始化。
- **继承**：一个类可以实现多个接口，因此接口提供了一种实现多继承的机制。

### 1.5.2.1 特点：

- 一个类可以实现多个接口。
- 接口中的方法只能声明，不能提供实现（除非是默认方法或静态方法）。
- 接口不能包含构造函数。
- 接口中的字段默认是 **public static final**，因此是常量。
- 接口提供了一种解耦合的方式，让不同类之间可以通过实现相同的接口来保持一致的行为。

### 1.5.2.2 示例：

```
● ● ●
1 interface Animal {
2     // 抽象方法，所有实现该接口的类必须实现该方法
3     void sound();
4
5     // 默认方法，具有默认实现
6     default void eat() {
7         System.out.println("Eating...");
8     }
9 }
10
11 class Dog implements Animal {
12     public void sound() {
13         System.out.println("Bark");
14     }
15 }
16
17 public class Main {
18     public static void main(String[] args) {
19         Dog dog = new Dog();
```

```

20     dog.sound(); // 输出: Bark
21     dog.eat();   // 输出: Eating...
22 }
23 }

```

Fence 13

### 1.5.3 abstract 类和 interface` 的主要区别

区别点	abstract 类	interface 接口
是否支持多继承	不支持 (只能继承一个 abstract 类)	支持 (一个类可以实现多个接口, 接口之间可继承)
方法实现	可以有抽象方法和已实现的方法	默认情况下, 所有方法都是抽象的 (Java 8 之后支持默认方法)
字段定义	可以包含实例字段 (成员变量), 也可以是常量	只能包含常量 (public static final 字段)
构造方法	可以有构造函数	没有构造函数
继承方式	使用 extends 关键字继承 (单继承)	使用 implements 关键字实现 (可以实现多个接口)
访问修饰符	方法和字段可以有不同的访问修饰符 (如 private , protected )	所有方法默认为 public , 字段也是 public static final
使用场景	适合用来定义类的共性行为并共享代码实现	适合用来定义行为契约, 支持多重实现和解耦

Table 2

### 1.5.4 在面向对象编程中的作用:

- **abstract 类** 提供了继承和代码复用的能力, 适用于一组类之间具有共同功能的情况。**抽象类不可被实例化。**
- **interface 接口** 强调的是行为规范, 适用于多个类需要共享某种行为, 但不关心具体实现的情况。**接口不可直接实例化。**

## 1.6 编写一个Java类

包含一个private整数成员变量和一个public构造器，该构造器初始化变量，并提供一个protected方法来返回这个变量的值。

```
● ● ●  
1 public class MyClass {  
2     // 私有成员变量  
3     private int number;  
4  
5     // 公有构造器，用于初始化成员变量  
6     public MyClass(int number) {  
7         this.number = number;  
8     }  
9  
10    // 受保护的方法，用于返回成员变量的值  
11    protected int getNumber() {  
12        return this.number;  
13    }  
14  
15    // 测试方法，显示成员变量的值  
16    public static void main(String[] args) {  
17        MyClass myObject = new MyClass(42);  
18        // 通过直接调用 getNumber 方法来访问 number 的值  
19        System.out.println("The value of number is: " +  
20        myObject.getNumber());  
21    }  
21 }
```

Fence 14

运行结果：

```
● ● ●  
1 | The value of number is: 42
```

Fence 15

## 1.7 创建一个Java接口

声明一个abstract方法calculateArea，然后在一个实现该接口的类中用final关键字实现这个方法。

```
● ● ●  
1 // 定义接口  
2 interface Shape {  
3     // 声明一个抽象方法
```

```
4     double calculateArea();
5 }
6
7 // 实现接口的类
8 class Circle implements Shape {
9     private double radius;
10
11    // 构造器, 用于初始化半径
12    public Circle(double radius) {
13        this.radius = radius;
14    }
15
16    // 使用final关键字实现calculateArea方法, 防止进一步重写
17    @Override
18    public final double calculateArea() {
19        return Math.PI * radius * radius; // 圆的面积公式:  $\pi * r^2$ 
20    }
21 }
22
23 // 测试类
24 public class Main {
25     public static void main(String[] args) {
26         // 创建一个Circle对象
27         Shape circle = new Circle(5.0);
28
29         // 计算并输出圆的面积
30         System.out.println("The area of the circle is: " +
31             circle.calculateArea());
32     }
33 }
```

Fence 16

运行结果



```
1 | The area of the circle is: 78.53981633974483
```

Fence 17

## 1.8 设计一个Java程序

包含一个static方法，该方法不接受任何参数，并打印出当前类的名称。



```
1 | public class MyClass {
2 |
3 |     // 静态方法, 用于打印当前类的名称
```

```
4 public static void printClassName() {  
5     // 获取当前类的名称并打印  
6     System.out.println("Current class name: " +  
7         MyClass.class.getName());  
8 }  
9 // 主方法, 用于测试  
10 public static void main(String[] args) {  
11     // 调用静态方法打印类的名称  
12     printClassName();  
13 }  
14 }
```

Fence 18

运行结果：

● ● ●  
1 | Current class name: MyClass

Fence 19

## 1.9 实现一个Java类

使用synchronized关键字同步一个方法，该方法内部生成一个随机数并打印。

● ● ●

```
1 import java.util.Random;  
2  
3 public class RandomNumberGenerator {  
4  
5     // 使用synchronized修饰的方法, 保证线程安全  
6     public synchronized void generateRandomNumber() {  
7         Random random = new Random();  
8         int randomNumber = random.nextInt(100); // 生成一个0到99之间的随机数  
9         System.out.println("Generated random number: " +  
randomNumber);  
10    }  
11  
12    public static void main(String[] args) {  
13        RandomNumberGenerator generator = new RandomNumberGenerator();  
14  
15        // 创建多个线程来模拟并发访问  
16        Thread t1 = new Thread(() ->  
generator.generateRandomNumber());  
17        Thread t2 = new Thread(() ->  
generator.generateRandomNumber());
```

```
18     Thread t3 = new Thread(() ->
19         generator.generateRandomNumber());
20     // 启动线程
21     t1.start();
22     t2.start();
23     t3.start();
24 }
25 }
```

Fence 20

一个可能的输出



```
1 Generated random number: 37
2 Generated random number: 8
3 Generated random number: 92
```

Fence 21

ps: **synchronized** 确保即使多个线程同时请求访问 `generateRandomNumber()` 方法，只有一个线程能够进入方法并执行，其他线程会被阻塞，直到当前线程完成方法执行。这样可以防止多个线程同时修改共享资源，避免数据竞争或其他并发问题。

## 1.10 编写一个Java程序

包含一个类变量和一个实例变量，两者都使用**volatile关键字修饰**，并创建一个线程安全的计数器方法。

- 在 Java 中，**volatile** 关键字用于修饰变量，确保该变量在多个线程中保持一致性。具体来说，**volatile** 关键字告诉 JVM，变量的值可能会被多个线程修改，因此每次读取该变量时，都应**直接**从主内存中读取，而不是从线程的缓存中读取，从而确保每个线程看到的是**最新的**值。



```
1 public class VolatileCounter {
2     // 类变量，使用 volatile 关键字修饰
3     private static volatile int classCount = 0;
4
5     // 实例变量，使用 volatile 关键字修饰
6     private volatile int instanceCount = 0;
7
8     // 线程安全的计数器方法
9     public synchronized void incrementCounters() {
10         // 增加实例变量和类变量的值
11         instanceCount++;
```

```
12         classCount++;
13     }
14
15     // 获取类变量的值
16     public static int getClassCount() {
17         return classCount;
18     }
19
20     // 获取实例变量的值
21     public int getInstanceCount() {
22         return instanceCount;
23     }
24
25     public static void main(String[] args) {
26         // 创建一个 VolatileCounter 对象
27         VolatileCounter counter = new VolatileCounter();
28
29         // 创建多个线程来测试计数器方法
30         Thread t1 = new Thread(() -> {
31             for (int i = 0; i < 1000; i++) {
32                 counter.incrementCounters();
33             }
34         });
35
36         Thread t2 = new Thread(() -> {
37             for (int i = 0; i < 1000; i++) {
38                 counter.incrementCounters();
39             }
40         });
41
42         // 启动线程
43         t1.start();
44         t2.start();
45
46         try {
47             // 等待两个线程完成
48             t1.join();
49             t2.join();
50         } catch (InterruptedException e) {
51             e.printStackTrace();
52         }
53
54         // 打印类变量和实例变量的最终值
55         System.out.println("Final classCount: " +
56                             VolatileCounter.getClassCount());
57         System.out.println("Final instanceCount: " +
58                             counter.getInstanceCount());
59     }
60 }
```

运行结果：



```
1 Final classCount: 2000  
2 Final instanceCount: 2000
```

Fence 23

**PS:**每次运行时，由于 `incrementCounters()` 方法的线程安全性，`classCount` 和 `instanceCount` 最终的值都会是 2000（每个线程执行了 1000 次自增操作）。

## 2. 002

# 2.1 Java异常处理的理解

解释Java中的try、catch和finally块的作用及其在异常处理中的重要性。

## 2.1.1 try 块

- 作用：**在 `try` 块中编写可能会抛出异常的代码。如果 `try` 块中的代码抛出异常，控制权将转移到与之关联的 `catch` 块。如果没有抛出异常，`catch` 块将被跳过。

## 2.1.2 catch 块

- 作用：**用于捕获并处理在 `try` 块中发生的异常。`catch` 块必须紧跟 `try` 块之后，它会捕获 `try` 中抛出的异常，并通过参数获取异常的详细信息。我们可以捕获多种类型的异常，或使用通配符 `Exception` 捕获所有异常。

## 2.1.3 finally 块

`finally` 块是一个可选的块，它无论是否抛出异常都会执行。它通常用于释放资源（如关闭文件、网络连接、数据库连接等）。无论是否发生异常，`finally` 块都会执行，即使在 `try` 或 `catch` 块中有 `return` 语句。

- 作用：**`finally` 块用于释放资源、清理操作等，它确保在异常发生后仍能执行关键的清理代码。
- 执行顺序：**无论是否抛出异常，`finally` 块都会执行；如果 `try` 和 `catch` 中有 `return` 语句，`finally` 也会在方法返回之前执行。

### 2.1.3.1 示例：

```
● ● ●  
1 try {  
2     int result = 10 / 2; // 没有异常  
3     System.out.println("Result: " + result);  
4 } catch (ArithmaticException e) {  
5     System.out.println("Caught an exception: " + e);  
6 } finally {  
7     System.out.println("This will always be executed.");  
8 }
```

Fence 24

### 2.1.3.2 输出：

```
● ● ●  
1 Result: 5  
2 This will always be executed.
```

Fence 25

如果 `try` 块中发生异常，`finally` 仍然会被执行。

```
● ● ●  
1 try {  
2     int result = 10 / 0; // 这行会抛出 ArithmaticException  
3 } catch (ArithmaticException e) {  
4     System.out.println("Caught an exception: " + e);  
5 } finally {  
6     System.out.println("This will always be executed.");  
7 }
```

Fence 26

### 2.1.3.3 输出：

```
● ● ●  
1 Caught an exception: java.lang.ArithmaticException: / by zero  
2 This will always be executed.
```

Fence 27

## 2.1.4 异常处理的流程

1. `try` 块：首先执行 `try` 块中的代码。如果代码正常执行完毕，`catch` 和 `finally` 块都不会执行，直接跳过。

2. **catch 块**：如果在 `try` 块中抛出异常，程序会跳转到与该异常类型匹配的 `catch` 块。异常对象会作为参数传递给 `catch` 块。如果 `try` 块没有抛出异常，`catch` 块将被跳过。
3. **finally 块**：`finally` 块会在 `try` 和 `catch` 块执行完后无条件地执行。如果在 `try` 或 `catch` 中有 `return` 语句，`finally` 块也会在方法返回之前执行。

## 2.1.5 使用场景和重要性

1. **确保资源释放**：`finally` 块常用于确保资源（如文件、数据库连接、网络连接等）在使用后被正确关闭，无论是否发生异常。例如：

```
● ● ●
1 public void readFile() {
2     FileReader fr = null;
3     try {
4         fr = new FileReader("file.txt");
5         // 读取文件操作
6     } catch (IOException e) {
7         System.out.println("Error reading file: " + e);
8     } finally {
9         // 确保文件被关闭
10        if (fr != null) {
11            try {
12                fr.close();
13            } catch (IOException e) {
14                System.out.println("Error closing file: " + e);
15            }
16        }
17    }
18 }
```

Fence 28

2. **提高代码的健壮性**：`try-catch` 语句帮助程序在遇到异常时不中断程序执行，可以根据业务需求处理异常。例如，可以记录日志、通知用户或采取其他恢复措施。
3. **区分异常类型**：通过多个 `catch` 块，可以区分不同类型的异常并做出不同的处理。例如，网络异常、IO异常和数据处理异常等可以有不同的处理方式。
4. **执行清理代码**：`finally` 块中的代码非常重要，因为它可以确保即使在 `try` 块或 `catch` 块中发生了异常，某些必要的清理操作仍然会被执行，如关闭文件流、释放数据库连接等。

## 2.2 编写一个Java程序

尝试打开并读取一个不存在的文件，并在 catch 块中处理 FileNotFoundException，在finally块中释放资源。



```
1 import java.io.*;
2
3 public class FileReadExample {
4     public static void main(String[] args) {
5         FileReader fileReader = null;
6         BufferedReader bufferedReader = null;
7
8         try {
9             // 尝试打开一个不存在的文件
10            fileReader = new FileReader("non_existent_file.txt");
11            bufferedReader = new BufferedReader(fileReader);
12
13            // 尝试读取文件内容
14            String line;
15            while ((line = bufferedReader.readLine()) != null) {
16                System.out.println(line);
17            }
18
19        } catch (FileNotFoundException e) {
20            // 处理文件未找到异常
21            System.out.println("Error: File not found. " +
e.getMessage());
22        } catch (IOException e) {
23            // 处理其他I/O异常
24            System.out.println("Error reading file. " +
e.getMessage());
25        } finally {
26            // 确保资源在最后被释放
27            try {
28                if (bufferedReader != null) {
29                    bufferedReader.close(); // 关闭 BufferedReader
30                }
31                if (fileReader != null) {
32                    fileReader.close(); // 关闭 FileReader
33                }
34            } catch (IOException e) {
35                System.out.println("Error closing file resources: " +
e.getMessage());
36            }
37        }
38    }
39 }
```

假设尝试读取的文件 `non_existent_file.txt` 不存在，输出结果为：



```
1 Error: File not found. non_existent_file.txt (No such file or
directory)
2 Error closing file resources: null
```

Fence 30

## 2.3 Java Checked和Unchecked异常的区别

描述Java中 `Checked` 异常和 `Unchecked` 异常（运行时异常）之间的区别，并给出每种异常的一个例子。

- **Checked 异常（已检查异常）**：`Checked` 异常是指那些在编译时被检查的异常。Java 编译器会要求程序员 **明确地处理** 这些异常（通常是通过 `try-catch` 块，或者通过 `throws` 声明）。如果程序中抛出了一个 `Checked` 异常，而没有适当的处理或声明，编译器将报错，例如 `IOException`。
- **Unchecked 异常（运行时异常）**：`Unchecked` 异常是指那些在编译时不需要强制检查的异常。它们是 `RuntimeException` 类及其子类的实例。程序员非必须在代码中显式地捕获这些异常，也非必须通过 `throws` 声明它们。这些异常通常是由程序的逻辑错误或不可预见的情况引起的，例如 `NullPointerException`、`ArrayIndexOutOfBoundsException`。



```
1 import java.io.*;
2
3 public class CheckedExceptionExample {
4     public static void main(String[] args) {
5         try {
6             // 尝试打开并读取一个文件（这可能会抛出 IOException）
7             FileReader file = new FileReader("nonexistentfile.txt");
8             BufferedReader reader = new BufferedReader(file);
9             System.out.println(reader.readLine());
10            reader.close();
11        } catch (IOException e) {
12            // 捕获并处理异常
13            System.out.println("Checked exception caught: " +
e.getMessage());
14        }
15        int t=10/0;//不用显式处理，属于Unchecked异常
16    }
17 }
```

Fence 31

特性	Checked 异常	Unchecked 异常
继承关系	继承自 <code>Exception</code> (但不继承 <code>RuntimeException</code> )	继承自 <code>RuntimeException</code>
编译时检查	必须在代码中处理 (通过 <code>try-catch</code> 或 <code>throws</code> 声明)	不强制要求处理, 编译器不检查
常见示例	<code>IOException</code> 、 <code>SQLException</code> 、 <code>FileNotFoundException</code>	<code>NullPointerException</code> 、 <code>ArrayIndexOutOfBoundsException</code> 、 <code>ArithmetricException</code>
处理要求	编译时强制要求处理 (捕获或声明)	程序员可选择是否处理
引发原因	通常是由外部因素 (如文件、网络、数据库等) 引起的	通常是程序逻辑错误或不当操作引起的

Table 3

## 2.4 创建一个方法

该方法接受一个整数数组并检查其中是否包含负数。如果包含，则抛出一个 `IllegalArgumentException` (Checked异常)，并在方法调用处进行处理。



```

1 public class NegativeNumberChecker {
2
3     // 方法接受一个整数数组并检查其中是否包含负数
4     public static void checkForNegativeNumbers(int[] numbers) throws
5         IllegalArgumentExeception {
6             // 遍历数组, 检查是否有负数

```

```
6     for (int number : numbers) {
7         if (number < 0) {
8             // 如果包含负数，抛出 IllegalArgumentException
9             throw new IllegalArgumentException("数组中包含负数: " +
10                number);
11        }
12    }
13
14    public static void main(String[] args) {
15        int[] numbers = {1, 2, -3, 4}; // 示例数组，包含负数
16
17        try {
18            // 调用方法并捕获可能抛出的异常
19            checkForNegativeNumbers(numbers);
20            System.out.println("数组中没有负数!");
21        } catch (IllegalArgumentException e) {
22            // 捕获 IllegalArgumentException，并输出异常信息
23            System.out.println("异常: " + e.getMessage());
24        }
25    }
26 }
```

Fence 32

输出：



1 | 异常：数组中包含负数： -3

Fence 33

## 2.5 Java异常链的理解

解释什么是Java异常链，并讨论在自定义异常时保留原始异常的原因。

Java 异常链（Exception Chaining）是一种在捕获异常时，将原始异常（被捕获的异常）作为新的异常的“根本原因”进行抛出的机制。在 Java 中，当发生异常时，可以将原始异常附加到新的异常对象上，从而建立一个异常链。Java 的异常链机制通过在异常的构造函数中传递原始异常来实现。这使得开发人员在抛出新的异常时，可以保留原始异常的信息，帮助调试和追踪问题的根源。



```
1 public class DatabaseOperationException extends Exception {
2
3     // 构造方法，传递消息和原始异常
4     public DatabaseOperationException(String message, Throwable cause)
{
5         super(message, cause);
}
```

```
6     }
7 }
8 public class DatabaseService {
9
10    public void executeQuery(String query) throws
11        DatabaseOperationException {
12        try {
13            // 可能会抛出 SQLException
14            throw new SQLException("Database connection error");
15        } catch (SQLException e) {
16            // 捕获 SQLException 并将其封装到自定义异常中
17            throw new DatabaseOperationException("Failed to execute
18                query", e);
19        }
20    }
21
22    public static void main(String[] args) {
23        DatabaseService service = new DatabaseService();
24        try {
25            service.executeQuery("SELECT * FROM users");
26        } catch (DatabaseOperationException e) {
27            // 输出异常堆栈跟踪
28            e.printStackTrace();
29        }
29 }
```

Fence 34

输出：



```
1 DatabaseOperationException: Failed to execute query
2     at DatabaseService.executeQuery(DatabaseService.java:12)
3     at DatabaseService.main(DatabaseService.java:18)
4 Caused by: java.sql.SQLException: Database connection error
5     at DatabaseService.executeQuery(DatabaseService.java:8)
6     ... 1 more
```

Fence 35

`DatabaseOperationException` 是自定义异常，它将原始的 `SQLException` 作为原因传递给了构造函数。当 `SQLException` 被捕获时，它被封装成 `DatabaseOperationException` 并抛出，调用者可以通过 `e.getCause()` 方法访问原始的 `SQLException`，从而知道更详细的错误信息。

## 2.5.1 为什么需要保留原始异常

### 2.5.1.1 帮助调试

- **追踪问题的根源**：当捕获一个异常并抛出新的自定义异常时，保留原始异常的信息可以帮助开发人员在调试时快速找到问题的根源。例如，一个高层次的异常可能只是另一个低层次异常的“包装”，如果没有原始异常的信息，开发人员可能无法准确理解问题发生的原因。
- **异常堆栈跟踪**：Java 的异常堆栈跟踪信息不仅会显示当前异常的堆栈信息，还会显示原始异常的堆栈跟踪信息。这样，开发人员可以清楚地看到异常链，从最底层的异常到最终的包装异常，帮助定位问题的具体来源。

### 2.5.1.2 增强代码可维护性

- 如果在抛出自定义异常时保留了原始异常，那么在以后的维护过程中，其他开发人员就能更容易地理解和处理代码。尤其是在大规模的企业级应用中，异常链帮助团队成员理解代码的执行流程，并且能够快速定位问题。

### 2.5.1.3 便于日志记录

- 当需要记录日志时，异常链可以提供更丰富的上下文信息。日志文件会显示异常链的完整信息（包括原始异常），这对于分析生产环境中的问题尤为重要。可以在日志中记录整个异常链，从而获得更多关于异常的上下文信息。

### 2.5.1.4 更好地遵循设计原则

- 在面向对象的编程中，异常处理是一个重要的设计原则。保留异常链符合 **“开放封闭原则”** 和 **“最小惊讶原则”**。即使是外部调用者无法直接处理底层的异常，它们仍然能通过异常链获取更多的背景信息，而不是直接丢失这些信息。

### 2.5.1.5 减少异常丢失

- 如果不保留原始异常，可能会丢失很多重要的上下文信息。例如，某个方法抛出了一个 `SQLException`，如果简单地抛出一个 `MyCustomException` 而不包含原始的 `SQLException`，就无法在上层代码中看到与数据库操作相关的具体错误信息。

## 2.6 编写一个自定义异常类

### MyCustomException

并在其构造器中接受另一个异常作为原因，并在抛出MyCustomException时，通过异常链传递原始异常。



```
1 // 自定义异常类 MyCustomException
2 public class MyCustomException extends Exception {
3
4     // 构造器, 接受错误消息和另一个异常作为原因
5     public MyCustomException(String message, Throwable cause) {
6         super(message, cause); // 调用父类构造器, 将消息和原因传递给父类
7     }
8
9     // 可选: 无参构造器
10    public MyCustomException() {
11        super();
12    }
13
14    // 可选: 只接受错误消息的构造器
15    public MyCustomException(String message) {
16        super(message);
17    }
18 }
19
20 import java.sql.SQLException;
21 // 模拟的数据库操作类
22 public class DatabaseService {
23
24     // 方法执行数据库查询
25     public void executeQuery(String query) throws MyCustomException {
26         try {
27             // 模拟抛出 SQLException
28             throw new SQLException("Database connection error");
29         } catch (SQLException e) {
30             // 捕获 SQLException, 并将其传递给 MyCustomException
31             throw new MyCustomException("Failed to execute query", e);
32         }
33     }
34
35     public static void main(String[] args) {
36         DatabaseService service = new DatabaseService();
37         try {
38             service.executeQuery("SELECT * FROM users");
39         } catch (MyCustomException e) {
40             // 输出异常链中的所有信息
41             e.printStackTrace();
42         }
43     }
44 }
```

```
42     }
43 }
44 }
```

Fence 36

## 2.6.1 代码说明：

### 1. `MyCustomException` 类：

- 继承自 `Exception` 类，构造器接受两个参数：错误消息 `message` 和 `Throwable` 类型的 `cause`，其中 `cause` 表示原始异常。在构造器中调用 `super(message, cause)`，将这两个信息传递给父类 `Exception`，建立异常链。

### 2. `DatabaseService` 类：

- `executeQuery` 方法模拟一个数据库查询，并在发生 `SQLException` 时将其捕获。
- 捕获到的 `SQLException` 被封装到 `MyCustomException` 中，并通过 `throw new MyCustomException("Failed to execute query", e)` 重新抛出，`e` 是原始异常 `SQLException`。

### 3. `main` 方法：

- 在 `main` 方法中调用 `executeQuery`，并使用 `try-catch` 块捕获 `MyCustomException`，调用 `e.printStackTrace()` 打印完整的异常信息及其异常链。

异常链的输出：



```
1 MyCustomException: Failed to execute query
2      at DatabaseService.executeQuery(DatabaseService.java:14)
3      at DatabaseService.main(DatabaseService.java:22)
4 Caused by: java.sql.SQLException: Database connection error
5      at DatabaseService.executeQuery(DatabaseService.java:10)
6      ... 1 more
```

Fence 37

## 2.7 Java中的`throw`和`throws`关键字

**解释Java中`throw`和`throws`关键字的区别和用途。**

在 Java 中，`throw` 和 `throws` 都与异常处理有关，但它们的作用和用途有所不同。下面将详细解释这两个关键字的区别和用途。

## 2.7.1 `throw` 关键字

`throw` 是一个 **语句**，用于显式地抛出一个异常。它可以在方法体内使用，抛出一个已创建的异常实例。通过 `throw`，你可以主动触发异常的发生，使得程序跳转到相应的异常处理块（即 `catch` 块）。

### 2.7.1.1 语法：



```
1 | throw new ExceptionType("Error message");
```

Fence 38

- `ExceptionType` 可以是任何类型的异常（包括自定义异常、标准异常等）。
- 通过 `throw` 语句抛出的异常会立即中断当前方法的执行，程序会跳转到最近的异常处理器（`catch` 块）。

## 2.7.2 `throws` 关键字

`throws` 是一个 **关键字**，用于声明一个方法可能会抛出的异常。它出现在方法的声明中，后跟一个或多个异常类型，用逗号分隔。通过 `throws` 关键字，方法的调用者知道该方法在执行过程中可能会抛出哪些异常，并且需要相应地处理这些异常。

### 2.7.2.1 语法：



```
1 | public void methodName() throws ExceptionType1, ExceptionType2 {  
2 |     // 方法体  
3 | }
```

Fence 39

- `throws` 关键字并不会抛出异常，而是仅仅告诉调用者该方法可能抛出的异常，调用者需要进行相应的异常处理。
- 如果一个方法声明了 `throws`，调用该方法时要么使用 `try-catch` 语句来捕获异常，要么继续声明抛出这些异常（如果它们是受检异常）。

## 2.7.3 示例：



```
1 | public class OrderProcessor {  
2 |  
3 |     // 声明该方法可能会抛出 SQLException 和 IllegalArgumentException  
4 |     public void processOrder(String orderId) throws SQLException,  
|         IllegalArgumentException {
```

```

5     if (orderId == null) {
6         throw new IllegalArgumentException("订单 ID 不能为空");
7     }
8
9     // 模拟数据库操作，可能会抛出 SQLException
10    if (orderId.equals("123")) {
11        throw new SQLException("数据库操作失败");
12    }
13
14    System.out.println("处理订单: " + orderId);
15}
16
17 public static void main(String[] args) {
18     OrderProcessor processor = new OrderProcessor();
19
20     try {
21         processor.processOrder(null); // 传入一个无效的订单 ID
22     } catch (SQLException | IllegalArgumentException e) {
23         System.out.println("捕获异常: " + e.getMessage());
24     }
25
26     try {
27         processor.processOrder("123"); // 传入一个会引发
28             SQLException 的订单 ID
29     } catch (SQLException | IllegalArgumentException e) {
30         System.out.println("捕获异常: " + e.getMessage());
31     }
32 }

```

Fence 40

**输出：**



- 1 捕获异常：订单 ID 不能为空
- 2 捕获异常：数据库操作失败

Fence 41

## 2.7.4 **throw** 和 **throws** 的区别总结

特性	<b>throw</b> 关键字	<b>throws</b> 关键字
<b>功能</b>	用于显式抛出一个异常实例	用于声明一个方法可能抛出的异常
<b>位置</b>	出现在方法体内，用于实际抛出异常	出现在方法声明中，用于声明异常

特性	<code>throw</code> 关键字	<code>throws</code> 关键字
<b>抛出方式</b>	通过 <code>throw</code> 后跟一个异常实例抛出异常	通过方法声明 <code>throws</code> 后跟一个或多个异常类型声明
<b>执行时机</b>	立即抛出异常，导致方法执行中断，转到 <code>catch</code> 块	不会抛出异常，仅用于声明方法可能抛出的异常
<b>异常类型</b>	可以抛出任何类型的异常（包括受检异常和未受检异常）	只能声明受检异常（ <code>checked exceptions</code> ）
<b>是否必须处理</b>	不要求调用者处理，调用者可以选择捕获异常或者声明抛出	调用者必须处理声明的异常，要么捕获要么继续声明抛出

Table 4

## 2.8 编写一个方法

该方法接受一个字符串参数并检查是否为空。如果为空，则使用`throw`关键字抛出一个 `NullPointerException`。



```

1 public class NullPointerExample {
2
3     // 方法接受一个字符串参数并检查是否为空
4     public static void checkString(String str) throws
5         NullPointerException{
6         // 检查字符串是否为空或为 null
7         if (str == null || str.isEmpty()) {
8             // 使用 throw 关键字抛出 NullPointerException
9             throw new NullPointerException("字符串不能为空或 null");
10        }
11
12        // 如果字符串不为空，则打印其内容
13        System.out.println("字符串内容: " + str);
14    }
15
16    public static void main(String[] args) {
17        try {
18            // 调用方法，传入空字符串（会抛出异常）
19            checkString("");
20            // 可以替换为 null 来测试
21        } catch (NullPointerException e) {
22            // 捕获并处理 NullPointerException
23            System.out.println("捕获到异常: " + e.getMessage());
24        }
25    }
26}
```

```
23
24     try {
25         // 调用方法，传入非空字符串
26         checkString("Hello, World!");
27     } catch (NullPointerException e) {
28         // 这块代码不会被执行
29         System.out.println("捕获到异常：" + e.getMessage());
30     }
31 }
32 }
```

Fence 42

输出：



```
1 | 捕获到异常：字符串不能为空或 null
2 | 字符串内容：Hello, World!
```

Fence 43

## 2.9 Java异常处理的最佳实践

讨论在Java异常处理中应遵循的最佳实践，包括何时捕获异常以及何时传播异常。

- **捕获最具体的异常**：应尽量捕获最具体的异常，而不是捕获更大的异常类（如 `Exception`）。这有助于更精确地处理问题。



```
1 try {
2     // 可能会抛出 SQLException 的代码
3 } catch (SQLException e) {
4     // 处理 SQLException
5 }
```

Fence 44

- **不要忽略异常**：捕获异常后应进行适当的处理，而不是简单地忽略。至少应记录异常信息，以便事后分析。



```
1 try {
2     // 可能会抛出异常的代码
3 } catch (Exception e) {
4     // 记录异常
5     e.printStackTrace();
6 }
```

- **使用合适的日志记录工具**: 使用如 `slf4j`、`Log4j` 等日志记录工具记录异常信息，而不是使用 `System.out.println`。



```

1 import org.slf4j.Logger;
2 import org.slf4j.LoggerFactory;
3
4 public class Example {
5     private static final Logger logger =
6         LoggerFactory.getLogger(Example.class);
7
8     public void exampleMethod() {
9         try {
10             // 可能会抛出异常的代码
11         } catch (Exception e) {
12             // 使用日志记录异常
13             logger.error("An error occurred: ", e);
14         }
15     }

```

Fence 46

- **避免过度使用异常**: 异常用于表示程序中的异常情况，而不是控制流程。过度使用异常会增加代码的复杂性和维护成本。
- **避免在 finally 块中抛出异常**: `finally` 块中的异常可能会掩盖 `try` 块中的异常，应尽量避免在 `finally` 块中抛出异常。

## 2.9.1 何时捕获异常：

- 捕获那些**能够有效处理**的异常。
- 捕获并记录无法恢复的异常，以便调查和记录。
- 使用**精确**的异常类型进行捕获，并避免捕获所有异常。

## 2.9.2 何时传播异常：

- 如果方法不能处理异常，应该将异常传播给调用者。
- 使用 `throws` 声明方法可能抛出的异常，并让调用者决定如何处理。
- 在某些情况下，可以通过自定义异常类型传递更多的上下文信息。

## 2.10 设计一个简单的银行账户类

包含存款和取款方法。在取款方法中，如果账户余额不足，则抛出一个 `InsufficientFundsException`，并在主方法中处理这个异常。

自定义异常 `InsufficientFundsException` 类：

```
● ● ●  
1 // 表示账户余额不足  
2 public class InsufficientFundsException extends Exception {  
3     public InsufficientFundsException(String message) {  
4         super(message); // 调用父类的构造器，传递错误信息  
5     }  
6 }
```

Fence 47

`BankAccount` 类：

```
● ● ●  
1  
2 public class BankAccount {  
3     private double balance; // 账户余额  
4  
5     // 构造器，初始化账户余额  
6     public BankAccount(double initialBalance) {  
7         this.balance = initialBalance;  
8     }  
9  
10    // 存款方法  
11    public void deposit(double amount) {  
12        if (amount > 0) {  
13            balance += amount;  
14            System.out.println("存款成功，当前余额: " + balance);  
15        } else {  
16            System.out.println("存款金额必须大于零");  
17        }  
18    }  
19  
20    // 取款方法  
21    public void withdraw(double amount) throws  
22        InsufficientFundsException {  
23        if (amount > balance) {  
24            // 如果余额不足，抛出 InsufficientFundsException 异常  
25            throw new InsufficientFundsException("账户余额不足，无法取  
款");  
26        } else if (amount > 0) {  
27            balance -= amount;  
28            System.out.println("取款成功，当前余额: " + balance);  
29        } else {  
30        }  
31    }
```

```
29         System.out.println("取款金额必须大于零");
30     }
31 }
32 // 获取账户余额
33 public double getBalance() {
34     return balance;
35 }
36 }
37 }
```

Fence 48

## 主函数 Main 类：



```
1 public class Main {
2     public static void main(String[] args) {
3         // 创建一个银行账户，初始余额为 1000
4         BankAccount account = new BankAccount(1000);
5
6         // 执行存款操作
7         account.deposit(500); // 当前余额: 1500
8
9         // 执行取款操作
10        try {
11            account.withdraw(2000); // 尝试取款 2000
12        } catch (InsufficientFundsException e) {
13            // 捕获并处理 InsufficientFundsException 异常
14            System.out.println("错误: " + e.getMessage());
15        }
16
17        // 继续执行其他取款操作
18        try {
19            account.withdraw(500); // 当前余额: 1500, 取款 500
20        } catch (InsufficientFundsException e) {
21            System.out.println("错误: " + e.getMessage());
22        }
23    }
24 }
```

Fence 49

## 输出：



- 1 存款成功，当前余额： 1500
- 2 错误：账户余额不足，无法取款
- 3 取款成功，当前余额： 1000

Fence 50

# 3. 003

## 3.1 Java继承的概念

解释Java中的继承是什么，并讨论它如何允许代码重用。

继承是面向对象编程（OOP）的三大特征之一。

### 3.1.1 继承的概念：

- 继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的属性和方法，或子类从父类继承方法，使得子类具有父类相同的行为。
- 程序中的继承是指类与类之间的继承关系。Java的继承通过extends关键字来实现，实现继承的类被称为子类，被继承的类称为父类（或叫基类、超类）。
- 父类和子类的关系是一种一般与特殊（is-a）的关系。例如苹果继承了水果，苹果是水果的子类，则苹果是一种特殊的水果。

### 3.1.2 代码重用

通过继承，子类可以直接使用父类的属性和方法，而无需重新编写这些代码。这避免了代码的冗余，减少了开发工作量，并且使得代码更加简洁和易于维护。



```
1 // 父类: Animal
2 class Animal {
3     void eat() {
4         System.out.println("This animal is eating.");
5     }
6
7     void sleep() {
8         System.out.println("This animal is sleeping.");
9     }
10 }
11
12 // 子类: Dog
13 class Dog extends Animal {
14     void bark() {
15         System.out.println("The dog is barking.");
16     }
17 }
18
19 // 子类: Cat
20 class Cat extends Animal {
21     void meow() {
```

```
22     System.out.println("The cat is meowing.");
23 }
24 }
25
26 public class Main {
27     public static void main(String[] args) {
28         Dog dog = new Dog();
29         dog.eat(); // 从 Animal 类继承的方法
30         dog.bark(); // Dog 类的特有方法
31
32         Cat cat = new Cat();
33         cat.sleep(); // 从 Animal 类继承的方法
34         cat.meow(); // Cat 类的特有方法
35     }
36 }
```

Fence 51

输出：



```
1 This animal is eating.
2 The dog is barking.
3 This animal is sleeping.
4 The cat is meowing.
```

Fence 52

在上面的示例中，Dog 和 Cat 类都继承了 Animal 类的 eat() 和 sleep() 方法，允许了代码重用。子类也可以添加自己的特有方法（如 bark() 和 meow()），同时保留父类的通用行为。

## 3.2 编写一个程序

创建一个基类 Animal 和两个继承自 Animal 的子类 Dog 和 Cat。每个类都有 makeSound 方法。



```
1 // 基类 Animal
2 abstract class Animal {
3     // 抽象方法 makeSound，子类必须实现
4     public abstract void makeSound();
5 }
6
7 // 子类 Dog
8 class Dog extends Animal {
9     @Override
```

```

10  public void makeSound() {
11      System.out.println("汪汪!");
12  }
13 }
14
15 // 子类 Cat
16 class Cat extends Animal {
17     @Override
18     public void makeSound() {
19         System.out.println("喵喵!");
20     }
21 }
22
23 // 测试类
24 public class Main {
25     public static void main(String[] args) {
26         // 创建 Dog 和 Cat 对象
27         Animal dog = new Dog();
28         Animal cat = new Cat();
29
30         // 调用它们的 makeSound 方法
31         dog.makeSound(); // 输出: 汪汪!
32         cat.makeSound(); // 输出: 喵喵!
33     }
34 }

```

Fence 53

## 3.3 Java多态性的理解

**描述Java中的多态性是什么，以及它如何影响方法调用。**

多态性 (Polymorphism) 是面向对象编程 (OOP) 中的一个核心概念，它指的是一个对象可以通过不同的方式表现（或者说是“多种形态”）。在Java中，多态性可以分为两种类型：

### 3.3.1 编译时多态

编译时多态是通过方法重载实现的。方法重载是指在同一个类中，存在多个同名的方法，但它们的参数列表不同（参数的个数、类型或顺序不同）。在编译时，Java编译器会根据实际参数的类型、个数和顺序来确定调用哪个重载方法。



```

1 public class OverloadExample {
2     public void display(int a) {
3         System.out.println("Display method with integer: " + a);
4     }
5

```

```
6 public void display(String a) {  
7     System.out.println("Display method with string: " + a);  
8 }  
9  
10 public static void main(String[] args) {  
11     OverloadExample obj = new OverloadExample();  
12     obj.display(10); // 调用第一个display方法  
13     obj.display("Hello"); // 调用第二个display方法  
14 }  
15 }
```

Fence 54

### 3.3.2 运行时多态

运行时多态是通过方法重写实现的。方法重写是指子类继承父类并重写父类中已有的方法。当父类引用指向子类对象时，调用的方法是**子类重写后**的方法，而不是父类的方法。



```
1 public class Animal {  
2     public void run() {  
3         System.out.println("父类Animal的run方法");  
4     }  
5 }  
6  
7 class Cat extends Animal {  
8     @Override  
9     public void run() {  
10        System.out.println("子类Cat的run方法");  
11    }  
12 }  
13  
14 public class Test {  
15     public static void main(String[] args) {  
16         Animal a = new Cat();  
17         a.run(); // 输出: 子类Cat的run方法  
18     }  
19 }
```

Fence 55

## 3.4 拓展题目2

创建一个Animal数组，包含Dog和Cat对象，并调用每个对象的makeSound方法。



```
1 // 定义Animal接口
2 interface Animal {
3     void makeSound(); // 所有动物都需要实现的发出声音的方法
4 }
5
6 // 定义Dog类, 实现Animal接口
7 class Dog implements Animal {
8     @Override
9     public void makeSound() {
10         System.out.println("Woof Woof");
11     }
12 }
13
14 // 定义Cat类, 实现Animal接口
15 class Cat implements Animal {
16     @Override
17     public void makeSound() {
18         System.out.println("Meow Meow");
19     }
20 }
21
22 // 主程序
23 public class Main {
24     public static void main(String[] args) {
25         // 创建一个Animal类型的数组, 包含Dog和Cat对象
26         Animal[] animals = new Animal[2];
27         animals[0] = new Dog();
28         animals[1] = new Cat();
29
30         // 遍历数组并调用每个对象的makeSound方法
31         for (Animal animal : animals) {
32             animal.makeSound();
33         }
34     }
35 }
```

Fence 56

输出



```
1 Woof Woof
2 Meow Meow
```

Fence 57

# 3.5 Java方法重写与重载

区分Java中的方法重写 (Override) 和重载 (Overload)。

在Java中，方法重写 (Override) 和方法重载 (Overload) 是两种不同的概念，它们在编程时用于不同的目的。下面是对这两者的详细解释及区别：

## 3.5.1 方法重写 (Override)

方法重写是指子类重新实现父类的一个方法。重写的目的是改变或增强父类方法的行为。方法重写需要满足以下条件：

- **方法名称、返回类型和参数列表**必须与父类中被重写的方法完全相同。
- 重写的方法可以有不同的访问权限（但不能比父类方法的访问权限更严格），通常是同样的访问权限或更宽松的权限。
- 被重写的方法可以抛出比父类方法更少或相同的异常，但不能抛出比父类方法更多或不同类型的异常。
- 方法重写是**运行时多态**的核心。



```
1 class Animal {  
2     void makeSound() {  
3         System.out.println("Animal makes a sound");  
4     }  
5 }  
6  
7 class Dog extends Animal {  
8     @Override  
9     void makeSound() {  
10        System.out.println("Woof Woof");  
11    }  
12 }  
13  
14 public class Main {  
15     public static void main(String[] args) {  
16         Animal myDog = new Dog();  
17         myDog.makeSound(); // 调用Dog类中的makeSound方法 (重写)  
18     }  
19 }
```

Fence 58

输出：



```
1 | Woof Woof
```

Fence 59

在这个例子中，`Dog` 类重写了 `Animal` 类中的 `makeSound` 方法。当调用 `myDog.makeSound()` 时，实际执行的是 `Dog` 类中的方法，而不是 `Animal` 类中的方法。

### 3.5.2 方法重载 (Overload)

方法重载是指在同一个类中，定义多个方法，它们具有相同的 **方法名称**，但 **参数列表不同**。重载的目的是根据不同的参数类型或参数数量来调用不同的方法。方法重载不会影响返回类型，但参数列表必须不同。

- 方法重载可以发生在同一个类中，也可以在继承关系中发生。
- 在方法重载中，**返回类型**是可以不同的，但仅凭返回类型无法区分方法重载。



```
1 class Calculator {  
2     // 方法重载，接受一个整数参数  
3     int add(int a) {  
4         return a + a;  
5     }  
6  
7     // 方法重载，接受两个整数参数  
8     int add(int a, int b) {  
9         return a + b;  
10    }  
11  
12    // 方法重载，接受两个浮点数参数  
13    double add(double a, double b) {  
14        return a + b;  
15    }  
16}  
17  
18 public class Main {  
19     public static void main(String[] args) {  
20         Calculator calc = new Calculator();  
21  
22         System.out.println(calc.add(5));           // 调用 int add(int a)  
23         System.out.println(calc.add(5, 10));       // 调用 int add(int a,  
int b)  
24         System.out.println(calc.add(5.5, 10.2)); // 调用 double  
add(double a, double b)  
25     }  
26 }
```

Fence 60

输出：



1 10  
2 15  
3 15.7

根据调用时传入的参数，Java会选择匹配的方法来执行。

### 3.5.3 方法重写 (Override) 与方法重载 (Overload) 的区别

特性	方法重写 (Override)	方法重载 (Overload)
定义	子类重新实现父类的已有方法。	在同一个类中，定义多个方法，方法名相同但参数列表不同。
方法名称	子类方法名称与父类方法名称相同。	方法名称相同。
参数列表	参数列表必须相同。	参数列表必须不同（数量或类型不同）。
返回类型	返回类型必须相同，或者是父类返回类型的子类型（协变返回类型）。	返回类型可以不同。
访问权限	子类方法的访问权限不能低于父类方法的访问权限。	访问权限可以相同，也可以不同。
异常	被重写的方法可以抛出比父类方法少的异常或相同类型的异常。	可以抛出不同的异常（与父类方法无关）。
多态性	支持运行时多态性。	不支持运行时多态性。
适用场景	用于改变父类方法的实现。	用于同一方法名适应不同参数的场景。

Table 5

## 3.6 节接上回

在Animal类中定义一个display方法，并在Dog和Cat类中重写这个方法。同时，在Animal类中重载display方法。



```
1 // Animal类，包含重载和重写display方法
2 class Animal {
3     // Animal类中的重载方法：不带参数的display方法
4     void display() {
5         System.out.println("This is an animal.");
6     }
7
8     // Animal类中的重载方法：带一个String参数的display方法
9     void display(String animalType) {
10        System.out.println("This is a " + animalType + ".");
11    }
12 }
13
14 // Dog类，继承Animal类并重写display方法
15 class Dog extends Animal {
16     @Override
17     void display() {
18         System.out.println("This is a dog.");
19     }
20 }
21
22 // Cat类，继承Animal类并重写display方法
23 class Cat extends Animal {
24     @Override
25     void display() {
26         System.out.println("This is a cat.");
27     }
28 }
29
30 public class Main {
31     public static void main(String[] args) {
32         // 创建Animal对象，调用不同的display方法
33         Animal animal = new Animal();
34         animal.display();           // 调用Animal类中的display()
35         animal.display("generic animal"); // 调用Animal类中的重载
36         display(String)
37
38         // 创建Dog对象，调用重写的display方法
39         Animal dog = new Dog();
40         dog.display();           // 调用Dog类中重写的display()
41
42         // 创建Cat对象，调用重写的display方法
43         Animal cat = new Cat();
```

```
43     cat.display(); // 调用Cat类中重写的display()
44 }
45 }
```

Fence 62

输出：

- ● ●

```
1 This is an animal.
2 This is a generic animal.
3 This is a dog.
4 This is a cat.
```

Fence 63

## 3.7 Java抽象类和接口

讨论Java中抽象类和接口的区别及其用途。

同001 第5题

## 3.8 创建一个抽象类Shape。

其中包含一个抽象方法draw。然后创建一个接口Colorable，包含一个方法setColor。实现这两个抽象概念的Circle类。

- ● ●

```
1 // 1. 定义一个抽象类Shape
2 abstract class Shape {
3     // 抽象方法draw, 必须在子类中实现
4     abstract void draw();
5 }
6
7 // 2. 定义一个接口Colorable
8 interface Colorable {
9     // 定义一个方法setColor
10    void setColor(String color);
11 }
12
13 // 3. 创建一个Circle类, 继承Shape并实现Colorable接口
14 class Circle extends Shape implements Colorable {
15     private String color;
16
17     // Circle类实现了抽象方法draw
18     @Override
```

```

19 void draw() {
20     System.out.println("Drawing a circle.");
21 }
22
23 // Circle类实现了Colorable接口的setColor方法
24 @Override
25 public void setColor(String color) {
26     this.color = color;
27     System.out.println("Setting the circle's color to " + color);
28 }
29
30 // 可选: 获取颜色的方法
31 public String getColor() {
32     return color;
33 }
34 }
35
36 // 主程序类
37 public class Main {
38     public static void main(String[] args) {
39         // 创建一个Circle对象
40         Circle circle = new Circle();
41
42         // 调用draw方法
43         circle.draw(); // 输出: Drawing a circle.
44
45         // 调用setColor方法
46         circle.setColor("Red"); // 输出: Setting the circle's color
47         to Red
48     }
49 }

```

Fence 64

**输出:**



```

1 Drawing a circle.
2 Setting the circle's color to Red

```

Fence 65

## 3.9 Java super关键字的使用

**解释Java中super关键字的用途和它在继承中的作用。**

**super** 关键字代指父类的对象。可在如下场景使用。

- **访问父类的成员 (方法或变量)** : 当子类继承父类时, 可以使用 super 来访问父类的成员, 尤其是当子类和父类的成员有**相同名称**时。

- 调用父类的构造函数 : `super()` 用来调用父类的构造函数 , 通常是子类构造函数的第一行代码。

- 调用父类的方法 : 如果子类重写了父类的方法 , 子类可以使用 `super` 调用父类的版本。



```
1 class Animal {
2     String name = "Animal";
3
4     void speak() {
5         System.out.println("Animal is speaking");
6     }
7 }
8
9 class Dog extends Animal {
10    String name = "Dog";
11
12    void display() {
13        System.out.println("Dog's name: " + this.name);           // 使用
this 访问当前类的name
14        System.out.println("Animal's name: " + super.name);      // 使用
super 访问父类的name
15        super.speak(); // 调用父类的speak方法
16    }
17 }
18
19 public class Main {
20     public static void main(String[] args) {
21         Dog dog = new Dog();
22         dog.display();
23         // 输出:
24         // Dog's name: Dog
25         // Animal's name: Animal
26         // Animal is speaking
27     }
28 }
```

Fence 66

PS:

- `super()` 必须是构造方法中的第一条语句 , 不能出现在其他代码之前。
- 如果子类的构造方法没有显式调用父类构造方法 , 则编译器会自动在子类构造方法的 **第一行** 调用父类的无参构造方法。如果父类没有无参构造方法 , 且子类没有显式调用父类的构造方法 , 则会导致编译错误。

## 3.10 创建一个基类Vehicle。

包含一个startEngine方法。创建一个子类Car，它重写startEngine方法，并在其中使用super.startEngine()。



```
1 // 基类Vehicle
2 class Vehicle {
3     // Vehicle类的startEngine方法
4     void startEngine() {
5         System.out.println("Vehicle engine is starting... ");
6     }
7 }
8
9 // 子类Car, 继承自Vehicle
10 class Car extends Vehicle {
11     // 重写startEngine方法
12     @Override
13     void startEngine() {
14         // 调用父类的startEngine方法
15         super.startEngine();
16         // 子类Car特有的行为
17         System.out.println("Car engine is starting with a roar!");
18     }
19 }
20
21 // 主程序类
22 public class Main {
23     public static void main(String[] args) {
24         // 创建Car对象
25         Car myCar = new Car();
26         // 调用重写的startEngine方法
27         myCar.startEngine();
28     }
29 }
```

Fence 67

输出：



```
1 Vehicle engine is starting...
2 Car engine is starting with a roar!
```

Fence 68

# 4. 004

## 4.1 Java this 关键字的作用

解释Java中this关键字的用途和它在方法中如何引用当前对象。

参考[Part 1第4题](#)

## 4.2 创建一个名为Person的类。

包含name和age属性，以及一个构造器和两个方法：getInfo()和updateAge(int age)。在getInfo()方法中使用this关键字来区分局部变量和成员变量。



```
1 // Person类定义
2 class Person {
3     // 成员变量
4     String name;
5     int age;
6
7     // 构造器: 初始化name 和age
8     Person(String name, int age) {
9         this.name = name; // 使用this来区分成员变量和局部变量
10        this.age = age;
11    }
12
13     // getInfo方法: 返回人的基本信息
14     void getInfo() {
15         // 使用this关键字区分成员变量和方法参数
16         System.out.println("Name: " + this.name + ", Age: " +
17             this.age);
18     }
19
20     // updateAge方法: 更新age 属性
21     void updateAge(int age) {
22         // 使用this关键字来区分成员变量age 和局部变量age
23         this.age = age;
24     }
25
26 // 主程序
27 public class Main {
28     public static void main(String[] args) {
29         // 创建一个Person对象
30         Person person = new Person("Alice", 30);
```

```
31  
32     // 获取并显示人的信息  
33     person.getInfo(); // 输出: Name: Alice, Age: 30  
34  
35     // 更新年龄  
36     person.updateAge(35);  
37  
38     // 再次获取并显示人的信息  
39     person.getInfo(); // 输出: Name: Alice, Age: 35  
40 }  
41 }
```

Fence 69

## 4.3 描述Java中参数传递是值传递还是引用传递，并举例说明

在 Java 中，**参数传递是值传递**，而不是引用传递。这意味着，无论传递的是基本数据类型（例如 `int`、`float`、`boolean` 等）还是对象类型（例如 `String`、`Array` 或自定义对象），Java 总是通过 **复制参数的值** 来进行传递。

### 4.3.1 值传递 (Pass-by-Value)

- **对于基本数据类型**，传递的是值的拷贝。即，方法接收到的是变量的副本，修改这个副本不会影响原始变量。
- **对于对象类型**，传递的是对象引用的拷贝。即，方法接收到的是对象的内存地址（引用）的副本，修改引用指向的对象的内容会影响原始对象，但重新赋值引用（让它指向其他对象）不会改变原始对象的引用。

### 4.3.2 详细解释：

#### 4.3.2.1 基本数据类型 (例如: int, float, char)

对于基本数据类型，传递的是变量的值的副本。如果在方法内部修改了副本的值，这不会影响原始的变量。

**示例：**



```
1 public class Main {  
2     public static void main(String[] args) {  
3         int x = 10;  
4         System.out.println("Before: " + x); // 输出: 10  
5         modifyValue(x);  
6         System.out.println("After: " + x); // 输出: 10  
7     }  
8  
9     // 修改基本数据类型的值  
10    static void modifyValue(int a) {  
11        a = 20;  
12    }  
13 }
```

Fence 70

输出:



```
1 Before: 10  
2 After: 10
```

Fence 71

在这个例子中，`modifyValue(x)` 传递的是 `x` 的副本（值）。即使在 `modifyValue()` 方法内将 `a` 设置为 20，原始变量 `x` 不会受到影响。

### 4.3.2.2 对象类型（例如：数组、对象）

对于对象类型，传递的是对象引用的副本（即引用的值）。这意味着方法内部可以修改对象的内容（因为传递的是对象的内存地址），但如果重新给引用赋值，原始对象的引用不会受到影响。

示例:



```
1 class Person {  
2     String name;  
3  
4     Person(String name) {  
5         this.name = name;  
6     }  
7 }  
8  
9 public class Main {  
10    public static void main(String[] args) {  
11        Person p = new Person("Alice");  
12        System.out.println("Before: " + p.name); // 输出: Alice
```

```

13     modifyObject(p);
14     System.out.println("After modification: " + p.name); // 输出:
Bob
15     changeReference(p);
16     System.out.println("After reference change: " + p.name); // 输出:
出: Bob
17 }
18
19 // 修改对象的内容
20 static void modifyObject(Person person) {
21     person.name = "Bob"; // 修改对象的字段
22 }
23
24 // 修改引用
25 static void changeReference(Person person) {
26     person = new Person("Charlie"); // 重新赋值引用, 指向一个新对象
27 }
28 }
```

Fence 72

**输出:**



```

1 Before: Alice
2 After modification: Bob
3 After reference change: Bob
```

Fence 73

**PS:**

- `modifyObject(p)` 修改了对象 `p` 的 `name` 属性，将其从 `"Alice"` 改为 `"Bob"`。
- `changeReference(p)` 试图改变 `p` 引用的对象，但由于 Java 是按值传递引用，因此 `p` 的引用副本被修改，而原始 `p` 仍然指向原来的对象。所以，`p.name` 仍然是 `"Bob"`。

## 4.4 编写一个方法swap

接受两个整数参数并交换它们的值，然后在主方法中测试这个方法。

## 4.4.1 方法1 (swap失败版)

```
● ● ●  
1 public class Main {  
2     // 方法swap, 交换两个整数  
3     static void swap(int a, int b) {  
4         System.out.println("Before swap: a = " + a + ", b = " + b);  
5  
6         // 交换两个整数的值  
7         int temp = a;  
8         a = b;  
9         b = temp;  
10  
11        // 输出交换后的值  
12        System.out.println("After swap: a = " + a + ", b = " + b);  
13    }  
14  
15    public static void main(String[] args) {  
16        int x = 10;  
17        int y = 20;  
18  
19        // 调用swap方法  
20        swap(x, y);  
21  
22        // 注意, Java是按值传递, x和y的值没有改变  
23        System.out.println("In main after swap call: x = " + x + ", y  
= " + y);  
24    }  
25 }
```

Fence 74

输出：

```
● ● ●  
1 | In main after swap call: x = 10 , y = 20
```

Fence 75

## 4.4.2 方法2 (swap成功版)

```
● ● ●  
1 public class TestSwap {  
2  
3     public static void main(String[] args){  
4         int a = 3;  
5         int b = 5;  
6         System.out.println("交换前: "+"a="+a+" b="+b);  
7         // 以数组接收后赋值, 注意赋值顺序, 注意对应关系
```

```
8     int[] arr = swap(a,b);
9     a = arr[0];
10    b = arr[1];
11    System.out.println("交换后: "+a+" "+b);
12 }
13
14 //交换
15 private static int[] swap(int x, int y){
16     //以数组形式返回
17     return new int[]{y,x};
18 }
19 }
```

Fence 76

输出：



```
1 | 交换前: a= 3, b=5
2 | 交换后: a= 5, b=3
```

Fence 77

## 4.5 Java方法重载

解释Java中方法重载是什么，以及它如何根据方法签名区分不同的方法。

参考Part 3第5题

## 4.6 在一个类Calculator中实现三个重载的add方法

分别接受两个整数、两个浮点数和三个整数作为参数。



```
1 class Calculator {
2     // 方法重载，接受三个整数参数
3     int add(int a,int b,int c) {
4         return a + b + c;
5     }
6
7     // 方法重载，接受两个整数参数
8     int add(int a, int b) {
9         return a + b;
10    }
```

```
11 // 方法重载, 接受两个浮点数参数
12 double add(double a, double b) {
13     return a + b;
14 }
15 }
16 }
17
18 public class Main {
19     public static void main(String[] args) {
20         Calculator calc = new Calculator();
21
22         System.out.println(calc.add(5,6,7));           // 调用 int
23         System.out.println(calc.add(5, 10));          // 调用 int add(int a,
24         int b)
25         System.out.println(calc.add(5.5, 10.2)); // 调用 double
26         add(double a, double b)
27     }
28 }
```

Fence 78

输出：

● ● ●  
1 18  
2 10  
3 15.7

Fence 79

## 4.7 Java方法重写

描述Java中方法重写的概念，并说明它与方法重载的区别。

参考Part 3 第5题

## 4.8 创建一个基类Animal和一个子类Bird。

重写Animal类中的makeSound方法。

● ● ●  
1 // 基类 Animal
2 class Animal {
3 // Animal类的makeSound方法
4 void makeSound() {

```

5     System.out.println("Animal makes a sound");
6 }
7 }
8
9 // 子类 Bird, 继承自 Animal
10 class Bird extends Animal {
11     // 重写父类的makeSound方法
12     @Override
13     void makeSound() {
14         System.out.println("Bird chirps");
15     }
16 }
17
18 // 主程序类
19 public class Main {
20     public static void main(String[] args) {
21         // 创建 Animal 对象并调用 makeSound 方法
22         Animal animal = new Animal();
23         animal.makeSound(); // 输出: Animal makes a sound
24
25         // 创建 Bird 对象并调用 makeSound 方法
26         Animal bird = new Bird();
27         bird.makeSound(); // 输出: Bird chirps
28     }
29 }

```

Fence 80

**输出:**



```

1 Animal makes a sound
2 Bird chirps

```

Fence 81

## 4.9 Java this 关键字与构造器

**讨论在Java构造器中使用this关键字调用另一个构造器的目的和语法。**

在 Java 中, **this** 关键字不仅可以用于指代当前对象的成员变量或方法, 还可以在构造器中用来调用当前类的另一个构造器。这种用法称为 **构造器链** (Constructor Chaining)。使用 **this** 调用另一个构造器的目的是为了避免代码重复, 简化构造器的实现。

## 4.9.1 目的和好处

- **代码复用**：多个构造器可能有相似的初始化逻辑，使用 `this` 可以避免重复代码，确保所有的构造器都能**共享同样的初始化逻辑**。
- **提高可维护性**：当你修改构造器的实现时，改动只需要在**一个**地方进行，而不需要在多个构造器中重复。
- **简化复杂构造器**：如果一个类有多个构造器，使用 `this` 可以将相似的初始化步骤合并到一个构造器中，其他构造器通过调用这个构造器来完成初始化。

## 4.9.2 语法和规则

在构造器中使用 `this` 调用另一个构造器的语法如下：

```
● ● ●  
1 | this(参数);
```

Fence 82

- **this(参数)**：用于调用当前类的另一个构造器，并传递参数。它必须是构造器中的第一条语句。即，`this()` 调用必须在构造器体中的最开始位置。

## 4.9.3 Example:

假设我们有一个 `Person` 类，它有两个构造器：一个是无参构造器，另一个是带有 `name` 和 `age` 参数的构造器。

```
● ● ●  
1 class Person {  
2     String name;  
3     int age;  
4  
5     // 无参构造器  
6     public Person() {  
7         this("Unknown", 0); // 调用另一个构造器，并传递默认值  
8         System.out.println("No-argument constructor called");  
9     }  
10  
11    // 带参构造器  
12    public Person(String name, int age) {  
13        this.name = name;  
14        this.age = age;  
15        System.out.println("Constructor with name and age called");  
16    }  
17  
18    // 方法打印对象信息  
19    public void displayInfo() {
```

```

20         System.out.println("Name: " + name + ", Age: " + age);
21     }
22 }
23
24 public class Main {
25     public static void main(String[] args) {
26         // 使用无参构造器
27         Person person1 = new Person();
28         person1.displayInfo(); // 输出: Name: Unknown, Age: 0
29
30         // 使用带参构造器
31         Person person2 = new Person("Alice", 30);
32         person2.displayInfo(); // 输出: Name: Alice, Age: 30
33     }
34 }
```

Fence 83

**输出：**

● ● ●

```

1 Constructor with name and age called
2 No-argument constructor called
3 Name: Unknown, Age: 0
4 Constructor with name and age called
5 Name: Alice, Age: 30
```

Fence 84

- 在 `Person` 类中，我们有两个构造器：
  - **无参构造器**：`this("Unknown", 0);` 调用了带有 `name` 和 `age` 参数的构造器，并为 `name` 和 `age` 提供了默认值 `"Unknown"` 和 `0`。
  - **带参构造器**：直接初始化 `name` 和 `age` 属性。
- 在 `main` 方法中，首先使用无参构造器创建 `person1`，然后使用带参构造器创建 `person2`。

#### 4.9.4 注意事项：

- **`this()` 必须是构造器中的第一条语句**：你不能在调用 `this()` 前写其他语句。如果尝试在 `this()` 调用前写其他代码，编译器将会报错。  
例如，以下代码是错误的：

● ● ●

```

1 public Person() {
2     System.out.println("Before this()");
3     this("Unknown", 0); // 错误, this() 必须是第一条语句
4 }
```

- **this()** 只能调用当前类的其他构造器，不能调用父类的构造器。如果需要调用父类的构造器，可以使用 **super()**。



```
1 // 错误示例: this() 只能调用当前类的构造器
2 public Person(String name) {
3     //super(name);    super() 应该用于调用父类构造器
4     this(name, 0); // 只能调用当前类的构造器
5 }
```

Fence 86

## 4.10 设计一个Car类

包含model、year和color属性，以及使用this关键字链式调用的多个构造器。



```
1 public class Car {
2     // 定义属性
3     private String model;
4     private int year;
5     private String color;
6
7     // 无参构造器
8     public Car() {
9         // 默认值
10        this("Unknown Model", 0, "Unknown Color");
11    }
12
13     // 一个参数构造器: model
14     public Car(String model) {
15         this(model, 0, "Unknown Color");
16     }
17
18     // 两个参数构造器: model, year
19     public Car(String model, int year) {
20         this(model, year, "Unknown Color");
21     }
22
23     // 三个参数构造器: model, year, color
24     public Car(String model, int year, String color) {
25         this.model = model;
26         this.year = year;
27         this.color = color;
28     }
29     // 打印汽车信息
30     public void printCarInfo() {
31         System.out.println("Car Model: " + model);
```

```

32     System.out.println("Year: " + year);
33     System.out.println("Color: " + color);
34 }
35
36 public static void main(String[] args) {
37     // 使用不同的构造器创建对象
38     Car car1 = new Car();
39     Car car2 = new Car("Tesla Model S");
40     Car car3 = new Car("BMW M3", 2022);
41     Car car4 = new Car("Audi A6", 2020, "Black");
42
43     // 打印各个汽车的信息
44     car1.printCarInfo();
45     System.out.println();
46     car2.printCarInfo();
47     System.out.println();
48     car3.printCarInfo();
49     System.out.println();
50     car4.printCarInfo();
51 }
52 }
```

Fence 87

## 5. 005

### 5.1 抽象类和接口的区别

**(1) 文字题：**解释Java中抽象类和接口的主要区别，并讨论他们各自适用的场景

特性	抽象类	接口
定义方式	使用 <b>abstract</b> 关键字定义类，类中可以有抽象方法和具体方法	使用 <b>interface</b> 关键字定义，类中的方法默认是抽象的（Java 8）
继承方式	一个类只能继承一个抽象类（Java不支持多重继承）	一个类可以实现多个接口（支持多重继承）

特性	抽象类	接口
成员变量	可以有成员变量，可以是实例变量或静态变量，且可以定义访问修饰符（如 <code>private</code> , <code>protected</code> , <code>public</code> ）	只能有 <code>public static final</code> 常量（默认常量），即只能定义常量，没有实例变量
方法	可以有抽象方法和具体方法（方法可以有实现，也可以都是具体方法）	只能有抽象方法（直到Java 8，接口中可以有默认方法和静态方法）
构造方法	可以有构造方法	不可以有构造方法
访问修饰符	抽象类的方法可以有不同的访问修饰符（如 <code>public</code> 、 <code>protected</code> 、 <code>private</code> 等）	接口中的方法默认是 <code>public abstract</code> 的，不能有其他访问修饰符
实现方式	一个类继承抽象类时，必须实现抽象类中所有的抽象方法（除非该类本身是抽象类）	类实现接口时，必须实现接口中的所有方法（除非该类本身是抽象类）
适用场景	用于类之间有“is a”关系的场景，且希望提供部分实现时	用于表示类具有某种能力，且不关心如何实现时

Table 6

## 【代码示例】

### 抽象类



```

1 abstract class Vehicle{
2     //非抽象方法，可以由子类继承并使用，抽象类本身也可使用
3     public void start(){
4         System.out.println("Vehicle started");
5     }
6
7     public void stop(){
8
9 }
```

```

8     System.out.println("Vehicle stopped");
9 }
10
11 //抽象方法，必须由子类覆写实现
12 public abstract void move();
13 }
14
15 //继承抽象类Vehicle的子类Car
16 class Car extends Vehicle{
17     //具体在子类中覆写实现抽象方法
18     @Override
19     public void move() {
20         System.out.println("Car moved");
21     }
22 }
23
24 //继承抽象类Vehicle的子类Bicycle
25 class Bicycle extends Vehicle{
26     //具体在子类中覆写实现抽象方法
27     @Override
28     public void move() {
29         System.out.println("Bicycle moved");
30     }
31 }

```

Fence 88

## 接口



```

1 //接口1：车载空调控制
2 interface Airconditioner{
3     void turn_on_airconditioner();
4     void turn_off_airconditioner();
5 }
6
7 //接口2：Carplay控制
8 interface Carplay{
9     void turn_on_Carplay();
10    void turn_off_Carplay();
11 }
12
13 //Car类调用两个接口
14 class Car implements Airconditioner,Carplay{
15     @Override
16     public void turn_on_airconditioner(){
17         System.out.println("Airconditioner is turned on");
18     }
19     @Override
20     public void turn_off_airconditioner(){
21         System.out.println("Airconditioner is turned off");
22 }

```

```
22 }
23 @Override
24 public void turn_on_Carplay(){
25     System.out.println("Carplay is turned on");
26 }
27 @Override
28 public void turn_off_Carplay(){
29     System.out.println("Carplay is turned off");
30 }
31 }
```

Fence 89

- 总的来说，抽象类是为子类的构成提供模板，且每一个子类只能继承一个模板；而接口是为类提供实现方法的模板，每个类可以实现多个接口。
- 抽象类和接口都不能实例化，抽象类跟c++的抽象类几乎一致，接口的定义更加严格。

① Note

抽象类和接口不能通过 new 的方式来实例化，但是可以通过子类的构造方法间接实例化，也可以通过生成匿名类的方式来实现。

## 子类的构造方法间接实例化



```
1 abstract class AbstractFruit {
2     public AbstractFruit(){
3         System.out.println("我是Fruit的抽象类，我被实例化了");
4     }
5     public abstract void say();
6 }
7
8 public class Orange extends AbstractFruit {
9     public Orange() {
10        System.out.println("我是Orange类，我被实例化了");
11    }
12    @Override
13    public void say() {
14        System.out.println("我是一个Orange");
15    }
16    public static void main(String[] args) {
17        Orange orange = new Orange();
18    }
19 }
20 }
```

Fence 90



- 1 我是Fruit的抽象类，我被实例化了
- 2 我是Orange类，我被实例化了

Fence 91

## 生成匿名类的方式



```
1 public static void main(String[] args) {  
2     //     Orange orange = new Orange();  
3     AbstractFruit abstractFruit = new AbstractFruit() {  
4         @Override  
5         public void say() {  
6             System.out.println("大家好我是abstractFruit");  
7         }  
8     };  
9     abstractFruit.say();  
10 }  
11 }
```

Fence 92



- 1 我是Fruit的抽象类，我被实例化了
- 2 大家好我是abstractFruit

Fence 93

**(2) 编程题：**创建一个抽象类Shape，包含一个抽象方法draw()，然后创建两个子类Circle和Rectangle实现这个抽象方法。



```
1 abstract class Shape{  
2     abstract void draw();  
3 }  
4  
5 class Rectangle extends Shape{  
6     @Override  
7     void draw(){  
8         System.out.println("Rectagle");  
9     }  
10 }  
11  
12 class Circle extends Shape{  
13     @Override
```

```

14     void draw(){
15         System.out.println("Circle");
16     }
17 }
18
19 class Main{
20     public static void main(String[] args){
21         Rectangle r=new Rectangle();
22         r.draw();
23         Circle c=new Circle();
24         c.draw();
25     }
26 }
```

Fence 94

## 5.2 抽象类中成员变量的访问

**(1) 文字题：**讨论在抽象类中是否可以有非抽象方法和实例变量，并解释它们如何在子类中被访问。

- **非抽象方法**：抽象类是不能被实例化的类，可以包含抽象方法和非抽象方法。它的目的是为子类提供公共的行为（方法）和状态（实例变量），从而减少代码重复和强制子类实现特定的行为。子类可以直接继承这些方法，也可以重写方法来实现特定实现
- **实例变量**：抽象类中可以定义实例变量（非静态变量）。这些实例变量可以被子类继承并访问，也可以在子类的构造方法中初始化。子类可以直接访问（继承）这些实例变量，除非它们被声明为 **private**



```

1 abstract class Animal {
2     // 实例变量
3     protected String name;
4
5     // 构造方法
6     public Animal(String name) {
7         this.name = name;
8     }
9
10    // 非抽象方法
11    public void eat() {
12        System.out.println(name + " is eating.");
13    }
14
15    // 抽象方法
16    public abstract void makeSound();
17 }
```

```
18  
19 class Dog extends Animal {  
20     public Dog(String name) {  
21         super(name); // 调用父类的构造方法  
22     //super(name) 是用来调用父类带有参数的构造方法，传递相应的参数。name 参数会传  
递给父类的构造方法。  
23     }  
24  
25     @Override  
26     public void makeSound() {  
27         System.out.println("Woof Woof");  
28     }  
29 }  
30  
31 public class Main {  
32     public static void main(String[] args) {  
33         Dog dog = new Dog("Buddy");  
34         dog.eat(); // 调用父类的非抽象方法  
35         dog.makeSound(); // 调用重写的抽象方法  
36     }  
37 }
```

Fence 95

(2) 编程题：设计一个抽象类Vehicle，其中包含一个实例变量wheels和一个非抽象方法startEngine()。创建一个子类Car继承Vehicle并重写startEngine()方法。

```
● ● ●  
1 abstract class Vehicle{  
2     protected String wheels;  
3  
4     public Vehicle(String wheels){  
5         this.wheels=wheels;  
6     }  
7  
8     void startEngine(){  
9         System.out.println("engine started");  
10    }  
11 }  
12  
13 class Car extends Vehicle{  
14     public Car(String wheels){  
15         super(wheels);  
16     }  
17  
18     @Override  
19     void startEngine() {  
20         System.out.println("Car:engine started");  
21     }
```

```
22 }
23
24 class Main{
25     public static void main(String[] args){
26         Car car=new Car("01");
27         car.startEngine();
28     }
29 }
```

Fence 96

## 5.3 抽象类与构造器

**(1) 文字题：**解释为什么抽象类不能被实例化，以及是否可以为抽象类提供构造器。

- **为什么抽象类不能被实例化？** 跳转

首先抽象类的设计目的就是为了作为其他类的基础父类，用于提供共同的行为和属性，不应该单独存在，就好比你可以实例化一个苹果，但你不能实例化一个水果，因此作为设计的理念，这是一种规定。

其次，抽象类包含抽象方法，这意味着该抽象方法不会在抽象类中实现，抽象类并不具备完整功能的实现。因此如果我们试图实例化一个抽象类的对象，**JVM** 会报错，**因为当关键字new申请访问内存时，抽象类中没有具体的成员变量或成员方法，没办法准确分配内存。**

- **是否可以为抽象类提供构造器？**

是的，抽象类是可以有构造器的。虽然抽象类不能被直接实例化，但它可以包含构造方法。构造方法用于初始化抽象类的状态（即成员变量），并且在子类实例化时可以被调用。

**(2) 编程题：**编写一个抽象类 Animal，包含一个构造器和一个抽象方法 **makeSound()**。创建一个子类Dog继承Animal并实现 **makeSound()** 方法。



```
1 abstract class Animal{
2     protected String name;
3
4     public Animal(String name){
5         this.name=name;
6         System.out.println("Animal: "+name);
7     }
}
```

```

8
9     abstract void makeSound();
10 }
11
12 class Dog extends Animal{
13     public Dog(String name){
14         super(name);
15     }
16
17     @Override
18     void makeSound(){
19         System.out.println("The dog: " +name+ " barks");
20     }
21 }
22
23 class Main{
24     public static void main(String[] args) {
25         Dog dog=new Dog("BaoZi");
26         dog.makeSound();
27     }
28 }
```

Fence 97

## 5.4 抽象类与多态

**(1) 文字题：**描述如何通过抽象类实现多态，并讨论它在设计模式中的应用。

- **如何通过抽象类实现多态？**

**多态** 是面向对象编程中的一个重要的概念，指的是“**同一个方法调用在不同情况下产生不同的行为**”。

**1. 定义一个抽象类**，其中包含抽象方法或非抽象方法，这些方法在不同的子类中可以有不同的实现。

**2. 创建多个子类**，继承抽象类，并实现其中的抽象方法，子类可以提供自己的具体实现。

**3. 通过父类引用指向子类对象**，从而实现多态。父类的引用可以指向任何一个子类的对象，而调用方法时会执行对应子类的方法实现。



```

1 // 定义一个抽象类 Animal
2 abstract class Animal {
3     public abstract void sound();
```

```

4 }
5
6 // Dog 类继承 Animal 类, 并实现 sound 方法
7 class Dog extends Animal {
8     @Override
9     public void sound() {
10         System.out.println("Woof Woof");
11     }
12 }
13
14 // Cat 类继承 Animal 类, 并实现 sound 方法
15 class Cat extends Animal {
16     @Override
17     public void sound() {
18         System.out.println("Meow");
19     }
20 }
21
22 public class Main {
23     public static void main(String[] args) {
24         // 创建 Animal 类型的引用, 可以指向 Dog 或 Cat 对象
25         Animal myDog = new Dog(); // 父类引用指向子类对象
26         Animal myCat = new Cat(); // 父类引用指向子类对象
27
28         // 调用不同子类的 sound 方法, 展示多态效果
29         myDog.sound(); // 输出: Woof Woof
30         myCat.sound(); // 输出: Meow
31     }
32 }

```

Fence 98

## • 多态在设计模式中的应用?

多态是设计模式中非常重要的一个概念, 它能够让我们编写更灵活、可扩展的代码。许多常见的设计模式都依赖于多态来解耦代码, 提高代码的复用性和灵活性。

以下是几种常见设计模式中多态的应用:

### 1.策略模式

策略模式是一种行为型设计模式, 它允许在运行时改变一个类的行为。通过定义一系列算法(策略), 并将这些算法封装成独立的类, 客户端可以根据需要选择使用不同的算法。

策略模式通过多态来选择不同的算法实现, 并将它们作为策略对象传递给上下文类。



```

1 // 定义一个策略接口
2 interface PaymentStrategy {
3     void pay(int amount);

```

```

4 }
5
6 // 具体策略类: 信用卡支付
7 class CreditCardPayment implements PaymentStrategy {
8     @Override
9     public void pay(int amount) {
10         System.out.println("Paid " + amount + " using Credit Card.");
11     }
12 }
13
14 // 具体策略类: 支付宝支付
15 class AlipayPayment implements PaymentStrategy {
16     @Override
17     public void pay(int amount) {
18         System.out.println("Paid " + amount + " using Alipay.");
19     }
20 }
21
22 // 上下文类
23 class PaymentContext {
24     private PaymentStrategy strategy;
25
26     public PaymentContext(PaymentStrategy strategy) {
27         this.strategy = strategy;
28     }
29
30     public void executePayment(int amount) {
31         strategy.pay(amount); // 根据不同的策略调用不同的实现
32     }
33 }
34
35 public class Main {
36     public static void main(String[] args) {
37         PaymentContext context = new PaymentContext(new
38             CreditCardPayment());
39         context.executePayment(100); // 使用 CreditCardPayment
40
41         context = new PaymentContext(new AlipayPayment());
42         context.executePayment(200); // 使用 AlipayPayment
43     }
44 }
```

Fence 99

**输出:**



```

1 Paid 100 using Credit Card.
2 Paid 200 using Alipay.
```

Fence 100

## 2.工厂方法模式

工厂方法模式是一种创建型设计模式，它定义了一个创建对象的接口，让子类决定实例化哪个类。通过工厂方法，客户端可以通过抽象接口创建具体的对象，避免直接依赖具体的类。

工厂方法模式利用多态，允许工厂方法返回不同类型的对象，而客户端只依赖于抽象类型。



```
1 // 抽象产品
2 abstract class Product {
3     public abstract void doSomething();
4 }
5
6 // 具体产品 A
7 class ProductA extends Product {
8     @Override
9     public void doSomething() {
10         System.out.println("Product A is doing something.");
11     }
12 }
13
14 // 具体产品 B
15 class ProductB extends Product {
16     @Override
17     public void doSomething() {
18         System.out.println("Product B is doing something.");
19     }
20 }
21
22 // 抽象工厂
23 abstract class Factory {
24     public abstract Product createProduct();
25 }
26
27 // 具体工厂 A
28 class FactoryA extends Factory {
29     @Override
30     public Product createProduct() {
31         return new ProductA(); // 创建 ProductA 对象
32     }
33 }
34
35 // 具体工厂 B
36 class FactoryB extends Factory {
37     @Override
38     public Product createProduct() {
39         return new ProductB(); // 创建 ProductB 对象
40     }
}
```

```

41 }
42
43 public class Main {
44     public static void main(String[] args) {
45         Factory factoryA = new FactoryA();
46         Product productA = factoryA.createProduct();
47         productA.doSomething(); // 输出: Product A is doing
48         something.
49
50         Factory factoryB = new FactoryB();
51         Product productB = factoryB.createProduct();
52         productB.doSomething(); // 输出: Product B is doing
53     }

```

Fence 101

**(2) 编程题:** 创建一个抽象类Painter，包含一个抽象方法paint()。创建两个子类HousePainter 和 CarPainter 实现 paint() 方法。编写一个方法paintObject(Painter painter)，接受Painter对象并调用paint()方法。



```

1 abstract class Painter {
2     // 抽象方法 paint()
3     public abstract void paint();
4 }
5
6 class HousePainter extends Painter {
7     @Override
8     public void paint() {
9         System.out.println("Painting a house.");
10    }
11 }
12
13 class CarPainter extends Painter {
14     @Override
15     public void paint() {
16         System.out.println("Painting a car.");
17    }
18 }
19
20 public class Main {
21     public static void main(String[] args) {
22         Painter housePainter = new HousePainter();
23         Painter carPainter = new CarPainter();
24
25         paintObject(housePainter); // 输出: Painting a house.

```

```

26     paintObject(carPainter);      // 输出: Painting a car.
27 }
28
29 // 定义一个方法, 接受 Painter 类型的对象并调用 paint() 方法
30 public static void paintObject(Painter painter) {
31     painter.paint();
32 }
33 }
```

Fence 102

## 5.5 抽象类与方法重写

**(1) 文字题：**讨论在抽象类中重写方法与在普通类中重写方法的区别和注意事项。

- **重写方法的区别：**

抽象类的方法不提供实现，子类必须重写抽象方法（除非子类是抽象类）；普通类的方法有实现，子类可以选择重写父类的普通方法

- **注意事项：**

- 在**抽象类**中重写方法时，子类必须重写所有的抽象方法。
- 父类中的方法在子类中必须可见。对于父类中的 **private** 方法，子类虽能继承，但无法访问和覆盖；对于父类中 **final** 的方法，子类继承但不能重写
- 子类和父类的**方法名、参数列表必须相同（定义）**，并且子类的返回值与父类相同或者是父类返回类型的子类型（**jdk1.5** 之后，否则会报错）。如果方法名称相同而参数列表不同，那么只是**方法的重载，而非重写**。
- 子类方法的访问权限 **不能小于** 父类方法的访问权限。访问权限由高到低：**public**、**protected**、包访问权限、**private**。
- 子类方法**不能比父类方法抛出更多的编译时异常**（不是运行时异常），即子类方法抛出的编译时异常或者和父类相同或者是父类异常的子类。

**(2) 编程题：**设计一个抽象类 **MusicPlayer**，包含一个抽象方法**play()**。创建一个子类 **MP3Player** 重写**play()**方法，并在主方法中创建 **MP3Player** 对象并调用**play()**方法。

```

● ● ●
1 abstract class MusicPlayer{
2     abstract void play();
3 }
4
5 class MP3Player extends MusicPlayer{
6     @Override
7     void play() {
```

```
8     System.out.println("Playing MP3!");
9 }
10}
11
12class Main{
13    public static void main(String[] args) {
14        MP3Player mp3 = new MP3Player();
15        mp3.play();
16    }
17}
```

Fence 103

## 6. 006

### 6.1 Java局部变量的作用域

**(1) 文字题：**描述Java中局部变量的作用域是什么，并说明它们何时被初始化。

在 Java 中，**局部变量**是指在方法、构造器或代码块中声明的变量。

- **局部变量的作用域**

局部变量只能在它被声明的**方法**、**构造器**或**代码块**内部访问。也就是说，局部变量只能在其所在的方法、构造器或代码块中有效，在方法调用外部是无法访问的。



```
1 // 1: 方法内部
2 public void Method(){
3     ...
4 }
5
6 // 2: 代码块内部
7 if(true){
8     ...
9 }
```

Fence 104

- **局部变量何时被初始化**

局部变量**必须显式初始化后才能使用**，即其必须在其作用域中被使用其值之前初始化，**编译器不会为局部变量分配默认值**

**(2) 编程题：**编写一个方法，该方法接受两个整数参数并返回它们的和。讨论局部变量a和b的作用域。



```
1 public class Main {  
2  
3     public static int sum(int a, int b) {  
4         // a 和 b 是方法的局部变量，它们的作用域只限于该方法内部  
5         return a + b;  
6     }  
7  
8     public static void main(String[] args) {  
9         int result = sum(5, 3); // 调用 sum 方法，传入 5 和 3  
10        System.out.println("The sum is: " + result);  
11    }  
12}  
13
```

Fence 105

## 6.2 Java实例变量和类变量的作用域

**(1) 文字题：**区分Java中的实例变量和类变量，并解释它们各自的作用域。

- **实例变量与类变量的比较**

特性	实例变量	类变量
定义方式	在类中声明时不使用 <code>static</code> 关键字	使用 <code>static</code> 关键字定义
存储位置	每个对象都有自己的一份副本	所有对象共享同一个副本
访问方式	通过对象来访问	可以通过 <b>类名</b> 或对象来访问
生命周期	由对象的创建和销毁控制	在类加载时创建，类卸载时销毁

特性	实例变量	类变量
作用域	类的实例，方法中访问	类的所有实例，整个类内都可访问
是否共享	不共享，每个对象有独立的实例变量	共享，所有实例共用一个类变量

Table 7

- 各自的作用域

**实例变量：**实例变量的作用域是类的整个实例，在类的所有方法中都可以访问（前提是这些方法能访问到该实例变量）。它们的作用范围是类的实例，不同对象的实例变量互相独立。

**类变量：**类变量的作用域是整个类，在类的所有方法中都可以访问（前提是这些方法能访问到该类变量）。类变量在整个应用程序中是唯一的，所有对象共享同一个类变量。

**(2) 编程题：**创建一个类Counter，包含一个实例变量count和一个类变量total。编写方法来增加count和total的值，并在主方法中演示它们的作用域。



```

1 public class Counter {
2     private int count = 0;
3     public static int total = 0;
4
5     public void incrementCount() {
6         count++;           // 每个对象的 count 增加
7     }
8
9     public static void incrementTotal() {
10        total++;          // 类变量 total 增加
11    }
12
13    public void display() {
14        System.out.println("Count: " + count);      // 显示实例变量 count
15        System.out.println("Total: " + total);        // 显示类变量 total
16    }
17
18    public static void main(String[] args) {
19        Counter counter1 = new Counter();
20        Counter counter2 = new Counter();
21
22        counter1.incrementCount(); // counter1 的 count 增加
23        counter1.incrementTotal(); // total 增加
24

```

```

25     counter2.incrementCount(); // counter2 的 count 增加
26     counter2.incrementTotal(); // total 增加
27
28     System.out.println("Counter 1:");
29     counter1.display(); // 输出: 1 2
30     System.out.println("Counter 2:");
31     counter2.display(); // 输出: 1 2
32
33     // 显示类变量 total 直接通过类名访问
34     System.out.println("Total (from class): " + Counter.total);
35 // 通过类名访问类变量
36 }

```

Fence 106

## 6.3 Java代码块作用域

**(1) 文字题：**解释Java中代码块作用域是什么，并给出一个代码块作用域的例子。

在 Java 中，**代码块的作用域**是指代码块中定义的变量仅在该代码块内部有效。在不同的代码块（如 **if** 语句、**for** 循环、**try-catch** 块等）中定义的局部变量的作用域是有限的，它们只在该块内部存在，不能在外部访问。



```

1 public class BlockScopeExample {
2     public static void main(String[] args) {
3         int x = 10; // 在 main 方法的作用域内声明
4
5         // 第一个代码块
6         {
7             int y = 20; // y 只在这个代码块内有效
8             System.out.println("Inside first block:");
9             System.out.println("x = " + x); // 可以访问外部的 x
10            System.out.println("y = " + y); // 可以访问 y, 因为 y 在此
          代码块内声明
11        }
12
13        // 第二个代码块
14        {
15            // int y = 30; // 错误! 此处的 y 已在前一个代码块中声明过
16            int z = 40; // z 只在这个代码块内有效
17            System.out.println("Inside second block:");
18            System.out.println("x = " + x); // 可以访问外部的 x
19            // System.out.println("y = " + y); // 错误! y 不在此作用域
          内

```

```

20         System.out.println("z = " + z); // 可以访问 z, 因为 z 在此
代码块内声明
21     }
22
23     // System.out.println("y = " + y); // 错误! y 不在 main 方法的
作用域内
24     // System.out.println("z = " + z); // 错误! z 不在 main 方法的
作用域内
25
26     System.out.println("Outside all blocks:");
27     System.out.println("x = " + x); // x 依然有效, 因为它在 main 方
法的作用域内
28 }
29 }
```

Fence 107

**(2) 编程题:** 在一个类中创建一个静态代码块和一个实例代码块，并在其中初始化类变量和实例变量。讨论这些变量的作用域。



```

1 public class BlockExample {
2     static int classVariable;
3
4     int instanceVariable;
5
6     // 静态代码块: 静态代码块只执行一次
7     static {
8         System.out.println("Static block executed.");
9         classVariable = 100; // 初始化类变量
10    }
11
12    // 实例代码块
13    {
14        System.out.println("Instance block executed.");
15        instanceVariable = 50; // 初始化实例变量
16    }
17
18    // 构造器
19    public BlockExample() {
20        System.out.println("Constructor executed.");
21    }
22
23    public void display() {
24        System.out.println("classVariable = " + classVariable); // 类
变量
25        System.out.println("instanceVariable = " + instanceVariable);
// 实例变量
26    }
}
```

```
27
28     public static void main(String[] args) {
29         System.out.println("Main method started.");
30
31         BlockExample obj1 = new BlockExample();
32         obj1.display();
33
34         System.out.println();
35
36         BlockExample obj2 = new BlockExample();
37         obj2.display();
38         System.out.println("Main method ended.");
39     }
40 }
```

Fence 108

## 输出:



```
1 Main method started.
2 Static block executed.
3 Instance block executed.
4 Constructor executed.
5 classVariable = 100
6 instanceVariable = 50
7
8 Instance block executed.
9 Constructor executed.
10 classVariable = 100
11 instanceVariable = 50
12 Main method ended.
```

Fence 109

## 作用域分析:

### 静态变量 `classVariable` :

- `classVariable` 是类变量，属于整个类的所有实例。它的作用域是整个类，所有对象共享这一个静态变量。在静态代码块中被初始化为 `100`，并且可以通过任何对象访问，也可以通过类名访问。例如：`BlockExample.classVariable`。
- 静态代码块中的 `classVariable` 在类加载时初始化，一旦类加载完成，它就可以在整个类的作用域中访问。

### 实例变量 `instanceVariable` :

- `instanceVariable` 是实例变量，属于每个对象的独立副本。它的作用域是该对象的生命周期。在每个对象创建时，都会通过实例代码块对它进行初始化，且每个对象的 `instanceVariable` 独立于其他对象。

- **instanceVariable** 在每次创建对象时初始化，每个对象都有自己的副本，互不干扰。

## 6.4 Java方法参数的作用域

**(1) 文字题：**描述Java中方法参数的作用域，并讨论它们与局部变量的关系。

**方法参数**是指在方法定义时声明的变量，它们用于接收调用方法时传递的实际参数值。方法参数在方法的作用域内有效，方法执行时它们会被初始化，并且只能在该方法的作用域内访问。

- **与局部变量的关系**

特点	方法参数	局部变量
<b>声明位置</b>	方法的参数列表中	在方法体或代码块中声明
<b>作用域</b>	方法体内	方法体或所在代码块内
<b>初始化</b>	方法调用时自动初始化	必须显式初始化
<b>生命周期</b>	方法调用时创建，方法结束时销毁	方法调用时创建，方法或代码块结束时销毁
<b>命名冲突</b>	方法参数和局部变量可以同名，但局部变量会隐藏方法参数	可以与方法参数同名，但会隐藏方法参数

Table 8

####

**(2) 编程题：**编写一个方法，该方法接受一个字符串参数，并在方法内部创建一个同名的局部变量。讨论参数和局部变量的作用域和关系。



```

1 public class VariableShadowingExample {
2
3     public static void printMessage(String message) {
4         // 创建一个同名的局部变量

```

```

5      String message = "Local variable message"; // 编译器会直接报错):
6
7      System.out.println("Parameter message: " + message); // 尝试访
问参数 message
8      System.out.println("Local variable message: " + message); // 访
问局部变量 message
9  }
10
11 public static void main(String[] args) {
12     String message = "Method parameter message";
13
14     // 调用方法
15     printMessage(message);
16 }
17 }
```

Fence 110

## 6.5 Java中this和super关键字与作用域

**(1) 文字题：**解释Java中this和super关键字如何影响成员变量和方法的作用域。

**this** 关键字指代当前对象的引用。在实例方法或构造器中，**this** 代表当前方法所属的对象，可以用它来访问当前对象的实例变量、调用当前对象的实例方法。

- 访问当前对象的实例变量。
- 调用当前对象的实例方法。
- 区分成员变量和局部变量（尤其是当它们同名时）。



```

1 public class Person {
2     private String name; // 实例变量
3
4     public Person(String name) {
5         // 使用 this 来区分局部变量 name 和实例变量 name
6         this.name = name;
7     }
8
9     public void printName() {
10        System.out.println("Name: " + this.name); // 通过 this 访问实
例变量
11    }
12
13    public static void main(String[] args) {
14        Person person = new Person("Alice");

```

```
15     person.printName(); // 输出 Name: Alice
16 }
17 }
```

Fence 111

## 作用域影响：

- **this** 关键字总是指向当前对象，因此它的作用域是当前对象的成员变量和方法。
- **this** 仅在实例方法和构造方法中有效，无法在静态方法中使用，因为静态方法属于类而不是对象。

**super** 关键字指代当前对象的父类对象。在子类中，**super** 用来访问父类的实例变量和实例方法，或者调用父类的构造方法。

- 访问父类的实例变量（如果子类没有覆盖父类的成员变量）。
- 调用父类的实例方法（如果子类没有覆盖父类的方法）。
- 在子类构造器中调用父类的构造器。



```
1 class Animal {
2     String name; // 父类成员变量
3
4     public Animal(String name) {
5         this.name = name; // 使用 this 访问父类的成员变量
6     }
7
8     public void speak() {
9         System.out.println("Animal speaks");
10    }
11 }
12
13 class Dog extends Animal {
14     String name; // 子类成员变量
15
16     public Dog(String name) {
17         super(name); // 使用 super 调用父类的构造器
18         this.name = "Dog: " + name; // 子类的成员变量
19     }
20
21     public void speak() {
22         super.speak(); // 使用 super 调用父类的 speak 方法
23         System.out.println("Dog barks");
24     }
25
26     public void printName() {
```

```

27     System.out.println("Dog's name: " + this.name); // 使用 this
    访问子类的成员变量
28     System.out.println("Animal's name: " + super.name); // 使用
    super 访问父类的成员变量
29 }
30
31 public static void main(String[] args) {
32     Dog dog = new Dog("Buddy");
33     dog.speak(); // 输出 Animal speaks \n Dog barks
34     dog.printName(); // 输出 Dog's name: Dog: Buddy \n Animal's
    name: Buddy
35 }
36 }
```

Fence 112

## 作用域影响：

- **super** 用于指代父类，因此它的作用域限于父类的成员变量和方法。
- **super** 只能在子类中使用，用于访问父类的成员（变量、方法等），不能在父类中使用。
- **super** 也可以用来调用父类的构造器，通常是在子类构造器的第一行。

**(2) 编程题：** 创建一个基类Person和一个子类Employee。在Person类中定义一个**getName()**方法，在Employee类中重写这个方法，并使用**super.getName()**来调用基类方法。讨论this和super如何影响方法调用。



```

1 class Person {
2     private String name;
3
4     // 构造方法
5     public Person(String name) {
6         this.name = name; // 使用 this 来访问当前对象的实例变量
7     }
8
9     public String getName() {
10        return name;
11    }
12 }
13
14 class Employee extends Person {
15     private String name;
16
17     // 构造方法
18     public Employee(String name) {
19         super(name); // 使用 super 调用父类构造器
20     }
21 }
```

```

20     this.name = "Employee: " + name; // 子类的 name 被重写
21 }
22
23 @Override
24 public String getName() {
25     return super.getName() + " (from Employee)";
26 }
27
28 public static void main(String[] args) {
29     Employee emp = new Employee("John Doe");
30     System.out.println(emp.getName()); // 调用重写后的 getName 方法
31 }
32 }

```

Fence 113

关键字	含义	作用
this	当前对象的引用	访问当前类的实例变量、实例方法或调用当前类的方法
super	父类对象的引用	访问父类的实例变量、实例方法或调用父类的构造器

Table 9

## 7. 007

### 7.1 Java接口的多实现

**(1) 文字题：**解释为什么一个Java接口可以被多个类实现，并讨论这种设计的好处。

- **为什么一个接口可以被多个类实现？**

Java 支持一个接口被多个类实现的原因是接口与类之间存在松耦合关系，同时接口定义了类应该具有的一组行为规范，而不限制具体的实现。

**1. 接口只定义方法签名：**接口中的方法没有具体实现（除非是 **default** 方法），它们只是行为的声明。这意味着类在实现接口时，必须提供方法的具体实现，或者直接继承抽象类来实现这些方法。

**2. 类与接口的实现是松耦合的：**类通过实现接口，声明自己符合某个行为规范，而不关心接口的具体实现细节。接口仅仅是行为的契约，而不是实现的细节。因此，多个类可以实现相同的接口，且每个类都可以提供自己的具体实现。

- **这种设计的好处？**

**1. 多态性：**由于多个类实现同一个接口，接口类型可以作为统一的引用类型处理不同的实现类对象。这种多态性使得可以通过接口引用来调用不同对象的相同方法，而不需要关心它们的具体实现。



```
1 Animal animal1 = new Dog();
2 Animal animal2 = new Cat();
3 animal1.makeSound(); // 输出 "Bark"
4 animal2.makeSound(); // 输出 "Meow"
```

Fence 114

**2. 解耦：**接口帮助类与类之间建立松耦合的关系。类实现接口时，只需关注接口中定义的行为，而不关心接口的具体实现。这使得系统中的各个部分可以独立开发、修改或替换，而不影响其他部分。

**3. 扩展性：**接口为系统提供了很好的扩展性。如果未来需要添加更多的类，只需让新的类实现现有的接口，而不需要修改原有类的代码。这遵循了**开闭原则**：对扩展开放，对修改封闭。

**(2) 编程题：**定义一个接口Flyable，包含一个方法fly()。创建两个类Bird和Airplane实现这个接口，并在主方法中创建它们的实例，调用fly()方法。



```
1 interface Flyable{
2     void fly();
3 }
4
5 class Bird implements Flyable{
6     public void fly(){
7         System.out.println("Bird fly");
8     }
9 }
10
11 class Airplane implements Flyable{
12     public void fly(){
13         System.out.println("Airplane fly");
14     }
15 }
16
17 public class Main {
```

```
18 public static void main(String[] args) {  
19     Flyable bird = new Bird();  
20     Flyable airplane = new Airplane();  
21     bird.fly();  
22     airplane.fly();  
23 }  
24 }
```

Fence 115

## 7.2 Java默认方法

**(1) 文字题：**描述Java 8中引入的默认方法是什么，以及它们如何影响接口的实现。

- 引入的默认方法是什么？

Java 8 引入了**默认方法 (default methods)** 的概念，这一特性允许接口中提供方法的实现，而不仅仅是方法的声明。默认方法通过 `default` 关键字进行定义，可以在接口中为某些方法提供具体实现，而不需要强制要求接口的实现类进行覆盖。默认方法使得接口的设计更加灵活，尤其是在接口演化和向后兼容性方面。



```
1 public interface MyInterface {  
2     // 普通的抽象方法  
3     void abstractMethod();  
4  
5     // 默认方法  
6     default void defaultMethod() {  
7         System.out.println("This is a default method");  
8     }  
9 }
```

Fence 116

- 如何影响接口的实现？

### 1. 向接口添加新方法而不破坏现有实现

默认方法的最大优势是，允许在接口中添加新的方法而不影响已有的实现类。这解决了传统的接口版本更新问题，因为如果我们向接口添加一个新的抽象方法，所有实现该接口的类都必须重新实现这个方法，而默认方法则避免了这种情况。

### 2. 接口可以有实现方法

在 Java 8 之前，接口只能包含抽象方法，所有实现类必须提供这些方法的实现。而在 Java 8 引入默认方法后，接口不仅可以定义方法签名，还可以提供方法的默认实现。这样，接口就变得更加灵活，不再要求实现类覆盖每个方法。

### 3. 实现类选择是否覆盖默认方法

实现类可以选择是否覆盖接口中的默认方法。如果实现类没有覆盖默认方法，默认方法的实现将被继承。如果实现类需要提供自己的实现，可以覆盖默认方法。默认方法为实现类提供了选择和扩展的灵活性。

### 4. 多个接口的冲突

如果一个类实现了多个接口，并且这些接口中都定义了相同的默认方法，那么该类必须显式地覆盖该方法并提供一个自己的实现。否则，编译器会提示错误，因为无法确定使用哪个接口的默认方法。

**(2) 编程题：**在一个接口Chargeable中定义一个默认方法charge()。实现这个接口，并在实现类中覆盖这个方法。

```
● ● ●
1 interface Chargeable{
2     default public void charge() {
3         System.out.println("Charging chargeable");
4     }
5 }
6
7 class Car implements Chargeable{
8     @Override
9     public void charge() {
10         System.out.println("Car chargeable");
11     }
12 }
13
14 public class Main{
15     public static void main(String[] args) {
16         Car car = new Car();
17         car.charge();
18     }
19 }
```

## 7.3 Java接口与抽象类的区别

(1) 文字题：讨论Java接口和抽象类的主要区别，并给出使用场景的例子。

见 005-题目1

(2) 编程题：创建一个接口Printable，包含一个方法print()。创建一个抽象类Document也包含一个方法print()。实现Printable接口和继承Document的两个不同类。

```
● ● ●
1 interface Printable{
2     void print();
3 }
4
5 abstract class Document{
6     abstract void print();
7 }
8
9 class Report implements Printable{
10    @Override
11    public void print(){
12        System.out.println("Printing report....");
13    }
14 }
15
16 class Invoice extends Document{
17    @Override
18    public void print(){
19        System.out.println("Printing Invoice... ");
20    }
21 }
22 public class Main {
23     public static void main(String[] args) {
24         Report report = new Report();
25         Invoice invoice = new Invoice();
26         report.print();
27         invoice.print();
28     }
29 }
```

## 7.4 Java接口中的静态方法

(1) 文字题：解释Java接口中可以包含静态方法，并讨论它们如何被调用。

在 Java 8 中，接口不仅可以包含抽象方法和默认方法（`default` 方法），还可以包含**静态方法**。接口中的静态方法与类中的静态方法类似，它们属于接口本身，而不是接口的实例。接口中的静态方法可以提供与接口相关的工具方法或辅助功能，但不能被接口的实现类继承或覆盖。



```
1 public interface MyInterface {  
2     // 静态方法  
3     static void staticMethod() {  
4         System.out.println("This is a static method in an interface.");  
5     }  
6 }
```

Fence 119

### • 如何被调用？

接口中的静态方法只能通过接口名来调用，而不能通过接口的实例或实现类来调用。



```
1 public interface MyInterface {  
2     // 静态方法  
3     static void staticMethod() {  
4         System.out.println("This is a static method in MyInterface.");  
5     }  
6 }  
7  
8 public class Main {  
9     public static void main(String[] args) {  
10         // 调用接口的静态方法  
11         MyInterface.staticMethod(); // 输出 "This is a static method  
12             in MyInterface."  
13     }  
14 }
```

Fence 120

(2) 编程题：在一个接口Calculator中定义一个静态方法add(int a, int b)。在主方法中调用这个静态方法。



```
1 interface Caculator{  
2     static void add(int a, int b){  
3         System.out.println(a+b);  
4     }  
5 }  
6  
7 public class Main {  
8     public static void main(String[] args) {  
9         Caculator.add(2,1);  
10    }  
11 }
```

Fence 121

## 7.5 Java接口作为参数传递

**(1) 文字题：**讨论将接口作为参数传递的好处，并给出一个实际的应用场景。

- **作为参数传递的好处：**

**1. 解耦合：**使用接口作为参数可以让代码更加解耦。接口定义了行为的规范，而不关心实现细节。通过接口传递参数，代码不依赖于具体的类实现，而依赖于行为的抽象。这样，当实现类发生变化时，调用者不需要修改自己的代码，只需要确保实现接口即可。

**2. 多态性：**接口作为参数能够充分利用多态性，允许我们将不同的实现传递给方法。这样，方法可以在不改变其代码的情况下处理不同类型的对象，实现灵活的代码复用。

**3. 可扩展性：**接口为代码提供了高度的扩展性和灵活性。因为接口可以由多个类实现，所以我们可以轻松地扩展系统的功能，而不需要修改现有的代码。例如，在接口的基础上新增新的实现类，并将这些类作为参数传递给现有方法，完全不影响原有的系统功能。

- **实际应用场景：** (排序)



```
1 public interface SortStrategy {  
2     void sort(int[] array); // 接口里面的类由具体的实现类实现  
3 }  
4  
5 // 实现快速排序策略类  
6 public class QuickSortStrategy implements SortStrategy {  
7     @Override  
8     public void sort(int[] array) {
```

```
9     System.out.println("QuickSort: Sorting array");
10    // 快速排序的具体实现
11    quickSort(array, 0, array.length - 1);
12 }
13
14 private void quickSort(int[] array, int low, int high) {
15     if (low < high) {
16         int pi = partition(array, low, high);
17         quickSort(array, low, pi - 1);
18         quickSort(array, pi + 1, high);
19     }
20 }
21
22 private int partition(int[] array, int low, int high) {
23     int pivot = array[high];
24     int i = (low - 1);
25     for (int j = low; j < high; j++) {
26         if (array[j] < pivot) {
27             i++;
28             int temp = array[i];
29             array[i] = array[j];
30             array[j] = temp;
31         }
32     }
33     int temp = array[i + 1];
34     array[i + 1] = array[high];
35     array[high] = temp;
36     return i + 1;
37 }
38 }
39
40 // 实现插入排序策略类
41 public class InsertionSortStrategy implements SortStrategy {
42     @Override
43     public void sort(int[] array) {
44         System.out.println("InsertionSort: Sorting array");
45         // 插入排序的具体实现
46         for (int i = 1; i < array.length; i++) {
47             int key = array[i];
48             int j = i - 1;
49             while (j >= 0 && array[j] > key) {
50                 array[j + 1] = array[j];
51                 j--;
52             }
53             array[j + 1] = key;
54         }
55     }
56 }
57
58 // 排序上下文
59 public class Sorter {
```

```

60     private SortStrategy sortStrategy;
61
62     // 构造函数接受一个 SortStrategy 接口的实现
63     public Sorter(SortStrategy sortStrategy) {
64         this.sortStrategy = sortStrategy;
65     }
66
67     // 排序方法，根据策略进行排序
68     public void sortArray(int[] array) {
69         sortStrategy.sort(array);
70     }
71 }
72
73 // 测试
74 public class Main {
75     public static void main(String[] args) {
76         int[] array = {5, 2, 9, 1, 5, 6};
77
78         // 使用快速排序策略
79         Sorter quickSorter = new Sorter(new QuickSortStrategy());
80         quickSorter.sortArray(array); // 输出 "QuickSort: Sorting
array"
81
82         // 使用插入排序策略
83         Sorter insertionSorter = new Sorter(new
InsertionSortStrategy());
84         insertionSorter.sortArray(array); // 输出 "InsertionSort:
Sorting array"
85     }
86 }
87

```

Fence 122

**(2) 编程题：** 定义一个接口 `RunnableTask`，包含一个方法`run()`。实现这个接口，并创建一个方法 `executeTask(RunnableTask task)`，它接受 `RunnableTask` 接口作为参数并执行`run()`方法。



```

1 interface RunnableTask{
2     void run();
3 }
4
5 //实现类
6 class PrintTask implements RunnableTask{
7     private String message;
8     public PrintTask(String message){
9         this.message = message;

```

```

10 }
11 public void run(){
12     System.out.println("Running task:"+message);
13 }
14 }
15
16 //执行类
17 class TaskExecutor{
18     public void excute(RunnableTask runnableTask){
19         runnableTask.run();
20     }
21 }
22
23 public class Main {
24     public static void main(String[] args) {
25         TaskExecutor taskExecutor = new TaskExecutor();
26         RunnableTask runnableTask = new PrintTask("Hello World");
27         taskExecutor.excute(runnableTask);
28     }
29 }

```

Fence 123

## 8. 008

# 8.1 Java中子类的构造方法如何调用父类的构造方法

**(1) 文字题：**解释Java中子类的构造方法如何调用父类的构造方法，以及super关键字的作用。

这题见[super与this](#)

**(2) 编程题：**创建一个基类Vehicle和一个子类Car。Vehicle有一个构造器，接受一个品牌参数。Car也有一个构造器，接受一个品牌和一个型号参数。在Car的构造器中调用Vehicle的构造器。



```

1 abstract class Vehicle{
2     private String brand;
3
4     public Vehicle(String brand){

```

```

5      this.brand = brand;
6  }
7
8  // 获取品牌的 getter 方法
9  public String getBrand() {
10    return brand;
11  }
12}
13
14 class Car extends Vehicle{
15  private String model;
16  public Car(String brand, String model){
17    super(brand);
18    this.model = model;
19  }
20
21  public String getModel() {
22    return model;
23  }
24
25  public void display(){
26    System.out.println("Car Brand: " + super.getBrand());
27    System.out.println("Car model: " + this.getModel());
28  }
29}
30
31 class Main {
32  public static void main(String[] args) {
33    Car car = new Car("Mercedes", "Ford");
34    car.display();
35  }
36}

```

Fence 124

## 8.2 Java中子类构造方法和父类成员变量的初始化顺序

**(1) 文字题：**描述Java中子类构造方法执行时，父类成员变量和子类成员变量的初始化顺序。

- 在类加载时，如果父类有静态变量或静态代码块，它们会先于任何实例化的代码执行。
- 子类构造方法调用之前，父类的实例变量和实例代码块会被初始化。初始化顺序是：

- 首先，父类的实例变量会被初始化（如果有默认值或显式初始化）。
- 然后，父类中的实例初始化块（如果有）会执行。
- 在子类构造方法执行之前，父类的构造方法会被调用。父类构造器的调用顺序：
  - 如果子类构造器没有显式调用父类构造器（即没有使用 `super()`），则默认调用父类的无参构造器（如果存在）。
  - 如果子类构造器显式调用了父类构造器（如 `super(param)`），则会先调用父类的构造器。
- 父类构造器执行完毕后，接着执行子类的实例变量初始化和实例初始化块。
- 最后，子类的构造方法执行。



```
1 class Animal {
2     // 父类实例变量
3     private String name;
4
5     // 父类实例初始化块
6     {
7         System.out.println("父类实例初始化块执行");
8         name = "Animal";
9     }
10
11    // 父类构造器
12    public Animal() {
13        System.out.println("父类构造器执行");
14    }
15
16    public String getName() {
17        return name;
18    }
19 }
20
21 class Dog extends Animal {
22     // 子类实例变量
23     private String breed;
24
25     // 子类实例初始化块
26     {
27         System.out.println("子类实例初始化块执行");
28         breed = "Labrador";
29     }
30
31    // 子类构造器
32    public Dog() {
33        // 显式调用父类构造器
34        super();
35        System.out.println("子类构造器执行");
36    }
}
```

```
37 }
38
39 public class Main {
40     public static void main(String[] args) {
41         Dog dog = new Dog();
42         System.out.println("犬种: " + dog.getName() + ", 品种: " +
43             dog.breed);
44 }
```

Fence 125

**(2) 编程题:** 创建一个基类Animal和一个子类Dog。Animal有一个成员变量name和一个构造器。Dog也有一个成员变量breed和一个构造器。展示在创建Dog对象时成员变量的初始化顺序。

● ● ●

```
1 abstract class Animal{
2     private String name;
3     public Animal(String name){
4         this.name = name;
5         System.out.println("Animal created");
6     }
7 }
8
9 class Dog extends Animal{
10    private String breed;
11    public Dog(String name, String breed){
12        super(name);
13        System.out.println(name + " created");
14        this.breed = breed;
15        System.out.println(breed + " created");
16    }
17 }
18
19 public class Main {
20     public static void main(String[] args) {
21         Dog dog = new Dog("Dog", "Bob");
22     }
23 }
```

Fence 126

## 8.3 Java中无参构造方法的调用

**(1) 文字题：**讨论Java中子类构造方法如果不显式调用父类构造方法，将会发生什么。

- **父类有无参构造方法时**：如果子类构造方法没有显式调用父类构造方法，Java 会自动插入 `super()` 调用父类的无参构造方法。
- **父类没有无参构造方法时**：如果子类构造方法没有显式调用父类的构造方法，编译器会报错，提示子类必须要显式调用父类的某个构造方法。

**(2) 编程题：**创建一个基类Shape和一个子类Circle。Shape有一个无参构造方法，打印一条消息。Circle有一个构造器，不显式调用Shape的构造器。创建Circle对象并观察输出。



```
1 abstract class Shape{
2     public String name;
3     public Shape(){
4         System.out.println("Creating a Shape");
5     }
6 }
7
8 class Circle extends Shape{
9     public String size;
10    public Circle(String size){
11        this.size = size;
12        System.out.println("Size: " + size);
13    }
14 }
15
16 public class Main{
17     static public void main(String[] args){
18         Circle circle = new Circle("Big");
19     }
20 }
21
22 //输出:
23 //Creating a Shape
24 //Size: Big
```

## 8.4 Java中构造方法和this关键字的使用

**(1) 文字题：**解释Java中this关键字在构造方法中的作用，以及如何使用它调用当前类的其他构造方法。

这题见 [this](#)

**(2) 编程题：**创建一个类Book，包含两个构造器。一个接受书名和作者，另一个接受书名、作者和出版年份。使用this关键字在这两个构造器之间进行调用。



```
1 public class Book {
2     private String title;
3     private String author;
4     private int publicationYear;
5
6     // 构造器1：接受书名和作者
7     public Book(String title, String author) {
8         this.title = title;
9         this.author = author;
10        // 默认出版年份为0
11        this.publicationYear = 0;
12    }
13
14     // 构造器2：接受书名、作者和出版年份
15     public Book(String title, String author, int publicationYear) {
16         // 调用第一个构造器
17         this(title, author);
18         this.publicationYear = publicationYear;
19     }
20
21     // 获取书名
22     public String getTitle() {
23         return title;
24     }
25
26     // 获取作者
27     public String getAuthor() {
28         return author;
29     }
30
31     // 获取出版年份
32     public int getPublicationYear() {
33         return publicationYear;
34     }
35
36     // 打印书籍信息
37     public void printBookInfo() {
38         System.out.println("Book title: " + title);
```

```
39     System.out.println("Author: " + author);
40     System.out.println("Publication year: " + (publicationYear ==
41     0 ? "Not specified" : publicationYear));
42 }
43 public static void main(String[] args) {
44     // 使用第一个构造器
45     Book book1 = new Book("1984", "George Orwell");
46     book1.printBookInfo();
47
48     // 使用第二个构造器
49     Book book2 = new Book("To Kill a Mockingbird", "Harper Lee",
50     1960);
51     book2.printBookInfo();
52 }
```

Fence 128

## 8.5 Java中父类构造方法的调用时机

**(1) 文字题：**讨论在Java中父类的构造方法在子类构造过程中何时被调用。

这题见 **本组题题目2**

**(2) 编程题：**创建一个基类Person和一个子类Employee。Person有一个构造器，接受一个名字参数。Employee有一个构造器，接受一个名字和一个员工ID参数。在Employee的构造器中调用Person的构造器，并讨论调用时机。

这个题跟4.2的编程题没区别

## 9. 009

## 9.1 Java静态方法和非静态方法的区别

解释Java中静态方法和非静态方法的主要区别，包括它们在调用和使用上的差异。

## 9.1.1 定义

- **静态方法**：用static关键字定义的方法。它属于类本身，而不是类的某个实例。
- **非静态方法**：没有static关键字的方法。它属于类的实例（对象），需要通过对对象来调用。

## 9.1.2 调用方式

### 静态方法的调用：

- 可以通过类名直接调用，例如：ClassName.staticMethod()
- 也可以通过对对象调用，但不推荐，因为静态方法与对象无关，例如：objectName.staticMethod()

### 非静态方法的调用：

- 必须通过对对象来调用，例如：objectName.nonStaticMethod()
- 不能通过类名直接调用。

## 9.1.3 访问权限

- **静态方法**：只能直接访问静态变量和静态方法，不能直接访问非静态变量和非静态方法。如果需要访问非静态成员，必须先创建对象。
- **非静态方法**：可以直接访问类中的所有成员，包括静态变量、静态方法、非静态变量和非静态方法。

## 9.1.4 作用域和生命周期

- **静态方法**：与类的生命周期相同，类加载时就存在，程序运行期间只会有一个静态方法的副本。
- **非静态方法**：其存在依赖于对象的生命周期，创建对象时方法才会存在，多个对象可以有各自的非静态方法副本（虽然方法本身是共享的）。

**编程题**：创建一个类MathUtils，包含一个静态方法add(int a, int b)和一个非静态方法subtract(int a, int b)。在主方法中调用这两个方法，并解释调用方式的不同。



```

1 public class MathUtils {
2
3     // 静态方法
4     public static int add(int a, int b) {
5         return a + b;
6     }
7
8     // 非静态方法
9     public int subtract(int a, int b) {
10        return a - b;
11    }
12
13    public static void main(String[] args) {
14        // 调用静态方法
15        int sum = MathUtils.add(5, 3);
16        System.out.println("Sum: " + sum); // 输出: Sum: 8
17
18        // 创建 MathUtils 的实例以调用非静态方法
19        MathUtils mathUtils = new MathUtils();
20        int difference = mathUtils.subtract(5, 3);
21        System.out.println("Difference: " + difference); // 输出:
22        Difference: 2
23    }
24}

```

Fence 129

- **静态方法调用**：静态方法 `add(int a, int b)` 是通过类名 `MathUtils` 直接调用的。因为静态方法属于类本身，而不是任何特定的对象，所以不需要创建 `MathUtils` 类的实例。
- **非静态方法调用**：非静态方法 `subtract(int a, int b)` 必须通过 `MathUtils` 类的实例来调用。在调用之前，我们需要创建一个 `MathUtils` 的对象（即 `mathUtils`）。

## 9.2 静态方法访问类成员

讨论为什么静态方法不能直接访问类的非静态成员变量和非静态方法。

- **类与实例的区别：** 静态方法属于类本身，而非静态方法和非静态成员变量属于类的实例（对象）。当你调用静态方法时，不需要创建类的实例，直接通过类名就可以调用。因此，静态方法没有对任何特定实例的引用。
- **没有this引用：** 在非静态方法中，可以使用 `this` 关键字来引用调用该方法的实例。而在静态方法中并不存在 `this`，因为静态方法并不依赖于任何特定的实例。因此，静态方法无法直接访问非静态成员变量和方法，因为它们需要通过实例来访问。

- **内存分配：** 非静态成员变量在每个实例中都有独立的存储空间，而静态成员变量只有一份存储空间，属于类本身。如果静态方法可以直接访问非静态成员变量，那么就会引发对特定实例状态的不确定性，导致静态方法可能在没有实例的情况下被调用。

**编程题：** 创建一个类Person，包含一个非静态成员变量name和一个静态方法printName(String name)。展示如何在静态方法中访问非静态成员变量。



```

1 public class Person {
2     // 非静态成员变量
3     private String name;
4
5     // 构造器
6     public Person(String name) {
7         this.name = name;
8     }
9
10    // 静态方法
11    public static void printName(String name) {
12        // 创建 Person 类的实例
13        Person person = new Person(name);
14        // 通过实例访问非静态成员变量
15        System.out.println("Person's name is: " + person.name);
16    }
17
18    public static void main(String[] args) {
19        // 调用静态方法
20        Person.printName("Alice");
21    }
22}
23

```

Fence 130

## 9.3 静态方法与构造器

解释Java中静态方法是否可以调用构造器，以及构造器是否可以调用静态方法。

- **静态方法不能直接调用构造器** 构造器是用来创建对象的，而静态方法属于类本身，而不是任何特定的对象。静态方法可以创建类的实例，并通过该实例调用构造器，但这种情况下是间接调用。
- **构造器可以调用静态方法** 由于静态方法是属于类的，因此在构造器中，可以直接调用静态方法。

**编程题**：创建一个类Product，包含一个静态方法loadConfig()和一个构造器。在构造器中调用loadConfig()方法，并在静态方法中创建Product对象。

```
● ● ●  
1 public class Product {  
2     // 构造器  
3     public Product() {  
4         // 在构造器中调用静态方法  
5         loadConfig();  
6     }  
7  
8     // 静态方法  
9     public static void loadConfig() {  
10        System.out.println("Loading product configuration...");  
11        // 可以在这里加载配置或其他相关操作  
12    }  
13  
14    // 静态方法用于创建 Product 对象  
15    public static Product createProduct() {  
16        // 在静态方法中创建 Product 对象  
17        return new Product();  
18    }  
19  
20    public static void main(String[] args) {  
21        // 调用静态方法以创建 Product 对象  
22        Product product = Product.createProduct();  
23    }  
24}  
25
```

Fence 131

## 9.4 静态方法与类变量

描述静态方法如何访问和操作类变量，并讨论它们在单例模式中的应用。

### 9.4.1 静态方法访问和操作类变量

在Java中，静态方法可以直接访问和操作类变量（静态变量）。类变量是属于类本身的，而不是属于某个特定的实例，因此静态方法可以直接通过变量名访问静态变量，而无需创建类的实例。下面是一个简单的示例：

```
● ● ●  
1 public class Counter {  
2     // 类变量（静态变量）  
3     private static int count = 0;  
4
```

```
5 // 静态方法
6 public static void increment() {
7     count++; // 操作类变量
8 }
9
10 public static int getCount() {
11     return count; // 访问类变量
12 }
13
14 public static void main(String[] args) {
15     Counter.increment();
16     System.out.println("Count: " + Counter.getCount()); // 输出
17     Count: 2
18 }
```

Fence 132

## 9.4.2 静态方法在单例模式中的应用

单例模式是一种设计模式，确保一个类只有一个实例，并提供全局访问点。静态方法在实现单例模式时非常重要，因为它们可以控制实例的创建。

以下是单例模式的一个经典实现：



```
1 public class Singleton {
2     // 静态变量，持有唯一实例
3     private static Singleton instance;
4
5     // 私有构造器，防止外部实例化
6     private Singleton() {
7         // 初始化代码
8     }
9
10    // 静态方法，提供全局访问点
11    public static Singleton getInstance() {
12        if (instance == null) {
13            instance = new Singleton(); // 创建唯一实例
14        }
15        return instance; // 返回唯一实例
16    }
17 }
```

Fence 133

**编程题**：设计一个单例类DatabaseConnection，包含一个类变量instance和一个静态方法getInstance()。确保getInstance()方法返回类的唯一实例。

```
● ● ●  
1 public class DatabaseConnection {  
2     // 类变量，用于存储唯一实例  
3     private static DatabaseConnection instance;  
4  
5     // 私有构造函数，防止外部实例化  
6     private DatabaseConnection() {  
7         // 在这里可以添加初始化代码  
8     }  
9  
10    // 静态方法，返回唯一实例  
11    public static DatabaseConnection getInstance() {  
12        if (instance == null) {  
13            instance = new DatabaseConnection();  
14        }  
15        return instance;  
16    }  
17  
18    // 示例方法  
19    public void connect() {  
20        System.out.println("Connecting to the database...");  
21    }  
22  
23    public void disconnect() {  
24        System.out.println("Disconnecting from the database...");  
25    }  
26  
27    // 测试代码  
28    public static void main(String[] args) {  
29        DatabaseConnection db1 = DatabaseConnection.getInstance();  
30        DatabaseConnection db2 = DatabaseConnection.getInstance();  
31  
32        // 验证db1和db2是同一个实例  
33        System.out.println(db1 == db2); // 输出: true  
34  
35        db1.connect();  
36        db2.disconnect();  
37    }  
38}
```

# 9.5 静态方法的继承和重写

讨论Java中静态方法是否可以被继承和重写，并解释为什么。

## 9.5.1 静态方法的继承

**可以被继承**：静态方法可以被子类继承。也就是说，子类可以使用从父类继承的静态方法。例如，如果一个父类定义了一个静态方法，子类可以直接调用这个静态方法。



```
1 class Parent {  
2     static void staticMethod() {  
3         System.out.println("Static method in Parent");  
4     }  
5 }  
6  
7 class Child extends Parent {  
8 }  
9  
10 public class Main {  
11     public static void main(String[] args) {  
12         Child.staticMethod(); // 输出: Static method in Parent  
13     }  
14 }
```

Fence 135

## 9.5.2 静态方法的重写

**不能被重写**：静态方法不能被重写。虽然子类可以定义一个与父类同名的静态方法，但这并不是重写，而是隐藏（hiding）。在这种情况下，父类和子类各自都有自己的静态方法。



```
1 class Parent {  
2     static void staticMethod() {  
3         System.out.println("Static method in Parent");  
4     }  
5 }  
6  
7 class Child extends Parent {  
8     static void staticMethod() {  
9         System.out.println("Static method in Child");  
10    }  
11 }  
12  
13 public class Main {  
14     public static void main(String[] args) {
```

```

15     Parent.staticMethod(); // 输出: Static method in Parent
16     Child.staticMethod(); // 输出: Static method in Child
17
18     Parent obj = new Child();
19     obj.staticMethod(); // 输出: Static method in Parent //如果是继
承那么输出的是child的函数
20 }
21 }
```

Fence 136

**编程题**：创建一个基类 Shape 和一个子类 Circle。Shape 类包含一个静态方法 calculateArea()。在Circle类中重写这个方法，并在主方法中调用这两个方法。



```

1 // 基类 Shape
2 class Shape {
3     // 静态方法, 用于计算面积
4     static void calculateArea() {
5         System.out.println("Calculating area in Shape class");
6     }
7     //非静态方法
8     void calculateArea1() {
9         System.out.println("Calculating area in Shape class");
10    }
11 }
12
13 // 子类 Circle
14 class Circle extends Shape {
15     // 静态方法, 隐藏父类的 calculateArea 方法
16     static void calculateArea() {
17         System.out.println("Calculating area in Circle class");
18     }
19     //非静态方法
20     void calculateArea1() {
21         System.out.println("Calculating area in Circle class");
22     }
23 }
24
25 public class Main {
26     public static void main(String[] args) {
27         // 调用基类的静态方法
28         Shape.calculateArea(); // 输出: Calculating area in Shape
29         class
30             // 调用子类的静态方法
31             Circle.calculateArea(); // 输出: Calculating area in Circle
32             class
33 }
```

```
33     // 使用父类引用指向子类
34     Shape shape = new Circle();
35     shape.calculateArea(); // 输出: Calculating area in Shape
36     shape.calculateArea1(); // 输出: Calculating area in Circle
37 }
38 }
```

Fence 137

## 易错点（记住静态方法与非静态方法在重写后的区别）

- 最后使用父类 Shape 的引用指向子类 Circle 的实例，调用 shape.calculateArea() 时，仍然调用的是 **父类的静态方法** Shape.calculateArea()，并不是子类的 Circle.calculateArea()。

# 10. 010

## 10.1 Java异常层次结构

描述Java异常层次结构的根类是什么，并解释检查型异常和非检查型异常的区别。

### 10.1.1 Java异常层次结构的根类

在Java中，异常层次结构的根类是 **Throwable**。**Throwable** 类是所有错误和异常的超类。它有两个主要的子类：

- Error**：表示严重的错误，通常是由 **JVM** 引发的，例如 **OutOfMemoryError**。这些错误通常不应被程序捕获和处理，因为它们表示系统级的问题。
- Exception**：表示程序中的异常情况，可以被程序捕获和处理。**Exception** 进一步分为两种类型：检查型异常（**Checked Exception**）和非检查型异常（**Unchecked Exception**）。

### 10.1.2 检查型异常

检查型异常是指在编译时被检查的异常。Java编译器要求开发者在代码中处理这些异常，**通常通过 try-catch 块或者在方法签名中使用 throws 声明**。这意味着，若方法可能抛出检查型异常，调用该方法的代码必须处理这些异常。常见的检查型异常有 **IOException**、**SQLException** 等。如：



```
1 public void readFile(String filePath) throws IOException {  
2     try(scanner sc=new scanner(new file(filepath))){  
3         //...  
4     }  
5     catch(IOException e){  
6         //...  
7     }  
8 }
```

Fence 138

### 10.1.3 非检查型异常

非检查型异常是指在运行时被检查的异常，编译器不会强制要求处理这些异常。  
**这类异常通常是程序逻辑错误、运行时错误，或者不合理的操作导致的**，例如 `NullPointerException`、`ArrayIndexOutOfBoundsException` 等。开发者可以选择捕获这些异常，但不需要强制处理。



```
1 public void divide(int a, int b) {  
2     int result = a / b; // 如果 b 为 0, 将抛出 ArithmeticException  
3 }
```

Fence 139

**编程题** 编写一个 Java 程序，尝试打开一个不存在的文件，并捕获并处理 `FileNotFoundException`（检查型异常）。



```
1 import java.io.File;  
2 import java.io.FileNotFoundException;  
3 import java.io.FileReader;  
4  
5 public class FileNotFoundException {  
6  
7     public static void main(String[] args) {  
8         // 指定一个不存在的文件路径  
9         String filePath = "non_existent_file.txt";  
10        File file = new File(filePath);  
11  
12        // 尝试打开文件并处理异常  
13        try {  
14            FileReader fileReader = new FileReader(file);  
15            System.out.println("文件成功打开!");  
16            // 这里可以添加文件读取的代码  
17        }
```

```
18     // 关闭文件读取器
19     fileReader.close();
20 } catch (FileNotFoundException e) {
21     System.out.println("文件未找到异常: " + e.getMessage());
22 } catch (Exception e) {
23     System.out.println("发生了其他异常: " + e.getMessage());
24 }
25 }
26 }
27 }
```

Fence 140

## 10.2 Java RuntimeException

解释Java中 **RuntimeException** 是什么，并给出几个继承自它的常见异常的例子。

**RuntimeException** 是 Java 中的一个异常类，属于检查型异常（Unchecked Exception）的一种。与检查型异常（如 **IOException**、**SQLException** 等）不同，运行时异常不需要在方法签名中声明，也不需要在代码中显式地捕获或处理。这意味着在运行时，如果发生运行时异常，程序会终止执行，而不需要在编译时处理这些异常。

### 特点

- **不需要强制捕获**：与检查型异常不同，**RuntimeException** 及其子类不需要在方法签名中声明，也不需要强制捕获。
- **通常指示程序中的错误**：运行时异常通常表示编程错误，比如逻辑错误、无效的参数等，通常是程序员可以避免的错误。
- **运行时发生**：这些异常通常在程序运行时发生，且不可预测。

### 10.2.1 常见的 RuntimeException 子类

- **NullPointerException**：当程序尝试访问或修改一个为 **null** 的对象时抛出此异常。例如，调用一个 **null** 对象的方法或访问它的属性时会抛出此异常。



```
1 String str = null;
2 int length = str.length(); // 将抛出 NullPointerException
3
```

Fence 141

- **ArrayIndexOutOfBoundsException**：当程序尝试访问数组中不存在的索引时抛出此异常。例如，访问负索引或超出数组大小的索引。



```
1 int[] arr = {1, 2, 3};  
2 int value = arr[5]; // 将抛出 ArrayIndexOutOfBoundsException  
3
```

Fence 142

- **ClassCastException**：当程序尝试将一个对象强制转换为不兼容的类型时抛出此异常。例如，将一个对象强制转换为它并不是的类型。



```
1 Object obj = new String("Hello");  
2 Integer num = (Integer) obj; // 将抛出 ClassCastException  
3
```

Fence 143

- **ArithmaticException**：当发生算术运算错误时抛出此异常，例如在整数除法中除以零。



```
1 int result = 10 / 0; // 将抛出 ArithmaticException  
2
```

Fence 144

**编程题**：编写一个方法，当传入的整数数组包含负数时，抛出 **IllegalArgumentException**，并在主方法中测试这个行为。



```
1 public class NegativeNumberExceptionExample {  
2  
3     // 方法: 检查数组是否包含负数  
4     public static void checkForNegativeNumbers(int[] numbers) {  
5         for (int number : numbers) {  
6             if (number < 0) {  
7                 throw new IllegalArgumentException("数组中包含负数: " +  
8                     number);  
9             }  
10        }  
11        System.out.println("数组中不包含负数。");  
12    }  
13  
14    public static void main(String[] args) {  
15        // 测试数组  
16        int[] validArray = {1, 2, 3, 4, 5}; // 不包含负数  
17        int[] invalidArray = {1, -2, 3, 4, 5}; // 包含负数  
18    }
```

```

18     // 测试有效数组
19     try {
20         checkForNegativeNumbers(validArray);
21     } catch (IllegalArgumentException e) {
22         System.out.println("捕获到异常: " + e.getMessage());
23     }
24
25     // 测试无效数组
26     try {
27         checkForNegativeNumbers(invalidArray);
28     } catch (IllegalArgumentException e) {
29         System.out.println("捕获到异常: " + e.getMessage());
30     }
31 }
32 }
33
34

```

Fence 145

## 10.3 Java finally块的作用

讨论Java中 **finally** 块的重要性以及它在异常处理中的作用。

### 10.3.1 Java中finally块的重要性

**确保代码执行:**

- **finally** 块中的代码无论是否发生异常，都会被执行。这使得它非常适合用于执行清理操作，例如关闭文件、释放资源、关闭数据库连接等。

**资源管理:**

- 在 Java 中，许多操作可能会占用外部资源（如文件、网络连接、数据库连接等）。使用 **finally** 块可以确保在完成操作后，无论操作是成功还是失败，都会正确释放这些资源，防止资源泄漏。

### 10.3.2 finally 块的工作机制

- **finally** 块是在 **try** 块中的代码结束后执行的。如果 **try** 块中没有抛出异常，**finally** 块会在 **try** 块正常结束后执行。
- 如果 **try** 块中抛出了异常并且被 **catch** 块捕获，**finally** 块仍然会执行。
- 如果 **try** 块中抛出了异常但没有被捕获，**finally** 块仍然会执行，然后异常将继续传播。

- 即使在 `try` 或 `catch` 中执行了 `System.exit()`，`finally` 块也可能不执行。通常情况下，`finally` 块会在正常执行或发生异常时执行，但在 `JVM` 关闭的情况下，`finally` 块可能不会执行。

**编程题：**编写一个Java程序，演示在 `try-catch` 块中打开一个资源，并在 `finally` 块中确保资源被关闭。



```
1 import java.io.FileReader;
2 import java.io.IOException;
3
4 public class Main {
5     public static void main(String[] args) {
6         FileReader fileReader = null;
7         try {
8             fileReader = new FileReader("somefile.txt");
9             // 进行文件读取操作，可能会抛出 IOException
10        } catch (IOException e) {
11            System.out.println("捕获到异常: " + e.getMessage());
12        } finally {
13            // 在这里确保文件流被关闭
14            System.out.println("这是里finally 块");
15            if (fileReader != null) {
16                try {
17                    fileReader.close();
18                    System.out.println("文件读取器已关闭。");
19                } catch (IOException e) {
20                    System.out.println("关闭文件读取器时发生异常: " +
21                        e.getMessage());
22                }
23            }
24        }
25    }
26}
```

Fence 146

## 10.4 Java自定义异常

解释何时应该创建自定义异常，并讨论自定义异常的命名约定。

## 10.4.1 何时应该创建自定义异常

- 特定业务逻辑：**当标准异常（如 `IllegalArgumentException`、`IOException` 等）不足以描述特定的业务逻辑错误时，可以考虑创建自定义异常。例如，如果在某个业务上下文中需要表示“用户未找到”或“订单已过期”，可以定义相应的自定义异常。
- 增强可读性：**自定义异常可以提高代码的可读性和可维护性。通过使用描述性名称的自定义异常，其他开发者能够更清晰地理解代码的意图。例如，使用 `UserNotFoundException` 明确表明这是与用户相关的异常。
- 异常分类：**如果需要对异常进行更细致的分类以便于处理，创建自定义异常是一种好的做法。例如，在处理不同类型的数据库错误时，可以创建多个自定义异常，如 `DatabaseConnectionException`、`DataNotFoundException` 等。
- 提供更多上下文信息：**自定义异常可以扩展默认异常，添加更多上下文信息。例如，您可以创建一个异常类，除了异常消息外，还包含错误代码、请求ID等信息，以便于调试和日志记录。
- 与外部库集成：**当与外部库或框架集成时，如果库的异常处理不符合特定需求，可以创建自定义异常，以便将外部异常转换为应用程序的特定异常类型。

## 10.4.2 自定义异常的命名约定

- 后缀使用：**自定义异常类的名称通常应该以 `Exception` 作为后缀，以便与 Java 内置异常区分开来。例如 `UserNotFoundException`、`InsufficientBalanceException`。
- 描述性命名：**自定义异常的名称应尽量描述其目的和上下文，避免模糊或不明确的名称。例如，`InvalidInputException` 比 `MyException` 更具描述性。
- 使用驼峰命名法：**遵循 Java 的命名约定，使用驼峰命名法（`CamelCase`）来命名自定义异常类。例如，`DataAccessException`。
- 避免过于通用的名称：**不要使用过于通用的名称，如 `ErrorException` 或 `CustomException`，因为这会使异常的意义不明确，导致代码可读性下降。

**编程题：**创建一个自定义异常 `InvalidAgeException`，并在处理用户年龄输入时抛出这个异常。



```
1 // 自定义异常类
2 class InvalidAgeException extends Exception {
3     public InvalidAgeException(String message) {
4         super(message);
5     }
6 }
7 }
```

```

8 // 主类
9 public class AgeValidator {
10
11     // 方法: 检查年龄
12     public static void validateAge(int age) throws InvalidAgeException
13     {
14         if (age < 0 || age > 120) { // 假设合法年龄范围是 0 到 120
15             throw new InvalidAgeException("无效年龄: " + age + ". 年龄
必须在 0 到 120 之间。");
16         }
17         System.out.println("年龄有效: " + age);
18     }
19
20     public static void main(String[] args) {
21         // 测试输入
22         int[] testAges = {25, -5, 130, 50};
23
24         for (int age : testAges) {
25             try {
26                 validateAge(age);
27             } catch (InvalidAgeException e) {
28                 System.out.println("捕获到异常: " + e.getMessage());
29             }
30         }
31     }
32
33 }
```

Fence 147

## 10.5 Java异常链

解释Java异常链的概念，并讨论它在调试异常时的好处。

### 10.5.1 java异常链的概念

- java 异常链 (Exception Chaining) 是指在一个异常的处理过程中，捕获一个异常并将其包装在另一个异常中，从而形成一个异常的“链”。这种机制允许开发者在抛出新异常时，保留原始异常的详细信息和上下文。这对于调试和排查问题非常有帮助。
- 构造函数：大多数异常类都有一个构造函数，接受一个 `Throwable` 类型的参数，这个参数通常是另一个异常对象。这允许你在新的异常中包含原始异常的信息。
- 使用 `cause` 方法：每个异常类都可以通过 `getCause()` 方法获取导致当前异常的原始异常。这使得在捕获和处理异常时，可以追踪到问题的根源。（案例可见编程题）

## 10.5.2 异常链的好处

- **保留上下文信息**：通过异常链，可以在新的异常中保留原始异常的信息，从而帮助开发者理解发生了什么错误。这样在调试时，可以更容易地追踪到问题的根源。
- **提高可读性**：将不同层次的异常信息结合在一起，能够提高代码的可读性。开发者可以从最外层的异常信息中快速获取上下文，同时还可以深入查看原始异常。
- **简化错误处理**：在复杂的应用程序中，异常链使得错误处理更加灵活。开发者可以根据不同的异常类型采取不同的处理措施，而不必在每一层都处理原始异常。
- **调试时的便利**：在调试时，异常链可以帮助开发者快速找到导致问题的具体原因。原始异常的堆栈跟踪信息可以提供有关错误发生位置的详细信息，使调试过程更加高效。
- **维护性**：使用异常链可以使代码更易于维护。当代码逻辑发生变化时，原始异常的信息仍然可以被保留，从而减少了对现有异常处理代码的修改。

**编程题：**编写一个方法，该方法接受一个文件路径作为参数，并在尝试打开文件时捕获 `IOException`，同时将原始异常作为新异常的 `cause` 抛出



```
1 import java.io.File;
2 import java.io.FileReader;
3 import java.io.IOException;
4
5 // 自定义异常类
6 class FileProcessingException extends Exception {
7     public FileProcessingException(String message, Throwable cause) {
8         super(message, cause);
9     }
10 }
11
12 public class FileHandler {
13
14     // 方法：尝试打开文件并处理异常
15     public static void openFile(String filePath) throws
16         FileProcessingException {
17         File file = new File(filePath);
18         FileReader fileReader = null;
19
20         try {
21             fileReader = new FileReader(file);
22             // 这里可以添加文件读取的逻辑
23             System.out.println("文件成功打开: " + filePath);
24         } catch (IOException e) {
25             // 将 IOException 包装成自定义异常，并将原始异常作为 cause
26             throw new FileProcessingException("无法打开文件: " +
27                 filePath, e);
28         }
29     }
30 }
```

```
26     } finally {
27         // 关闭 FileReader
28         if (fileReader != null) {
29             try {
30                 fileReader.close();
31             } catch (IOException e) {
32                 // 处理关闭文件时的异常
33                 System.out.println("关闭文件时发生异常: " +
34                     e.getMessage());
35             }
36         }
37     }
38
39     public static void main(String[] args) {
40         String filePath = "non_existent_file.txt"; // 指定一个不存在的文
件路径
41
42         try {
43             openFile(filePath);
44         } catch (FileProcessingException e) {
45             // 捕获自定义异常并打印相关信息
46             System.out.println("捕获到异常: " + e.getMessage());
47             // 打印原始异常信息
48             System.out.println("原始异常: " + e.getCause());
49         }
50     }
51 }
52
53
```

Fence 148

可以看到以下输出:



- 1 | 捕获到异常: 无法打开文件: non\_existent\_file.txt
- 2 | 原始异常: java.io.FileNotFoundException: non\_existent\_file.txt (系统找不到指定的文件。)
- 3 |

Fence 149

## 11.1 方法重写与方法签名

阐述在Java中方法重写需要满足哪些条件，并解释方法签名的含义。

### 11.1.1 方法重写

在Java中，方法重写（Overriding）是指子类重新定义父类中已经存在的方法，以提供特定的实现。要成功重写一个方法，需要满足以下条件：

- **方法名称相同**：子类的方法名称必须与父类被重写的方法名称完全一致。
- **参数列表相同**：子类的方法参数类型、参数数量和参数顺序必须与父类的方法完全一致。这一点非常重要，因为Java使用方法签名来确定方法的唯一性。
- **返回类型兼容**：子类重写的方法的返回类型必须与父类的方法的返回类型相同，或者是其返回类型的子类（即实现协变返回类型）。
- **访问修饰符**：子类重写的方法的访问修饰符不能比父类的方法更严格。例如，如果父类的方法是public，那么子类的方法不能是private，可以是public或protected。
- **异常抛出**：子类重写的方法不能抛出比父类方法更多的检查异常（checked exceptions）。也就是说，如果父类方法抛出了某个检查异常，子类可以选择不抛出该异常，或者抛出相同的异常或其子类，但不能抛出父类方法未抛出的其他检查异常。

### 11.1.2 方法签名的含义

在Java中，方法签名是指方法的名称和参数类型的组合。具体来说，方法签名包括以下内容：

- 方法名称
- 参数的数量和类型（不包括返回类型）

方法签名是确定方法唯一性的关键因素。在重载（Overloading）中，方法签名的不同使得同一个类中可以定义多个同名的方法，而在重写中，方法签名的相同使得子类的方法可以替代父类的方法。

例如，以下两个方法在同一个类中是重载的，因为它们的参数列表不同：



```
1 public void exampleMethod(int a) { ... }      // 只有一个 int 参数
2 public void exampleMethod(double a) { ... }    // 只有一个 double 参数
3
```

Fence 150

而以下两个方法在不同的类中是重写的，因为它们的方法签名相同：



```
1 class Parent {
2     public void exampleMethod(int a) { ... }
3 }
4
5 class Child extends Parent {
6     @Override
7     public void exampleMethod(int a) { ... } // 重写了父类的方法
8 }
9
```

Fence 151

**编程题** 定义一个基类 `Animal` 和一个子类 `Dog`。在 `Animal` 类中定义一个方法 `makeSound()`，然后在 `Dog` 类中重写这个方法。



```
1 // 定义基类 Animal
2 class Animal {
3     // 定义一个方法 makeSound()
4     public String makeSound() {
5         return "Some generic animal sound";
6     }
7 }
8
9 // 定义子类 Dog，继承自 Animal
10 class Dog extends Animal {
11     // 重写 makeSound() 方法
12     @Override
13     public String makeSound() {
14         return "Bark";
15     }
16 }
17
18 // 主类
19 public class Main {
20     public static void main(String[] args) {
21         // 创建 Animal 对象
22         Animal animal = new Animal();
```

```

23     System.out.println(animal.makeSound()); // 输出: Some generic
24     animal sound
25
26     // 创建 Dog 对象
27     Dog dog = new Dog();
28     System.out.println(dog.makeSound()); // 输出: Bark
29 }
30

```

Fence 152

## 11.2 重写与访问控制

讨论在子类中重写父类方法时，如何通过访问控制符影响方法的可见性。

### 11.2.1 Java中常用的访问控制符：

- **public**：该方法对所有类可见，包括不同包中的类。
- **protected**：该方法只能被同一包中的类以及所有子类访问，不论子类是否在同一包中。
- **default**（包私有，未指定访问修饰符）：该方法只能被同一包中的类访问。
- **private**：该方法只能在定义它的类内部访问，不能被子类重写。

### 11.2.2 影响方法可见性

#### 重写时的可见性要求：

- 当子类重写父类的方法时，子类的方法的访问级别必须与父类的方法相同或更宽松。例如：
- 如果父类的方法是 **public**，子类可以将其重写为 **public** 或 **protected**，但不能将其重写为 **private**。
- 如果父类的方法是 **protected**，子类可以将其重写为 **protected** 或 **public**，但不能将其重写为 **private** 或默认（包私有）。
- 如果父类的方法是默认（包私有），子类可以将其重写为默认或 **protected**，但不能将其重写为 **public** 或 **private**。

#### 影响父类方法访问：

- 如果父类的方法是 **private**，则子类 **无法重写** 该方法，因为它对子类是不可见的。这意味着子类不能改变父类的实现。
- 使用 **protected** 或 **public** 访问修饰符 **可以使** 方法在子类中可见，并允许子类重写它们。

**编程题：**创建一个基类 **Vehicle**，其中包含一个 **startEngine()** 方法。然后创建一个子类 **Car**，重写 **startEngine()** 方法，并尝试降低访问级别。



```

1 // 基类 Vehicle
2 class Vehicle {
3     // public 方法，允许任何类访问
4     public void startEngine() {
5         System.out.println("Vehicle engine started");
6     }
7 }
8
9 // 子类 Car
10 class Car extends Vehicle {
11     // 尝试降低访问级别会导致编译错误
12     /*
13     @Override
14     private void startEngine() { // 编译错误: Cannot reduce the
visibility of the inherited method from Vehicle
15         System.out.println("Car engine started");
16     }
17     */
18
19     // 正确的重写方法，保持访问级别
20     @Override
21     public void startEngine() {
22         System.out.println("Car engine started");
23     }
24 }
25
26 // 主类
27 public class Main {
28     public static void main(String[] args) {
29         Vehicle myVehicle = new Vehicle();
30         myVehicle.startEngine(); // 输出: Vehicle engine started
31
32         Car myCar = new Car();
33         myCar.startEngine(); // 输出: Car engine started
34     }
35 }
36

```

在Java中，重写方法时不能降低访问级别，必须保持父类方法的可见性或提高可见性。尝试降低访问级别会导致编译错误。

## 11.3 重写与协变返回类型

解释Java中的协变返回类型是什么，并讨论它与方法重写的关系。

**协变返回类型**（Covariant Return Type）是Java中的一个特性，它允许重写的方法返回一个更具体的子类类型，而不仅仅是父类类型。这种特性使得在方法重写时，返回类型可以是父类返回类型的子类。

### 11.3.1 协变返回类型的特性

- 返回类型的灵活性：在子类中重写父类的方法时，返回类型可以是父类方法的返回类型的子类。这样，子类可以提供更具体的返回值，而不必始终返回父类类型。
- 增强可读性和可用性：通过使用协变返回类型，代码可以更清晰，因为它允许方法提供更具体的信息。例如，子类可以返回它自己的实例，而不是父类实例，从而使调用者能够直接使用子类特有的方法。



```
1 // 基类 Animal
2 class Animal {
3     public Animal makeSound() {
4         System.out.println("Some generic animal sound");
5         return new Animal();
6     }
7 }
8
9 // 子类 Dog
10 class Dog extends Animal {
11     @Override
12     public Dog makeSound() { // 返回类型为 Dog，子类类型
13         System.out.println("Bark");
14         return new Dog();
15     }
16 }
17
18 // 主类
19 public class Main {
20     public static void main(String[] args) {
21         Animal myAnimal = new Animal();
22         myAnimal.makeSound(); // 输出: Some generic animal sound
```

```

23
24     Dog myDog = new Dog();
25     myDog.makeSound(); // 输出: Bark
26
27     // 使用协变返回类型
28     Animal dogAsAnimal = myDog.makeSound(); // 返回类型为 Dog, 但可
以被赋值给 Animal
29 }
30 }
31
32

```

Fence 154

### 11.3.2 与方法重写的关系

- 方法重写：当子类提供了父类方法的具体实现时，称为方法重写。协变返回类型是方法重写的一种扩展，允许重写方法的返回类型更具体。
- 类型安全：协变返回类型在编译时保持类型安全，因为Java会检查返回类型是否是父类类型的子类。这意味着调用者可以安全地将返回的对象视为子类类型。
- 增强继承结构的灵活性：协变返回类型使得继承和多态的使用更加灵活，允许开发者设计更具可扩展性和可维护性的类结构。

**编程题：**定义一个基类 `Shape` 和一个子类 `Circle`。`Shape` 类包含一个返回 `Shape` 类型的方法 `clone()`。在 `Circle` 类中重写这个方法，使其返回 `Circle` 类型。



```

1 // 基类 Shape
2 class Shape {
3     // clone 方法返回 Shape 类型
4     public Shape clone() {
5         System.out.println("Cloning a Shape");
6         return new Shape();
7     }
8 }
9
10 // 子类 Circle
11 class Circle extends Shape {
12     // 重写 clone 方法，返回 Circle 类型
13     @Override
14     public Circle clone() {
15         System.out.println("Cloning a Circle");
16         return new Circle();
17     }
18 }
19
20 // 主类

```

```
21 public class Main {  
22     public static void main(String[] args) {  
23         Shape myShape = new Shape();  
24         Shape clonedShape = myShape.clone(); // 输出: Cloning a Shape  
25  
26         Circle myCircle = new Circle();  
27         Circle clonedCircle = myCircle.clone(); // 输出: Cloning a  
28         Circle  
29         // 可以将 Circle 类型的实例赋值给 Shape 类型的变量  
30         Shape anotherShape = myCircle.clone(); // 仍然可以赋值为 Shape  
31         类型  
32     }  
33  
34 }
```

Fence 155

## 11.4 重写与静态方法

讨论为什么静态方法不能被重写，并解释它们在继承中的行为。

### 11.4.1 静态方法不能被重写的原因

静态方法在Java中是与类本身而不是与类的实例关联的。这意味着静态方法是属于类的，而不是属于某个对象。由于这一特性，静态方法的行为和实例方法有很大的不同。以下是静态方法不能被重写的主要原因：

- **静态绑定**：静态方法是在编译时进行绑定的。这意味着在编译时，Java编译器会根据引用类型（即声明类型）来决定调用哪个静态方法，而不是根据对象的实际类型。这与实例方法不同，后者是在运行时进行动态绑定。
- **类级别的性质**：静态方法是与类本身相关联的，不依赖于任何对象的状态。这使得静态方法不能被子类重写，因为重写涉及到多态性，而多态性是基于对象的实际类型（即对象的实例）而不是类的类型。
- **隐藏**：当子类定义一个与父类同名的静态方法时，这实际上是方法隐藏，而不是重写。子类的方法并不覆盖父类的静态方法，而是隐藏了父类的方法。这意味着如果你通过父类引用调用静态方法，调用的是父类的方法；如果通过子类引用调用，则调用的是子类的方法。

## 11.4.2 静态方法在继承中的行为

- **不可重写**：正如前面所述，静态方法不能被重写。子类可以定义一个与父类静态方法同名的方法，但这只是隐藏了父类的静态方法，而不是重写。
- **调用方式**：静态方法可以通过类名直接调用，而不需要创建类的实例。例如，`ClassName.staticMethod()`。这与实例方法不同，实例方法需要通过对象来调用。
- **访问限制**：如果父类的静态方法是 `private`，则子类无法访问和调用这个方法，即使子类定义了一个同名的静态方法。

● ● ●

```
1 class Parent {  
2     static void staticMethod() {  
3         System.out.println("Static method in Parent");  
4     }  
5  
6     void instanceMethod() {  
7         System.out.println("Instance method in Parent");  
8     }  
9 }  
10  
11 class Child extends Parent {  
12     static void staticMethod() {  
13         System.out.println("Static method in Child");  
14     }  
15  
16     @Override  
17     void instanceMethod() {  
18         System.out.println("Instance method in Child");  
19     }  
20 }  
21  
22 public class Main {  
23     public static void main(String[] args) {  
24         Parent parent = new Parent();  
25         parent.staticMethod(); // 输出: Static method in Parent  
26         parent.instanceMethod(); // 输出: Instance method in Parent  
27  
28         Child child = new Child();  
29         child.staticMethod(); // 输出: Static method in Child  
30         child.instanceMethod(); // 输出: Instance method in Child  
31  
32         Parent parentRefToChild = new Child();  
33         parentRefToChild.staticMethod(); // 输出: Static method in  
Parent (静态绑定)  
34             parentRefToChild.instanceMethod(); // 输出: Instance method in  
Child (动态绑定)  
35     }
```

```
36 }  
37  
38  
39
```

Fence 156

## 11.5 重写与构造方法

### 11.5.1 为什么构造方法不能被重写

构造方法在Java中用于初始化新创建的对象。与普通方法不同，构造方法具有以下特性，使得它们不能被重写：

- **构造方法没有返回类型**：构造方法不需要声明返回类型，包括 `void`，而且它的名称必须与类名相同。这使得构造方法本身不被视为一种可重写的方法，因为没有返回值来进行重写的关联。
- **构造方法是专门用于对象初始化的**：构造方法用于创建类的实例，并且在实例被创建时被自动调用。它的主要目的是初始化对象的状态，而不是执行其他逻辑。因此，构造方法的行为与普通方法不同。
- **构造方法在类的实例化过程中被调用**：当一个对象被创建时，构造方法会被调用来分配内存和初始化对象的属性。由于构造方法是与对象的实例化过程紧密相关的，所以它不支持重写。

### 11.5.2 子类构造方法与父类构造方法的关系

#### 调用父类构造方法：

- 在创建子类对象时，子类的构造方法会隐式或显式地调用父类的构造方法。Java会首先执行父类的构造方法，以确保父类的部分被正确初始化。默认情况下，如果没有显式调用，Java会调用父类的无参构造方法。
- 如果父类没有无参构造方法，则子类必须显式调用父类的某个构造方法，使用 `super()` 语句。

#### 构造方法链：

- 在子类构造方法中，可以使用 `super()` 关键字来调用父类的构造方法。这样可以确保在子类构造之前，父类的状态得到正确初始化。  
这种构造方法链确保了对象的完整性，从父类到子类的初始化顺序。

#### 访问控制：

- 子类构造方法可以访问父类的公有和受保护的构造方法，但不能访问父类的私有构造方法。

**编程题**：创建一个基类 Person 和一个子类 Student。Person 类有一个接受名字的构造方法。在 Student 类中提供一个构造方法，它除了接受名字外还接受一个学号，并在 Student 的构造方法中调用 Person 的构造方法。



```
1 // 基类 Person
2 class Person {
3     private String name;
4
5     // 接受名字的构造方法
6     public Person(String name) {
7         this.name = name;
8         System.out.println("Person constructor called: " + name);
9     }
10
11    // 获取名字的方法
12    public String getName() {
13        return name;
14    }
15 }
16
17 // 子类 Student
18 class Student extends Person {
19     private String studentId;
20
21     // 接受名字和学号的构造方法
22     public Student(String name, String studentId) {
23         super(name); // 调用父类的构造方法
24         this.studentId = studentId;
25         System.out.println("Student constructor called: " + name + ", "
26             + "Student ID: " + studentId);
27     }
28
29     // 获取学号的方法
30     public String getStudentId() {
31         return studentId;
32     }
33 }
34
35 // 主类
36 public class Main {
37     public static void main(String[] args) {
38         // 创建 Student 对象
39         Student student = new Student("Alice", "S12345");
40     }
41 }
```

```
40     // 输出学生的名字和学号
41     System.out.println("Student Name: " + student.getName());
42     System.out.println("Student ID: " + student.getStudentId());
43 }
44 }
45
46
```

Fence 157

## 12. 012

### 12.1 基本类型与包装类的转换

解释Java中基本数据类型和它们的包装类之间如何进行自动装箱和拆箱。

在Java中，基本数据类型（primitive types）和它们的包装类（wrapper classes）之间的转换称为自动装箱（autoboxing）和拆箱（unboxing）。这两个过程使得基本数据类型和对象之间的转换变得更加简单和直观。

#### 12.1.1 基本数据类型和包装类

Java中的基本数据类型包括：

- `int`：整数类型
- `double`：双精度浮点类型
- `float`：单精度浮点类型
- `char`：字符类型
- `boolean`：布尔类型
- `byte`：字节类型
- `short`：短整型
- `long`：长整型

对应的包装类如下：

- `Integer`：包装类，表示 int
- `Double`：包装类，表示 double
- `Float`：包装类，表示 float
- `Character`：包装类，表示 char
- `Boolean`：包装类，表示 boolean

- **Byte** : 包装类, 表示 byte
- **Short** : 包装类, 表示 short
- **Long** : 包装类, 表示 long

## 12.1.2 自动装箱 (Autoboxing)

自动装箱是指将基本数据类型转换为对应的包装类的过程。在某些上下文中（如将基本数据类型赋值给一个对象类型），Java会自动完成这种转换。

示例：



```
1 int primitiveInt = 5;
2 Integer wrappedInt = primitiveInt; // 自动装箱
3 System.out.println(wrappedInt); // 输出: 5
4
```

Fence 158

在这个例子中，`int` 类型的 `primitiveInt` 被自动转换为 `Integer` 类型的 `wrappedInt`。

## 12.1.3 拆箱 (Unboxing)

拆箱是指将包装类转换为对应的基本数据类型的过程。当需要使用基本数据类型时，Java会自动完成这种转换。

示例：



```
1 Integer wrappedInt = 10; // 自动装箱
2 int primitiveInt = wrappedInt; // 拆箱
3 System.out.println(primitiveInt); // 输出: 10
4
5
```

Fence 159

在这个例子中，`Integer` 类型的 `wrappedInt` 被自动转换为 `int` 类型的 `primitiveInt`。

**编程题** 编写一个Java程序，将 `Integer` 对象转换为 `int` 基本类型，并演示自动拆箱的过程。



```
1 public class AutoUnboxingExample {  
2     public static void main(String[] args) {  
3         // 创建一个 Integer 对象  
4         Integer wrappedInteger = 42; // 自动装箱  
5  
6         // 自动拆箱, 将 Integer 转换为 int  
7         int primitiveInt = wrappedInteger; // 自动拆箱  
8  
9         // 输出结果  
10        System.out.println("Wrapped Integer: " + wrappedInteger); //  
输出: Wrapped Integer: 42  
11        System.out.println("Primitive int: " + primitiveInt); // 输出:  
Primitive int: 42  
12  
13        // 进一步演示自动拆箱的使用  
14        Integer anotherWrappedInteger = 100; // 自动装箱  
15        int result = wrappedInteger + anotherWrappedInteger; // 拆箱后  
进行加法  
16  
17        // 输出加法结果  
18        System.out.println("Result of addition: " + result); // 输出:  
Result of addition: 142  
19    }  
20 }  
21  
22
```

Fence 160

## 12.2 对象的向上转型和向下转型

讨论Java中对象的向上转型和向下转型的区别，并解释何时需要强制类型转换。

### 12.2.1 向上转型 (Upcasting)

#### 12.2.1.1 定义：

- 向上转型是指将子类对象引用赋值给父类变量。这种转型是安全的，因为子类对象是父类对象的一种特化形式。

## 12.2.1.2 特点：

- 向上转型是隐式转换，不需要强制类型转换。
- 通常用于多态性，允许通过父类引用调用子类实现的方法。
- 在向上转型后，**无法访问子类特有的方法和属性**。

示例：

```
● ● ●
1 class Animal {
2     void makeSound() {
3         System.out.println("Animal sound");
4     }
5 }
6
7 class Dog extends Animal {
8     void makeSound() {
9         System.out.println("Bark");
10    }
11
12    void fetch() {
13        System.out.println("Dog fetching");
14    }
15 }
16
17 public class UpcastingExample {
18     public static void main(String[] args) {
19         Animal animal = new Dog(); // 向上转型
20         animal.makeSound(); // 输出: Bark
21
22         // animal.fetch(); // 编译错误，无法访问 Dog 类的 fetch 方法
23     }
24 }
25
26 }
```

Fence 161

## 12.2.2 向下转型 (Downcasting)

### 12.2.2.1 定义：

- 向下转型是将父类对象引用赋值给子类变量。这种转型需要显式地进行，因为它可能会导致运行时错误。

## 12.2.2.2 特点：

- 向下转型是显式转换，需要使用强制类型转换。
- 在进行向下转型时，**必须确保对象的实际类型是子类类型**，否则会抛出 ClassCastException。
- 可以访问子类特有的方法和属性。

示例：

```
● ● ●
1 public class DowncastingExample {
2     public static void main(String[] args) {
3         Animal animal = new Dog(); // 向上转型
4         Dog dog = (Dog) animal; // 向下转型，强制类型转换
5         dog.fetch(); // 输出: Dog fetching
6
7         // 如果尝试将一个 Animal 对象转型为 Dog 对象，会抛出异常
8         Animal anotherAnimal = new Animal();
9         // Dog anotherDog = (Dog) anotherAnimal; // 编译器不会报错，但会在运行时抛出 ClassCastException
10    }
11 }
12
13 }
```

Fence 162

## 12.2.3 何时需要强制类型转换

访问子类特有方法：

- 当需要访问子类中特有的方法或属性时，需要进行向下转型。例如，当你有一个父类的引用指向一个子类对象，并且想要调用子类的方法时，必须进行向下转型。

多态行为的实现：

- 在某些情况下，可能需要根据对象的具体类型执行不同的操作。可以通过 instanceof 关键字检查对象类型，然后进行安全的向下转型。

**编程题**：创建一个基类 Animal 和两个子类 Dog 和 Cat。编写代码演示如何将 Animal 类型的对象向下转型为 Dog 类型。

```
● ● ●
1 // 基类 Animal
2 class Animal {
3     void makeSound() {
```

```
4         System.out.println("Animal sound");
5     }
6 }
7
8 // 子类 Dog
9 class Dog extends Animal {
10     @Override
11     void makeSound() {
12         System.out.println("Bark");
13     }
14
15     void fetch() {
16         System.out.println("Dog fetching the ball");
17     }
18 }
19
20 // 子类 Cat
21 class Cat extends Animal {
22     @Override
23     void makeSound() {
24         System.out.println("Meow");
25     }
26
27     void scratch() {
28         System.out.println("Cat scratching the furniture");
29     }
30 }
31
32 // 主类
33 public class DowncastingExample {
34     public static void main(String[] args) {
35         // 创建一个 Animal 类型的对象, 实际是 Dog 类型的实例
36         Animal animal = new Dog(); // 向上转型
37         animal.makeSound(); // 输出: Bark
38
39         // 向下转型, 安全地将 Animal 引用转换为 Dog 类型
40         if (animal instanceof Dog) { // 检查类型
41             Dog dog = (Dog) animal; // 向下转型
42             dog.fetch(); // 调用 Dog 类特有的方法
43         } else {
44             System.out.println("The animal is not a Dog.");
45         }
46
47         // 尝试向下转型为 Cat 类型(这是不安全的示例)
48         try {
49             Cat cat = (Cat) animal; // 这里会抛出 ClassCastException
50             cat.scratch(); // 不会执行, 因为会抛出异常
51         } catch (ClassCastException e) {
52             System.out.println("Cannot cast Animal to Cat: " +
e.getMessage());
53         }
54 }
```

```
54 }  
55 }  
56  
57  
58
```

Fence 163

## 12.3 多态与类型转换

解释在多态的情况下，为什么可能需要进行强制类型转换，并讨论其安全性。

### 12.3.1 强制类型转换的必要性

- 访问子类特有的方法：

当你使用父类引用（如 `Animal`）来指向一个子类对象（如 `Dog`），你只能访问父类中定义的方法。如果你想调用子类中特有的方法（如 `fetch()`），就需要将父类引用强制转换为子类类型。



```
1 Animal animal = new Dog();  
2 animal.makeSound(); // 调用父类方法  
3  
4 // 需要强制类型转换来调用 Dog 类特有的方法  
5 Dog dog = (Dog) animal; // 强制转换  
6 dog.fetch(); // 调用 Dog 类特有的方法  
7
```

Fence 164

- 多态行为的实现：

在一些情况下，基于对象的具体类型执行特定的操作是有必要的。通过强制类型转换，可以根据实例的真实类型进行不同的处理。



```
1 if (animal instanceof Dog) {  
2     Dog dog = (Dog) animal;  
3     dog.fetch();  
4 }  
5
```

Fence 165

## 12.3.2 安全性问题

强制类型转换虽然在某些情况下是必要的，但同时也带来了安全性问题。主要包括：

- **ClassCastException** :

如果你试图将一个对象向下转型为一个不兼容的类型，程序将抛出 **ClassCastException**。例如，如果你有一个 **Animal** 引用指向一个 **Cat** 对象，但你尝试将它转换为 **Dog** 类型，就会导致异常。



```
1 Animal animal = new Cat();
2 Dog dog = (Dog) animal; // 抛出 ClassCastException
3
```

Fence 166

- 使用 **instanceof** 检查 :

为了确保安全性，通常在进行强制类型转换之前，使用 **instanceof** 关键字检查对象的实际类型。这可以避免在运行时发生 **ClassCastException**。



```
1 if (animal instanceof Dog) {
2     Dog dog = (Dog) animal; // 安全的强制类型转换
3     dog.fetch();
4 } else {
5     System.out.println("The animal is not a Dog.");
6 }
7
8
```

Fence 167

**编程题**：创建一个方法，接受 **Animal** 类型的参数并调用一个 **makeSound** 方法。在子类中重写这个方法，并在主方法中演示如何调用子类的特定实现。



```
1 // 基类 Animal
2 class Animal {
3     void makeSound() {
4         System.out.println("Animal sound");
5     }
6 }
7
8 // 子类 Dog
9 class Dog extends Animal {
10    @Override
```

```

11 void makeSound() {
12     System.out.println("Bark");
13 }
14 }
15
16 // 子类 Cat
17 class Cat extends Animal {
18     @Override
19     void makeSound() {
20         System.out.println("Meow");
21     }
22 }
23
24 // 主类
25 public class AnimalSoundDemo {
26     // 接受 Animal 类型参数的方法
27     public static void showAnimalSound(Animal animal) {
28         animal.makeSound(); // 调用 makeSound 方法
29     }
30
31     public static void main(String[] args) {
32         Animal dog = new Dog(); // 向上转型
33         Animal cat = new Cat(); // 向上转型
34
35         // 调用方法，展示子类特定实现
36         showAnimalSound(dog); // 输出: Bark
37         showAnimalSound(cat); // 输出: Meow
38     }
39 }
40
41

```

Fence 168

## 12.4 数组与泛型的类型转换

讨论Java中数组的协变返回类型，并解释为什么数组不能进行泛型的协变。

### 12.4.1 数组的协变返回类型

在Java中，数组是协变的，这意味着一个数组的子类型可以被赋值给父类型的数组变量。例如，`Dog` 是 `Animal` 的子类，因此 `Dog[]` 是 `Animal[]` 的子类。这使得我们可以创建一个 `Dog` 类型的数组，并将其赋值给一个 `Animal` 类型的数组变量。

示例：



```

1 class Animal {
2     void makeSound() {
3         System.out.println("Some animal sound");
4     }
5 }
6
7 class Dog extends Animal {
8     @Override
9     void makeSound() {
10        System.out.println("Bark");
11    }
12 }
13
14 public class ArrayCovarianceExample {
15     public static void main(String[] args) {
16         Dog[] dogs = new Dog[2]; // 创建 Dog 类型的数组
17         dogs[0] = new Dog();
18         dogs[1] = new Dog();
19
20         Animal[] animals = dogs; // 协变, Dog[] 转换为 Animal[]
21         animals[0].makeSound(); // 输出: Bark
22     }
23 }
24
25

```

Fence 169

在这个例子中，Dog[] 被赋值给 Animal[]，这是一种协变行为。当我们调用 makeSound() 方法时，仍然会执行 Dog 类中重写的实现。

## 12.4.2 为什么数组不能进行泛型的协变

尽管数组支持协变，但Java中的泛型不支持协变。这是由于以下原因：

- 类型擦除**：Java中的泛型使用类型擦除机制来实现。在编译时，泛型类型信息被擦除，转换为原始类型（raw types）。这意味着在运行时，泛型信息不可用，导致类型安全问题。
- 类型不安全**：如果允许泛型的协变，可能会导致类型不安全的问题。例如，如果 List<Dog> 可以被赋值给 List<Animal>，那么我们可以在 List<Animal> 中添加 Animal 类型的对象，从而破坏 List<Dog> 的类型安全性。



```
1 List<Dog> dogs = new ArrayList<Dog>();
2 List<Animal> animals = dogs; // 编译时允许, 但不安全
3 animals.add(new Animal()); // 添加一个 Animal 对象
4
5 Dog dog = dogs.get(0); // 运行时错误, ClassCastException
6
7
```

Fence 170

**编程题**：编写一个泛型方法，返回一个泛型类型的数组，并尝试将返回值转换为特定类型的数组。



```
1 import java.lang.reflect.Array;
2
3 public class GenericArrayExample {
4
5     // 泛型方法, 创建并返回一个泛型类型的数组
6     public static <T> T[] createArray(Class<T> clazz, int size) {
7         // 使用反射创建数组
8         T[] array = (T[]) Array.newInstance(clazz, size);
9         return array;
10    }
11
12    public static void main(String[] args) {
13        // 创建一个 Integer 类型的数组
14        Integer[] intArray = createArray(Integer.class, 5);
15        for (int i = 0; i < intArray.length; i++) {
16            intArray[i] = i + 1; // 填充数组
17        }
18
19        System.out.println("Integer Array:");
20        for (Integer num : intArray) {
21            System.out.println(num); // 输出: 1, 2, 3, 4, 5
22        }
23
24        // 创建一个 String 类型的数组
25        String[] strArray = createArray(String.class, 3);
26        strArray[0] = "Hello";
27        strArray[1] = "Generic";
28        strArray[2] = "Array";
29
30        System.out.println("\nString Array:");
31        for (String str : strArray) {
32            System.out.println(str); // 输出: Hello, Generic, Array
33        }
34}
```

```
34    }
35 }
36
37
38
```

Fence 171

## 12.5 强制类型转换与异常

解释在Java中进行强制类型转换时可能抛出的异常，并讨论如何安全地进行类型转换。

在Java中，进行强制类型转换时可能会抛出 **ClassCastException**。这种异常发生在尝试将一个对象转换为不兼容的类型时。

### 12.5.1 ClassCastException 异常

- **定义**：**ClassCastException** 是一种运行时异常，表示在强制类型转换时，目标类型与实际类型不兼容。例如，如果你有一个 **Animal** 类的对象，但试图将其转换为 **Dog** 类型，而该对象实际上是 **Cat** 类型的实例，Java 将抛出 **ClassCastException**。



```
1 class Animal {}
2 class Dog extends Animal {}
3 class Cat extends Animal {}

4

5 public class Main {
6     public static void main(String[] args) {
7         Animal animal = new Cat(); // animal 是 Cat 类型的对象
8
9         // 尝试将 animal 转换为 Dog 类型
10        Dog dog = (Dog) animal; // 抛出 ClassCastException
11    }
12}
13}
```

Fence 172

## 12.5.2 如何安全地进行类型转换

### 12.5.2.1 使用 instanceof 检查

在进行强制类型转换之前，使用 `instanceof` 关键字检查对象的实际类型。这可以确保在转换之前确认对象的类型，从而避免抛出异常。



```
1 public class Main {  
2     public static void main(String[] args) {  
3         Animal animal = new Cat(); // animal 是 Cat 类型的对象  
4  
5         // 使用 instanceof 检查类型  
6         if (animal instanceof Dog) {  
7             Dog dog = (Dog) animal; // 安全的转换  
8             // 进行 Dog 类型特有的操作  
9         } else {  
10             System.out.println("animal is not a Dog.");  
11         }  
12     }  
13 }  
14  
15
```

Fence 173

### 12.5.2.2 使用多态

在设计类时，可以利用多态来避免强制类型转换。通过父类引用调用子类重写的方法，这样可以避免直接进行类型转换。



```
1 class Animal {  
2     void makeSound() {  
3         System.out.println("Animal sound");  
4     }  
5 }  
6  
7 class Dog extends Animal {  
8     @Override  
9     void makeSound() {  
10         System.out.println("Bark");  
11     }  
12 }  
13  
14 public class Main {  
15     public static void main(String[] args) {  
16         Animal animal = new Dog(); // 向上转型  
17         animal.makeSound(); // 调用 Dog 的 makeSound 方法，输出: Bark
```

```
18    }
19 }
20
21 }
```

Fence 174

**编程题：**编写一个程序，尝试将一个 `Object` 类型的变量强制类型转换为 `String`，并处理可能出现的 `ClassCastException`。



```
1 public class ClassCastExceptionExample {
2     public static void main(String[] args) {
3         // 创建一个 Object 类型的变量，并赋值为一个 String
4         Object obj = "Hello, World!"; // 这是一个合法的 String 对象
5
6         // 尝试将 Object 转换为 String
7         try {
8             String str = (String) obj; // 强制类型转换
9             System.out.println("Converted String: " + str);
10        } catch (ClassCastException e) {
11            System.out.println("ClassCastException: Cannot cast to
String - " + e.getMessage());
12        }
13
14        // 创建另一个 Object 类型的变量，并赋值为一个 Integer
15        obj = 42; // 这是一个 Integer 对象
16
17        // 尝试将 Object 转换为 String
18        try {
19            String str = (String) obj; // 强制类型转换
20            System.out.println("Converted String: " + str);
21        } catch (ClassCastException e) {
22            System.out.println("ClassCastException: Cannot cast to
String - " + e.getMessage());
23        }
24
25        // 使用 instanceof 检查类型并安全转换
26        if (obj instanceof String) {
27            String safeStr = (String) obj; // 安全的强制转换
28            System.out.println("Safely converted String: " + safeStr);
29        } else {
30            System.out.println("The object is not a String.");
31        }
32    }
33 }
34 }
```

输出结果为：



```
1 Converted String: Hello, World!
2 ClassCastException: Cannot cast to String - Cannot cast
   java.lang.Integer to java.lang.String
3 The object is not a String.
4
```

## 13. 013

本节前置知识：需要了解启动线程的方法。

### 13.1 题目1：线程间通信

#### 13.1.1 解释Java中线程间如何进行通信。

在 Java 中，线程间通信通常依赖于共享资源和同步机制。可以通过暂停一个线程，让其他的线程完成共享数据的处理以后再激活它实现线程间的同步。

#### 13.1.2 synchronized介绍

- **synchronized** 是 Java 中的一种关键字，用于实现线程同步。它可以确保多个线程在同一时刻只能有一个线程访问某些代码块或方法，从而避免并发冲突，保证共享资源的安全性。
- 该关键词相当于设置了共享对象，每一个共享对象有一个对象锁，该对象锁每一次只能锁住一个线程，只有当该线程释放了对象锁后其他的线程才能执行该对象的其他 **synchronized** 方法。同时，每一个共享对象有一个等待队列，在 **wait()** 方法和 **notify()** 方法中会起到作用。注意，**wait()** 方法和 **notify()** 方法只有在 **synchronized** 的代码块中才有意义。
- 用法：
  - 同步实例方法：

锁定某个对象的同步方法，当一个线程调用该对象的某一个同步方法时，其他的线程无法调用该对象的任何同步方法。

```
● ● ●
1 public class Counter {
2     private int count = 0;
3
4     public synchronized void increment() {
5         count++;
6     }
7
8     public synchronized int getCount() {
9         return count;
10    }
11 }
```

Fence 177

- 同步静态方法：

锁定当前类的 **Class** 对象。当一个线程调用该类的某一个静态同步方法时，其他的线程无法调用该类的任何静态同步方法。

```
● ● ●
1 public class Counter {
2     private static int count = 0;
3
4     public static synchronized void increment() {
5         count++;
6     }
7
8     public static synchronized int getCount() {
9         return count;
10    }
11 }
```

Fence 178

- 同步代码块：

锁定一个特定对象，可以精确控制加锁范围，提高性能。

```
● ● ●
1 public class Counter {
2     private int count = 0;
3
4     public void increment() {
5         synchronized (this) {
6             count++;
7         }
8     }
9 }
```

```
9  
10     public int getCount() {  
11         synchronized (this) {  
12             return count;  
13         }  
14     }  
15 }
```

Fence 179

## 13.1.3 通讯方法

Java 提供了 `wait()`、`notify()` 和 `notifyAll()` 方法来实现线程间通信。这些方法是 `Object` 类的一部分，而不是 `Thread` 类，因为它们作用于共享的对象上，而不是线程本身。

### 13.1.3.1 `wait()` 方法

- 作用：
  - 使调用该方法的线程进入等待状态（挂起），直到另一个线程调用同一对象的 `notify()` 或 `notifyAll()` 方法。
  - 调用 `wait()` 时，线程会释放当前持有的对象锁。
- 使用场景：
  - 常用于线程等待某个条件的满足。
- 注意：
  - 必须在同步块（`synchronized`）中调用，否则会抛出 `IllegalMonitorStateException`。对于在同一个 `synchronized` 的对象中，调用了 `wait()` 方法后，会进入该共享对象的等待队列中，只有在该共享对象的其他方法中调用了 `notify()`，才会开始释放队列中的线程。
  - 线程进入等待状态后，必须重新获得对象的锁才能继续执行。

### 13.1.3.2 `notify()` 方法

- 作用：
  - 唤醒一个正在等待对象锁的线程。
  - 被唤醒的线程不会立即执行，必须等调用 `notify()` 的线程释放锁后，等待线程才能获得锁并继续执行。
- 使用场景：
  - 通知某个等待中的线程可以继续执行。
- 注意：
  - 如果有多个线程在等待，具体唤醒哪个线程是不确定的（取决于 JVM 的实现）。

### 13.1.3.3 `notifyAll()` 方法

**作用：**

- 唤醒所有正在等待对象锁的线程。
- 被唤醒的线程依次竞争锁，只有一个线程会获得锁并执行，其余线程会继续等待。

**使用场景：**

- 需要通知所有等待线程以重新检查条件。

**注意：**

- 如果没有合理的条件检查，可能会引发资源争用。

**13.1.4 创建两个线程，一个生产者线程和一个消费者线程，使用一个共享资源（如一个整数）。生产者线程增加共享资源的值，消费者线程减少它的值。确保线程间正确通信，避免数据不一致。**



```
1 package Homework;
2
3 class SharedResource {
4     private int data;
5
6     public synchronized void product(){
7         data++;
8         System.out.println(Thread.currentThread().getName() + " "
9             product + data);
10        notify(); //通知消费者
11    }
12    public synchronized void consume() throws InterruptedException {
13        if(data == 0){
14            wait();
15        }
16        data--;
17        System.out.println(Thread.currentThread().getName() + " "
18            consume + data);
19    }
20
21 public class ThreadDemo {
22     public static void main(String[] args) {
```

```

23     SharedResource sharedResource = new SharedResource();
24
25
26     Thread producer = new Thread("producer"){
27         @Override
28         public void run() {
29             for(int i = 0; i < 999; i++){
30                 sharedResource.product();
31             }
32         }
33     };
34     Thread consumer = new Thread("consumer"){
35         @Override
36         public void run() {
37             try { //注意这里消费者要多一个，所以最后肯定线程会暂停
38                 for(int i = 0; i < 1000; i++){
39                     sharedResource.consume();
40                 }
41             } catch (InterruptedException e) {
42                 throw new RuntimeException(e);
43             }
44         }
45     };
46     consumer.start();
47     producer.start();
48
49
50 }
51 }
52

```

Fence 180

## 13.2 死锁

### 13.2.1 死锁概念

死锁是一种程序在多线程环境中可能出现的情况，它指的是两个或多个线程因为相互等待对方释放资源而陷入永久阻塞的状态。简单来说，死锁会导致线程无法继续执行，也无法退出，最终程序挂起。

死锁出现需要满足下面的条件：

#### 互斥条件：

- 至少有一个资源只能被一个线程占用。

#### 占用且等待条件：

- 一个线程持有一个资源的同时，还在等待另一个资源，而该资源被其他线程占用。

### 不可抢占条件：

- 已经分配给线程的资源不能被强行抢占，只能由持有它的线程释放。

### 循环等待条件：

- 存在一个线程循环等待链，例如线程 A 等待线程 B 持有的资源，而线程 B 又等待线程 A 持有的资源。

## 13.2.2 死锁示例



```

1 class SharedResource {
2     public static final Object lock1 = new Object();
3     public static final Object lock2 = new Object();
4 }
5
6 public class ThreadDemo {
7     public static void main(String[] args) {
8         Thread t1 = new Thread("Thread1"){
9             @Override
10            public void run(){
11                 System.out.println(Thread.currentThread().getName() +
12                     "尝试获取锁1");
13                 synchronized (SharedResource.lock1){
14                     System.out.println(Thread.currentThread().getName() + "获取锁1");
15                     try {
16                         Thread.sleep(100);
17                     } catch (InterruptedException e) {
18                         throw new RuntimeException(e);
19                     }
20                     System.out.println(Thread.currentThread().getName() + "尝试获取锁2");
21                     synchronized (SharedResource.lock2){
22                         System.out.println(Thread.currentThread().getName() + "获取锁2");
23                     }
24                 }
25             };
26         Thread t2 = new Thread("Thread2"){
27             @Override
28             public void run(){
29                 System.out.println(Thread.currentThread().getName() +
30                     "尝试获取锁2");
31                 synchronized (SharedResource.lock2){
32                     System.out.println(Thread.currentThread().getName() + "获取锁2");
33                 }
34             };
35         }
36     }
37 }
```

```

31     System.out.println(Thread.currentThread().getName() + " 获取锁2");
32         try {
33             Thread.sleep(100);
34         } catch (InterruptedException e) {
35             throw new RuntimeException(e);
36         }
37
38     System.out.println(Thread.currentThread().getName() + " 尝试获取锁1");
39         synchronized (SharedResource.lock1){
40
41         System.out.println(Thread.currentThread().getName() + " 获取锁2");
42         }
43     };
44     t1.start();
45     t2.start();
46
47
48 }
49 }
```

Fence 181

## 13.2.3 如何避免死锁

### 13.2.3.1 按顺序获取锁

确保所有线程以相同的顺序获取资源锁。例如在上例中将锁的获取顺序都设置为先lock1再lock2。

### 13.2.3.2 使用 `tryLock()`

Java 的 `ReentrantLock` 提供了 `tryLock()` 方法，可以尝试获取锁，而不是无限期等待。`tryLock()` 返回 `boolean` 类型，可以作为判断条件出现在 `if` 语句中，如果获取失败则返回 `false`。

### 13.2.3.3 限制锁的作用范围

尽量缩小锁的作用范围，减少线程持有锁的时间，从而降低发生死锁的概率。

### 13.2.3.4 避免嵌套锁

尽量避免嵌套同步块（一个 `synchronized` 块中再嵌套另一个 `synchronized` 块），减少锁依赖的复杂性。

- 如果获取不到锁，可以选择退出或尝试其他操作，避免死锁。

# 13.3 线程池

## 13.3.1 简述

线程池（`ThreadPool`）是一个多线程管理工具，维护了一组固定数量的线程，用于执行任务。线程池允许程序将任务提交到线程队列中，线程池中的工作线程会依次处理这些任务，从而避免频繁创建和销毁线程的开销。线程池有很多的优点。

### 提高性能：

- 重用线程池中的线程，避免了频繁创建和销毁线程的开销。
- 减少线程的启动时间，从而加快任务执行速度。

### 资源管理：

- 线程池可以限制线程的数量，避免过多线程竞争系统资源，从而提高系统的稳定性。

### 任务管理：

- 可以控制任务的执行顺序（如 FIFO）。
- 支持任务的调度和执行。

### 简化并发管理：

- 提供了统一的接口来管理线程（如任务提交、取消、监控等），简化多线程开发。

## 13.3.2 一个简单的线程池实现。

Java中我们可以自己构建线程池，也可以使用一些框架下的线程池，例如：

Java 提供了多种线程池实现，主要通过 `Executor` 框架实现：

- `newFixedThreadPool()`：创建一个固定大小的线程池。
- `newCachedThreadPool()`：创建一个可缓存的线程池，线程数可动态增长。
- `newSingleThreadExecutor()`：创建一个单线程池。
- `newScheduledThreadPool()`：创建一个支持定时任务的线程池。
- 上面的都没什么用，因为我们不会用(bushi)。
- 现在懂点概念就行了，这玩意考代码得要老命

以下是一个简单实现的线程池，它使用固定数量的工作线程来执行提交的任务。



```
1 package Homework;  
2
```

```
3 import java.util.LinkedList;
4 import java.util.Queue;
5 class SimpleThreadPool {
6     private final int poolSize; // 线程池大小
7     private final WorkerThread[] workers; // 工作线程数组
8     private final Queue<Runnable> taskQueue; // 任务队列
9     private boolean isShutdown = false; // 是否关闭线程池
10
11    // 构造方法，初始化线程池
12    public SimpleThreadPool(int poolSize) {
13        this.poolSize = poolSize;
14        this.taskQueue = new LinkedList<>();
15        this.workers = new WorkerThread[poolSize];
16
17        // 创建并启动工作线程
18        for (int i = 0; i < poolSize; i++) {
19            workers[i] = new WorkerThread();
20            workers[i].start();
21        }
22    }
23
24    // 提交任务到线程池
25    public synchronized void submit(Runnable task) {
26        if (isShutdown) {
27            throw new IllegalStateException("ThreadPool is shutdown");
28        }
29        taskQueue.offer(task); // 将任务加入队列
30        notify(); // 通知工作线程
31    }
32
33    // 关闭线程池
34    public synchronized void shutdown() {
35        isShutdown = true;
36        for (WorkerThread worker : workers) {
37            worker.interrupt(); // 中断所有工作线程
38        }
39    }
40
41    // 工作线程类
42    private class WorkerThread extends Thread {
43        @Override
44        public void run() {
45            while (true) {
46                Runnable task;
47                synchronized (SimpleThreadPool.this) {
48                    while (taskQueue.isEmpty() && !isShutdown) {
49                        try {
50                            SimpleThreadPool.this.wait(); // 等待任务
51                        } catch (InterruptedException e) {
52                            return; // 线程被中断，退出
53                        }
54                    }
55                    task = taskQueue.poll();
56                }
57                if (task != null) {
58                    task.run();
59                }
60            }
61        }
62    }
63}
```

```
54         }
55         if (isShutdown && taskQueue.isEmpty()) {
56             return; // 如果线程池关闭且没有任务，退出
57         }
58         task = taskQueue.poll(); // 获取任务
59     }
60     try {
61         if (task != null) {
62             task.run(); // 执行任务
63         }
64     } catch (Exception e) {
65         e.printStackTrace();
66     }
67 }
68 }
69 }
70 }
71
72 public class ThreadDemo {
73     public static void main(String[] args) {
74         // 创建线程池
75         SimpleThreadPool threadPool = new SimpleThreadPool(5);
76
77         // 提交任务
78         for (int i = 1; i <= 10; i++) {
79             int taskNumber = i;
80             threadPool.submit(new Runnable() {
81                 @Override
82                 public void run() {
83
84                     System.out.println(Thread.currentThread().getName() + " is executing
85                     task " + taskNumber);
86                     int sum = 34242113;
87                     for(int i = 0; i < 123122023; i++){
88                         sum ^= i;
89                     }
90                     System.out.println(taskNumber+ ":" + sum);
91                 };
92             });
93         }
94         // 关闭线程池
95         threadPool.shutdown();
96     }
97 }
98 }
```

## 13.4 Callable和Future

### 13.4.1 讨论Java中Callable接口与Runnable接口的区别

#### 13.4.1.1 \*\* Callable 和 Runnable 的定义\*\*

- **Runnable**

:

- 是 Java 提供的一个函数式接口，用于定义一个任务，任务执行时不会返回结果，也不会抛出检查异常。

- 方法签名：



```
1 public void run();
```

Fence 183

- **Callable** :

- 是 Java 提供的一个泛型接口，用于定义一个任务，任务执行后会返回一个结果，并且可以抛出检查异常。

- 方法签名：



```
1 public V call() throws Exception;
```

Fence 184

### 13.4.2 主要区别

特性	Runnable	Callable
返 回 值	无返回值。	有返回值，可以通过 Future 获取。
异 常 处 理	不允许抛出检查异常（ Checked Exception ），只能捕获和处理。	可以抛出检查异常，由调用者捕获。

特性	Runnable	Callable
方法名称	<code>run()</code>	<code>call()</code>
适用场景	用于执行简单任务，不需要返回结果或处理异常的场景。	用于需要返回结果或处理异常的任务。
使用方式	可以直接通过 <code>Thread</code> 或 <code>ExecutorService</code> 执行。	必须通过 <code>ExecutorService</code> 执行，并结合 <code>Future</code> 。

Table 10

### 13.4.3 Future对象

`Future` 是一个接口，用于表示一个任务的异步计算结果。通过 `Future` 对象，主线程可以获取任务的执行状态和结果。

#### Future 的常用方法

方法	描述
<code>get()</code>	获取任务执行的结果，如果任务尚未完成，则阻塞当前线程。
<code>get(long timeout, TimeUnit unit)</code>	带超时的 <code>get()</code> 方法，如果任务未在指定时间内完成，会抛出 <code>TimeoutException</code> 。
<code>isDone()</code>	检查任务是否已经完成，返回 <code>true</code> 或 <code>false</code> 。
<code>isCancelled()</code>	检查任务是否被取消。
<code>cancel(boolean mayInterruptIfRunning)</code>	取消任务，如果任务已经完成或无法取消，则返回 <code>false</code> 。

Table 11

# 13.5 原子变量和可见性

## 13.5.1 解释Java中原子变量的作用，并讨论它们如何保证线程间操作的原子性和可见性。

Java 中的原子变量是 `java.util.concurrent.atomic` 包提供的一组类，由于底层差别，可以保证这些操作中不需要带锁。

原子变量通过底层的无锁机制（CAS），在多线程环境中保证：

1. **操作的原子性**：保证变量的更新操作（如加法、减法等）是不可分割的，多个线程同时更新变量时不会引发竞争问题。
2. **操作的可见性**：保证一个线程修改变量后，其他线程能够立即看到最新值。

### 13.5.1.1 原子性

**CAS (Compare-And-Swap)** 是一种硬件级别的原子操作，包含以下三个步骤：

1. 比较内存中的变量值和预期值是否相等。
2. 如果相等，将变量值更新为新值。
3. 如果不相等，返回当前值，不进行更新。

### 13.5.1.2 可见性

在 Java 中，线程通常会在自己的工作内存中缓存变量，线程对变量的修改可能不会立即同步到主内存，而其他线程可能无法看到最新值。

#### 解决可见性

原子变量内部使用了 **volatile 修饰** 的变量，`volatile` 保证了以下两点：

1. 可见性：
  - 一个线程修改了 `volatile` 变量的值，其他线程能立即看到该修改。
2. 禁止指令重排：
  - 确保对 `volatile` 变量的读写顺序不会被编译器或 CPU 重排序优化。

## 13.5.2 编程题：创建一个多线程程序，其中一个线程更新一个共享计数器，而另一个线程读取这个计数器。使用 AtomicInteger来确保更新对所有线程都是可见的，并且是原子操作。



```
1 package Homework;
2
3 import java.util.concurrent.atomic.AtomicInteger;
4
5 public class ThreadDemo{
6     public static void main(String[] args) {
7         // 共享计数器
8         AtomicInteger counter = new AtomicInteger(0);
9         Thread adder = new Thread("Adder"){
10             @Override
11             public void run(){
12                 for(int i = 0; i < 10; i++){
13                     int value = counter.incrementAndGet();
14                     System.out.println(Thread.currentThread().getName()
15                         + " " + value);
16                     try {
17                         sleep(100);
18                     } catch (InterruptedException e) {
19                         throw new RuntimeException(e);
20                     }
21                 }
22             };
23
24         Thread reader = new Thread("Reader"){
25             @Override
26             public void run(){
27                 for(int i = 0; i < 10; i++){
28                     int value = counter.get();
29                     System.out.println("Reader " + value);
30                     try {
31                         sleep(100);
32                     } catch (InterruptedException e) {
33                         throw new RuntimeException(e);
34                     }
35                 }
36             };
37         };
38         reader.start();
39         adder.start();
40     }
41 }
```

# 14. 014

## 14.1 Java异常处理基础

异常处理（Exception Handling）是确保程序健壮性和稳定性的重要机制。通过异常处理，开发者能够有效地应对程序运行过程中可能出现的各种错误和异常情况，避免程序意外终止，并提供友好的错误提示或恢复措施。

### 14.1.1 基本使用方法



```

1 try{
2     // 在此处throw Exception
3 }catch(Exception e){
4     // 捕获Exception后的处理
5 }finally{ // 可选
6     // 无论是否捕获异常 都将执行
7 }

```

- try 块：** `try` 块用于包围可能会抛出异常的代码。当程序执行到 `try` 块中的代码时，如果发生异常，控制权将转移到相应的 `catch` 块进行处理。
- catch 块：** `catch` 块用于捕捉并处理 `try` 块中抛出的特定类型的异常。可以有多个 `catch` 块来处理不同类型的异常。可以使用多个 `catch` 来捕获同一 `try` 块的内容，或者将多个 `Exception` 捕获进入同一 `catch` 块。一般按照从具体到一般的顺序排列。



```

1 try {
2     int num = 10 / 0;
3 }catch (ArithmaticException | NullPointerException e){ // 处理多个异常
4     System.out.println("Catch1");
5 }catch (Exception e) {
6     System.out.println("Catch2" + e.getMessage());
7 }

```

3. **finally 块**: **finally** 块中的代码无论是否发生异常，都会被执行。通常用于释放资源、关闭文件流、释放数据库连接等操作，确保资源得到正确的释放。**finally**块为可选的。

```
● ● ●  
1 Scanner sc = null;  
2     try{  
3         File file = new File("in.txt");  
4         sc = new Scanner(file);  
5     } catch (FileNotFoundException e) {  
6         throw new RuntimeException(e);  
7     } finally {  
8         if (sc != null) {  
9             sc.close();  
10        }  
11    }
```

Fence 188

## 14.1.2 编写一个Java程序，尝试解析用户输入的整数，并处理可能发生的 **NumberFormatException**

注意，**NumberFormatException** 属于 **RuntimeException**，所以抛出时并不一定需要try块。

```
● ● ●  
1 try{  
2     int num = Integer.parseInt("Hello World");  
3 }catch (NumberFormatException e){  
4     System.out.println(e.getMessage());  
5 }
```

Fence 189

## 14.2 Java自定义异常

### 14.2.1 解释何时需要创建自定义异常，并讨论自定义异常与标准异常的区别。

#### 14.2.1.1 创建自定义异常的情况

- Java自带的异常无法满足开发需求。
- 自定义更加清晰的异常，可以使代码易读性提高。
- 自定义异常可以隐藏具体的实现细节，提供一致的错误接口。由于子类异常可以使用其父类异常捕获，自定义异常可以更加方便管理。

### 14.2.1.2 区别对比

方面	标准异常	自定义异常
定义来源	Java标准库预定义，如 <code>NullPointerException</code>	开发者根据需求自行定义
用途	处理通用错误，如空指针、IO错误	处理特定业务或应用场景下的错误
继承体系	通常继承自 <code>Exception</code> 或 <code>RuntimeException</code>	继承自 <code>Exception</code> 或 <code>RuntimeException</code>
信息表达	表达通用错误信息	提供更具体、语义化的错误信息
可读性	适用于广泛场景	提高代码可读性和维护性

Table 12

### 14.2.2 创建一个自定义异常 `InvalidDataException`，并在处理特定业务逻辑时抛出此异常。



```

1 public class ExceptionDemo {
2     public static void main(String[] args) {
3         Scanner sc = new Scanner(System.in);
4         try{
5             int age = sc.nextInt();
6             if(age<18){
7                 throw new InvalidDataException("Can't reach");
8             }
9         }catch (InvalidDataException e){
10             System.out.println(e.getMessage());
11         }
12     }
13 }
14

```

```
15 class InvalidDataException extends Exception {  
16     public InvalidDataException(String message) {  
17         super(message);  
18     }  
19 }  
20
```

Fence 190

## 14.3 Java Checked和Unchecked异常

### 14.3.1 异常分类解释

Java 中的异常分为两大类： **检查型异常 (Checked Exceptions)** 和 **非查型异常 (Unchecked Exceptions)**。

#### 14.3.1.1 检查型异常 (Checked Exceptions)

- **定义**：检查型异常是指在编译时被强制检查的异常。Java编译器要求程序员必须对这些异常进行处理，否则代码将无法通过编译。直接继承于 `java.lang.Exception`。
- **使用**：使用Checked Exceptions的时候，要求使用 `try-catch` 块捕获异常或者在方法签名中使用 `throws` 声明抛出异常。检查型异常通常用于表示可预见且可恢复的错误。



```
1 public static void main(String[] args) {  
2     try{  
3         f();  
4     }catch (IOException e){  
5         System.out.println(e.getMessage());  
6     }  
7 }  
8 static void f() throws IOException{  
9     throw new IOException();  
10 }
```

Fence 191

- **使用场景**：检查型异常通常用于表示可预见且可恢复的错误，例如：
  - `IOException`
  - `FileNotFoundException`
  - `ClassNotFoundException`

### 14.3.1.2 非检查型异常 (Unchecked Exceptions)

- **定义**：非检查型异常是在运行时抛出的异常，编译器不会强制要求程序员处理这些异常。继承自 `java.lang.RuntimeException`。
- **使用场景**：非检查型异常通常用于表示**编程错误或不可预见的错误**，例如：
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
  - `IllegalArgumentException`

### 14.3.1.3 检查型异常与非检查型异常的区别

方面	检查型异常 (Checked Exceptions)	非检查型异常 (Unchecked Exceptions)
继承层次	继承自 <code>Exception</code> 但不继承自 <code>RuntimeException</code>	继承自 <code>RuntimeException</code>
编译器检查	必须捕获或声明抛出，否则编译错误	不需要捕获或声明，可以选择处理
适用场景	预见且可恢复的错误，如IO操作、网络通信等	编程错误或不可预见的错误，如空指针、数组越界等
设计意图	强制程序员处理可能发生的异常	表示程序中的逻辑错误或运行时错误
示例异常	<code>IOException</code> , <code>SQLException</code>	<code>NullPointerException</code> , <code>IllegalArgumentException</code>

Table 13

### 14.3.2 编写一个程序，该程序包含一个可能抛出Checked异常的方法，并在调用处适当处理这个异常。



```
1 public static void main(String[] args) {  
2     try{  
3         f();  
4     }catch(FileNotFoundException e){  
5         System.out.println(e.getMessage());  
6     }  
7 }  
8 static void f() throws FileNotFoundException {  
9     File file = new File("in.txt");  
10    Scanner sc = new Scanner(file); // 可能会抛出FileNotFoundException  
11 }
```

Fence 192

## 14.4 Java异常链

### 14.4.1 讨论Java异常链的概念，并解释它如何帮助调试。

**异常链 (Exception Chaining)** 是一种将一个异常与另一个异常关联起来的机制。通过异常链，开发者可以在捕获一个异常后，抛出一个新的异常，并将原始异常作为新异常的原因 (cause) 进行传递。

#### 14.4.1.1 Exception详解

**Throwable** 是Java中所有错误和异常的超类。**Throwable** 是Java中所有错误和异常的超类。其中有两个主要的子类，**Exception** 和 **Error**。

**Error** :

- 表示严重的错误，通常是无法恢复的，如 **OutOfMemoryError** (内存溢出错误)。
- 一般不建议程序捕捉和处理 **Error**，因为它们通常表示程序无法继续运行的严重问题。

**Exception** :

- 表示程序可以捕捉并处理的异常情况。一般可以处理。又分为检查型异常 (Checked Exception) 和非检查型异常 (Unchecked Exception)。

Throwable包含以下构造方法：



```
1 public class Throwable {  
2     public Throwable() { ... }  
3     public Throwable(String message) { ... }  
4     public Throwable(String message, Throwable cause) { ... }  
5     public Throwable(Throwable cause) { ... }  
6 }
```

Fence 193

其中String的内容作为消息，也就是e.getMessage()的内容。Throwable作为原因，也就是e.getCause()的内容。

以及常用方法：

- `getMessage()`：返回异常的详细信息。
- `getCause()`：返回引发此异常的原因。
- `printStackTrace()`：打印异常的堆栈跟踪信息，帮助调试。

#### 14.4.1.2 一个简单的异常链实现



```
1 try {  
2     try {  
3         // 模拟一个底层异常  
4         throw new NullPointerException("底层空指针异常");  
5     } catch (NullPointerException e) {  
6         // 抛出一个新的异常，并将原始异常作为原因传递  
7         throw new Throwable("高级别异常", e);  
8     }  
9 } catch (Throwable t) {  
10    System.out.println("异常消息: " + t.getMessage());  
11    System.out.println("原因: " + t.getCause());  
12    t.printStackTrace();  
13 }  
14 }
```

Fence 194

原因的传递除了使用默认的构造函数，还可以使用 `initCause(Throwable)` 函数实现。

#### 14.4.1.3 异常链如何帮助调试

异常链通过保留多个异常的堆栈信息，能够快速定位异常发生的路径，并且了解异常产生的原因和异常消息。

## 14.4.2 编写一个方法，该方法打开一个文件并读取内容。 在捕获 `IOException` 的同时，将原始异常包装在新的异常中抛出。

```
● ● ●  
1 public class ExceptionDemo {  
2     public static void main(String[] args) {  
3         try{  
4             f();  
5         }catch(Exception e){  
6             System.out.println(e.getMessage());  
7             System.out.println(e.getCause());  
8         }  
9     }  
10    static void f() throws IOException{  
11        try {  
12            Scanner sc = new Scanner(new File("noFile.txt"));  
13        } catch (IOException e) {  
14            throw new ReadException("File not found", e);  
15        }  
16    }  
17 }  
18  
19 class ReadException extends IOException{  
20     ReadException(String message, Throwable cause) {  
21         super(message, cause);  
22     }  
23 }
```

Fence 195

## 14.5 Java finally块与资源释放

### 14.5.1 讨论Java中finally块的重要性，特别是在资源管理（如关闭文件流）中的应用。

`finally` 块确保无论程序是否抛出异常，特定的代码段总会被执行。这在资源管理（如关闭文件流、数据库连接等）中尤为关键。

在进行资源管理时，未正确关闭的资源可能导致内存泄漏、文件锁定等问题。`finally` 块提供了一种可靠的方式来实现这一点。

## 14.5.2 编写一个Java程序，使用 try-catch-finally 块打开和关闭文件。确保即使在发生异常的情况下，文件也能被正确关闭。



```
1 Scanner sc = null;
2     try {
3         sc = new Scanner(new File("in.txt"));
4         int a= sc.nextInt();
5     } catch (Exception e) {
6         if(sc != null){
7             System.out.println("sc未正常关闭");
8             sc.close();
9         }
10    }
```

Fence 196

# 15. 015

## 15.1 抽象类的应用

### 15.1.1 抽象类简述

**抽象类 (Abstract Class)** 是实现面向对象编程 (OOP) 中抽象化和模块化的重要工具。抽象类允许开发者定义一个类的基本结构和行为，而不需要提供所有具体实现细节。这不仅促进了代码的复用和组织，还增强了系统的可维护性和扩展性。

#### 15.1.1.1 使用方法：

**抽象类** 是使用关键字 **abstract** 声明的类。它不能被实例化（即不能直接创建对象），但可以被继承。只有在抽象类中才能创建抽象方法。抽象方法可以不用给出实现代码，而是交由子类去实现。

#### 15.1.1.2 与接口的对比

接口和抽象类十分相似，下表可以帮助理解两者的区别：

特性	抽象类	接口
声明方式	使用 <code>abstract</code> 关键字	使用 <code>interface</code> 关键字
方法实现	可以有抽象方法和具体方法	只能有抽象方法 (Java 8后支持默认方法和静态方法)
字段	可以有实例字段和常量	只能有常量 (默认是 <code>public static final</code> )
继承	子类使用 <code>extends</code> 继承抽象类	实现类使用 <code>implements</code> 实现接口
多继承	不支持多继承，一个类只能继承一个抽象类	支持多继承，一个类可以实现多个接口
构造方法	可以有构造方法	不能有构造方法
访问修饰符	方法可以有不同的访问修饰符	方法默认是 <code>public</code> ，不能有其他访问修饰符

Table 14

### 15.1.1.3 抽象类在代码抽象化和模块化中的作用

- 提供通用模板

抽象类允许开发者定义一组通用的方法和字段，这些方法和字段可以被多个子类共享，从而避免代码重复。

- 强制子类实现特定行为

通过定义抽象方法，抽象类可以强制子类实现特定的方法，确保子类具备某些核心功能。

- 提供部分实现

抽象类可以包含部分实现，子类可以直接继承这些实现，或者根据需要重写。

## 15.1.2 设计一个抽象类 Shape，包含一个抽象方法 calculateArea()，然后创建两个子类 Circle 和 Rectangle 实现这个抽象方法。

● ● ●

```
1 abstract class Shape{
2     abstract double calculateArea();
3 }
4
5 class Rectangle extends Shape{
6     double length = 0;
7     double width = 0;
8
9     public Rectangle(double length, double width) {
10         this.length = length;
11         this.width = width;
12     }
13
14     public Rectangle() {
15     }
16
17     @Override
18     double calculateArea() {
19         return length * width;
20     }
21 }
22
23 class Circle extends Shape{
24     double radius = 0;
25     public Circle(double radius) {
26         this.radius = radius;
27     }
28     public Circle() {
29
30     }
31     double calculateArea() {
32         return Math.PI * radius * radius;
33     }
34 }
```

# 15.2 泛型类和泛型接口的使用

## 15.2.1 泛型、泛型类和泛型接口简述

**泛型**允许在类、接口和方法中使用类型参数，使得代码在处理不同数据类型时具有更高的灵活性和类型安全性。

### 15.2.1.1 使用语法

- 类

```
● ● ●  
1 // T可以为任何类型  
2 class Base1<T>{  
3     T item;  
4 }  
5 // T只能是Integer的子类  
6 class Base2<T extends Integer>{  
7     T item;  
8     void show(){  
9         System.out.println(item.intValue());  
10    }  
11 }
```

Fence 198

- 接口

```
● ● ●  
1 interface Test<T>{  
2     void test(T t);  
3 }  
4  
5 // 使用类的泛型作为接口的泛型  
6 class Base1<T> implements Test<T>{  
7     @Override  
8     public void test(T t){  
9         System.out.println(t);  
10    }  
11 }  
12  
13 // 自己设置泛型  
14 class Base2<T> implements Test<String>{  
15     @Override  
16     public void test(String t){  
17         System.out.println(t);  
18     }  
19 }
```

- 函数调用



```
1 package Homework;
2
3 import java.io.*;
4 import java.util.*;
5
6 public class GenericsDemo {
7     public static void main(String[] args) {
8         Base<IOException> base = new Base(new IOException("Hello
World"));
9         // f(base);      // 调用失败 无法识别Base<Exception> 和
10        Base<IOException>
11        g(base);      // 可以识别
12        h(base);      // 可以识别
13        k(base);      // 可以识别
14    }
15
16    static void f(Base<Exception> base) {
17        base.show();
18    }
19
20    // 接受Exception的所有子类 相当于下界
21    static void g(Base<? extends Exception> base){
22        base.show();
23    }
24
25    // 接受FileNotFoundException的所有父类 相当于上界
26    static void h(Base<? super FileNotFoundException> base){
27        base.show();
28    }
29
30    // 接受所有的类 相当于Base<> base
31    static void k(Base<?> base){
32        base.show();
33    }
34
35 class Base<T>{
36     T item;
37     Base(T item){
38         this.item = item;
39     }
40     void show(){
41         System.out.println(item);
42     }
43 }
```

### 15.2.1.2 Java中泛型类和泛型接口的优势。

- 代码重用性 (Code Reusability)

泛型允许开发者编写通用的类和方法，适用于不同的数据类型，减少了代码的重复，提高了代码的复用性。

- 类型安全 (Type Safety)

泛型通过在编译时进行类型检查，确保在使用集合类时，元素的类型是一致的，避免了运行时的 `ClassCastException`。

- 增强可读性

通过明确指定集合或类中的元素类型，代码的意图更加清晰，易于理解和维护。

### 15.2.2 创建一个泛型类 `Box<T>`，它包含一个泛型类型T的私有成员变量，并提供相应的 `getter` 和 `setter` 方法。然后创建一个泛型接口 `ComparableBox<T>`，其中包含一个比较两个泛型类型对象的方法。



```
1 class Box<T extends Comparable<T>> implements ComparableBox<T> {
2     private T value;// 注意这Comarable<T> 这个T能够保证比较限制在T类型
3     Box(T value){
4         this.value = value;
5     }
6
7     public T getValue() {
8         return value;
9     }
10
11    public void setValue(T value) {
12        this.value = value;
13    }
14
15    @Override
16    public int Compare(T a) {
17        return value.compareTo(a);
18    }
19 }
20
21 interface ComparableBox<T>{
22     int Compare(T a);
23 }
```

# 15.3 集合框架的应用 Fence 201

## 15.3.1 Collection接口

Java中 **Collection** 是最基本的集合接口，提供了操作集合的通用方法。其主要的子接口有：

- **List**：有序集合，允许重复元素。
- **Set**：无序集合，不允许重复元素。
- **Queue**：用于定义队列结构。

以下是 **Collection** 接口中定义的常用方法：（简单了解即可）

方法	描述
<code>boolean add(E e)</code>	添加一个元素到集合中。
<code>boolean addAll(Collection&lt;? extends E&gt; c)</code>	将另一个集合中的所有元素添加到当前集合。
<code>void clear()</code>	清空集合中的所有元素。
<code>boolean contains(Object o)</code>	检查集合中是否包含指定元素。
<code>boolean containsAll(Collection&lt;?&gt; c)</code>	检查当前集合是否包含另一个集合的所有元素。
<code>boolean isEmpty()</code>	判断集合是否为空。
<code>Iterator&lt;E&gt; iterator()</code>	返回一个迭代器，用于遍历集合中的元素。
<code>boolean remove(Object o)</code>	删除集合中的指定元素。
<code>boolean removeAll(Collection&lt;?&gt; c)</code>	删除当前集合中所有与指定集合相同的元素。
<code>boolean retainAll(Collection&lt;?&gt; c)</code>	仅保留当前集合中与指定集合相同的元素。
<code>int size()</code>	返回集合中元素的个数。
<code>Object[] toArray()</code>	将集合中的元素转换为一个数组。

Table 15

其中，常用的实现类包括：

### List 的实现类：

- `ArrayList`：动态数组，随机访问效率高。
- `LinkedList`：链表实现，插入和删除效率高。

### Set 的实现类：

- `HashSet`：基于哈希表实现，无序且不允许重复。
- `TreeSet`：基于红黑树实现，有序且不允许重复。

注意：`Map` 并不是 `Collection` 的实现类。

## 15.3.2 描述Java集合框架中 `List`、`Set` 和 `Map` 接口的主要区别和用途。

特性	<code>List</code>	<code>Set</code>	<code>Map</code>
<b>是否允许重复</b>	允许	不允许	键不允许重复，值可以重复
<b>是否有序</b>	保持插入顺序	通常无序（特殊实现有序）	通常无序（特殊实现有序）
<b>是否基于索引</b>	基于索引访问（通过索引操作）	不支持索引访问	通过键访问值
<b>典型实现类</b>	<code>ArrayList</code> 、 <code>LinkedList</code>	<code>HashSet</code> 、 <code>TreeSet</code>	<code>HashMap</code> 、 <code>TreeMap</code>

Table 16

### 15.3.2.1 List

#### 特点：

1. **有序性**：`List` 保持元素的插入顺序，即元素的存储顺序与遍历顺序一致。
2. **允许重复**：`List` 可以包含重复的元素。
3. **索引访问**：`List` 提供通过索引访问元素的方法（如 `get(index)` 和 `set(index, element)`）。
4. **实现类**：常见实现类有 `ArrayList`、`LinkedList` 和 `Vector`。

### 15.3.2.2 Set

- **特点：**

1. **无序性**：Set 通常不保证元素的存储顺序（某些实现如 `LinkedHashSet` 保留插入顺序，`TreeSet` 保证排序顺序）。
2. **不允许重复**：Set 中的每个元素都是唯一的，不允许重复。
3. **实现类**：常见实现类有 `HashSet`、`TreeSet`。

### 15.3.2.3 \*\* Map\*\*

- **特点：**

1. **键值对存储**：Map 用于存储键值对，每个键（Key）必须是唯一的，值（Value）可以重复。
2. **无序性**：Map 的键值对通常不保证顺序（`LinkedHashMap` 保留插入顺序，`TreeMap` 保证键的排序顺序）。
3. **实现类**：常见实现类有 `HashMap`、`TreeMap`。

### 15.3.3 编写一个程序，使用Java集合框架实现一个简单的电话簿，允许添加、删除和查找联系人信息。



```
1 public class GenericsDemo {
2     public static void main(String[] args) {
3         PhoneBook pb = new PhoneBook();
4         pb.add("WHU", "2024");
5         pb.find("JAVA");
6         pb.find("WHU");
7         pb.remove("JAVA");
8         pb.remove("WHU");
9     }
10 }
11
12 class PhoneBook{
13     Map<String, String> Contacts = new HashMap<>();
14
15     PhoneBook(){
16
17     }
18     PhoneBook(PhoneBook tmp){
19         // 注意这里，实现深拷贝
20         this.Contacts = new HashMap<> (tmp.Contacts);
21     }
22
23     void add(String name, String number){}
```

```

24     Contacts.put(name, number);
25     System.out.println("添加成功: " + name + " - " + number);
26 }
27 void remove(String name){
28     String tmp = Contacts.remove(name);
29     if(tmp != null){
30         System.out.println("成功删除 " + name + " - " + tmp);
31     }else {
32         System.out.println("删除失败, 没有找到对应的联系人");
33     }
34 }
35 void find(String name){
36     if(Contacts.containsKey(name)){
37         System.out.println(
38             "查找成功 " + name + " - " + Contacts.get(name)
39         );
40     }else {
41         System.out.println("查找失败");
42     }
43 }
44 }

```

Fence 202

- `map.get()` 根据 `key` 查找 `value`。
- `map.put()` 添加键值对。
- `map.remove()` 根据 `key` 删除对应的键值对。删除成功则返回 `value`，否则返回 `null`。
- `containsKey()` 根据 `key` 查找是否含有对应的键值对，返回 `boolean` 值。

## 15.4 设计模式

针对考试，简单介绍单例模式与工厂模式。

### 15.4.1 单例模式

单例模式确保一个类只有一个实例，并提供一个全局访问点。这在需要控制某个资源的唯一性时非常有用，例如配置管理器、日志记录器或数据库连接池。

单例模式可以简单分为两种：懒汉式和饿汉式。

懒汉式在首次访问的时候才会创建，占用较小的内存，缺点就是一般情况下线程不安全。

饿汉式在类加载的时候就创建了对象，线程安全

## 15.4.1.1 应用场景

1. **配置管理**：应用程序的配置通常是全局唯一的，使用单例模式可以确保所有部分访问同一个配置实例。
2. **日志记录**：日志记录器通常需要统一管理日志输出，单例模式确保日志记录器的唯一性。
3. **数据库连接池**：管理数据库连接的池通常由单一实例负责，避免多个连接池的创建。
4. **设备驱动程序**：硬件设备的驱动程序通常需要一个唯一的实例来管理设备的访问。

## 15.4.1.2 优点

1. **控制实例数量**：确保系统中只有一个实例，节省资源。
2. **全局访问点**：提供全局访问点，方便在不同模块中访问。
3. **延迟实例化**：可以实现延迟实例化，即在第一次使用时才创建实例。

## 15.4.1.3 缺点

1. **隐藏的依赖性**：全局访问点可能导致代码中隐藏的依赖关系，增加维护难度。
2. **并发问题**：在多线程环境下，必须小心处理实例的创建，避免多线程同时创建多个实例。
3. **扩展性差**：由于单例限制了类的实例数量，可能会影响系统的扩展性。

## 15.4.1.4 示例

```
● ● ●  
1 public class GenericsDemo {  
2     public static void main(String[] args) {  
3         Singleton s1 = Singleton.getInstance();  
4         s1.addCnt();  
5         s1.addCnt();  
6         System.out.println(s1.getCnt());  
7         Singleton s2 = Singleton.getInstance();  
8         System.out.println(s2.getCnt());  
9     }  
10 }  
11  
12 class Singleton{  
13     private int cnt = 0;  
14     private static Singleton instance = null;  
15  
16     private Singleton(){  
17     }  
18 }
```

```
19
20     public static Singleton getInstance(){
21         if(instance == null){
22             instance = new Singleton();
23         }
24         return instance;
25     }
26
27     public void addCnt(){
28         cnt++;
29     }
30     public int getCnt(){
31         return cnt;
32     }
33 }
```

Fence 203

## 15.4.2 工厂模式

工厂模式通过定义一个用于创建对象的接口，让子类决定实例化哪一个类。它使得对象的创建延迟到子类进行，促进了代码的解耦。



```
1 abstract class Shape{
2     abstract void show();
3 }
4
5 class Circle extends Shape{
6     Circle(){
7
8     }
9     void show(){
10         System.out.println("This is a Circle");
11     }
12 }
13
14 class Square extends Shape{
15     Square(){
16
17     }
18     void show(){
19         System.out.println("This is a Square");
20     }
21 }
```

Fence 204

## 15.4.2.1 分类

常见的工厂模式有这几种：

- **简单工厂模式 (Simple Factory)** : 通过一个工厂类根据参数决定创建哪种对象。



```
1 public class GenericsDemo {
2     public static void main(String[] args) {
3         Shape tmp = Factory.CreateShape("Circle");
4         tmp.show();
5     }
6 }
7
8 class Factory{
9     public static Shape CreateShape(String type){
10        switch (type){
11            case "Circle" -> {
12                return new Circle();
13            }
14            case "Square"->{
15                return new Square();
16            }
17            default -> {
18                return null;
19            }
20        }
21    }
22 }
```

Fence 205

**工厂方法模式 (Factory Method)** : 定义一个创建对象的接口，由子类实现具体的创建过程。



```
1 public class GenericsDemo {
2     public static void main(String[] args) {
3         String type = "Circle";
4         Shape shape;
5         ShapeFactory factory= getFactory(type);
6         shape = factory.CreateShape();
7         shape.show();
8     }
9     public static ShapeFactory getFactory(String type){
10        switch (type){
11            case "Circle": return new CircleFactory();
12            case "Square": return new SquareFactory();
```

```

13         default: return null;
14     }
15
16 }
17 }
18
19 interface ShapeFactory{
20     Shape CreateShape();
21 }
22 class CircleFactory implements ShapeFactory{
23     // 注意使用public修饰
24     @Override
25     public Shape CreateShape(){
26         return new Circle();
27     }
28 }
29 class SquareFactory implements ShapeFactory{
30     @Override
31     public Shape CreateShape() {
32         return new Square();
33     }
34 }

```

Fence 206

### 15.4.2.2 应用场景

- 对象创建复杂**: 当对象的创建过程复杂，或者需要动态决定创建哪种对象时。
- 代码解耦**: 客户端不需要知道具体类的创建过程，只需要通过工厂接口获取对象。
- 扩展性强**: 需要增加新的产品类时，只需增加相应的工厂类，符合开放/关闭原则。
- 统一管理对象创建**: 集中管理对象的创建逻辑，便于维护和修改。

### 15.4.2.3 优点

**单一职责**: 将对象的创建职责集中到工厂类，提高代码的可维护性。

**扩展性强**: 新增产品类时，无需修改客户端代码，只需扩展工厂类。

**提高灵活性**: 通过工厂接口，可以灵活地切换不同的实现类。

### 15.4.2.4 缺点

**增加类的数量**: 引入工厂类会增加系统中的类数量，可能导致系统复杂度增加。

**代码理解难度**: 对于简单对象创建，引入工厂模式可能显得过于复杂，增加理解难度。

**维护成本**: 在产品种类频繁变化时，工厂类需要频繁修改，增加维护成本。

# 15.5 方法重写和重载

在Java中，**方法重写**（Override）和**方法重载**（Overload）是两种重要的多态性实现方式。

## 15.5.1 方法重写

方法重写是指子类在继承父类时，重新定义父类中已经存在的方法。重写的方法必须具有与被重写方法相同的方法签名（方法名、参数列表）和返回类型（Java 5及以后允许返回类型的协变）。

**注意点：**

- **访问权限**：子类重写方法的访问权限不能低于父类方法。例如，父类方法是 `public`，子类方法也必须是 `public`。
- **注解**：通常使用 `@Override` 注解标识，以增强代码的可读性和检查正确性。
- 重写发生在继承关系中。

## 15.5.2 方法重载

方法重载是指在同一个类中，定义多个方法名相同但参数列表不同（参数个数、类型或顺序不同）的方法。重载的方法可以具有不同的返回类型和访问修饰符。

**注意：**只靠返回类型无法判断方法重载

## 15.5.3 方法重写与方法重载的区别

特性	方法重写（Override）	方法重载（Overload）
<b>发生位置</b>	子类继承父类时，子类中重新定义父类的方法。	同一个类中，多个方法名相同但参数列表不同。
<b>方法签名</b>	方法名和参数列表相同。	方法名相同，参数列表不同（参数个数、类型或顺序）。
<b>返回类型</b>	必须相同或协变。	可以相同或不同，但不能仅通过返回类型区分重载。
<b>访问权限</b>	子类方法的访问权限不能低于父类方法。	无限制，可以有不同的访问修饰符。

特性	方法重写 (Override)	方法重载 (Overload)
多态性类型	实现运行时多态。	实现编译时多态。
注解	通常使用 <code>@Override</code> 注解。	不需要特定注解。
用途	修改或扩展父类的行为。	实现同一操作的不同实现方式。

Table 17

### 15.5.3.1 方法重写的作用

1. **实现多态性**：通过重写，程序可以在运行时根据对象的实际类型调用相应的方法，增强系统的灵活性和可扩展性。
2. **代码复用与维护**：子类可以重用父类的代码，同时根据需要修改特定的行为，简化代码维护。
3. **接口实现**：当类实现接口或抽象类时，必须重写接口或抽象类中定义的方法，确保类具备接口规定的行为。

### 15.5.3.2 方法重载的作用

1. **增强方法的可用性**：通过提供多种参数组合的同名方法，满足不同的调用需求，提升API的友好性。
2. **简化代码**：避免为相似功能定义不同的方法名，减少代码量，提高可读性。

15.5.4 创建一个类 `Calculator`，包含方法 `add(int a, int b)` 和 `add(double a, double b)`，展示方法重载。然后在子类 `ScientificCalculator` 中重写 `add` 方法，添加额外的功能。



```

1 class Calculator{
2     int add(int a, int b){
3         return a + b;
4     }
5     double add(double a, double b){
6         return a + b;
7     }
8 }
```

```
9 class ScientificCalculator extends Calculator{  
10     @Override  
11     public int add(int a, int b) {  
12         int sum = super.add(a, b);  
13         System.out.println("平方和: " + (sum * sum));  
14         return sum;  
15     }  
16     @Override  
17     public double add(double a, double b) {  
18         System.out.println("ScientificCalculator: 执行双精度浮点数加法,  
并显示平方结果");  
19         double sum = super.add(a, b);  
20         System.out.println("平方和: " + (sum * sum));  
21         return sum;  
22     }  
23 }
```

Fence 207

# 16. 016

## 16.1 接口的多实现

### 16.1.1 解释为什么一个Java接口可以被多个类实现，并讨论这种设计的好处

**接口**是一种抽象类型，用于定义一组方法的签名（即方法名、参数列表和返回类型），而不提供方法的具体实现。接口用于规范类的行为，确保实现接口的类提供特定的方法。

#### 1. 实现多态性 (Polymorphism)

通过接口，可以强行使子类实现接口中的函数，从而实现多态性。

#### 2. 提高代码的灵活性和可维护性

接口定义了行为的规范，而具体的实现可以根据需求进行更改或扩展。这种设计使得代码更加灵活，易于维护和升级。

16.1.2 定义一个接口 `Flyable`，包含一个方法 `fly()`。创建两个类 `Bird` 和 `Airplane` 实现这个接口，并在主方法中创建它们的实例，调用 `fly()` 方法。

```
● ● ●  
1 interface Flyable{  
2     void fly();  
3 }  
4  
5 class Bird implements Flyable{  
6     @Override  
7     public void fly(){  
8         System.out.println("Bird Fly");  
9     }  
10 }  
11  
12 class AirPlane implements Flyable{  
13     @Override  
14     public void fly(){  
15         System.out.println(  
16             "Airplane Fly"  
17         );  
18     }  
19 }
```

Fence 208

## 16.2 多态性的应用

### 16.2.1 多态性的概念

多态指的是同一个方法或操作作用于不同的对象时，产生不同的行为。这种差异在编译阶段时时体现不出来的，只有在运行阶段才能真正地体现出同一个方法在不同对象上的差异。

常用的实现方法就是 **方法重写** 和 **方法重载**。

16.2.2 创建一个基类 Animal 和一个子类 Dog。Animal 类有一个方法 makeSound()，在 Dog 类中重写这个方法。创建一个 Animal 数组，包含 Dog 对象，并调用 makeSound() 方法。



```
1 class Animal {
2     public void makeSound() {
3         System.out.println("Animal makes a sound");
4     }
5 }
6
7 class Dog extends Animal {
8     @Override
9     public void makeSound() {
10         System.out.println("Dog barks");
11     }
12 }
13
14 public class InterfaceDemo {
15     public static void main(String[] args) {
16         Animal[] animals = new Animal[5];
17         animals[0] = new Dog();
18         for(int i = 1; i < animals.length; i++){
19             animals[i] = new Animal();
20         }
21         for (Animal animal : animals) {
22             animal.makeSound();
23         }
24     }
25 }
26 }
```

Fence 209

## 16.3 组合/聚合关系

组合和聚合是java中的两种常见的对象关系，尽管都是一个类包含了另外一个类，但是具体的生命周期有所不同。

## 16.3.1 组合关系

组合是一种强关系，其中一个对象包含另一个对象，而且被包含的对象的生命周期由包含它的对象管理。如果容器对象被销毁，那么它包含的对象也会被销毁。

**使用场景：**

- 当一个对象（如 `Car`）不能独立存在时，它需要包含另一个对象（如 `Engine`）。这表示对象之间的强依赖关系，通常用于表示“整体-部分”的关系。
- 适用于表示一种“生命周期依赖”的关系。

**示例：**

设计一个Car类，其中包含 `Engine` 类的实例作为属性（组合关系）。

```
● ● ●
1 class Engine{
2     void run(){
3         System.out.println("Engine run");
4     }
5 }
6
7 class Car{
8     Engine engine = new Engine();
9     void carStart(){
10        engine.run();
11        System.out.println("Car Start");
12    }
13 }
14
```

Fence 210

## 16.3.2 聚合关系

聚合是一种较弱的关系，表示一个对象是另一个对象的组成部分，但这些部分对象在逻辑上是独立的。被聚合的对象的生命周期不依赖于聚合对象的生命周期。

**使用场景：**

- 当一个对象（如 `TrafficLight`）拥有多个子对象（如 `Car`），但这些子对象在逻辑上是独立的，且可以独立于容器对象存在时。
- 适用于表示“部分-整体”的关系，但部分对象的生命周期不依赖于整体对象。

```
● ● ●
1 public class InterfaceDemo{
2     public static void main(String[] args) {
3         Car car1 = new Car("Car1");
```

```

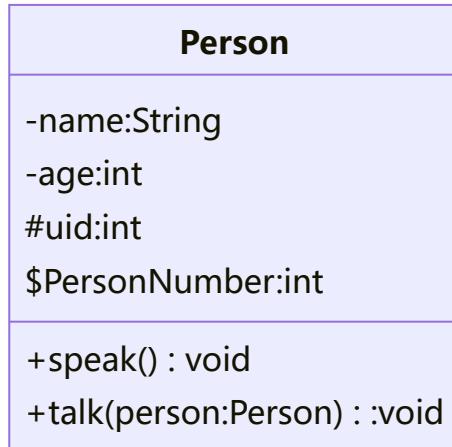
4     Car car2 = new Car("Car2");
5     Car car3 = new Car("Car3");
6     Road(car1, car2, car3);
7     car1.carStart();
8 }
9 public static void Road(Car... cars){
10    TrafficLight trafficLight = new TrafficLight();
11    for(var car : cars){
12        trafficLight.add(car);
13    }
14    trafficLight.GreenLed();
15 }
16 }
17
18 class Engine{
19     void run(){
20         System.out.println("Engine run");
21     }
22 }
23
24 class Car{
25     String name;
26     Engine engine = new Engine();
27     Car(){
28
29     }
30     Car(String name){
31         this.name = name;
32     }
33     void carStart(){
34         engine.run();
35         System.out.println(name + " Start");
36     }
37 }
38
39 class TrafficLight{
40     private List<Car> waitList = new ArrayList<>();
41     void add(Car car){
42         waitList.add(car);
43     }
44     void GreenLed(){
45         for(Car car: waitList){
46             car.carStart();
47         }
48     }
49 }
50

```

## 16.3.3 UML图

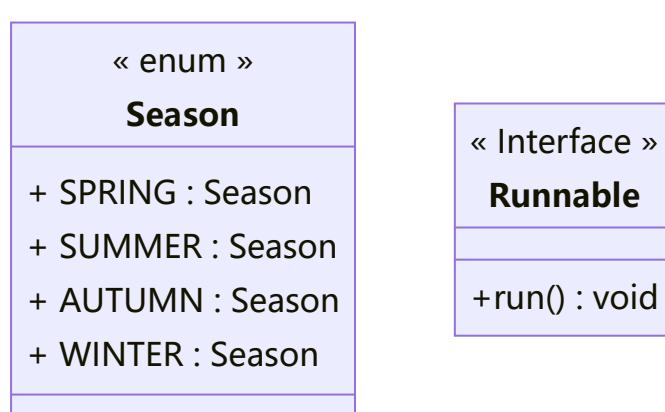
**UML类图 (Unified Modeling Language Class Diagram)** 是一种用于表示系统中类 (Class) 、它们的属性 (Attributes) 、方法 (Methods) , 以及类之间关系的图形化模型。

### 16.3.3.1 基本表示



### 16.3.3.2 类注释

用于标记一个类的元素据，以 `<<` 开始，以 `>>` 结束，如 `<<interface>>`。



### 16.3.3.3 关系基数

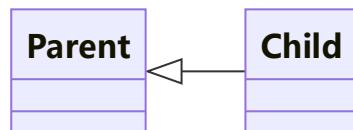
关系基数主要用于 聚合 与 组合，表名类与类之间的关联关系。

基数	含义
1	有且只有1个
0..1	0个或1个
1..*	1个或多个
*	多个
n	n个, n大于1
0..n	0至n个, n大于1
1..n	1至n个, n大于1

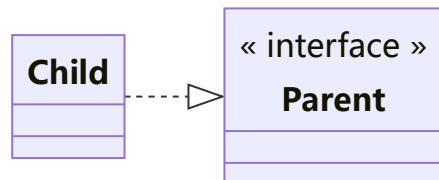
Table 18

#### 16.3.3.4 类关系

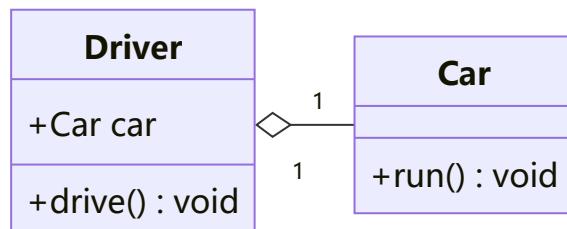
- 继承



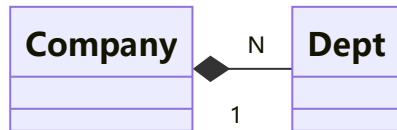
- 实现



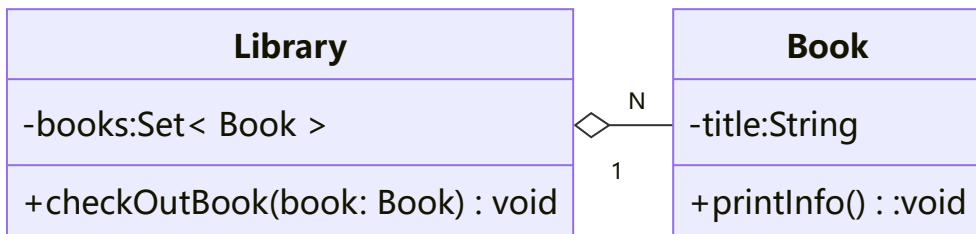
- 聚合



- 组合



**16.3.4 根据以下描述绘制一个UML类图：一个Library类包含Book类的集合，Book类有一个String类型的title属性和一个 printInfo() 方法。 Library 类 有 一 个 checkOutBook(Book book)方法。**



## 16.4 接口与抽象类的区别

详见抽象类：[跳转连接](#)

### 16.4.1 接口的使用场景

#### 1. 定义行为规范（契约）：

- 接口适合用来定义类的行为规范，不涉及具体实现。
- 例如：`List`、`Set` 和 `Map` 等集合框架类都实现了 `Collection` 接口。

#### 2. 多继承需求：

- 当一个类需要继承多个功能时，可以使用接口。
- 例如：一个类既需要实现 `Serializable`，又需要实现 `Comparable`。

#### 3. 松耦合设计：

- 接口使得代码与实现解耦，便于扩展和测试。
- 例如：通过接口定义服务，再用不同类提供具体实现。

16.4.2 创建一个抽象类 `Shape`，包含一个抽象方法 `draw()`。然后创建一个接口 `Colorable`，包含一个方法 `setColor(String color)`。实现这两个抽象概念的 `Circle` 类，它既是一个 `Shape` 也是一个 `Colorable`。



```
1 abstract class Shape{
2     abstract void draw();
3 }
4
5 interface Colorable{
6     void setColor(String color);
7 }
8
9 class Circle extends Shape implements Colorable{
10    String color;
11    public void setColor(String color){
12        this.color = color;
13    }
14    public void draw(){
15        System.out.println("画一个"+color+"的Circle");
16    }
17 }
```