

# 1.接口

## 1.1黑马信息管理系统集合改进 (应用)

- 使用数组容器的弊端
  1. 容器长度是固定的，不能根据添加功能自动增长
  2. 没有提供用于增删改查的方法
- 优化步骤
  1. 创建新的StudentDao类， OtherStudentDao
  2. 创建ArrayList集合容器对象
  3. OtherStudentDao中的方法声明，需要跟StudentDao保持一致  
注意：如果不一致，StudentService中的代码就需要进行修改
  4. 完善方法（添加、删除、修改、查看）
  5. 替换StudentService中的Dao对象
- 代码实现

OtherStudentDao类

```
public class OtherStudentDao {  
    // 集合容器  
    private static ArrayList<Student> stus = new ArrayList<>();  
  
    static {  
        Student stu1 = new Student("heima001", "张三", "23", "1999-11-11");  
        Student stu2 = new Student("heima002", "李四", "24", "2000-11-11");  
  
        stus.add(stu1);  
        stus.add(stu2);  
    }  
  
    // 添加学生方法  
    public boolean addStudent(Student stu) {  
        stus.add(stu);  
        return true;  
    }  
  
    // 查看学生方法  
    public Student[] findAllStudent() {  
  
        Student[] students = new Student[stus.size()];  
  
        for (int i = 0; i < students.length; i++) {  
            students[i] = stus.get(i);  
        }  
  
        return students;  
    }  
}
```

```

    }

    public void deleteStudentById(String delId) {
        // 1. 查找id在容器中所在的索引位置
        int index = getIndex(delId);
        stus.remove(index);
    }

    public int getIndex(String id){
        int index = -1;
        for (int i = 0; i < stus.size(); i++) {
            Student stu = stus.get(i);
            if(stu != null && stu.getId().equals(id)){
                index = i;
                break;
            }
        }
        return index;
    }

    public void updateStudent(String updateId, Student newStu) {
        // 1. 查找updateId, 在容器中的索引位置
        int index = getIndex(updateId);
        stus.set(index, newStu);
    }
}

```

StudentService类

```

public class StudentService {
    // 创建StudentDao (库管)
    private OtherStudentDao studentDao = new OtherStudentDao();
    // 其他方法没有变化,此处省略...
}

```

## 1.2黑马信息管理系统抽取Dao (应用)

- 优化步骤
  1. 将方法向上抽取，抽取出一个父类（BaseStudentDao）
  2. 方法的功能实现在父类中无法给出具体明确，定义为抽象方法
  3. 让两个类分别继承 BaseStudentDao，重写内部抽象方法

- 代码实现

BaseStudentDao类

```

public abstract class BaseStudentDao {
    // 添加学生方法
    public abstract boolean addStudent(Student stu);
    // 查看学生方法
    public abstract Student[] findAllStudent();
    // 删除学生方法
    public abstract void deleteStudentById(String delId);
    // 根据id找索引方法
    public abstract int getIndex(String id);
    // 修改学生方法
    public abstract void updateStudent(String updateId, Student newStu);
}

```

StudentDao类

```

public class StudentDao extends BaseStudentDao {
    // 其他内容不变,此处省略
}

```

OtherStudentDao类

```

public class OtherStudentDao extends BaseStudentDao {
    // 其他内容不变,此处省略
}

```

### 1.3接口的概述（理解）

- 接口就是一种公共的规范标准，只要符合规范标准，大家都可以通用。
- Java中接口存在的两个意义
  1. 用来定义规范
  2. 用来做功能的拓展

### 1.4接口的特点（记忆）

- 接口用关键字interface修饰

```

public interface 接口名 {}

```

- 类实现接口用implements表示

```

public class 类名 implements 接口名 {}

```

- 接口不能实例化  
我们可以创建接口的实现类对象使用
- 接口的子类  
要么重写接口中的所有抽象方法  
要么子类也是抽象类

## 1.5接口的成员特点（记忆）

- 成员特点
  - 成员变量  
只能是常量 默认修饰符：public static final
  - 构造方法  
没有，因为接口主要是扩展功能的，而没有具体存在
  - 成员方法  
只能是抽象方法  
默认修饰符：public abstract  
关于接口中的方法，JDK8和JDK9中有一些新特性，后面再讲解
- 代码演示
  - 接口

```
public interface Inter {  
    public static final int NUM = 10;  
  
    public abstract void show();  
}
```

- 实现类

```
class InterImpl implements Inter{  
  
    public void method(){  
        // NUM = 20;  
        System.out.println(NUM);  
    }  
  
    public void show(){  
  
    }  
}
```

- 测试类

```

public class TestInterface {
    /*
        成员变量：只能是常量 系统会默认加入三个关键字
            public static final
        构造方法：没有
        成员方法：只能是抽象方法，系统会默认加入两个关键字
            public abstract
    */
    public static void main(String[] args) {
        System.out.println(Inter.NUM);
    }
}

```

## 1.6类和接口的关系（记忆）

- 类与类的关系  
继承关系，只能单继承，但是可以多层继承
- 类与接口的关系  
实现关系，可以单实现，也可以多实现，还可以在继承一个类的同时实现多个接口
- 接口与接口的关系  
继承关系，可以单继承，也可以多继承

## 1.7黑马信息管理系统使用接口改进（应用）

- 实现步骤
  1. 将 BaseStudentDao 改进为一个接口
  2. 让 StudentDao 和 OtherStudentDao 去实现这个接口
- 代码实现  
BaseStudentDao接口

```

public interface BaseStudentDao {
    // 添加学生方法
    public abstract boolean addStudent(Student stu);
    // 查看学生方法
    public abstract Student[] findAllStudent();
    // 删除学生方法
    public abstract void deleteStudentById(String delId);
    // 根据id找索引方法
    public abstract int getIndex(String id);
    // 修改学生方法
    public abstract void updateStudent(String updateId, Student newStu);
}

```

StudentDao类

```
public class StudentDao implements BaseStudentDao {  
    // 其他内容不变,此处省略  
}
```

OtherStudentDao类

```
public class OtherStudentDao implements BaseStudentDao {  
    // 其他内容不变,此处省略  
}
```

## 1.8黑马信息管理系统解耦合改进 (应用)

- 实现步骤
  1. 创建factory包, 创建 StudentDaoFactory (工厂类)
  2. 提供 static 修改的 getStudentDao 方法, 该方法用于创建StudentDao对象并返回

- 代码实现

StudentDaoFactory类

```
public class StudentDaoFactory {  
    public static OtherStudentDao getStudentDao(){  
        return new OtherStudentDao();  
    }  
}
```

StudentService类

```
public class StudentService {  
    // 创建StudentDao (库管)  
    // private OtherStudentDao studentDao = new OtherStudentDao();  
  
    // 通过学生库管工厂类, 获取库管对象  
    private OtherStudentDao studentDao = StudentDaoFactory.getStudentDao();  
}
```

## 2.接口组成更新

### 2.1接口组成更新概述【理解】

- 常量  
public static final
- 抽象方法  
public abstract
- 默认方法(Java 8)
- 静态方法(Java 8)
- 私有方法(Java 9)

## 2.2接口中默认方法【应用】

- 格式

`public default 返回值类型 方法名(参数列表) {}`

- 作用

解决接口升级的问题

- 范例

```
public default void show3() {  
}
```

- 注意事项

- 默认方法不是抽象方法，所以不强制被重写。但是可以被重写，重写的时候去掉default关键字
- public可以省略，default不能省略
- 如果实现了多个接口，多个接口中存在相同的方法声明，子类就必须对该方法进行重写

## 2.3接口中静态方法【应用】

- 格式

`public static 返回值类型 方法名(参数列表) {}`

- 范例

```
public static void show() {  
}
```

- 注意事项

- 静态方法只能通过接口名调用，不能通过实现类名或者对象名调用
- public可以省略，static不能省略

## 2.4接口中私有方法【应用】

- 私有方法产生原因

Java 9中新增了带方法体的私有方法，这其实在Java 8中就埋下了伏笔：Java 8允许在接口中定义带方法体的默认方法和静态方法。这样可能会引发一个问题：当两个默认方法或者静态方法中包含一段相同的代码实现时，程序必然考虑将这段实现代码抽取成一个共性方法，而这个共性方法是不需要让别人使用的，因此用私有给隐藏起来，这就是Java 9增加私有方法的必然性

- 定义格式

- 格式1

`private 返回值类型 方法名(参数列表) {}`

- 范例1

```
private void show() {  
}
```

- 格式2

private static 返回值类型 方法名(参数列表){}

- 范例2

```
private static void method() {  
}
```

- 注意事项
  - 默认方法可以调用私有的静态方法和非静态方法
  - 静态方法只能调用私有的静态方法

## 3.多态

### 3.1多态的概述（记忆）

- 什么是多态

同一个对象，在不同时刻表现出来的不同形态
- 多态的前提
  - 要有继承或实现关系
  - 要有方法的重写
  - 要有父类引用指向子类对象
- 代码演示

```
class Animal {  
    public void eat(){  
        System.out.println("动物吃饭");  
    }  
}  
  
class Cat extends Animal {  
    @Override  
    public void eat() {  
        System.out.println("猫吃鱼");  
    }  
}  
  
public class Test1Polymorphic {  
    /*  
        多态的前提：  
  
        1. 要有(继承 \ 实现)关系  
        2. 要有方法重写  
        3. 要有父类引用，指向子类对象  
    */  
    public static void main(String[] args) {  
        // 当前事物，是一只猫  
        Cat c = new Cat();  
  
        // 当前事物，是一只动物
```



```

        Animal a = new Cat();
        a.eat();

    }
}

```

## 3.2多态中的成员访问特点（记忆）

- 成员访问特点
  - 成员变量  
编译看父类，运行看父类
  - 成员方法  
编译看父类，运行看子类
- 代码演示

```

class Fu {
    int num = 10;

    public void method(){
        System.out.println("Fu.. method");
    }
}

class Zi extends Fu {
    int num = 20;

    public void method(){
        System.out.println("Zi.. method");
    }
}

public class Test2Polymorplic {
    /*
        多态的成员访问特点：

        成员变量：编译看左边（父类），运行看左边（父类）

        成员方法：编译看左边（父类），运行看右边（子类）

    */
    public static void main(String[] args) {
        Fu f = new Zi();
        System.out.println(f.num);
        f.method();
    }
}

```

## 3.3多态的好处和弊端（记忆）

- 好处

提高程序的扩展性。定义方法时候，使用父类型作为参数，在使用的时候，使用具体的子类型参与操作

- 弊端

不能使用子类的特有成员

### 3.4多态中的转型（应用）

- 向上转型

父类引用指向子类对象就是向上转型

- 向下转型

格式：子类型 对象名 = (子类型)父类引用;

- 代码演示

```
class Fu {
    public void show(){
        System.out.println("Fu..show...");
    }
}

class Zi extends Fu {
    @Override
    public void show() {
        System.out.println("Zi..show...");
    }

    public void method(){
        System.out.println("我是子类特有的方法，method");
    }
}

public class Test3Polymorphic {
    public static void main(String[] args) {
        // 1. 向上转型：父类引用指向子类对象
        Fu f = new Zi();
        f.show();
        // 多态的弊端：不能调用子类特有的成员
        // f.method();

        // A: 直接创建子类对象
        // B: 向下转型

        // 2. 向下转型：从父类类型，转换回子类类型
        Zi z = (Zi) f;
        z.method();
    }
}
```

### 3.5多态中转型存在的风险和解决方案 (应用)

- 风险

如果被转的引用类型变量,对应的实际类型和目标类型不是同一种类型,那么在转换的时候就会出现 ClassCastException

- 解决方案

- 关键字

- instanceof

- 使用格式

- 变量名 instanceof 类型

- 通俗的理解: 判断关键字左边的变量, 是否是右边的类型, 返回boolean类型结果

- 代码演示

```
abstract class Animal {
    public abstract void eat();
}

class Dog extends Animal {
    public void eat() {
        System.out.println("狗吃肉");
    }

    public void watchHome(){
        System.out.println("看家");
    }
}

class Cat extends Animal {
    public void eat() {
        System.out.println("猫吃鱼");
    }
}

public class Test4Polymorphic {
    public static void main(String[] args) {
        useAnimal(new Dog());
        useAnimal(new Cat());
    }

    public static void useAnimal(Animal a){ // Animal a = new Dog();
                                           // Animal a = new Cat();

        a.eat();
        //a.watchHome();

        // Dog dog = (Dog) a;
        // dog.watchHome(); // ClassCastException 类型转换异常

        // 判断a变量记录的类型, 是否是Dog
        if(a instanceof Dog){
            Dog dog = (Dog) a;
            dog.watchHome();
        }
    }
}
```

```
}
```

## 3.6 黑马信息管理系统多态改进 (应用)

- 实现步骤
  1. StudentDaoFactory类中方法的返回值定义成父类类型BaseStudentDao
  2. StudentService中接收方法返回值的类型定义成父类类型BaseStudentDao
- 代码实现

StudentDaoFactory类

```
public class StudentDaoFactory {  
    public static BaseStudentDao getStudentDao(){  
        return new OtherStudentDao();  
    }  
}
```

StudentService类

```
public class StudentService {  
    // 创建StudentDao (库管)  
    // private OtherStudentDao studentDao = new OtherStudentDao();  
  
    // 通过学生库管工厂类，获取库管对象  
    private BaseStudentDao studentDao = StudentDaoFactory.getStudentDao();  
}
```

## 4. 内部类

### 4.1 内部类的基本使用 (理解)

- 内部类概念
  - 在一个类中定义一个类。举例：在一个类A的内部定义一个类B，类B就被称为内部类
- 内部类定义格式
  - 格式&举例：

```
/*  
    格式：  
    class 外部类名{  
        修饰符 class 内部类名{  
  
        }  
    }  
*/  
  
class Outer {  
    public class Inner {
```

```
}  
}
```

- 内部类的访问特点
  - 内部类可以直接访问外部类的成员，包括私有
  - 外部类要访问内部类的成员，必须创建对象
- 示例代码：

```
/*  
    内部类访问特点：  
    内部类可以直接访问外部类的成员，包括私有  
    外部类要访问内部类的成员，必须创建对象  
*/  
public class Outer {  
    private int num = 10;  
    public class Inner {  
        public void show() {  
            System.out.println(num);  
        }  
    }  
    public void method() {  
        Inner i = new Inner();  
        i.show();  
    }  
}
```

## 2.2 成员内部类（理解）

- 成员内部类的定义位置
  - 在类中方法，跟成员变量是一个位置
- 外界创建成员内部类格式
  - 格式：外部类名.内部类名 对象名 = 外部类对象.内部类对象;
  - 举例：Outer.Inner oi = new Outer().new Inner();
- 私有成员内部类
  - 将一个类，设计为内部类的目的，大多数都是不想让外界去访问，所以内部类的定义应该私有化，私有化之后，再提供一个可以让外界调用的方法，方法内部创建内部类对象并调用。
  - 示例代码：

```
class Outer {  
    private int num = 10;  
    private class Inner {  
        public void show() {  
            System.out.println(num);  
        }  
    }  
    public void method() {  
        Inner i = new Inner();  
        i.show();  
    }  
}
```

```

    }
}
public class InnerDemo {
    public static void main(String[] args) {
        //Outer.Inner oi = new Outer().new Inner();
        //oi.show();
        Outer o = new Outer();
        o.method();
    }
}

```

- 静态成员内部类

- 静态成员内部类访问格式：外部类名.内部类名 对象名 = new 外部类名.内部类名();
- 静态成员内部类中的静态方法：外部类名.内部类名.方法名();
- 示例代码

```

class Outer {
    static class Inner {
        public void show(){
            System.out.println("inner..show");
        }

        public static void method(){
            System.out.println("inner..method");
        }
    }
}

public class Test3Innerclass {
    /*
        静态成员内部类演示
    */
    public static void main(String[] args) {
        // 外部类名.内部类名 对象名 = new 外部类名.内部类名();
        Outer.Inner oi = new Outer.Inner();
        oi.show();

        Outer.Inner.method();
    }
}

```

## 2.3 局部内部类（理解）

- 局部内部类定义位置
  - 局部内部类是在方法中定义的类
- 局部内部类方式方式
  - 局部内部类，外界是无法直接使用，需要在方法内部创建对象并使用
  - 该类可以直接访问外部类的成员，也可以访问方法内的局部变量
- 示例代码

```

class Outer {
    private int num = 10;
    public void method() {
        int num2 = 20;
        class Inner {
            public void show() {
                System.out.println(num);
                System.out.println(num2);
            }
        }
        Inner i = new Inner();
        i.show();
    }
}

public class OuterDemo {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.method();
    }
}

```

## 2.4 匿名内部类（应用）

- 匿名内部类的前提
  - 存在一个类或者接口，这里的类可以是具体类也可以是抽象类
- 匿名内部类的格式
  - 格式：new 类名 () { 重写方法 } new 接口名 () { 重写方法 }
  - 举例：

```

new Inter(){
    @Override
    public void method(){}
}

```

- 匿名内部类的本质
  - 本质：是一个继承了该类或者实现了该接口的子类匿名对象
- 匿名内部类的细节
  - 匿名内部类可以通过多态的形式接受

```

Inter i = new Inter(){
    @Override
    public void method(){

    }
}

```

- 匿名内部类直接调用方法

```

interface Inter{
    void method();
}

class Test{
    public static void main(String[] args){
        new Inter(){
            @Override
            public void method(){
                System.out.println("我是匿名内部类");
            }
        }.method(); // 直接调用方法
    }
}

```

## 2.4 匿名内部类在开发中的使用（应用）

- 匿名内部类在开发中的使用
  - 当发现某个方法需要，接口或抽象类的子类对象，我们就可以传递一个匿名内部类过去，来简化传统的代码
- 示例代码：

```

/*
    游泳接口
*/
interface Swimming {
    void swim();
}

public class TestSwimming {
    public static void main(String[] args) {
        goSwimming(new Swimming() {
            @Override
            public void swim() {
                System.out.println("铁汁，我们去游泳吧");
            }
        });
    }

    /**
     * 使用接口的方法
     */
    public static void goSwimming(Swimming swimming){
        /*
            Swimming swim = new Swimming() {
                @Override
                public void swim() {
                    System.out.println("铁汁，我们去游泳吧");
                }
            }
        */
    }
}

```



```
        swimming.swim();
    }
}
```

## 5.Lambda表达式

### 5.1体验Lambda表达式【理解】

- 代码演示

```
/*
    游泳接口
*/
interface Swimming {
    void swim();
}

public class TestSwimming {
    public static void main(String[] args) {
        // 通过匿名内部类实现
        goSwimming(new Swimming() {
            @Override
            public void swim() {
                System.out.println("铁汁，我们去游泳吧");
            }
        });

        /* 通过Lambda表达式实现
           理解：对于Lambda表达式，对匿名内部类进行了优化
        */
        goSwimming(() -> System.out.println("铁汁，我们去游泳吧"));
    }

    /**
     * 使用接口的方法
     */
    public static void goSwimming(Swimming swimming) {
        swimming.swim();
    }
}
```

- 函数式编程思想概述

在数学中，函数就是有输入量、输出量的一套计算方案，也就是“拿数据做操作”

面向对象思想强调“必须通过对对象的形式来做事情”

函数式思想则尽量忽略面向对象的复杂语法：“强调做什么，而不是以什么形式去做”

而我们要学习的Lambda表达式就是函数式思想的体现

### 5.2Lambda表达式的标准格式【理解】

- 格式：

(形式参数) -> {代码块}

- 形式参数：如果有多个参数，参数之间用逗号隔开；如果没有参数，留空即可
  - ->：由英文中画线和大于符号组成，固定写法。代表指向动作
  - 代码块：是我们具体要做的事情，也就是以前我们写的方法体内容
- 组成Lambda表达式的三要素：
    - 形式参数，箭头，代码块

## 5.3 Lambda表达式练习1【应用】

- Lambda表达式的使用前提
  - 有一个接口
  - 接口中有且仅有一个抽象方法
- 练习描述

无参无返回值抽象方法的练习
- 操作步骤
  - 定义一个接口(Eatable)，里面定义一个抽象方法：void eat();
  - 定义一个测试类(EatableDemo)，在测试类中提供两个方法
    - 一个方法是：useEatable(Eatable e)
    - 一个方法是主方法，在主方法中调用useEatable方法
- 示例代码

```
//接口
public interface Eatable {
    void eat();
}

//实现类
public class EatableImpl implements Eatable {
    @Override
    public void eat() {
        System.out.println("一天一苹果，医生远离我");
    }
}

//测试类
public class EatableDemo {
    public static void main(String[] args) {
        //在主方法中调用useEatable方法
        Eatable e = new EatableImpl();
        useEatable(e);

        //匿名内部类
        useEatable(new Eatable() {
            @Override
            public void eat() {
                System.out.println("一天一苹果，医生远离我");
            }
        });
    }
}
```

```

//Lambda表达式
useEatable(() -> {
    System.out.println("一天一苹果，医生远离我");
});

private static void useEatable(Eatable e) {
    e.eat();
}
}

```

## 5.4Lambda表达式练习2【应用】

- 练习描述

有参无返回值抽象方法的练习

- 操作步骤

- 定义一个接口(Flyable)，里面定义一个抽象方法：void fly(String s);
- 定义一个测试类(FlyableDemo)，在测试类中提供两个方法
  - 一个方法是：useFlyable(Flyable f)
  - 一个方法是主方法，在主方法中调用useFlyable方法

- 示例代码

```

public interface Flyable {
    void fly(String s);
}

public class FlyableDemo {
    public static void main(String[] args) {
        //在主方法中调用useFlyable方法
        //匿名内部类
        useFlyable(new Flyable() {
            @Override
            public void fly(String s) {
                System.out.println(s);
                System.out.println("飞机自驾游");
            }
        });
        System.out.println("-----");

        //Lambda
        useFlyable((String s) -> {
            System.out.println(s);
            System.out.println("飞机自驾游");
        });

        private static void useFlyable(Flyable f) {
            f.fly("风和日丽，晴空万里");
        }
    }
}

```

```
}
```

## 5.5 Lambda表达式练习3【应用】

- 练习描述  
有参有返回值抽象方法的练习
- 操作步骤
  - 定义一个接口(Addable), 里面定义一个抽象方法: `int add(int x,int y);`
  - 定义一个测试类(AddableDemo), 在测试类中提供两个方法
    - 一个方法是: `useAddable(Addable a)`
    - 一个方法是主方法, 在主方法中调用`useAddable`方法
- 示例代码

```
public interface Addable {  
    int add(int x,int y);  
}  
  
public class AddableDemo {  
    public static void main(String[] args) {  
        //在主方法中调用useAddable方法  
        useAddable((int x,int y) -> {  
            return x + y;  
        });  
    }  
  
    private static void useAddable(Addable a) {  
        int sum = a.add(10, 20);  
        System.out.println(sum);  
    }  
}
```

## 5.6 Lambda表达式的省略模式【应用】

- 省略的规则
  - 参数类型可以省略。但是有多个参数的情况下, 不能只省略一个
  - 如果参数有且仅有一个, 那么小括号可以省略
  - 如果代码块的语句只有一条, 可以省略大括号和分号, 和`return`关键字
- 代码演示

```
public interface Addable {  
    int add(int x, int y);  
}  
  
public interface Flyable {  
    void fly(String s);  
}
```

```

public class LambdaDemo {
    public static void main(String[] args) {
        //      useAddable((int x,int y) -> {
        //          return x + y;
        //      });
        //参数的类型可以省略
        useAddable((x, y) -> {
            return x + y;
        });

        //      useFlyable((String s) -> {
        //          System.out.println(s);
        //      });
        //如果参数有且仅有一个，那么小括号可以省略
        useFlyable(s -> {
            System.out.println(s);
        });

        //如果代码块的语句只有一条，可以省略大括号和分号
        useFlyable(s -> System.out.println(s));

        //如果代码块的语句只有一条，可以省略大括号和分号，如果有return，return也要省略掉
        useAddable((x, y) -> x + y);
    }

    private static void useFlyable(Flyable f) {
        f.fly("风和日丽，晴空万里");
    }

    private static void useAddable(Addable a) {
        int sum = a.add(10, 20);
        System.out.println(sum);
    }
}

```

## 5.7 Lambda表达式的使用前提【理解】

- 使用Lambda必须要有接口
- 并且要求接口中有且仅有一个抽象方法

## 5.8 Lambda表达式和匿名内部类的区别【理解】

- 所需类型不同
  - 匿名内部类：可以是接口，也可以是抽象类，还可以是具体类
  - Lambda表达式：只能是接口
- 使用限制不同
  - 如果接口中有且仅有一个抽象方法，可以使用Lambda表达式，也可以使用匿名内部类
  - 如果接口中多于一个抽象方法，只能使用匿名内部类，而不能使用Lambda表达式
- 实现原理不同
  - 匿名内部类：编译之后，产生一个单独的.class字节码文件

- Lambda表达式：编译之后，没有一个单独的.class字节码文件。对应的字节码会在运行的时候动态生成