《面向对象》之静态（static）

笔记本： JAVA
创建时间： 2018/7/24 14:16 **更新时间：** 2018/8/5 14:24
作者： Debao Xu

静态关键字static的使用

先来一个程序：

```
class Students
{
	private int num; //关键字private用于对变量进行封装
	private String name;
	Students(int num,String name) // 构造函数用于初始化对象
	{
		this.num = num;   // this关键字用于区别成员变量和局部变量
		this.name = name;
	}
	public void show()
	{
		System.out.println(num+" "+name);
	}
	public void sleep()
	{
		System.out.println("huhu");
	}

}
public class StudentsDemo
{
	public static void main(String[] args)
	{
		Students s = new Students(26,"Rex");
		s.show();
		s.sleep();
	}
}
运行结果：
26 Rex
huhu
```

这个程序的编译和运行都没有问题，但是内存上有问题，问题就是在调用 s.sleep() 这个方法时并没有用到对象中数据（即num = 26，name = Rex）,所以我们想要不创建对象就可以调用函数sleep()，实现这种方法的途径就是，在该函数前面加上关键字static，然后就可以通过类名来访问类中的函数。即有：

```
public static void sleep()
{
   System.out.println("huhu");
}
```

这样就可以直接通过类名来调用类中的方法而不需要创建对象，如下：

```
Students.sleep();
```

那么，某个函数什么时候需要用静态关键字static修饰呢？
答：如果该函数没有访问过对象中的属性时（如上面输出时只是打印了“huhu”），就需要用静态修饰。

上面是关于用 static 来修饰函数的，再有，static也是可以用来修饰成员变量的，如下：

```
static String country  = "CN";
.........
System.out.println(Students.country); //在主方法中可以直接调用
```

总结：
静态关键字static是什么？　成员修饰符。
有什么特点？
1、被静态修饰的成员，可以直接被类名所调用。
2、静态成员优先于对象存在。（即先加载静态成员，再创建对象）
3、静态成员随着类的加载而加载，随着类的消失而消失，静态成员的生命周期很长。

注意事项：
1、静态方法只能访问静态成员，不能访问非静态成员。这就是静态方法的访问局限性，而非静态方法可以访问静态成员。
2、静态的方法中不能出现this或者 super关键字，原因是：静态的数据先在，对象后在，如果静态中有this的话，那么这个this就要去找对应的对象，而此时对象还没有被创建，这就矛盾了。
3、主函数是静态的。

什么时候用static？
　成员变量：如果数据在所有对象中的都是一样的（例如对于“国籍”这一属性，所有的中国人都是“中国”），则直接静态修饰。
　成员函数：如果函数没有访问过对象中的属性数据，那么该函数就用静态修饰。

成员变量和静态变量的区别？
1、名称上的区别
成员变量也叫实例变重。
静态变量也叫类变量。
2、内存存储上的区别。
成员变量存储到堆内存的对象中。静态变量存储到方法区的静态区中。
3、生命周期不同。
成员变量随着对象的出现而出现，随着对象的消失而消失。静态变量随着类的出现而出现，随着类的消失而消失。

静态代码块、构造代码块、局部代码块

静态代码块：随着类的加载而执行，而且只执行一次

作用：给类进行初始化的。【复习】：构造函数是给对象初始化的，构造函数也只执行一次
举例：

```
class CodeDemo
{
	static int num;
	static
	{
		num = 10;
		System.out.println("StaticCode");
	}
	static void show()
	{
		System.out.println("A");
	}
}
public class StaticCode
{
	public static void main(String[] args)
	{
		CodeDemo.show();
		CodeDemo.show();
	}
}
运行结果：
StaticCode
A
A
```

从上面的运行结果可以看出，先执行了静态代码块中的内容，对类进行了初始化，输出了 StaticCode，接着在主函数中执行了两次CodeDemo.show().

构造代码块

构造函数是给对应的对象进行初始化的，而构造代码块是给所有的对象进行初始化的。

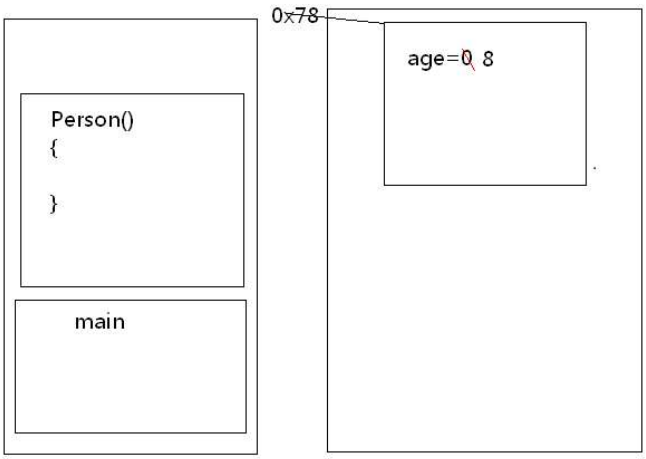
创建一个对象的流程（例如下面创建的对象p，及对应的内存图）：
1、加载指定的字节码文件（.class文件）进内存
2、通过new在堆内存中开辟空间，分配首地址值（0x78）
3、对对象中的属性进行默认初始化（age = 0）
4、调用与之对应的构造函数，构造函数压栈（Person()压栈）
5、构造函数中执行隐式的语句super() 访问父类中的构造函数
6、对属性进行显示初始化（age = 8）。
7、调用类中的构造代码块（{ Sys.out.println("constructor code run。。。"+age); }）
8、执行构造函数中自定义的初始化代码（Person()）
9、初始化完毕，将地址赋给指定的引用（此处赋给引用 p）

```
创建一个对象p， Person p = new Person();
class Person
{
        private int age = 8;//显示初始化。

        {//构造代码块。给所有对象进行初始化的。
          System.out.println("constructor code run。。。"+age);

        }
        Person()
        {
          1,super();//调用父类构造函数。没学。
          2,显示初始化。
          3,构造代码块初始化。
          System.out.println("person run");
        }
        Person(int age)
        {
          this.age = age;
          System.out.println("Perosn(age) run");
        }

}
```



局部代码块

局部代码块可以控制局部变量的生命周期，如下，

```
public class StaticCode
{
        public static void main(String[] args)
        {
                CodeDemo.show();
                CodeDemo.show();

                // 局部代码块
                {
                        int age = 34;
                        System.out.println("hehe");
                }
        }
}
```

可以看出变量 age = 34 只能在局部代码块中有效

单例设计模式

解决的问题是：保证一个类的对象在内存中的唯一性。

应用场景：当多个程序（假如有A、B两个程序）都在操作同一个配置文件时，程序B需要对程序A操作后的结果进一步操作。然而，这里有一个前提就是，这些程序操作的数据都存储在配置文件的对象中，那么这就要求程序A和程序B操作的配置文件对象是同一个。

问题：怎么保证这个类只产生一个对象呢?
思路：
1、这里的问题就是每一个程序都可以通过new来创建该类的对象，那么关于这个类的对象的数量就无法控制？解决方法就是：不让其他的程序new对象，不就行了吗?
2、那第一步的问题也产生了，就是其他的程序不就没有对象了吗？解决方案：在本类中自己new一个本类对象，这样的好处是，不让别的程序new对象，可以实现对象数量的控制
3、当然为了解决第一步的问题，我们需要对外提供让其他程序获取该对象的方式就可以了。

解决步骤：
1、为了不让其他的程序new该类对象，我们就将该类中的构造函数私有化（这样的话，如果其他程序想创建对象，就无法利用构造函数进行初始化，也就没法创建出对象了）
2、在本类中new一个对象
3、定义一个方法返回该对象

代码体现：

```
class Single
{
	private Single() {}  //构造函数私有化（不让其他程序创建对象）

	static Single S = new Single();  //在本类中创建一个对象（为了给主函数调用，必须是static的）
	static Single getInstance() //为了能够用类名进行访问，该方法也必须是static的。另外，为什么要通过一个方法来访问对象S，而不直接访问S呢，原因就是：通过方法访问S时，能够达到“可控”的效果
	{
		return S;
	}
}
public class SingleDemo
{
	public static void main(String[] args)
	{
		Single s1 = Single.getInstance();
		Single s2 = Single.getInstance();
		System.out.println(s1 == s2); // 运行结果为true，表明s1和s2是同一个对象
	}
}
```

运行结果：

```
ture
```

单例模式的具体应用

```
class Student
{
	private String name; //私有化变量
	private Student(String name)  //私有化构造函数
	{
		this.name = name;
	}
	private static Student S = new Student("Rexall");//创建对象并初始化
	static Student getIstance() //调用函数，返回对象S
	{
		return S;
	}

	public void setName(String name)
	{
		this.name = name;
	}
	public String getName()
	{
```

```
                return name;
        }
}
public class StudentDemo
{
        public static void main(String[] args)
        {
                Student s1 = Student.getIstance();//调用函数返回对象
                System.out.println(s1.getName());//将对象中的初始化值输出

                s1.setName("Debao");//改变同一个对象的属性值
                System.out.println(s1.getName());//输出改变之后的属性值
        }
}
```

输出结果

```
Rexall
Debao
```