1.内部类

1.1 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();
    }
}
```

1.2 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();
    }
}
```

1.3 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();
    }
}
```

1.4 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
    }
}
```

1.5 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();
    }
}
```

2.Lambda表达式

2.1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

2.2Lambda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

2.3Lambda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("输入重写内容");
        });
    }

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

2.4Lambda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");
    }
}
```

2.5Lambda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);
    }
}
```

2.6Lambda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);
    }
}
```

2.7Lambda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

2.8Lambda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