Java程序设计
(进阶)

2020.5.5

isszym　sysu.edu.cn

目录

抽象类

- 如果一个类的某个方法只能由子类给出具体实现，就可以把它定义为**抽象类**。该方法只需给出头部，不给出主体（body）。例如：Shape类的方法draw()，只有等着子类给出具体形状才能draw。
- 没有方法主体的方法要定义为**抽象方法**。具有抽象方法的类必须定义为**抽象类**。抽象类不能被用于创建实例，但是可以用于定义子类。
- 下面的ShapeAbs必须定义为抽象类，因为它有一个抽象方法draw：

```
abstract class ShapeAbs {                                    // 抽象类
    public ShapeAbs() {                                      // 构造函数(constructor)
        System.out.println("Shape Initialized!");
    }
    public abstract void draw();                            // 抽象方法(method)
}
```

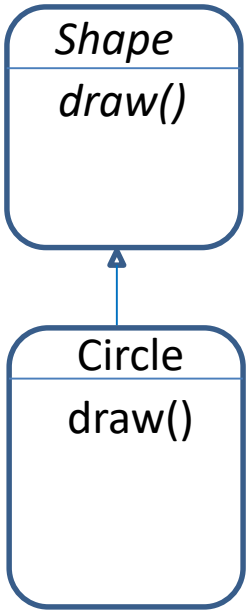
- 子类Circle实现了父类的唯一的抽象方法draw，没有其他抽象方法，因此可以作为具体(concrete)类，并在ShapeAbsTest类中创建了实例。

ShapeAbsTest.java

```
class Circle extends ShapeAbs {
    public void draw() {
      System.out.println("Circle draw() is called!");
    }
}
public class ShapeAbsTest {
  public static void main(String args[]){
      Circle circle1 = new Circle();
      circle1.draw();
  }
}
```

执行结果：

```
Shape Initialized!
Circle draw() is called!
```



接口 /interface

- 接口用于描述类应该具有什么功能而不是每个功能的具体实现。
- 接口可以看成是一种极端的抽象类，它只能定义常量和方法头，不能定义任何数据域和方法体。接口的方法默认都是public的和abstract的。
- 接口的子类必须实现其所有方法。下面定义的接口Door定义了方法open()和close()的方法头。它的子类OrdinaryDoor给出了方法open()和close()的主体部分。

```
public interface Door {
    void open();   // 开门
    void close();  // 关门
}
public class OrdinaryDoor implements Door {
    public void open() { System.out.println("open door!");}
    public void close(){ System.out.println("close door!");}
}
```

依赖倒置原则（Dependence Inversion Principle）程序要依赖于抽象接口，不要依赖于具体实现。

-- 面向对象设计原则之一

- 类DoorTest创建了接口Door的子类OrdinaryDoor的实例，并向上转型为接口Door的类型，再调用enter()。
- 类DoorTest的方法enter()的参数door的类型采用接口Door类型。

```
public class DoorTest {
    public static void main(String args[]){
        Door door1 = new OrdinaryDoor();     //向上转型
        Door door2 = new OrdinaryDoor();
        enter(door1);
        enter(door2);
    }
    public static void enter(Door door){    // 采用接口作为参数
      door.open();
      door.close();
    }
}
```

运行结果：

```
open door!
close door!
open door!
close door!
```

- 让上层模块或外部调用者保持稳定是使用接口的主要优点。增加了接口的子类或修改了其子类方法的实现，引用了该接口的方法或类不需要作任何改变。

```
public class SecurityDoor implements Door {    //增加子类
        public void open(){ System.out.println("open security door!");}
        public void close(){ System.out.println("close security door!");}
        public void alarm(){ System.out.println("alarm!");}
}
public class DoorTest2 {
   public static void main(String args[]){
      Door door1 = new OrdinaryDoor();          //向上转型
      Door door2 = new OrdinaryDoor();
      Door door3 = new SecurityDoor();          //增加一个子类
      enter(door1);
      enter(door2);
      enter(door3);                             //增加的调用
      door3.alarm();                            //出错！！（接口Door没有该方法）
   }
   public static void enter(Door door){         //不变
     door.open();
     door.close();
   }
}
```

执行结果：

```
open door!
close door!
open door!
close door!
open security door!
close security door!
```

开闭原则（Open Closed Principle）要求设计的模块对于扩展是开放的，对于修改是关闭的。

-- 面向对象设计原则之一

- 接口中不能定义方法主体，每个子类都必须实现其所有方法。JDK 1.8之后允许在接口中定义static和default的方法和方法体，使得其子类可以不定义这些方法，一种简写方式。

```
interface IHello {
    void sayHi();
    static void sayHello(){System.out.println("say hello!");}
    default void sayByebye(){System.out.println("say byebye!");}
}
class HelloImpl implements IHello {
    @Override
    public void sayHi() {System.out.println("say hi!");  }
}
public class IntDefault {
    public static void main(String[] args) {
        HelloImpl helloImpl = new HelloImpl();
        helloImpl.sayHi();          // 抽象方法调用
        helloImpl.sayByebye();      // 实例对象调用
        IHello.sayHello();          // 静态方法调用
        new IHello() {              // 匿名实现类
            @Override
            public void sayHi() {
                System.out.println("say hi(2)!");
            }
        }.sayHi();
    }
}
```

执行结果：

```
say hi!
say byebye!
say hello!
say hi(2)!
```

去掉default和static会出错: 接口抽象方法不能带有主体;
再去掉sayHello()的主体还会出错： HelloImpl不是抽象的, 并且未覆盖IHello中的抽象方法sayHello()。

匿名子类和回调函数 /anonyintf

- 如果不想为一个接口或类的子类取名，可以使用**匿名子类**。采用 “new 接口名(){方法; 方法; …}” 创建一个匿名子类的实例。该子类都会被单独编译为一个class文件，其文件名由系统给出。
- 下面定义了一个接口Door的匿名子类。

```
Door door = new Door(){
    @Override
    public void open(){  //回调函数
        System.out.println("open anonymous door!");
    }
    @Override
    public void close(){  //回调函数
        System.out.println("close anonymous door!");
    };
};
```

相当于

```
public class AnonyDoor implements Door {
    ……
}
Door door = new AnonyDoor();
```

- 下面给出了完整程序，方法enter()调用了Door子类的方法open()和close()，这两个方法是重写了接口Door的方法。

DoorTest3.java

```
public class DoorTest3 {
   static void enter(Door door){
         door.open();
         door.close();
   }
   public static void main(String[] args){
      Door door= new Door(){
               @Override
               public void open(){
                    System.out.println("open anonymous door!");
               }
               @Override
               public void close(){
                  System.out.println("close anonymous door!");
               };
      };
      enter(door);
   }
}
```

执行结果：

```
open anonymous door!
close anonymous door!
```

* 系统给这个匿名类的字节码文件的名称是DoorTest3$1.class

- 下例直接把匿名子类的实例作为参数。
- 与上例一样，其两个方法open()和close()都是**回调函数**(Callback function) 。
- 在系统调用时，通过把**回调函数**作为参数，可以实现系统内部调用外部程序的功能。

DoorTest4.java

```
public class DoorTest4 {
    static void enter(Door door){
          door.open();                    //回调函数
          door.close();                   //回调函数
    }
    public static void main(String[] args){
        enter(new Door(){
                  @Override
                  private void open(){
                       System.out.println("open anonymous door!");
                  }
                  @Override
                  private void close(){
                      System.out.println("close anonymous door!");
                  };
              };
         );
    }
}
```

如果enter是一个系统类的方法，Door是系统的一个接口，就可以说明问题。这里简化了。

多重继承 /multiherit

- Java的类没有多重继承，但是可以通过接口实现类似多重继承的功能。
- 下面的SecurityDoorX类实现了两个接口Door和Alarm。

```
interface Door { void open(); void close(); }
interface Alarm { void alarm(); }
class SecurityDoorX implements Door,Alarm {
        ……
}
```

> **接口隔离原则**（Interface Segregation Principle）：使用多个专门的接口比使用单一的总接口要好。
>
> -- 面向对象设计原则之一

- 一个Java类可以继承一个类并实现多个接口。

```
class SlidingDoor {
        ……
}
class SecuritySlidingDoor extends SlidingDoor implements Door,Alarm {
        ……
}
```

- 一个接口可以继承多个接口

```
interface SecurityDoorX extends Door,Alarm {
        ……
}
```

- 下面是完整的程序：

Door.java
Alarm.java
SecurityDoorX.java
SecuritySlidingDoor.java

```
interface Door { void open(); void close(); }
interface Alarm { void alarm(); }
class SecurityDoorX implements Door,Alarm {
   public void open(){  System.out.println("open door!");  }
   public void close(){  System.out.println("close door!");  }
   public void alarm(){  System.out.println("alarm!");  }
}
```

SlidingDoor.java

```
class SlidingDoor {
     void slidingOpen(){System.out.println("open sliding door!");}
     void slidingClose(){System.out.println("close sliding door!");}
}
class SecuritySlidingDoor extends SlidingDoor implements Door,Alarm {
   public void open(){
      slidingOpen();
   }
   public void close(){
      slidingClose();
   }
   public void alarm(){ System.out.println("alarm!");  }
}
```

DoorTest5.java

```
public class DoorTest5 {
   public static void main(String args[]){
      SecurityDoorX door1 = new SecurityDoorX();
      SecuritySlidingDoor door2 = new SecuritySlidingDoor();
      enter(door1);
      alarm(door1);
      enter(door2);
      alarm(door2);
   }
   public static void enter(Door door){
      door.open();
      door.close();
   }
   public static void alarm(Alarm alarm){
     alarm.alarm();
   }
}
```

执行结果：

```
open security door!
close security door!
alarm!
open sliding door!
close sliding door!
alarm!
```

通过多重继承，一个类可以为不同的用途提供不同的**外观(facade)**，上例的SecurityDoorX提供了door的功能又具有alarm的功能。

内部类

- 在一个类的内部定义的类称为**内部类**。内部类中还可以定义内部类。内部类可以直接使用外部类的数据域或方法。
- 内部类分为*静态内部类*和*非静态内部类*。非静态内部类又可以分为*成员内部类*、*方法内部类*和*匿名内部类*，*它们都不能在外部类之外使用*。
- **静态内部类**可以在外部类之外直接使用。

DoorTest6.java

```
public class DoorTest6 {
   String door1 = "door1";
   static String door2 = "door6";
   static class Door {                              // 静态内部类
       void open(){
          System.out.println(door1);                // 访问非静态成员，出错!!
          System.out.println(door2 + " open!");     // 可以访问静态成员
       }
   }

   public static void main(String args[]){
      Door door = new Door();
      door.open();
   }
}
```

执行结果(删除错误语句后)：

```
door6 open!
```

可以外部类之外的类使用：

```
DoorTest6.Door door = new DoorTest6.Door();
door.open();
```

- 在一个类的内部定义的接口称为**内部接口**。内部接口与静态内部类一样，可以直接在外部类之外使用。

DoorTest14.java

```
class DoorEx {
  public interface Door {
    void open();
  }
}
class DoorTest14 {
   public static void main(String[] args){
        new DoorEx.Door() {
            public void open(){
               System.out.println("open door! ");
            }
        }.open();

    }
}
```

执行结果：

```
open door!
```

  - 内部类只是出现在编译阶段，实际运行时是没有内部类的。
  - 采用内部类可以实现多重继承问题，还是类的另一种封装方式（采用包是一种）。
  - 内部类直接访问外部类的数据域和方法。

- **成员内部类**(普通内部类)就是把它放在外部类的数据域和方法的同等位置。成员内部类可以直接访问外部类的数据域和方法。
- 访问成员内部类，必须先建立外部类的实例，然后通过.new建立内部类的实例。

DoorTest7.java

```
class DoorTest7 {
    String door1 = "door7";
    class Door {                                    // 成员内部类
       void open(){
          System.out.println(door1 + " open!");
       }
    }

    public static void main(String args[]){
      DoorTest7 doorTest = new DoorTest7();
      Door door = doorTest.new Door();
      door.open();
    }
}
```

执行结果(删除错误语句后)：

```
door7 open!
```

- 成员内部类可以定义为private、public、protected和无修饰词，这与成员的访问权限相似。
- 下面是使用*无修饰词成员内部类*的例子，只有同一个包（这里是默认包）的类才可以访问。

DoorTest8.java

```
class DoorTest8 {
    String door1 = "door8";
    class Door {
       void open(){
          System.out.println(door1 + " open!");
       }
    }
}
```

// 成员内部类（无修饰词）

DoorTest9.java

```
class DoorTest9 {
    public static void main(String args[]){
      DoorTest8 doorTest = new DoorTest8();
      DoorTest8.Door door = doorTest.new Door();
      door.open();
    }
}
```

执行结果：

```
door8 open!
```

- private成员内部类除了其外部类，不能被其他类访问，public成员内部类可以被所有其它类访问，而protected成员内部类可以被同一个包和外部类的子类所访问。
- 下面是使用*protected成员内部类*的例子：

DoorTest10.java

```
class DoorTest10 {
    String door = "door10";
    protected class Door {
       void open(){
          System.out.println(door + " open!");
       }
    }
}
```

// 成员内部类(用于继承)

DoorTest11.java

```
class DoorTest11 extends DoorTest10 {
    public static void main(String args[]){
      DoorTest10 doorTest = new DoorTest10();
      Door door = doorTest.new Door();
      door.open();
    }
}
```

执行结果：

```
door10 open!
```

- **方法内部类**是定义在一个方法内部的类。方法内部类只能在方法中使用。
- 下面是使用方法内部类的例子：

DoorTest12.java

```
class DoorTest12 {
    String door = "door12";
    void openDoor(){
        class Door {                          // 方法内部类
            void open(){
                System.out.println(door + " open!");
            }
        }
        Door door = new Door();
        door.open();
    }
    public static void main(String args[]){
      DoorTest12 doorTest = new DoorTest12();
      doorTest.openDoor();
    }
}
```

执行结果：

```
door12 open!
```

- **匿名内部类**是不取名称的方法内部类，可以作为方法参数或方法在方法内部。
- 下面是使用匿名内部类的例子，其中new OnClickListener(){}定义了一个接口OnClickListener的匿名子类(系统取名为AnonyTest$1.class)。

OnClickListener.java

```
interface OnClickListener {
    void onClick();
}
```

AnonyTest.java

```
class AnonyTest {
    public static void main(String args[]){
       OnClickListener onClickListener = new OnClickListener(){
          @Override
          public void onClick(){
            System.out.println("anonymous class click!");
          }
       };
       onClickListener.onClick();
    }
}
```

执行结果：

```
anonymous class click!
```

* 可以不定义实例直接调用：new OnClickListener(){}.onClick();

- 如果外围类和内部类都实现接口或继承方法，就可以实现类似**多重继承**的功能。
- 下面的内部类SchoolAlarmDoor同时继承了两个类Door和Alarm。

DoorTest13.java

```
class Door {
   void open(){System.out.println("open door!");};
   void close(){System.out.println("close door!");};
}
class Alarm {
   void alarm(){System.out.println("alarm!");};
}
public class DoorTest13 extends Door{
   class SchoolAlarmDoor extends Alarm{
        void enter(){
           open(); alarm(); close();
        }
   }
   public static void main(String[] args){
      DoorTest13 doorTest = new DoorTest13();
      SchoolAlarmDoor door = doorTest.new SchoolAlarmDoor();
      door.enter();
   }
}
```

执行结果：

```
open door!
alarm!
close door!
```

泛型 / generic

- 泛型(Generic Type)为参数化的类型。在定义时只给类型标识符，在使用时才给出真实类型。采用< >包括类型标识符。
- 泛型可以用于类、接口、方法和类型限定四种情况。
- 下面是**泛型用于接口**的例子，把泛型T和U用在其方法show()上：

```
interface Show <T, U> {  // 泛型接口，T和U是泛型
    void show(T t,U u);
}
```

下面实现这个接口的类把泛型T和U分别对应类型String和Date：

```
class GenericInt implements Show<String,Date>{
    @Override
    public void show(String str,Date date) {
        System.out.println(str);
        System.out.println(date);
    }
}
```

* 泛型用于类的方法和接口相似

使用上面的类的例子： GenericTest1.java

```
GenericInt test = new GenericInt();
test.show("Hello!",new Date());
```

执行结果：

```
Hello!
Thu Apr 09 22:29:14 CST 2020
```

- 下面是**泛型用于定义方法**的例子，把泛型X、Y分别用于方法getE()的参数和返回值：

```
class GenericMethod {
    public static <X, Y> Y getE(X x, Y y) {   // 泛型方法 ， X和Y是泛型
        if(x!=null)
            return y;
        else
            return null;
    }
}
```

使用上面泛型的例子：　　GenericTest2.java

```
String str= GenericMethod.getE("Hello", "World");
System.out.println(str);

int i = GenericMethod.getE(100,200);
System.out.println(i);
```

执行结果：

```
World
200
```

- 下面是**泛型用于类型限定**的例子，把泛型T用于方法getF()的参数和返回值，T被限定为接口Comparable的子类：

```
class GenericRestrict {
    public static <T extends Comparable> T getF(T t1,T t2) {
        if(t1.compareTo(t2)>=0);
        return t1;
    }
}
```

*接口Comparable的方法compareTo()用于比较其子类实例的大小。
*可以用&并列两个接口，例如： <T extends Comparable&Serialize>

下面是使用类型限定泛型的例子：

```
String str= GenericRestrict.getF("Hello", "World");
System.out.println(str);
int i = GenericRestrict.getF(200, 100);
System.out.println(i);
```

GenericTest3.java

执行结果：

```
C:\Java2\generic>javac GenericTest3.java -Xlint:unchecked
C:\Java2\generic>java GenericTest3
Hello
200
```

- 对于**类型限定泛型**，我们可以用?表示未知类型，一般用于带入参数，也用于限定类型，例如：“? super Apple”和“? extends Apple”限定只能使用祖先类或子类（包括本类）。

- 下面是另一个例子，Dish类使用了泛型T：

GenericTest4.java

```
class Dish<T>{        // T为泛型
     T food;
     public Dish(T t){food = t;}
}
```

```
class Meat{}
class Pork extends Meat{}
class Beef extends Meat{}
```

创建的Dish实例时要求使用Meat的子类或子孙类的语句：

```
Dish<? extends Meat> dish1;
dish1 = new Dish<Pork>(new Pork()); //Pork可以换成Beef，但是不能换成Meat
```

创建的Dish实例时要求使用Pork的父类或祖先类的语句：

```
Dish<? super Pork> dish2;
dish2 = new Dish<Meat>(new Meat()); // Meat换成Beef会出错
```

方法eat()的参数要求使用Meat的子类或子孙类：

```
public static void eat(Dish<? extends Meat> dish){
       System.out.println("eat "+dish.food.getClass());
}
```

```
eat(dish1);
eat(dish2);      // 出错!
```

执行结果：

```
eat class Pork
```

反射 / reflect

- 通过反射机制，程序可以在运行时取得和使用一个对象或类的所有数据域和方法。这里假设这些数据域和方法在编程时是未知的。
- 获取Class对象有三种方式：

(1)给定对象，通过Class类的方法getClass() ：

```
Class cls1 = new String("").getClass(); // String对象
```

(2)给出包名和类名，通过Class类的方法forName()：

```
Class cls2 = Class.forName("com.group.MyClassName");
```

(3)给出类，通过Class类的属性class：

```
Class cls3 = MyClass.class;
Class cls4 = int.class;       // 整数类（Integer）
Class cls5 = Double[].class; // 数组类（Array）
```

- Java的类在使用时才加载。JVM加载程序ClassLoader先找到class文件，在加载后根据其中的类信息来生成一个Class对象（java.lang.Class 类），用来保存该类的属性和方法等信息。这些信息也用于产生该类的实例。
- 除非迫不得已，一般不主张使用反射的方法获得类信息。

下面的类Person被后面的反射举例所用：

Person.java

```
class Person{
   private String name;
   private int age;
   Person() {
        this.name="Li";
        this.age=30;
   }
   Person(String name,int age) {
        this.name=name;
        this.age=age;
   }
   void setAge(int age){
      this.age=age;
   }
   @Override
   public String toString(){
       return "["+this.name+"  "+this.age+"]";
   }
}
```

例1、下面程序是使用反射创建一个“未知”类实例的例子，其中?是泛型，newInstance()用于创建实例，getDeclaredConstructor()用于获得构造器。

ReflectTest1.java

```
import java.lang.reflect.*;
class ReflectTest1{
    public static void main(String[] args) throws Exception{
        Class<?> cls = Class.forName("Person");

        System.out.println(cls.getName());
        Person per1 = (Person) cls.newInstance();
        per1.setAge(16);
        System.out.println(per1);
        Constructor PerConst = cls.getDeclaredConstructor(
                                    new Class[]{String.class,int.class});
        Person per2=(Person)PerConst.newInstance(new Object[]{"Chen",6});
        System.out.println(per2);
    }
}
```

另外两种获得Class实例的例子：

```
Person per = new Person();
Class<?> cls = per.getClass();

Class<?> cls = Person.class;
```

执行结果：

```
Person
[Li 16]
[Chen 6]
```

例2、下面程序展示了如何通过类Class的方法getDeclaredConstructors()、方法getDeclaredMethods()和getDeclaredFields()获得一个类的所有构造器、方法和属性。

ReflectTest2.java

```
import java.lang.reflect.*;

class ReflectTest2{
    public static void main(String[] args) throws Exception{
        String sBuffer = "";
        Class<?> cls = Class.forName("Person");

        System.out.println(cls.getName()+"的构造器有：");
        Constructor[] allConstructs = cls.getDeclaredConstructors();
        for (int i = 0; i < allConstructs.length; i++) {
            sBuffer = Modifier.toString(allConstructs[i].getModifiers());
            sBuffer =  sBuffer + " " + allConstructs[i].getName() + "(";
            Class<?>[] ptypes = allConstructs[i].getParameterTypes();
            for (int j = 0; j < ptypes.length; j++) {
                 sBuffer =  sBuffer + " "+ ptypes[j].getSimpleName();
            }
            System.out.println(sBuffer + ")");
        }
```

```
        System.out.println("\n" + cls.getName()+"的方法有：");
        Method[] allMethods = cls.getDeclaredMethods();
        for (int i = 0; i < allMethods.length; i++) {
            sBuffer = Modifier.toString(allMethods[i].getModifiers());
            sBuffer =  sBuffer + " "
                                + allMethods[i].getReturnType().getSimpleName();
            sBuffer =  allMethods[i].getName() + "(";
            Class<?>[] ptypes = allMethods[i].getParameterTypes();
            for (int j = 0; j < ptypes.length; j++) {
                 sBuffer =  sBuffer + " " + ptypes[j].getSimpleName();
            }
            System.out.println(sBuffer + ")");
        }
        System.out.println("\n" + cls.getName()+"的属性有：");
        Field[] fields = cls.getDeclaredFields();
        for (int i = 0; i < fields.length; i++) {
            sBuffer = Modifier.toString(fields[i].getModifiers());
            sBuffer = sBuffer + " "+ fields[i].getType().getSimpleName();
            sBuffer = sBuffer + " "+ fields[i].getName();
            System.out.println(sBuffer);
        }
    }
}
```

执行结果：

```
Person的构造器有：
 Person()
 Person( String int)

Person的方法有：
toString()
setAge( int)

Person的属性有：
private String name
private int age
```

例3、下面程序展示了如何通过类Class的方法newInstance()和Invoke()创建和调用反射获得的类的方法。

```
package com.group;
public class Pork{
   public void eat() {
       System.out.println("Eat pork!");
   }
}

import java.lang.reflect.*;
class ReflectTest3{
    public static void main(String[] args) throws Exception{
       Class c=Class.forName("com.group.Pork");   // 包名.类名
       Method method=c.getMethod("eat");
       method.invoke(c.newInstance());  // Java9之后显示已过时
    }
}
```

Java 9 使用以下指令:

```
method.invoke(c.getDeclaredConstructor().newInstance());
```

正则表达式

\regex

- 如果你需要在字符串中查找或替换具有某种模式的子串，你就可以正则表达式(Regular Expression)。

  模式[Jj]ava.+可以用来查找以Java或java开头的字符串。其中，“[Jj]”表示第一个字符必须是J或j，然后必须出现三个字符“ava”，最后“.+”表示需要以一个或若干个任意字符结尾。字符串“Javanese”可以匹配该模式，而“core java”和“Java”则不能匹配该模式。

- 如何用正则表达式确定字符串“Javanese”是否匹配模式[Jj]ava.+?
  - Pattern类用于编译正则表达式创建一个匹配模式，Matcher类使用Pattern实例提供的模式信息对字符串进行匹配。Matcher的matches()要求整个字符串匹配。

```
Pattern pattern = Pattern.compile("[Jj]ava.+");          RegExTest1.java
Matcher matcher = pattern.matcher("Javanese");
boolean mat2 = matcher.matches();                         // true
```

  - 用类Pattern的方法matches直接匹配正则表达式：

```
boolean mat1 = Pattern.matches("[Jj]ava.+","Javanese");  //true
```

RegExTest2.java

* 除了整体匹配之外，Matcher还可以用find()有一种找出所有子串。

- 单字符选择匹配[]：

[Jj]表示匹配一个字符J或j，其它例子：

[abc]	匹配方括号之间的任何一个字符，这里是匹配a或b或c。
[^abc]	匹配任何不在方括号之间的字符，即只有不是abc就行。
[0-9]	查找任何从 0 至 9 的数字
[a-z]	查找任何从小写 a 到小写 z 的字符
[A-Z]	查找任何从大写 A 到大写 Z 的字符
[A-z]	查找任何从大写 A 到小写 z 的字符。
[A-Z0-9]	查找任何大写字母和数字

给出模式: (1) Java[A-Z][0-9] [0-9]　(2) [^A-Z][0-9] [^a-z]　(3) [A-x0-9]Room
以下哪些可以匹配以上模式？

① JavaX15 ① √
② xyRoom ② X
③ A0A ③ X
④ JavaS99 ④ √
⑤ JavY26 ⑤ X
⑥ A5a ⑥ X
⑦ Javas88 ⑦ X
⑧ a0A ⑧ √
⑨ 9WRoom ⑨ X
⑩ a65 ⑩ X
⑪ aRoom ⑪ √
⑫ s9S ⑫ √
⑬ Zroom ⑬ X
⑭ JavaT970 ⑭ X
⑮ 6Room ⑮ √
⑯ B9Roo ⑯ X

- [a-zA-Z0-9]可以表示字母数字，还可以用“\w”表示。用“.”可以表示任意字符。这些都是转义字符的表示方法。转义字符还可以用于表示一些不可显示的字符。

.	除了换行符之外的任意字符，等价于 [^\n]
\w	任何单个字母数字，等价于[a-zA-Z0-9]
\W	任何非单个字母数字，等价于[^a-zA-Z0-9]
\s	任何空白符,等价于[\t\n\r\f\v]
\S	任何非空白符,等价于[^ \t\n\r\f\v]
\d	任何数字,等价于[0-9]
\D	除了数字之外的任何字符,等价于[^0-9]
\ f	换页符
\ nnn	十进制数 nnn指定的ASCII码字符
\ xnn	十六进制数 nn 指定的ASCII码字符
\unn	十六进制数 nn 指定UNICODE码字符
\ cX	控制字符^X。例如，\cI等价于\t，\cJ等价于 \n
\ /	字符/的转义，还有字符\ . * + ? | () []{}需要转义。

\ n	换行符
\ r	回车
\ t	制表符
\ v	垂直制表符
\0	字符nul
\b	单词边界
\B	非单词边界
[\b]	一个退格直接量(特例)

\s 和\b都可以匹配空格，但是\s 会匹配掉空格，而\b不会，它只是找到单词的边界。

RegExTest3. java

*Java必须把匹配模式中的\写成\\(因为在字符串中)

- 量词模式“.+”可以用来匹配一个或若干字符，其它的量词匹配为:

x{n}	x出现n次
x{n, m}	x出现n~m次
x{n, }	x出现至少n次
x?	x出现0次或1次，等价于 {0, 1}
x+	x出现至少1次，等价于{1,}
x*	x出现至少0次，等价于{0,}

例子：

```
a{5}              // 匹配5个a，即aaaaa
a{3,}             // 至少出现3个a，即aaa…
\d{2, 4}          // 匹配2到4个数字，即[0~9]{2,4}，例，3993，256
\w{3} \d?         // 匹配三个单字字符和一个数字， [a-zA-Z0-9]{3}[0~9]?
\s+java\s+        // 匹配字符串“java“ ,前后至少有一个空白符。
\bjava\b.s+       // 匹配”java lang”为true，而\sjava\s.+为false。
[^”] *            // 匹配零个或多个非引号字符。
```

- 如果要表示以java结尾，可以使用位置匹配模式“java$”，所有的位置匹配模式如下：

x$	匹配任何结尾为 x 的字符串。
^x	匹配任何开头为 x 的字符串。
?=x	匹配任何其后紧接指定字符串 x 的字符串。
?!x	匹配任何其后没有紧接指定字符串 x 的字符串。

- 怎么表示分别以3个数字开头，然后大写字母+数字至少重复一次？例如，234A1，490B4G5S8.

\d\d\d[A-Z]\d+ 可以吗？

\d\d\d([A-Z]\d)+ //( )表示一个整体，用于隔开前面的内容。

- 怎么表示分别3个小写字母，加3个数字或者6个数字，再加4个小写字母的模式呢?

  [a-z]{3}(\d{3}|\d{6})[a-z]{4}

  |是一种多模式选择方式，它会扩展到整体的边界

  怎么表示3个小写字母接3个数字或者6个数字接4个小写字母的模式?

  ([a-z]{3}\d{3})|(\d{6}[a-z]{4}) 或者 [a-z]{3}\d{3}|\d{6}[a-z]{4}

贪婪重复：对于量词+和*，只要符合匹配模式就会尽可能多地包含字符。
非贪婪重复：在匹配量词+和*后加上?，表示最小匹配，即匹配尽可能少的字符。

例如，对于字符串abcabcabc，正则表达式(abc)+?只会匹配第一个abc，而(abc)+会匹配整个字符串。

- 类Matcher的方法find()用于从上次查找位置开始查找下一个匹配子串。用find(indexStart)可以重置起始查找位置。找到后可以用Matcher的方法start()和end()得到匹配子串的起止位置，用group()得到匹配的子串。

```
import java.util.regex.*;
public class RegExTest4 {
   public static void main(String args[]){
         String regEx = "[Ss]tu";
         String s = "IastUmastuaaaStuosTuoostuvvv";
         Pattern p = Pattern.compile(regEx);
         Matcher m = p.matcher(s);

         while(m.find()) {
            System.out.println("start: “ + m.start());
            System.out.println("end: ” + m.end());
            System.out.println(“group: ” + m.group());
        }
        System.out.println(m.find());              // false
        System.out.println(m.find(0));             // true
        System.out.println("start: "+m.start()); //7
   }

}
```

运行结果：

```
start: 7
end: 10
group: stu
start: 13
end: 16
group: Stu
start: 22
end: 25
group: stu
false
true
start: 7
```

- 调用find()之后，可以用group()返回匹配的子表达式。group()和group(0)都是指的整个匹配子串，group(1) 指的是第一个括号的匹配子串，group(2)第二个括号的匹配子串。
- 在"Iamastudent101thabc"匹配模式"student(\\d+)(th)ab"时， group(0) 匹配的子串为"student101thab"，group(1)为(\\d+)匹配的子串"101"，group(2)为(th)匹配的子串"th"， group(3)会出错。

RegExTest5.java

```
import java.util.regex.*;
public class RegExTest5 {
   public static void main(String args[]){
         String regEx = "student(\\d+)(th)ab";
         String s = "Iamastudent101thabc";
         Pattern pat = Pattern.compile(regEx);
         Matcher mat = pat.matcher(s);
         if(mat.find()){
            System.out.println(mat.group());   //student101thab
            System.out.println(mat.group(0));  //student101thab
            System.out.println(mat.group(1));  //101
            System.out.println(mat.group(2));  //th
            System.out.println(mat.group(3));  // 出错
         }
   }
}
```

运行结果：

```
student101thab
student101thab
101
th
```

- Matcher的lookingAt()和matches()都用来尝试匹配一个输入序列模式，不同是matches()要求整个序列都匹配，而lookingAt()不要求，但是需要从第一个字符开始匹配。

RegExTest6.java

```
import java.util.regex.*;
public class RegExTest6 {
     static String regex = "foo";
     static String s1 = "fooooooooooooooo";
     static String s2 = "ooooofooooooooooo";
     static Pattern pat;
     static Matcher mat1;
     static Matcher mat2;
     public static void main( String args[] ){
       pat  = Pattern.compile(regex);
       mat1 = pat.matcher(s1);
       mat2 = pat.matcher(s2);
       System.out.println("lookingAt(): "+mat1.lookingAt());
       System.out.println("matches(): "+mat1.matches());
       System.out.println("lookingAt(): "+mat2.lookingAt());
    }
}
```

运行结果：

```
lookingAt(): true
matches(): false
lookingAt(): false
```

- 字符串的方法replaceAll()用于替换子串，而方法split()用于分割字符串，它们都可以使用正则表达式。

RegExTest7.java

```
import java.util.regex.*;
import java.util.Arrays;

public class RegExTest7 {
  public static void main(String[] args) {
        String s1 = "okaa, seeaaa aa youa";
        String regex = "a+";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(s1);
        String s2 = matcher.replaceAll("A");
        System.out.println(s2);       //okA, seeA A youA
        String s3 = s1.replaceAll("a+","A");
        System.out.println(s3);       //okA, seeA A youA
        String arr[]=s1.split("a+");
        Arrays.stream(arr).forEach(s -> System.out.print("["+s+"] "));
    }
}
```

// [ok] [, see] [ ] [you]

运行结果：

```
okA, seeA A youA
okA, seeA A youA
[ok] [, see] [ ] [ you]
```

附录1、Class类的方法

Class类是一个系统类。系统在装载类时会创建Class的实例来保存所装载类的信息。

-- getName()

返回String形式的该类的简要描述。数组Class对象调用该方法会产生奇怪的结果。

-- newInstance()

该方法可以根据某个Class对象产生其对应类的实例。需要强调的是，它调用的是此类的默认构造方法。例如：

```
MyObject  x = new MyObject();
MyObject  y = x.getClass().newInstance();
```

-- getClassLoader()

返回该Class对象对应的类的类加载器。

-- getComponentType()

该方法针对数组对象的Class对象，可以得到该数组的组成元素所对应对象的Class对象。例如：

```
int[] ints = new int[]{1,2,3};
Class class1 = ints.getClass();
Class class2 = class1.getComponentType();
```

而这里得到的class2对象所对应的就应该是int这个基本类型的Class对象。

-- getSuperClass()
返回某子类所对应的直接父类所对应的Class对象。

-- isArray()
判定此Class对象所对应的是否是一个数组对象。

-- forName(String)
找到并返回类。参数是类名的字符串。forName(“com.group.Cookie”)。

-- getDeclaredConstructors()
获得该类的所有构造器。

-- getDeclaredMethods()
获得该类的所有方法。

-- getDeclaredFields()
获得该类的所有数据域。

附录2、参考资料

- 内部类：https://blog.csdn.net/hacker_zhidian/article/details/82193100
- 内部类： https://mp.weixin.qq.com/s/mBEaMGI1SrwU4ENDJk-0AQ
- 反射： http://blog.csdn.net/nieweilin/article/details/5908165
- 反射： http://www.cnblogs.com/cr330326/p/5695474.html