Chp8 三个修饰符

本章导读

本章将向读者介绍 Java 中的三个非常重要的修饰符：static、final 和 abstract。

学习修饰符，应该始终明确：该修饰符能够修饰什么程序组件，而修饰某个组件的时候，又表示了什么含义。

1 static

static 修饰符也被称为静态修饰符。这个修饰符能够修饰三种程序组件：属性、方法、初始化代码块。static 在修饰这三个不同的组件的时候，分别表示不同的含义。

需要注意的是，static 不能修饰局部变量和类。

下面根据修饰的组件不同，我们分别阐述 static 修饰属性、方法以及初始化代码块所代表的含义。

1.1 静态属性

static 修饰属性，则该属性就成为静态属性。静态属性是全类公有的属性。例如，有如下代码：

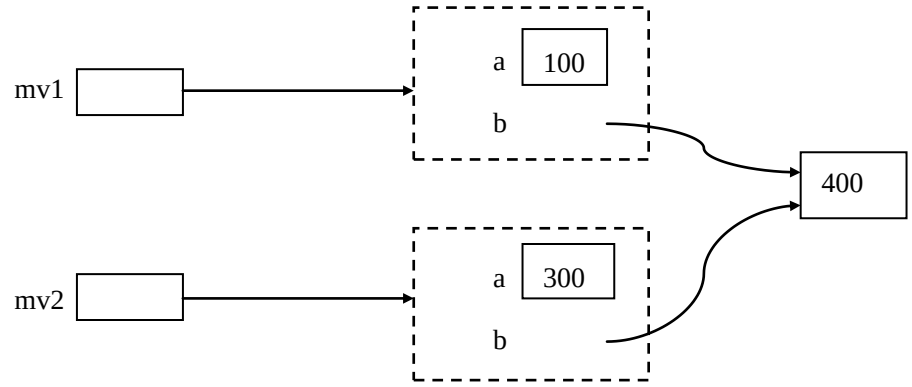
```
class MyValue{
    int a;
    static int b;
}
public class TestStatic{
    public static void main(String args[]){
        MyValue mv1 = new MyValue();
        MyValue mv2 = new MyValue();
        mv1.a = 100;
        mv1.b = 200;
        mv2.a = 300;
        mv2.b = 400;
        System.out.println(mv1.a);
        System.out.println(mv1.b);
        System.out.println(mv2.a);
        System.out.println(mv2.b);
    }
}
```

编译运行之后，输出结果为：

```
100
400
300
400
```

要注意，400 这个数字出现了两次，200 没有出现。

原因在于：b 属性是一个静态属性。MyValue 类的所有对象，都公用一个 b 属性，每一个对象的 b 属性，都指向同一块内存。如下图所示：



可以看到，mv1 对象的 a 属性和 mv2 对象的 a 属性，彼此之间是相互独立的。而 mv1 和 mv2 两个对象的 b 属性，指向的是内存中的同一块区域。因此，当代码执行到 mv1.b = 200; 时，把这块区域设置为 200；执行到 mv2.b = 400 时，把内存中的同一块区域设置为 400。这时，无论通过 mv1.b，还是 mv2.b，读取到的都是同一块内存区域的值，因此输出语句输出的都是 400。

每个对象的静态属性都指向同一块内存区域，事实上，这个属性不属于任何一个特定对象，而属于“类”。因此，可以使用类名直接调用静态属性。例如，可以把上面的 TestStatic 程序改写成下面的样子：

```
class MyValue{
    int a;
    static int b;
}
public class TestStatic{
    public static void main(String args[]){
        MyValue mv1 = new MyValue();
        MyValue mv2 = new MyValue();
        mv1.a = 100;
        MyValue.b = 200;
        mv2.a = 300;
        MyValue.b = 400;
        System.out.println(mv1.a);
        System.out.println(MyValue.b);
        System.out.println(mv2.a);
        System.out.println(MyValue.b);
    }
}
```

这段代码中，对 b 属性的使用都是用类名直接调用。

在实际编程过程中，为了避免错误和误会，对静态属性的使用，应当尽量用“类名直接调用”的写法。用这种写法能够把静态属性和实例变量加以区分，从而提高程序的可读性。

要注意的是，由于静态属性不属于任何一个对象，因此，在一个对象都没有创建的情况之下，照样可以使用静态属性。这个时候，就只能用类名直接访问静态属性。例如下面的代

码：

```
class MyValue{
    int a;
    static int b;
}
public class TestStatic{
    public static void main(String args[]){
        //没有创建任何 MyValue 对象就直接使用属性
        MyValue.b = 200;
        System.out.println(MyValue.b);
    }
}
```

从概念上怎么来理解静态属性呢？举一个生活中的例子，例如：在一个班级中，每个学生对象都有一个属性：“Java 老师”。对于同一个班级的同学来说，Java 老师这个属性总是一样的，即使上课过程中更换老师，班级中所有学生对象的“Java 老师”属性都会进行同样的改变。因此，如果将学生看做是一个类，“Java 老师”这个属性，就属于这个类的“公有”属性，也就是静态属性。

最后，介绍一下几个名词。在之前的课程中，我们接触过的“属性”就是指的实例变量。现在，我们接触到了静态属性这个概念，再提“属性”这个概念，就分为静态属性和非静态属性两种。其中，静态属性也可以叫“类变量”，而非静态属性就是我们所说的“实例变量”。这几个名词的关系如下：

1.2 静态方法

用 static 修饰的方法称之为静态方法。首先我们看一个代码的例子，这个例子中反映了静态方法与非静态方法分别能访问什么样的属性和方法。

```
class TestStatic{
    int a = 10;              //非静态属性
    static int b = 20;       //静态属性
    public void ma(){}   //非静态方法
    public static void mb(){} //静态方法

    public void fa(){    //fa 是一个非静态方法
        System.out.println(a);  //非静态方法能够访问非静态属性
        System.out.println(b);  //非静态方法能够访问静态属性
        ma();  //非静态方法中，能够调用非静态方法
        mb();  //非静态方法中，能够调用静态方法
    }
    public static void fb(){    //fb 是一个静态方法
        System.out.println(a);  //编译错误 静态方法中不能访问非静态属性
        System.out.println(b);  //静态方法中可以访问静态属性
```

```
        ma(); //编译错误， 静态方法中调用非静态方法
        mb(); //静态方法中可以调用静态方法
    }
}
```

从上面这个例子中，我们可以总结出如下规律：

在非静态方法中，无论方法或属性是否是静态的，都能够访问；

而在静态方法中，只能访问静态属性和方法。

静态方法和静态属性统称为静态成员。以上的规律可以记成：静态方法中只能访问静态成员。需要注明的是，在静态方法中不能使用 this 关键字。

除此之外，静态方法与静态属性一样，也能够用类名直接调用。例如下面的代码：

```
public class TestStaticMethod{
    public static void main(String args[]){
        TestStatic.fb();
    }
}
```

静态方法是属于全类公有的方法。从概念上来理解，调用静态方法时，并不针对某个特定的对象，这个方法是全类共同的方法。例如，对于班级来说，有一个“办联欢会”的方法。想要办好一个班级联欢会，不能依靠某个特定同学（也就是某个对象）的努力，而应该是全班同学共同配合完成。因此，这个方法不属于某个特定对象，而是“全类公有”的方法。

除了上面所说的特点之外，静态方法还有一个非常重要的特性，这个特性跟方法覆盖有关。看下面的代码：

```
class Super{
    public void m1(){
        System.out.println(“m1 in Super”);
    }
    public static void m2(){
        System.out.println(“m2 in Super”);
    }
}
class Sub1 extends Super{
    public void m1(){ //非静态方法覆盖非静态方法，编译通过
        System.out.println(“m1 in Sub”);
    }
    public static void m2(){ //静态方法覆盖静态方法，编译通过
        System.out.println(“m2 in Sub”);
    }
}

class Sub2 extends Super{
    public static void m1(){} // 静态方法覆盖非静态方法，编译出错！
    public void m2(){} // 非静态方法覆盖静态方法，编译出错！
}
```

根据上面的例子，可以发现：静态方法只能被静态方法覆盖，非静态方法只能被非静态

方法覆盖。

除此之外，我们再写一个程序来测试一个 Super 类和 Sub1 类。

```
public class TestStaticOverride{
    public static void main(String args[]){
        Super sup = new Sub1();
        sup.m1();
        sup.m2();
        Sub1 sub = (Sub) sup;
        sub.m1();
        sub.m2();
    }
}
```

编译运行，结果如下：

```
m1 in Sub
m2 in Super
m1 in Sub
m2 in Sub
```

注意到，对于 m1 这个非静态方法来说。无论引用类型是 Super 还是 Sub1，调用 m1 方法，结果都是 m1 in Sub。这是上一章我们所讲的多态特性：运行时会根据对象的实际类型调用子类覆盖以后的方法。

然后，如果是 m2 这个静态方法，情况则大大不同。在引用类型为 Super 时，调用的是 m2 in Super；当引用类型为 Sub1 时，调用的是 m2 in Sub。这是静态方法很重要的一个特性：静态方法没有多态。当我们对一个引用调用静态方法的时候，等同于对这个引用的引用类型调用静态方法：所以代码

```
super.m2();  相当于  Super.m2();
sub.m2(); 相当于 Sub.m2();
```

总结一下静态方法的几个性质：

1. 静态方法可以用类名直接调用。
2. 静态方法中只能访问类的静态成员。
3. 静态方法只能被静态方法覆盖，并且没有多态。

1.3 静态初始化代码块

static 修饰符修饰初始化代码块，就称之为静态初始化代码块。

首先，简单介绍一下初始化代码块。看如下的代码例子

```
public class MyClass{
    {
        //此处为初始化代码块
    }
    int a;
    public MyClass(){
        System.out.println("MyClass()");
    }
}
```

在类的里面，所有方法的外面定义的代码块称之为初始化代码块（如上面的例子所示）。

能够用 static 修饰初始化代码块，使之成为静态初始化代码块。例如：

```
public class MyClass{
    static {
        //此处为静态初始化代码块
        System.out.println("In MyClass Static");
    }
    int a;
    public MyClass(){
        System.out.println("MyClass()");
    }
}
```

这样就定义好了静态初始化代码块。

那么这个代码块什么时候执行呢？静态初始化代码块执行的时机非常特别，下面我们就来介绍这个问题。

1.3.1 类加载

首先我们来简单考虑一下下面的代码。

```
//TestStudent.java
01: class Student{
02:     static {
03:         System.out.println("in Student static");
04:     }
05:     public Student(){
06:         System.out.println("Student()");
07:     }
08: }
09: public class TestStudent{
10:     public static void main(String args[]){
11:         Student stu1 = new Student();
12:         Student stu2 = new Student();
13:     }
14: }
```

上面的 TestStudent.java 文件，编译生成两个.class 文件：一个 Student.class，一个 TestStudent.class。

生成这两个.class 文件之后，执行

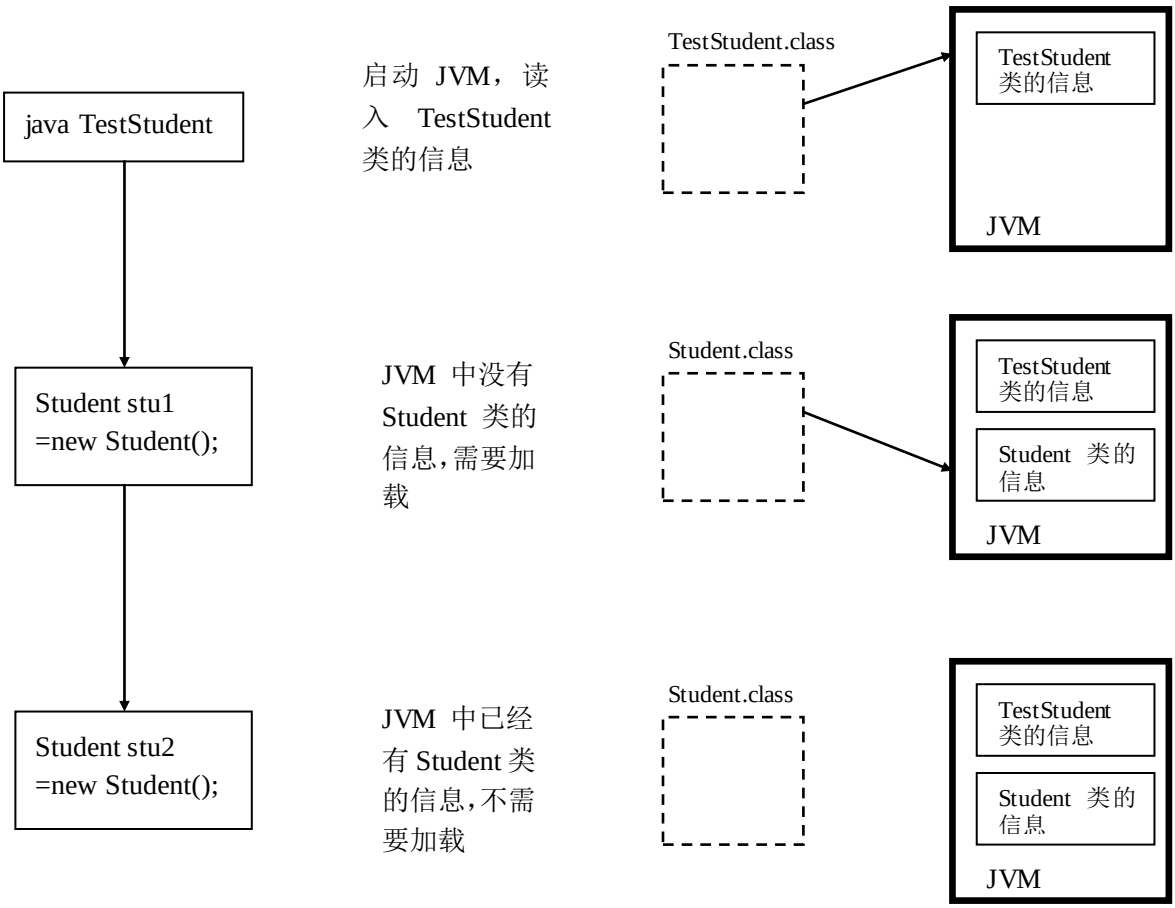
```
java TestStudent
```

则，首先启动 JVM，然后在硬盘上找到 TestStudent.class 文件，读入这个文件，并开始解释执行。此时，JVM 中只有 TestStudent.class 这个类的信息。

然后，执行主方法。在第 11 行时，遇到 Student stu1 = new Student();这句话。这个语句要求创建一个 Student 类的对象，但是此时，在 JVM 中，只有 TestStudent 类的信息，而没有 Student 类的信息！

这个时候，JVM 会自动的通过 CLASSPATH 环境变量，去硬盘上寻找相应的 Student.class 文件。当 JVM 找到这个文件之后，会把这个文件中所保存的 Student 类的信息读入到 JVM 中，并保存起来。此时，JVM 中保存有 Student 类和 TestStudent 类两个类的信息。示意图如

下：



当创建了第一个 Student 对象之后，创建第二个 Student 对象时，由于 JVM 中已经存在 Student 类的信息，因此就不需要重新读入 Student.class 文件。

类加载就是把.class 文件读入 JVM 的过程。也就是说，当 JVM 第一次遇到某个类时，会通过 CLASSPATH 找到相应的.class 文件，读入这个文件并把类的信息保存起来，这个过程叫做类加载。

而静态初始化代码块会在类加载的时候执行。

因此，执行 TestStudent，会输出：

```
in Student Static
Student()
Student()
```

当创建第一个 Student 对象时，由于是第一次遇到 Student 类，因此 JVM 会对 Student 类进行类加载，在类加载的时候，Student 类的静态初始代码块将会运行。此时会输出：in Student Static。

类加载完毕之后，创建对象时，会调用对象的构造方法，因此输出：Student()。调用完构造方法之后，第一个对象创建完毕。

创建第二个对象时，由于 JVM 中已经有 Student 类的信息，此时不需要类加载。创建对象时，调用对象的构造方法，于是输出 Student()。

另外，类加载除了读取类的信息，执行静态初始化代码块之外，还会为类的静态属性分配空间，并初始化其值为默认值。

2 final

final 属性能够修饰变量、方法和类。注意，所谓变量，既包括属性，又包括局部变量。也就是说，final 能够修饰属性、局部变量、方法参数（注意方法参数也是特殊的局部变量）。

2.1 常量

用 final 修饰的变量则称为常量。怎么来定义常量呢？可以把常量理解为：一旦赋值，其值不能改变的变量。

例如下面这个程序：

```
public class TestFinalVar{
    public static void main(String args[]){
        final int N; //定义一个常量，注意常量名要求全大写
        N = 100;      //为常量第一次赋值
        N++; //编译出错，为常量赋值之后，常量值不可以改变
        System.out.println(N); //获得常量的值
    }
}
```

对于基本类型的变量而言，所谓的不能改变，指的是不能改变变量的值。下面看另一个例子。

```
01: class MyValue{
02:     int value;
03: }
04: public class TestMyValue{
05:     public static void main(String args[]){
06:         final MyValue MV; //定义一个 MyValue 类型的常量
07:         MV = new MyValue(); //创建一个对象
08:         MV.value = 10;     //为 MV 的 value 属性赋值
09:         MV.value = 100;    //为 MV 的属性再次复制！编译通过
10:         MV = new MyValue(); // 编译错误 MV 不能指向另一个对象！
11:     }
12: }
```

如上面的例子，MV 为一个对象类型的变量。对象类型的变量中，保存的是对象的地址，因此所谓“一旦赋值，不能改变”，是指的对象的地址不能改变。也就是说，对于对象类型的常量而言，一旦指向某个对象以后，就不能指向其他的对象。在 07 行，创建了一个对象，并让 MV 常量指向这个新对象。因此，在 10 行，不能创建新对象，让 MV 指向这个新对象。

但是，让 MV 指向一个对象之后，完全可以改变这个对象的属性，如第 08、09 行显示。这并不违反常量“一旦赋值，不能改变”的要求。

了解了 final 修饰不同类型变量的情况，下面我们讨论一下 final 修饰属性时的情况。例如，有如下代码：

```
class MyValue{
    final int value;
}
```

上面演示了 final 修饰实例变量的情况。要注意的是，上面这段代码无法编译通过！

考虑一下对象创建的过程。对于一般的实例变量而言，创建时第一步是分配空间，在分

配空间的同时，还会为实例变量赋一个默认值。然而，对于 final 类型的实例变量而言，如果在分配空间的同时赋默认值的话，那根据“一旦赋值，不能改变”的定义，final 实例变量的值在成为默认值之后将不能改变。这样的话，final 的实例变量将只能有默认值，不能再赋值为其他的值。在上述例子中，value 属性的值将被赋值为 0，且永远为 0。这显然并不是程序员所希望的。

正因为如此，对于 final 属性而言，虚拟机在分配完空间之后，将不会为其赋默认值，从而把为 final 属性赋值的那一次机会留给程序员。

但是，Java 语言规定，一旦对象创建完成，则对象的 final 类型的属性也就不能赋值了。因此，对于 final 类型的实例变量，能够赋值的时机，就是在分配空间之后，对象创建完成之前。实际上，在这段时间内，有两次赋值的机会：1. 初始化属性；2. 调用构造方法。

下面的两个例子演示了利用初始化属性以及调用构造方法为 final 属性赋值时的情况。

```
//使用属性初始化为 final 属性赋值
class MyValue1{
    final int value  = 100; //初始化属性时赋值
}
//使用构造方法为 final 属性赋值
class MyValue2{
    final int value;
    public MyValue2(int value){
        this.value = value;
    }
}
```

由上面这两个代码例子，我们可以学习到怎样为 final 属性赋值。然而，有三个要注意的细节。

首先，为 final 属性赋值有两个时机。对于程序员来说，必须抓住两个时机中的一次，但是不能试图两次机会都抓住。例如下面的代码例子：

```
//注意！下面的代码会编译出错！
class MyValue3{
    final int value = 100;
    public MyValue3(int value){
        this.value = value;
    }
}
```

上面这段代码，由于既利用了初始化属性赋值，又使用了构造方法赋值，因此会编译出错！

其次，对于使用构造方法赋值的情况。如果有多个构造方法，则多个构造方法都必须对 final 属性赋值。

例如下面的例子：

```
class MyValue4{
    final int value;

    public MyValue4(){
```

```
    }

    public MyValue4(int value){
        this.value = value;
    }
}
```

这段代码将编译失败，因为在 MyValue4 类的无参构造方法中，没有为 value 属性赋值。如果用户利用这个构造方法来创建对象，那么这个对象的 value 属性将没有赋值，这显然是错误的。

因此，在一个类中所有的构造方法里，都要对 final 属性赋值。这样就保证了，当一个对象创建的时候，无论调用的是哪一个构造方法，final 属性都被正确的赋值了。

上面是 final 属性的介绍。接下来考虑一个初始化 final 属性的问题。假设程序员选择了在定义属性的时候直接对 final 属性进行赋值，如下面代码：

```
class MyValue6{
    final int value = 200;
}
public class TestMyValue6{
    public static void main(String args[]){
        MyValue6 mv1 = new MyValue6();
        MyValue6 mv2 = new MyValue6();
    }
}
```

考虑上面的代码。在这段代码中，创建了两个 MyValue6 类型的对象。这两个对象的空间中，分别保留着一块内存空间，用来保存 value 属性。

然而，由于 value 属性在初始化的时候被直接赋值为 200，因此对于所有对象来说，value 属性的值都为 200。并且，由于 final 属性的含义，所有对象的 value 属性都不能改变。因此，造成了这样的局面：每个对象中都保留了一块空间用来存放 value 属性，这个 value 属性的值都一样，而且修改不了。

这样，实际上造成了内存空间的浪费。由于这种属性全类所有对象都一致，因此可以把这个属性写成 static 的。即把 MyValue6 改为：

```
class MyValue6{
    final static int value = 200;
}
```

要注意的是，由于 static 属性是在类加载的时候分配空间的，因此静态的 final 属性不能在构造方法中赋值。我们可以选择在定义这个属性的时候赋值，或是在静态初始代码块中为这个属性赋值。

2.2 final 方法

相对 final 属性，final 方法的含义相对简单。final 修饰方法，表示该方法不能被子类覆盖。例如以下代码：

```
class Super{
    public void m1(){}
    public final void m2(){}
}
```

```
class Sub extends Super {
    public void m1(){} // 能够覆盖父类方法
    public void m2(){} // 编译出错！无法覆盖父类方法！
}
```

如上面的代码例子所示，Super 类中有两个方法，m1 方法没有被 final 修饰，因此子类方法能够覆盖父类方法。m2 方法被 final 修饰，因此子类方法不能够覆盖父类方法。

2.3 final 类

final 修饰符也可以用来修饰类。final 类表示这个类不能被继承。例如：

```
final class MyClass{
}
class Sub extends MyClass{} // 编译出错，无法继承一个 final 类
```

如上面的代码所示，final 类不能被继承。

要注意区分 final 修饰方法和 final 修饰类的区别。一个类中有 final 方法，这个类能够被继承，但是无法覆盖 final 方法；而一个类如果本身就是 final 的，则这个类无法被继承，这样的话，它所有方法都无法被覆盖。

3 abstract

abstract 可以用来修饰类和方法。abstract 单词本身表示“抽象”，是 java 中一个很重要的修饰符。

3.1 抽象类

abstract 修饰类，则这个类就成为一个抽象类。抽象类的特点是：抽象类只能用来声明引用，不能用来创建对象。例如下面的例子：

```
abstract class MyAbstract{
    public void m(){}
}
public class TestAbstract1{
    public static void main(String args[]){
        MyAbstract ma;   //可以声明抽象类的引用类型
        ma = new MyAbstract(); // 编译错误 不能创建抽象类的对象
    }
}
```

如上面的代码所示，抽象类可以声明引用，但是不能用来创建对象。

这样的类有什么用呢？虽然抽象类不能创建对象，但是抽象类可以被继承，从而创建子类的对象。例如下面的例子：

```
abstract class MyAbstract{
    public void m(){}
}
class MySubClass extends MyAbstract{
}
```

```
public class TestAbstract1{
    public static void main(String args[]){
        MyAbstract  ma;   //可以声明抽象类的引用类型
        // ma = new MyAbstract(); 错误的代码，不能创建抽象类的对象
        ma = new MySubClass();  //但是可以创建子类对象
    }
}
```

虽然抽象类不能创建对象，但是抽象类能够声明引用，并让这个引用指向子类对象。从某种意义上说，写抽象类的目的就是为了能够让子类继承。

更多的情况是把抽象类和抽象方法结合在一起使用，下面我们为大家介绍抽象方法。

3.2 抽象方法

用 abstract 修饰的方法称为抽象方法。抽象方法是 Java 中一个很重要的概念，在大家今后的编程实践中，会在很多地方用到抽象方法的概念。

我们在前面讲到“方法”这个概念的时候强调过，定义一个方法分两个部分：方法的声明，方法的实现。

抽象方法指的是：一个只有声明，没有实现的方法。对于抽象方法来说，方法的实现部分用一个分号来代替。例如：

```
class MyAbstract{
    public abstract void m1();
    public void m2(){}
}
```

其中的 m1 方法就是一个抽象方法。在这个方法的声明后面，没有代码块，只有一个分号，也就是说没有方法的实现。与之对应的是 m2 方法。m2 方法提供了一个空的方法实现，在实现之后没有分号。m2 方法不是一个抽象方法。

抽象方法是一种“只有声明，没有实现”的方法。方法的实现留给子类来完成。因此，如果一个类中有抽象方法，我们就可以认为这个类是一个“半成品”（因为这个类中的抽象方法缺少实现，实现的工作要由子类来继续完成）。一个“半成品”的类是不能用来创建对象的。

因此，如果一个类中有抽象方法，这个类就必须是抽象类。因此，上面的代码应该改写为：

```
abstract class MyAbstract{
    public abstract void m1();
    public void m2(){}
}
```

注意：有抽象方法的类必须是抽象类，但是反过来，抽象类中未必有抽象方法。

前面提到过，抽象方法是留给子类来实现的，因此，我们可以写一个子类来继承 MyAbstract 类：

```
class MySubClass extends MyAbstract{
}
```

这个类将无法编译通过，因为 MySubClass 类将从父类中继承到抽象的 m1 方法，也就相当于，在 MySubClass 类中出现了一个抽象方法，因此 MySubClass 类也必须是抽象类。

如果我们不希望 MySubClass 类也成为抽象类，就必须想方设法的“去除掉”这个类中的抽象方法。

我们可以用一个有实现的，完整的 m1 方法，覆盖掉父类继承下来的抽象的 m1 方法。如下代码：

```
class MySubClass extends MyAbstract{
    public void m1(){
        System.out.println("In SubClass");
    }
}
```

这样，MySubClass 类中的抽象方法就被“覆盖”掉了。在前面的章节中，我们给出的“覆盖”的定义为：子类用特殊的方法实现替换掉父类的一般的实现。而这里的情况略有不同：子类覆盖父类的抽象方法时，并不是用一个特殊的实现替换一个一般的实现，而是在父类没有方法实现的情况下，子类给出一个方法的实现。因此，像这样，用一个有方法体的方法覆盖一个没有方法体的方法，也可以称之为“实现”了该方法。例如，我们就可以说，MySubClass 类实现了父类中的 m1 方法。

因此我们得出结论：子类继承一个抽象类，如果我们不希望子类也成为抽象类，就必须让子类实现父类中定义的所有抽象方法。

我们再看主方法：

```
01: public  class  TestAbstract{
02:     public static void main(String args){
03:         MyAbstract ma = new MySubClass();
04:         ma.m1();
05:     }
06: }
```

对于第 3 行代码，MyAbstract 类是抽象类，可以用来声明引用，而子类 MySubClass 实现了所有的抽象方法，不是抽象类，因此可以用来创建对象。这句代码是正确的。

对于第 4 行代码，能否编译通过呢？我们来回忆一下多态的几条原则：

- 只能对一个引用调用引用类型中定义的方法。
- 运行时会根据对象类型调用子类覆盖之后的方法。

在 MyAbstract 类中，我们确实定义了 m1 方法（尽管没有实现），因此对 ma 引用，我们完全可以调用 m1 方法，这句代码是能够编译通过的。进而，在运行时，将调用子类（也就是 MySubClass 类）中实现之后的 m1 方法，因此，运行时屏幕上将打印出：

```
In SubClass
```

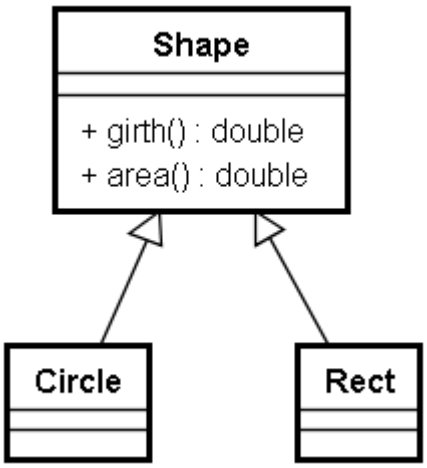
由此看出，抽象的语法和多态是紧密配合的。

总结一下：抽象方法的语法特征有如下几条：

1. 抽象方法只有声明，没有实现。实现的部分用分号表示。
2. 一个拥有抽象方法的类必须是抽象类。
3. 子类继承抽象类，要么也成为抽象类，要么就必须实现抽象类中的所有抽象方法。

3.3 抽象的作用

首先来看下面这个例子：写一个 Shape 类，表示一个形状，并为这个类提供两个方法：一个用来计算周长，一个用来计算面积。为这个类提供两个子类，一个是 Circle 类，表示一个圆形；一个是 Rect 类，表示一个矩形。下面是类的继承关系图：



其中，girth 方法表示求周长，area 方法表示求面积。
下面是代码的实现：

```
01: class  Shape{
02:     public double girth(){
03:         return 0;
04:     }
05:     public double area(){
06:         return 0;
07:     }
08: }
09: class  Circle  extends  Shape{
10:     private double r;
11:     private static final double PI = 3.1415926;
12:     public Circle(double r){
13:         this.r = r;
14:     }
15:     public double girth(){
16:         return 2*PI*r;
17:     }
18:     public double area(){
19:         return PI*r*r;
20:     }
21: }
22: class  Rect  extends  Shape{
23:     private double a;
24:     private double b;
25:     public Rect(double a, double b){
26:         this.a = a;
27:         this.b = b;
28:     }
29:     public double girth(){
```

```
30:         return 2*(a+b);
31:     }
32:     public double area(){
33:         return a*b;
34:     }
35: }
```

在这个继承关系中，Shape 中定义了 girth 和 area 方法，因为所有图形都能够计算周长和面积，这是所有图形的共性，应当放在父类。但是，对于不同的 Shape 中，对 girth 和 area 方法的实现都不相同，因此，这两个方法肯定需要被子类覆盖。然而，这两个方法虽然需要被覆盖，但是，在 Shape 类中，却不能不写“return 0”这个实现。因为如果不写，这两个方法就没有按照声明的要求，返回 double 值。这样会导致编译失败。

因此“return 0”这句代码就比较尴尬了，不写是不行的，写了又永远不会执行（因为根据多态，被调用的永远是子类覆盖之后的方法）。那么如何解决这个问题呢？

上面描述的 girth 与 area 方法，具有这样的特点：方法的声明是共性，方法的实现是特性。也就是说，所有形状都能够求周长和求面积，这是共性；而形状类不同的子类，求周长和求面积的方式不同，这是特性。遇到这种情况，我们就可以利用抽象方法来描述这种关系。

我们改写一下 Shape 类，结果为：

```
abstract class Shape{
    abstract public double girth();
    abstract public double area();
}
```

我们以前曾经分析过，方法的声明代表“对象能做什么”，而方法的实现代表“对象怎么做”，那么对于 Shape 类型来说，我们只能确定，任何一个形状对象都有“求周长”和“求面积”这两个功能，但是无法确定形状对象“怎么求周长”“怎么求面积”，具体周长和面积的算法要留给不同的子类，给出不同的实现。

因此，使用抽象方法，就可以让我们把方法的声明放在父类中，把方法的实现留在子类中。这样一方面，能够很好的体现出“共性放在父类”这个基本的原则，另一方面，用父类类型的引用又可以调用这些方法，并不影响多态语法的正常使用。

在 Shape 类中出现抽象方法的同时，这个类也成为了抽象类。这也是合理的，因为“形状”这个概念本身就是一个抽象的概念，在实际中，只会有矩形对象，圆形对象，菱形对象等等，不可能出现一个孤立的“形状”对象。因此，形状这个类就应该是一个抽象类。

我们换一个例子。在生活中，我们可以见到狗对象，猫对象，猴子对象等等，因此我们有了狗类，猫类，猴子类。从这些类中，我们归纳出了“动物类”。这些类都是动物类的子类。但是，谁又见过一个既非狗又非猫，什么具体动物都不是的“动物对象”呢？动物类是从那些具体的类中抽象出来的，而这个类本身又不会有任何对象，因此，动物类只能是个抽象类。

在动物类中，我们可以定义“吃”这个方法，因为我们确认，所有动物都会吃。但是我们又无法实现这个“吃”方法，因为不同的动物在“吃”方面有不同的行为方式，只能在不同的动物的子类中去分别实现“吃”方法。因此，“吃”方法在动物类中，就只能是个抽象方法。

由此可见，抽象的语法和其他的面向对象语法一样，都是从我们的实际生活中归纳总结出来的。