



主讲教师 张 智  
计算机学院软件工程系  
课程群: 421694618

## 7 抽象类和接口

### 7.1 抽象类

### 7.2 接口

### 7.3 匿名类

### 7.4 Lambda表达式

### 7.5 函数式接口

### 【附录】枚举类

## 7.1 抽象类 -- 引例

- 父类Shape：无属性，面积为0；
- 子类：矩形类(Rectangle)和圆(Circle)类，有自有属性，能求面积
- 多态方式测试。

```
public class Shape {  
    public double getArea() {  
        return 0.0;  
    }  
}
```

这个返回值，意义不大  
改成abstract抽象方法

```
class Rectangle extends Shape {  
    private int length;  
    private int width;  
    public Rectangle() {  
    }  
    public Rectangle(int length, int width) {  
        this.length = length;  
        this.width = width;  
    }  
    public double getArea() {  
        return length * width;  
    }  
}
```

```
class Circle extends Shape {  
    private double r;  
    public Circle() {  
    }  
    public Circle(double r) {  
        this.r = r;  
    }  
    public double getArea() {  
        return Math.PI * r * r;  
    }  
}
```

```
class Test {  
    public static void main(String[] args) {  
        Shape s = new Rectangle(3,4);  
        //Shape s = new Circle(1);  
        System.out.println( s.getArea() );  
    }  
}
```

## 抽象类概念

- Java类可定义一些不含方法体的方法，它的实现交给子类根据自己的情况去实现，这样的方法就是抽象方法，包含抽象方法的类叫抽象类。

简单来说：如果一个类中没有包含足够的信息来描绘一个具体对象，那么这样的类称为抽象类

- 抽象关键字：**abstract**

## 抽象类特点

- 抽象类只是一个类型的部分实现，所以不能被实例化(不能new对象)。 (那有啥用?)
- 抽象方法没有方法体，必须存在于抽象类中。
- `abstract`不能用于`static`方法或者构造函数，也不能与`private`、`final`共同修饰同一个方法。 (想想)
- 子类可以继承抽象父类，一般要重写父类所有的抽象方法，如果没有，则该子类还要声明为抽象类。

## 抽象类示例

```
public abstract class A {
```

```
    private int i=1;
```

```
    abstract int aa(int x,int y);
```

```
    abstract void bb();
```

```
    public void cc() {
```

```
        System.out.println(i);
```

```
    }
```

```
}
```

注：如果子类没有重写父类所有的抽象方法，则该子类还要声明为抽象类

```
class B extends A {
```

```
    int aa(int x,int y) { return x+y; }
```

```
    void bb() { } //空操作也算实现
```

```
}
```

实现抽象方法

实现抽象方法

一个抽象类，可以有 0~n 个抽象方法，以及 0~n 个具体方法

注意 0

## 抽象类示例 (带构造函数的抽象类)

### ■ 抽象类Animal:

- 公有属性: animalName(String)
- 2个构造函数: Animal()、Animal(String)
- 抽象方法: eat()

### ■ 子类: Dog和Cat类

- 各自都有构造函数: Dog()/Dog(String)、Cat()/Cat(String)
- 各自实现抽象方法: eat()
- 分别输出: 某某狗吃骨头!, 某某猫吃老鼠!



## 抽象类 -- Animal

```
public abstract class Animal {  
    public String animalName;
```

```
    public Animal() {  
    }
```

抽象类可以有构造函数，但不能声明为抽象

```
    public Animal(String animalName) {  
        this.animalName = animalName;  
    }
```

```
    public abstract void eat();  
}
```

抽象方法

## Dog类和Cat类

```
public class Dog extends Animal {
```

```
    public Dog() {
```

```
        //super();
```

```
    }
```

```
    public Dog(String s) {
```

```
        super(s);
```

```
    }
```

```
    public void eat() {
```

```
        System.out.println(animalName + "吃骨头!");
```

```
    }
```

```
}
```

调用父类默认构造函数

```
public class Cat extends Animal {
```

```
    public Cat() {
```

```
        //super();
```

```
    }
```

```
    public Cat(String s) {
```

```
        super(s);
```

```
    }
```

```
    public void eat() {
```

```
        System.out.println(animalName + "吃老鼠!");
```

```
    }
```

```
}
```

显式调用父类构造函数

实现抽象方法

## 测试类

```
public class Test {  
    public static void main(String args[]) {  
        //Animal a = new Animal();    //报错  
        Animal a = new Dog("哈士奇");  
        //Animal a = new Cat("大橘");  
        a.eat();  
    }  
}
```

原因：抽象类可声明对象(多态)，不能被实例化(不能new对象)

## 抽象类归纳：

一个抽象类，可以有 0~n 个抽象方法，以及 0~n 个具体方法

- 含有抽象方法的类必须被声明为抽象类。但抽象类中不一定包含的都是抽象方法
- 抽象类可以有构造方法，但构造方法不能声明为抽象★
- 抽象类提供一个类型的部分实现，所以不能被实例化(即不能用new去产生对象)，但可声明对象(用于多态) ★
- 抽象类不能用final来修饰，即一个类不能既是最终类又是抽象类
- 抽象类的子类必须重写所有抽象方法后才能被实例化，否则子类还是个抽象类
- 抽象方法只需声明，而不需实现
- abstract不能与private、static、final并列修饰同一个方法

[【返回】](#)

## 7.2 接口

- [接口概念](#)
- [接口常量](#)
- [接口default/static方法](#)
- [多接口](#)
- [接口继承多接口](#)
- [接口特点总结](#)

[【返回】](#)

## 1. 接口概念

### ■ 接口是一种特殊的抽象类：

- 如果一个抽象类中的所有方法都是抽象的，这个类就定义为接口 (interface)  
Java8以前版本
- 接口的所有方法通常由子类全部实现 (implements)，不同子类的实现可以具有不同的功能  
不是extends
- 如果一个类没有全部实现某个接口的所有方法，则这个类必须声明为抽象的

## 接口示例1

- 定义Shape接口，包含两个抽象方法：

- `double getPerimeter();`   //求周长

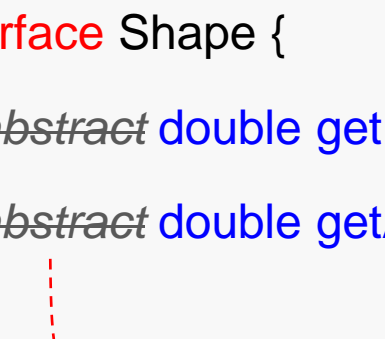
- `double getArea();`         //求面积

- 两个实现类：

- 矩形类(Rectangle)和圆(Circle)类

- 有自有属性，能求周长和面积

## Shape接口：



```
public interface Shape {  
    public abstract double getPerimeter();    //求周长  
    public abstract double getArea();        //求面积  
}
```

public abstract可以省略，即接口内方法默认是公有、抽象的



## Rectangle和Circle类：

实现接口

```
class Rectangle implements Shape {  
    private double height;  
    private double width;  
    public Rectangle(double height, double width) {  
        this.height = height;  
        this.width = width;  
    }  
    @Override  
    public double getPerimeter() {  
        return 2 * (height + width);  
    }  
    @Override  
    public double getArea() {  
        return height * width;  
    }  
}
```

实现接口所有方法

```
class Circle implements Shape {  
    private double radius;  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    @Override  
    public double getPerimeter() {  
        return 2 * Math.PI * radius;  
    }  
    @Override  
    public double getArea() {  
        return Math.PI * radius * radius;  
    }  
}
```

## 测试类

```
public class Test {  
    public static void main(String[] args) {  
        Shape r = new Rectangle(10,10);  
        // Shape r=new Circle(10);  
        System.out.println("矩形面积="+ r.getArea() );  
        System.out.println("矩形周长="+ r.getPerimeter() );  
    }  
}
```

多态体现

Shape s = new Shape(); ×接口也不能被实例化

思考：什么时候用抽象类，什么时候用接口？

## 接口示例2

- 定义一个整形堆栈IntStack接口：

- 包含2个方法：void push(int item) 和 int pop()

- 编写两个实现类：

- (1) FixedStack实现类：实现一个固定长度的整数堆栈

- (2) DynStack实现类：实现一个动态栈，每个栈都以一个初始长度构造，如果初始化长度被超出，则堆栈的大小将成倍增加（原有数据要保留）

```

public interface IntStack {
    void push(int item);
    int pop();
}

class FixedStack implements IntStack {
    private int stck[];
    private int pos;
    public FixedStack( int size ) {
        stck = new int[size];
        pos = -1;
    }
    @Override
    public int pop() {
        if (pos < 0) {
            System.out.print("Stack is underflow.");
            return -1;    //不是特别好
        }
        else
            return stck[pos--];
    }
}

```

```

@Override
public void push(int item) {
    if (pos == stck.length - 1)
        System.out.print("Stack is full.");
    else
        stck[++pos] = item;
}
}

class TestStack1 {
    public static void main(String[] args) {
        IntStack fixedStack = new FixedStack(5);
        for (int i = 0; i < 5; i++)
            fixedStack.push(i);
        System.out.print( "Stack in mystack:" );
        for (int i = 0; i < 5; i++)
            System.out.print( fixedStack.pop() + " " );
    }
}

```

运行结果

```
Stack in mystack:4 3 2 1 0
```

```

class DynStack implements IntStack {
    private int stck[];
    private int pos;

    public DynStack( int size ) {
        stck = new int[size];
        pos = -1;
    }

    @Override
    public int pop() {
        if (pos < 0) {
            System.out.print("Stack is underflow.");
            return -1;
        } else
            return stck[pos--];
    }

    @Override
    public void push(int item) {
        if (pos == stck.length - 1) {
            int temp[] = new int[stck.length * 2];

```

```

            for (int i = 0; i < stck.length; i++)
                temp[i] = stck[i];    //转存
            stck = temp;
            stck[++pos] = item;
        } else
            stck[++pos] = item;
    }
}

class TestStack2 {
    public static void main(String[] args) {
        IntStack dynStack = new DynStack(5);
        for (int i = 0; i < 10; i++)
            dynStack.push(i);
        System.out.print("Stack in mystack:");
        for (int i = 0; i < 10; i++)
            System.out.print(dynStack.pop() + " ");
    }
}

```

运行结果

Stack in mystack:9 8 7 6 5 4 3 2 1 0

[【返回】](#)

## 2. 接口常量

- 接口中不能有变量，但可有常量
- 常量默认是 `public static final` 类型（公有全局静态常量），且必须被显示初始化。

why

- 为什么不能定义变量：由于接口中的方法都是抽象的，如果接口可以定义变量，那么在接口中无法通过行为(方法)来修改属性
- 为什么是public static：接口提供的是一种统一的"协议"，接口中的属性也属于"协议"成员，所有实现类都可以共享这一"协议"
- 为什么是final：实现接口的对象如果修改了接口中的属性，那么所有对象也都会自动拥有这一改变后的值，这和接口提供的统一的抽象这种思想相抵触

## 接口常量示例：

接口常量调用方法：接口名.常量

```
public interface A {  
    public static final int END=0; // OK 标准写法  
    private final int OK=1; // × 不能为private  
    int a; // × 没有初始化属于变量，接口不能有变量  
    注意→ int b=1; // OK 初始化的变量会自动被识别为public static final常量  
    public final int NO=-1; // OK 不写也默认是static  
    static int share=0; // OK 不写也默认是public、final  
}
```

建议public static final都写全

## 接口常量用法

```
public interface Person {  
    public static final int MALE = 1;  
    public static final int FEMALE = 2;  
}
```

简写



```
public interface Person {  
    int MALE = 1;  
    int FEMALE = 2;  
}
```

使用接口常量：

```
if( gender == Person.MALE ) {  
    ...  
}  
if( gender == Person.FEMALE ) {  
    ...  
}
```

接口.常量

[【返回】](#)



### 3. 接口default/static方法（了解）

- Java 8 以后，接口可添加 default 或者 static 方法 (有实现的)



**default方法：**实现接口的类可以继承，可以选择重写 (注：多接口实现如有同名default方法则必须重写)

**static方法：**实现接口的类或者子接口不会继承(不能重写)，用接口.方法名()调用

## 接口default方法:

```
public interface A {  
    public default void foo(){  
        System.out.println("这是default方法");  
    }  
}  
  
class B implements A {  
    public void foo() {  
        System.out.println("重写default方法");  
    }  
}  
  
class Test {  
    public static void main(String[] args) {  
        A a = new B();  
        a.foo(); // 多态, 输出 重写default方法  
    }  
}
```

default不能省略

实现类可以继承default方法或根据需要重写 (重写时不加default)

## 接口static方法：

```
public interface A {  
    public static void a(){  
        System.out.println("这是A");  
    }  
}  
  
public class B implements A {  
}  
  
public class test {  
    public static void main(String[] args) {  
        A a = new B();  
        a.a(); // × ← 实现类不会继承static方法(不能重写)  
        A.a(); // 正确调用  
    }  
}
```

static不能省略

[【返回】](#)

## 4. 多接口

- Java一个类可以实现多个接口，从而间接的实现多继承。

```
interface A {  
    double fun1();  
}
```

```
Interface B {  
    double fun2();  
}
```

用逗号分隔

```
class C implements A , B {
```

```
    public double fun1() {  
        //实现方法  
    }
```

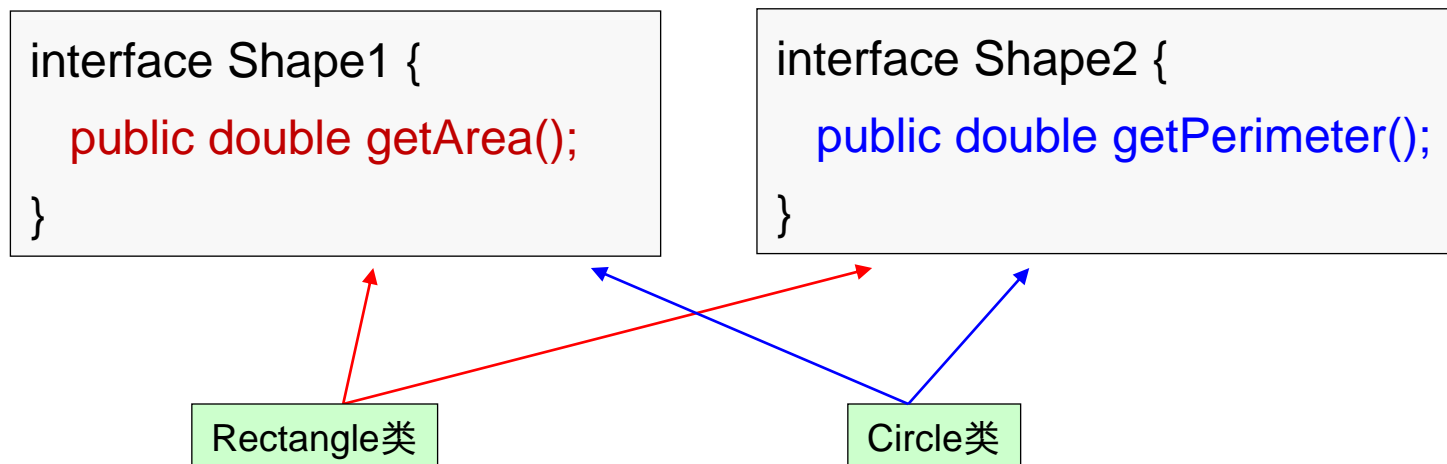
```
    public double fun2() {  
        //实现方法  
    }
```

```
}
```

实现类要实现所有接口的所有方法否则要声明为抽象类

## 多接口编程

- 将Shape接口用多接口实现：



## Rectangle类实现多接口：

```
class Rectangle implements Shape1, Shape2 {
```

← 实现多接口

```
    private double height;
```

```
    private double width;
```

```
    public Rectangle(double height, double width) {
```

```
        this.height = height;
```

```
        this.width = width;
```

```
    }
```

```
    @Override
```

```
    public double getArea() {
```

```
        return height * width;
```

```
    }
```

```
    @Override
```

```
    public double getPerimeter() {
```

```
        return 2 * (height + width);
```

```
    }
```

```
}
```

实现Shape1接口的所有方法

实现Shape2接口的所有方法

注：内部代码没有任何改变

## Circle类实现多接口：

```
class Circle implements Shape1, Shape2 {
```

← 实现多接口

```
    private double radius;
```

```
    public Circle(double radius) {
```

```
        this.radius = radius;
```

```
    }
```

```
    @Override
```

```
    public double getArea() {
```

```
        return Math.PI * radius * radius;
```

```
    }
```

```
    @Override
```

```
    public double getPerimeter() {
```

```
        return 2 * Math.PI * radius;
```

```
    }
```

```
}
```

实现shape1接口的所有方法

实现shape2接口的所有方法

注：内部代码没有任何改变

[【返回】](#)

## 5. 接口继承多接口

Java类只支持单继承，不支持多继承  
但接口可以继承多个其他接口

```
public interface A {  
    void fa();  
}
```

```
public interface B {  
    void fb();  
}
```

```
public interface C extends A, B {  
    void fc();  
}
```

用逗号分隔

接口继承多个其它的接口

```
public class E implements C {
```

...//必须实现接口(包括父接口)所有的方法，如fa()、fb()、fc()

```
}
```

[【返回】](#)



## 6. 接口的特点总结

- 接口中的方法默认都是 **public abstract** 类型(可省略)，没有方法体
- 接口不能有构造函数，不能被实例化 注:抽象类可以有构造函数
- 接口中不能有变量，常量默认是 **public static final**，必须被显示初始化
- 接口中不能包含静态抽象方法，可以有default 或者 static 方法 (Java 8以后)

```
public interface A {  
    int var=1;                // OK,实际是public static final常量  
    public A() { }            // 报错,接口不能有构造函数  
    void method1() { }        // 报错,接口抽象方法没有方法体  
    protected void method2(); // 报错,接口方法必须是public abstract  
    static void method3();    // 报错,接口不能包含静态抽象方法  
    static void method4() { } // OK,接口可以包含静态方法  
    default void method5() { } // OK,接口可以包含default方法  
}
```

## 接口的特点(续)

- 当一个类实现某个接口时，它**必须重写**这个接口的所有抽象方法(包括这个接口的父接口中的方法)，**否则这个类必须声明为抽象的。**
- 一个类可以实现多个接口 → `class C implements A , B { ... }`
- 一个接口可继承多个其它的接口 ★ → `interface C extends A, B { ... }`
- 一个接口不能实现(implements)另一个接口 → `interface a implements b{ ... }` //错，接口不能实现接口

## 接口的特点(续)

- 类在继承父类的同时，可实现一个或多个接口，但`extends`关键字必须位于`implements`关键字之前：

```
public class A extends B implements C, D {...}
```

↑  
继承在接口实现前

- 接口不能实例化，但允许定义接口类型的引用变量，该引用变量可引用实现该接口的类的实例(多态)

## 抽象类和接口对比：

No.	区别点	抽象类	接口
1	定义	包含抽象方法的类	主要是抽象方法和全局常量的集合
2	组成	构造方法、抽象方法、普通方法、常量、变量	常量、抽象方法、(jdk8.0:默认方法、静态方法)
3	使用	子类继承抽象类(extends)	子类实现接口(implements)
4	关系	抽象类可以实现多个接口	接口不能继承抽象类，但允许继承多个接口
5	常见设计模式	模板方法	简单工厂、工厂方法、代理模式
6	对象	都通过对象的多态性产生实例化对象	
7	局限	抽象类有单继承的局限	接口没有此局限
8	实际	作为一个模板	是作为一个标准或是表示一种能力
9	选择	如果抽象类和接口都可以使用的话，优先使用接口，因为避免单继承的局限	

牛客@CoderMa

[【返回】](#)

## 7.3 匿名类

- **匿名类**：是指没有类名的内部类，必须在创建时使用 new 语句来声明
- 语法形式如下：

编译成功后还是会安排一个特殊名字

↓

```
ClassName 对象名 = new ClassName() {  
    // 方法  
};
```

左边是父类(如抽象类、接口或普通父类都可以)

右边new创建匿名类

匿名类

匿名类内部要重写父类方法或实现接口方法

分号结束

- 匿名类通常继承一个父类或实现一个接口
- 匿名类使得代码更加简洁、紧凑，模块化程度更高

## 常规写法

```
public interface Person { ← 接口
    void speak();
}
```

普通实现类

```
class Boy implements Person {
    @Override
    public void speak() {
        System.out.print("boy");
    }
}
```

```
class Test {
    public static void main(String[] args) {
        Person p = new Boy();
        p.speak();
    }
}
```

实现接口方法

## 匿名类写法

```
class Boy implements Person { ← 无需定义有名字的实现类
}
```

```
class Test {
    public static void main(String[] args) {
        Person p = new Person() {
            @Override
            public void speak() {
                System.out.print("boy");
            }
        };
        p.speak();
    }
}
```

匿名类实现

注：匿名类实质是一个内部类，匿名类编译生成的类名为：Test\$1.class

数字是匿名类序号值

## 匿名类特点：

- 匿名类总是一个内部类，并且不能是static的
- 匿名类不能是抽象的，必须要实现继承的类或者实现的接口的所有抽象方法
- 匿名类总是隐式的final（不能继承）
- 匿名内部类中是不能定义构造函数的（但可用{ 代码块 } 进行初始化）
- 匿名内部类中不能存在静态成员变量和静态方法(静态常量可以有 static final)
- 匿名类的class文件命名是：主类\$1,2,3.....

[【返回】](#)

## 7.4 Lambda表达式

- Lambda 表达式是一个匿名函数（也称为箭头函数），基于数学中的 $\lambda$ 演算得名，现在很多语言都支持 Lambda 表达式
- 使用 Lambda 表达式能使代码更简洁紧凑，Lambda语法格式：

参数

函数主体

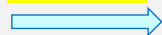
```
(类型 参数) -> { 代码语句; }
```



Lambda 表达式示例

```
(int a, int b) -> {  
    int c = a + b;  
    return c;  
}
```

进一步省略



```
(a, b) -> a + b
```



## 示例：

- `() -> 5`      // 无参数,返回值为 5    `return`都被省略
- `(int x, int y) -> x + y`    // 2个int参数，返回和
- `(x, y) -> x - y`    // 2个参数(数字)，返回差值
- `x -> 2 * x`      // 接收一个参数(数字)，返回其2倍    `x`仅仅是一个形参名
- `(String s) -> System.out.print(s)`    // String参数，只输出，无返回值

### 省略说明：

- 类型：不需要声明参数类型，编译器可以统一识别参数值。
- 参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。
- 可选的大括号：如果主体包含了一个语句，就不需要使用大括号。
- `return`关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定明表达式返回了一个数值。

【返回】

## 7.5 函数式接口

- 函数式接口：有且只有一个抽象方法的接口
- **@FunctionalInterface**注解：当接口中声明的抽象方法多于或少于一个时就会报错

```
@FunctionalInterface  
interface Person {  
    void speak() ;  
    void eat() ;  
}
```

添加函数式接口注解 (注意：报错)

原因：函数式接口抽象方法多于或少于1个

## 函数式接口的匿名类实现：

@FunctionalInterface

```
interface Person {  
    void speak();  
}
```

← 函数式接口

```
public class Test {  
    public static void main(String[] args) {  
        Person p = new Person() {  
            @Override  
            public void speak() {  
                System.out.println("boy");  
            }  
        };  
        p.speak();  
    }  
}
```

匿名类实现

↓  
引入Lambda表达式后代码会进一步简化(下页)

## 函数式接口的Lambda表达式实现

- Lambda 表达式简化了匿名类使用的方法，给予Java强大的函数化编程能力

@FunctionalInterface

```
interface Person {  
    void speak();  
}
```

← 函数式接口

```
public class Test {  
    public static void main(String[] args) {  
        Person p = ()->System.out.println("boy");  
        p.speak();  
    }  
}
```

↑  
Lambda表达式

用法理解说明：

- "="左边确定是Person这个函数式接口，由于函数式接口只有1个抽象方法，因此"="右边Lambda表达式这个匿名函数就只能实现这个speak()方法。
- 另外如果接口方法有形参(类型)，Lambda表达式也能自动侦测，所以Lambda表达式的参数类型都可以省略

## Lambda表达式示例2

@FunctionalInterface

```
interface MathOperation { ← 函数式接口
    int operation(int a, int b);
}
```

```
public class Test {
    public static void main(String[] args) {
        MathOperation add = (int a, int b) -> { return a + b; };
        MathOperation sub = (a, b) -> a - b ;
        MathOperation multi = (a, b) -> a * b ;
        MathOperation div = (a, b) -> a / b ;
        System.out.println("10 + 5 = " + add.operation(10, 5) );
        System.out.println("10 - 5 = " + sub.operation(10, 5) );
        System.out.println("10 x 5 = " + multi.operation(10, 5) );
        System.out.println("10 / 5 = " + div.operation(10, 5) );
    }
}
```

Lambda表达式

运行结果

```
10 + 5 = 15
10 - 5 = 5
10 x 5 = 50
10 / 5 = 2
```

[【返回】](#)

## 【附录】枚举类：enum

- 枚举的本质是类，枚举屏蔽了枚举值的类型信息
- enum类型继承自java.lang.Enum，且无法被继承
- 枚举是用来构建常量数据结构的模板，这个模板可扩展
- 枚举的使用增强了程序的健壮性，比如在引用一个不存在的枚举值的时候，编译器会报错（针对接口常量）

## 枚举类示例：

```
public enum Gender {  
    MALE, FEMALE;  
}
```

Gender.java

枚举值用逗号分隔，无类型信息


```
Gender gender = Gender.MALE; // 定义枚举变量，无需new
```

```
if( gender == Gender.MALE ) {  
    System.out.println("男生");  
}
```

枚举比较直接用== (equals也可以)

此时：if ( gender == 0 ) 编译就会报错，程序健壮性得到加强

## 枚举类在switch中使用：

```
public static void show( Gender gender ) {  
    switch ( gender ) {  
        case MALE:    
            System.out.println("男生");  
            break;  
        case FEMALE:  
            System.out.println("女生");  
            break;  
        default:  
            System.out.println("性别不明");  
            break;  
    }  
}
```

直接使用枚举值，不能用：Gender.MALE ×



## 枚举类的两个方法

- `name()`: 返回枚举常量名
- `ordinal()`: 返回常量的序号值, 默认从0开始

```
Gender gender = Gender.FEMALE;  
System.out.println( gender.name() );    // FEMALE  
System.out.println( gender.ordinal() );  // 1
```

## 枚举类的扩展：

```
public enum Gender {  
    MALE(100, "男生"), FEMALE(200, "女生");  
    private int index;  
    private String name;  
    private Gender(int index, String name) {  
        this.index = index;  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public int getIndex() {  
        return index;  
    }  
}
```

① 枚举值先添加扩展，括号里的值在下面定义

② 扩展信息对应的变量和类型

③ 枚举构造函数(一般用私有)

④ 根据需要定义成员方法

和  
class  
定义  
一样

## 枚举类扩展测试：

```
Gender gender = Gender.FEMALE;  
System.out.println( gender.getName() );    // 女生  
System.out.println( gender.getIndex() );    // 200  
System.out.println( gender.name() );        // FEMALE  
System.out.println( gender.ordinal() );      // 1
```

【完】