Java 中的 constructor

1、构造函数的特点

构造函数有以下特点：

（1）、构造函数名与类名相同；

（2）、构造函数不返回任何值，也没有返回类型，不能有任何非访问性质的修改符；

（3）、每一类可以有零个或多个构造方法；

（4）、构造方法在创建对象时自动执行，一般不用显示地直接调用。

其次，就上面几个需要说明的特点进一步进行解释：

特点一：构造函数名与类名相同。这个很简单，只要知道 Java 语言是区分大小写即可；

特点二：这一特点需要说明，并加以强调。构造函数不返回任何值，也没有返回类型（有了返回类型的话就是不是构造方法了，而是实例方法），因此在构造函数前面不可添加各种基本数据类型，也不可添加引用类型。

和实例方法一样，构造器可以有任何访问的修饰符，public、private、protected 或者没有修饰符，都可以对构造方法进行修饰。不同于实例方法的是构造方法不能有任何非访问性质的修饰符修饰，例如 static、final、synchronized、abstract 等都不能修饰构造方法。（解释：构造方法用于初始化一个实例对象，所以 static 修饰是没有任何意义的；多个线程不会同时创建内存地址相同的同一个对象，所以 synchronized 修饰没有意义；构造方法不能被子类继承，所以 final 和 abstract 修饰没有意义。）

特点三：每一类可以有零个或多个构造方法。如果类没有构造函数，编译器会自动添加默认的无参构造函数，当调用默认的构造函数时，就会利用构造函数为类的成员变量进行初始化，当然不同的类型其默认的初始值不同。一旦用户定义了构造函数，则不会产生默认的构造函数。当有多个构造函数时，系统会根据产生对象时，所带参数的不同而选择调用不同的构造函数。

我们知道，java 语言中规定每个类至少要有一个构造方法，为了保证这一点，

当用户没有给 java 类定义明确的构造方法的时候，java 为我们提供了一个默认的构造方法，这个构造方法没有参数，修饰符是 public 并且方法体为空。

其实默认的构造方法还分为两种，一种就是刚刚说过的隐藏的构造方法，另一种就是显示定义的默认构造方法。

如果一个类中定义了一个或者多个构造方法，并且每一个构造方法都是带有参数形式的，那么这个类就没有默认的构造方法，看下面的例子。

```
//DemoWithImplicitDefaultConstructor中有一个隐式的默认构造方法（编译器提供）
public class DemoWithImplicitDefaultConstructor {
private String demoName;
public String getDemoName() {
	return demoName;
}
public void setDemoName(String demoName) {
	this.demoName = demoName;
}
}

//DemoWithExplicitDefaultConstructor中有一个显示的默认构造函数，编译器不会提供隐式的默认构造函数
public class DemoWithExplicitDefaultConstructor {
   private String demoName;
   public String getDemoName() {
      return demoName;
   }
   public void setDemoName(String demoName) {
      this.demoName = demoName;
   }
   public DemoWithExplicitDefaultConstructor(){
   setDemoName("demoWithExplicitDefaultConstructor");
```

```
    }
}

//当类中有带参的构造函数时，编译器就不会提供隐式的默认构造函数
public class DemoNoDefaultConstructor {
    private String demoName;
    public String getDemoName() {
        return demoName;
    }
    public void setDemoName(String demoName) {
        this.demoName = demoName;
    }
    public DemoNoDefaultConstructor(String demoName){
        setDemoName(demoName);
    }
}
```

上面的三个类中 DemoWithImplicitDefaultConstructor 有一个隐式的默认构造方法，下列语句 DemoWithImplicitDefaultConstructor demo1=new DemoWithImplicitDefaultConstructor（）合法

DemoWithExplicitDefaultConstructor 有一个显示的默认构造方法，所以以下语句 DemoWithExplicitDefaultConstructor demo2=new DemoWithExplicitDefaultConstructor()；合法。

DemoNoDefaultConstructor 没有默认的构造方法，下列语句 DemoNoDefaultConstructor demo3=new DemoNoDefaultConstructor（）不合法，执行会编译错误

特点四：构造方法在创建对象时自动执行，一般不用显示地直接调用。构造函数的作用是对类的成员变量进行初始化操作，因此都是在创建类的对象是自动

执行。

2、构造方法与实例方法的区别

2.1、主要的区别在于三个方面：修饰符、返回值、命名

（1）、和实例方法一样，构造器可以有任何访问的修饰符，public、private、protected 或者没有修饰符，都可以对构造方法进行修饰。不同于实例方法的是构造方法不能有任何非访问性质的修饰符修饰，例如 static、final、synchronized、abstract 等都不能修饰构造方法。（解释：构造方法用于初始化一个实例对象，所以 static 修饰是没有任何意义的；多个线程不会同时创建内存地址相同的同一个对象，所以 synchronized 修饰没有意义；构造方法不能被子类继承，所以 final 和 abstract 修饰没有意义。）

例如，在单例模式的实现中经常把构造函的访问修饰符改为 private，以达到禁止其它对象创建该类的实例。

（2）、返回类型是非常重要的，实例方法可以返回任何类型的值或者是无返回值（void），而构造方法是没有返回类型的，void 也不行。

（3）、至于命名就是构造方法与类名相同，当然了实例方法也可以与类名相同，但是习惯上我们为实例方法命名的时候通常是小写的，另一方面也是与构造方法区分开。而构造方法与类名相同，所以首字母一般大写。

下面看的例子熟悉一下：

```
/**
 * 实例方法的名字可以与类名同名
 *
 * */
public class DemoMethodNameSameAsClassName {
    private String demoName;

    public String getDemoName() {
```

```
        return demoName;
    }

    public void setDemoName(String demoName) {
        this.demoName = demoName;
    }

    //构造方法
    public DemoMethodNameSameAsClassName(){
        demoName = "Constructor method";
        System.out.println("DemoMethodNameSameAsClassName
constructor...");
        System.out.println("demoName is " + getDemoName());
    }

    //不是构造方法，而是实例方法。因为其有返回类型void(为了与构造方法
区分开。而构造方法与类名相同，所以首字母一般大写。)
    public void DemoMethodNameSameAsClassName(){
        demoName = "Instance method";
        System.out.println("DemoMethodNameSameAsClassName :
void");
        System.out.println("demoName is " + getDemoName());
    }

    //不是构造方法，而是实例方法。因为其有参数和返回值(为了与构造方法
区分开。而构造方法与类名相同，所以首字母一般大写。)
    public String DemoMethodNameSameAsClassName(String title){
        System.out.println("DemoMethodNameSameAsClassName :
String");
        System.out.println("demoName is " + getDemoName());
```

```
        return getDemoName() + ":" + title;
    }

    //普通方法
    public void run(){
        System.out.println("the instance of
DemoMethodNameSameAsClassName run...");
    }
}
```

测试类：

```
import static org.junit.Assert.*;

import org.junit.Test;

public class DemoMethodNameSameAsClassNameTest {

    @Test
    public void testRun() {
        DemoMethodNameSameAsClassName demo = new
DemoMethodNameSameAsClassName();
        demo.run();
        /*
         * 测试输出：DemoMethodNameSameAsClassName constructor...
                demoName is Constructor method
                the instance of DemoMethodNameSameAsClassName
run...

         * */
    }
```

```
    @Test
    public void testDemoMethodNameSameAsClassName(){
        DemoMethodNameSameAsClassName demo = new
DemoMethodNameSameAsClassName();
        demo.DemoMethodNameSameAsClassName();

    System.out.println(demo.DemoMethodNameSameAsClassName("test
"));
        /*
         * 测试输出：DemoMethodNameSameAsClassName constructor...
                demoName is Constructor method
                DemoMethodNameSameAsClassName : void
                demoName is Instance method
                DemoMethodNameSameAsClassName : String
                demoName is Instance method
                Instance method:test
                说明：实例方法可以与类名重名
         * */
    }
}
```

2.2、实例方法和构造方法中 this、super 的使用

2.2.1、"this"的用法

实例方法中可以使用 this 关键字，它指向正在执行方法的类的实例对象，当然 static 方法中是不可以使用 this 对象的，因为静态方法不属于类的实例对象；而构造方法中同样可以使用 this 关键字，构造器中的 this 是指向同一个对象中不同参数的另一个构造器。

让我们来看下面的一段代码：

```
public class DemoConstructorUseThis {
    private String demoName;

    public String getDemoName() {
        return demoName;
    }
    public void setDemoName(String demoName) {
        this.demoName = demoName;
    }
    public DemoConstructorUseThis (String demoName){
        this.demoName = demoName;
    }
    public DemoConstructorUseThis (){
        this("useThis");
    }
}
```

测试类：

```
import static org.junit.Assert.*;
import org.junit.Test;
public class DemoConstructorUseThisTest {
    @Test
    public void test() {
        DemoConstructorUseThis demo1 = new
DemoConstructorUseThis("demoNoUseThis");
        DemoConstructorUseThis demo2 = new
DemoConstructorUseThis();
        System.out.println(demo1.getDemoName() + "---" +
demo2.getDemoName());
        /*
```

```
        * 测试输出：demoNoUseThis---useThis
        */
    }
}
```

需要注意的三个地方是：

（1)、构造方法中通过 this 关键字调用其他构造方法时，那么这句代码必须放在第一行，否则会编译错误。

（2)、构造方法中只能通过 this 调用一次其他的构造方法。(同一个对象只能构造一个次)。

（3)、实例方法可以通过 this 关键字调用其它的实例方法(可以多次调用)，并且可以放在任意位置。

2.2.2、"super"的用法

实例方法和构造方法中的 super 关键字都用于去指向父类，子类构造方法中调用父类的构造方法和子类实例方法中的 super 关键字是去调用父类当中的某个方法，看下面的代码：

父类：

```
public class DemoConstructorUseSuperOfFather {
    public DemoConstructorUseSuperOfFather(){
        System.out.println("Constructor of father");
    }
    public DemoConstructorUseSuperOfFather(String s){
        System.out.println("Constructor of father:" + s);
    }
    public void fatherRun(){
        System.out.println("father run...");
    }
}
```

子类：

```
public class DemoConstructorUseSuperOfSon extends
DemoConstructorUseSuperOfFather{
    public DemoConstructorUseSuperOfSon(){
        super();
        //可以传参数
        //super("s1");
        //子类构造函数中调用父类的构造函数，super(..)只能放在第一行
        System.out.println("Constructor of son");

    }
    public void sonRun(){
        System.out.println("son run...");
        super.fatherRun();
        /*
         * 子类实例方法调用父类中的实例方法,super.methodName(..)可以
放在任意位置
         * */
    }

}
```

需要注意的三个地方是：

（1)、子类构造方法中通过 super 关键字调用父类构造方法时，那么这句代码必须放在第一行，否则会编译错误。并且可以通过传入相应的参数来调用父类的其它构造方法。

（2)、子类构造方法中只能通过 super 调用一次父类的构造方法。(同一个对象只能构造一个次)

（3)、子类的实例方法可以通过 super 关键字调用其父类的实例方法（可以多次调用)，并且可以放在任意位置。

3、构造函数的调用顺序

类的继承机制使得子类可以调用父类的功能，下面介绍类在继承关系的初始化顺序问题。我们在实例化子类对象时，程序会先调用父类的默认构造方法，然后再执行子类的构造方法。（不举例了）

类的成员变量、静态变量的构造函数的调用顺序测试代码如下：

```
public class DemoConstructorSequenceOfOne {
    public DemoConstructorSequenceOfOne (String str){
        System.out.println("The instance of
DemoConstructorSequenceOfOne : " + str);
    }

}

public class DemoConstructorSequenceOfTwo {
    DemoConstructorSequenceOfOne demoOne1 = new
DemoConstructorSequenceOfOne("demoOne1");
    DemoConstructorSequenceOfOne demoOne2 = new
DemoConstructorSequenceOfOne("demoOne2");
    static DemoConstructorSequenceOfOne demoOne3 = new
DemoConstructorSequenceOfOne("demoOne3");

    public DemoConstructorSequenceOfTwo(String str){
        System.out.println("The instance of
DemoConstructorSequenceOfTwo : " + str);
    }

}
```

测试类：

```
public class DemoConstructorSequenceTest {
    static DemoConstructorSequenceOfTwo demoTwo3 = new
DemoConstructorSequenceOfTwo("demoTwo3");

    @Test
    public  void testConstructorSequence() {
        System.out.println("The test start...");
        DemoConstructorSequenceOfTwo deomTwo1 = new
DemoConstructorSequenceOfTwo("demoTwo1");
        System.out.println("<------------------------分割线
------------------------->");
        DemoConstructorSequenceOfTwo deomTwo2 = new
DemoConstructorSequenceOfTwo("demoTwo2");
        /*
         * 输出为:
         * The instance of DemoConstructorSequenceOfOne : demoOne3
         * The instance of DemoConstructorSequenceOfOne : demoOne2
         * The instance of DemoConstructorSequenceOfTwo : demoTwo3
         * The test start...
         * The instance of DemoConstructorSequenceOfOne : demoOne1
         * The instance of DemoConstructorSequenceOfOne : demoOne2
         * The instance of DemoConstructorSequenceOfTwo : demoTwo1
         * <------------------------分割线
------------------------->
         * The instance of DemoConstructorSequenceOfOne : demoOne1
         * The instance of DemoConstructorSequenceOfOne : demoOne2
         * The instance of DemoConstructorSequenceOfTwo : demoTwo2
         */
    }
```

}

在实例化类的对象时，类中的成员变量会首先进行初始化，如果其中的成员变量有对象，那么它们也会按照顺序执行初始化工作。在所有类成员初始化完成后，才调用对象所在类的构造方法创建对象。构造方法作用就是初始化。

如果一个类中有静态对象，那么他会在非静态对象初始化前进行初始化，但只初始化一次。而非静态对象每次调用时都要初始化。

程序中主类的静态变量会在 testConstructorSequence（）方法执行前初始化。结果中只输出了一次 demoOne3,这也说明：如果一个类中有静态对象，那么它会在非静态对象前初始化，但只初始化一次。非静态对象每次调用时都要初始化。

总结初始化顺序：

（1）、主类的静态成员首先初始化。

（2）、主类的父类的构造方法被调用。

（3）、主类的非静态对象（变量）初始化。

（4）、调用主类的构造方法。