2、具体内容

在类之中的组成就是属性与方法，一般而言方法都是对外提供服务的，所以是不会进行封装处理的，而对于属性由于需要较高的安全性，所以往往需要对其进行保护，这个时候就需要采用封装性对属性进行保护。

在默认的情况下，对于类中的属性是可以通过其它类利用对象进行调用的。

范例：属性在不封装情况下的问题

```
class Person    //定义一个类
{
        String name;    //人员的姓名
        int age;            //人的年龄
        public void tell(){
                System.out.println("姓名："+name+"年龄:"+age);
        }
}

public class JavaDemo{
        public static void main(String[] args) {
                Person per = new Person();   //声明并实例化对象
                per.name = "sanzhang";       //在类外部修改属性
                per.age = -18;                       //在类外部修改属性
                per1.tell();      //进行方法的调用
        }
}
```

此时在person类中提供的name与age两个属性并没有进行封装处理，这样外部就可以直接进行调用了，但是有可能所设置的数据是错误的数据。如果想要解决这样的问题就可以利用private关键字对属性进行封装处理。

范例：对属性进行封装

```
class Person    //定义一个类
{
        private String name;      //人员的姓名
```

```
    private int age;            //人的年龄
    public void tell(){
        System.out.println("姓名："+name+"年龄:"+age);
    }
}
```

而属性一旦封装之后外部将不能够直接访问，即：对外部不可见，但是对类的内部是可见的，那么如果想让外部的程序可以访问封装的属性，则在Java开发标准中提供有如下要求：

- 【setter、getter】设置或取得属性可以使用setXxx()、getXxx()方法，以：private String name为例；

|- 设置属性方法：public void setName(String n);

|- 获取属性方法：public string getName();

范例：实现封装

```
class Person    //定义一个类
{
    private String name;      //人员的姓名
    private int age;          //人的年龄
    public void tell(){
        System.out.println("姓名："+name+"年龄:"+age);
    }

    public void setName(String n){
        name = n;
    }

    public void setAge(int a){
        if(a>=0)
            age = a;
    }

    public void getName(){
        return name;
    }

    public void getAge(){
        return age;
    }
}

public class JavaDemo{
    public static void main(String[] args) {
        Person per = new Person();  //声明并实例化对象
        per.setName("sanzhang");
        per.getAge(18);
        per1.tell();      //进行方法的调用
    }
}
```

在以后进行任何类定义的时候一定要记住，类中的所有属性都必须使用private封装（98%），并且属性要进行访问必须要提供有setter、getter方法。

2、具体内容

现在的程序在使用类的时候一般都按照了如下的步骤进行：

- 声明并实例化对象，这个时候实例化对象中的属性并没有任何的数据存在，都是其对应数据类型的默认值；
- 需要通过一系列的setter方法为类中的属性设置内容。

等于现在要想真正获得一个可以正常使用的实例化对象，必须经过两个步骤才可以完成。

范例：传统调用

```
class Person    //定义一个类
{
        private String name;        //人员的姓名
        private int age;            //人的年龄
        public void tell(){
                System.out.println("姓名："+name+"年龄:"+age);
        }

        public void setName(String n){
                name = n;
        }

        public void setAge(int a){
                        age = a;
        }

        public void getName(){
                return name;
        }
```

```
    public void getAge(){
        return age;
    }
}

public class JavaDemo{
    public static void main(String[] args) {
        //1、对象初始化准备
        Person per = new Person();  //声明并实例化对象
        per.setName("sanzhang");
        per.getAge(18);
        //2、对象的使用
        per1.tell();      //进行方法的调用
    }
}
```

但是如果按照这样的方式来进行思考的话就会发现一个问题：假设说现在类中的属性有很多个(8个)，那么按照之前的做法，此时就需要调用8次的setter方法进行内容设置，这样的调用实在是太啰嗦了，所以在Java里面考虑到对象初始化的问题，专门提供有构造方法，即：**可以通过构造方法实现实例化对象中的属性初始化处理。**只有在关键字new的时候使用构造方法，在Java程序里面构造方法的定义要求如下：

- 构造方法名称必须与类名称保持一致；
- 构造方法不允许设置任何的返回值类型，即：没有返回值定义；
- 构造方法是在使用关键字new实例化对象的时候自动调用的。

范例：定义构造方法

```
class Person    //定义一个类
{
    private String name;      //人员的姓名
    private int age;          //人的年龄
    //方法名称与类名称相同，并且无返回值定义
    public Person(){
        name = n;//为类中属性赋值(初始化)
        age = a;//为类中的属性赋值(初始化)
    }
    public void tell(){
        System.out.println("姓名："+name+"年龄:"+age);
    }

}

public class JavaDemo{
    public static void main(String[] args) {
        //1、对象初始化准备
        Person per = new Person("sanzhang",18); //声明并实例化对象
        //2、对象的使用
        per1.tell();      //进行方法的调用
    }
```

```
}
```

下面针对于当前的对象实例化格式与之前的对象实例化格式做一个比较：

- 之前的对象实例化格式：①Person ②per = ③new ④Person();
- 当前的对象实例化格式：①Person ②per = ③new ④Person("sanzhang",18);
  - |- “①Person”：主要是定义对象的所属类型，类型决定了你可以调用的方法；
  - |- “②per”：实例化对象的名称，所有的操作通过对象来进行访问；
  - |- “③new”：开辟一块新的堆内存空间；
  - |- "④Person("sanzhang",18)"：调用有参构造、"④Person()"：调用无参构造；

在Java程序里面结构的完整性，所以所有的类都会提供构造方法，也就是说如果现在你的类中没有定义任何的构造方法，那么一定会默认提供有一个无参的，什么都不做的构造方法，这个构造方法是在程序编译的时候自动创建的。如果你现在已经在类中明确的定义有一个构造方法的时候，那么这个默认的构造方法将不会被自动创建。

结论：一个类至少存在有一个构造方法，永恒存在。

疑问：为什么构造方法上不允许设置返回值类型?
既然构造方法是一个方法，那么为什么不让它定义返回值类型呢?
既然构造方法不会返回数据，为什么不使用void定义呢?

分析：程序编译器是根据代码结构进行编译处理的，执行的时候也是根据代码结构来处理的。

```
public Person(String n,int a){}
public void Person(String n,int a){}
```

如果在构造方法上使用了void，那么此结构就与普通方法的结构完全相同了，这样编译器会认为此方法是一个普通方法，而普通方法和构造方法最大的区别：构造方法是在类对象实例化的时候调用的，而普通方法是在类对象实例化产生之后调用的。

既然构造方法本身是一个方法，那么方法就具有重载的特点，而构造方法重载的时候只需要考虑参数的类型与个数即可。

范例：构造方法重载

```
class Person    //定义一个类
{
    private String name;      //人员的姓名
    private int age;          //人的年龄
    //方法名称与类名称相同，并且无返回值定义
    public Person(){
        name = "wumingshi";
        age = -1;
    }
    public Person(String n){}
    public void tell(){
        System.out.println("姓名："+name+"年龄:"+age);
    }
}

public class JavaDemo{
```

```
	public static void main(String[] args) {
		//1、对象初始化准备
		 Person per = new Person();  //声明并实例化对象
		 //2、对象的使用
		 per1.tell();	//进行方法的调用
	}
}
```

在进行多个构造方法定义的时候强烈建议大家有一些定义的顺序，例如：可以按照参数的个数降序(升序)排列。

经过分析可以发现，构造方法的确是可以进行数据的设置，而对于setter也可以进行数据的设置，这个时候一定要清楚，构造方法是在对象实例化的时候为属性设置初始化内容，而setter除了拥有设置数据的动能以外，还具有修改数据的功能。

范例：使用setter修改数据

```
class Person	//定义一个类
{
	private String name;	//人员的姓名
	private int age;		//人的年龄
	//方法名称与类名称相同，并且无返回值定义
	public Person(){}
	public Person(String n){}
	public Person(String n,int a){}
	public void setName(String n){
		name = n;
	}

	public void setAge(int a){
			age = a;
	}

	public void getName(){
		return name;
	}

	public void getAge(){
		return age;
	}
	public void tell(){
		System.out.println("姓名："+name+"年龄:"+age);
	}


}

public class JavaDemo{
	public static void main(String[] args) {
		//1、对象初始化准备
		 Person per = new Person();  //声明并实例化对象
		 //2、对象的使用
```

```
            per.setAge(!8);//修改属性内容
            per1.tell();      //进行方法的调用
      }
}
```

经过了分析之后可以发现，利用构造方法可以传递属性数据，于是现在进一步分析对象的产生格式：

- 定义对象的名称：类名称 对象名称 = null；
- 实例化对象：对象名称 = new 类名称()；

如果这个时候只是通过实例化对象来进行类的操作也是可以的，而这种形式的对象由于没有名字就称为匿名对象。

范例：观察匿名对象

```
public class JavaDemo{
      public static void main(String[] args) {
            new Person("sanzhang",10).tell();  //进行方法的调用
      }
}
```

此时依然通过了对象进行了类中tell()方法的调用，但是由于此对象没有任何的引用名称，所以该对象使用一次之后就将成为垃圾，而所有的垃圾将被GC进行回收与释放。

现在发现此时的程序里面已经存在构造方法了，那么下面通过一个程序来利用构造方法进行一次内存分析。

范例：编写一个分析程序

```
class Message
{
      private String title;
      public Message(String t){
            title = t;
      }
      public String getTitle(){
            return title;
      }
      public String setTitle(String t){      //具有修改功能
            title = t;
      }
}

class Person    //定义一个类
{
      private String name;      //人员的姓名
      private int age;          //人的年龄
      public Person(Message msg,int a){
            name = msg.getTitle();   //为类中的属性赋值(初始化)
            age = a;   //为类中的属性赋值(初始化)
      }
      public Message getInfo(){
            return new Message(name+": "+age);
```

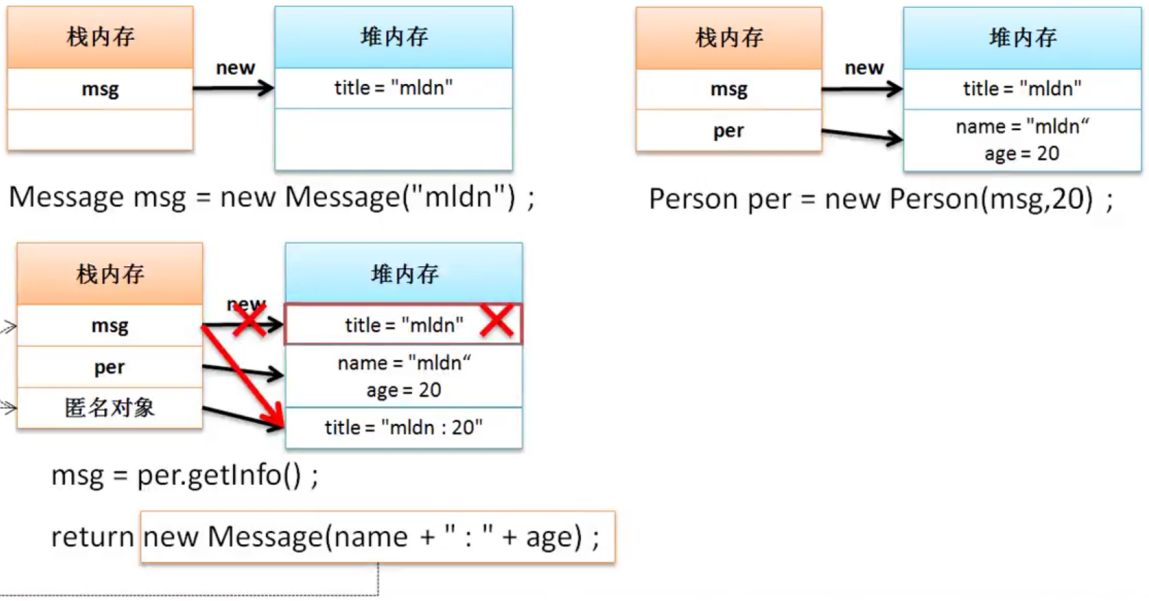
```
    }
    public void tell(){
        System.out.println("姓名： "+name+"年龄:"+age);
    }


}

public class JavaDemo{
    public static void main(String[] args) {
        Message msg = new Message("mldn");
        Person per = new Person(msg,20);
        msg =per.getInfo();
        System.out.println(msg.getTitle());
    }
}
```

下面通过此程序进行一个简短的内存分析。

内存分析



只要是方法都可以传递任意的数据类型(基本数据类型、引用数据类型)。