

Ch6 对象和类

本章导读

本章是 Java 学习中最核心的内容之一。这一章中，我们会第一次接触“对象”和“类”这两个概念，第一次接触“面向对象”的编程思想。这部分的知识，可以说是 Java 中最核心的知识之一，Java 庞大的知识体系和结构，都是建筑在“面向对象”的思想之上的。因此，本章中既有大量新的语法要掌握，也有很多新的概念要理解和消化。

1 面向对象基本概念

本小结将给读者阐述面向对象中的一些基本特点和概念。在这部分中，讲主要对面向对象的一些基本思想，做概念上的阐述和分析。

1.1 编程思想

首先我们来介绍编程思想的概念。什么是编程思想呢？所谓的编程思想，简单的说，就是程序员的思考方式。程序员在编程的时候，需要按照一定的思考方式，把需求编程具体的代码，这种思考方式，就是编程思想。

例如，我们曾经介绍过“面向过程”的编程思想。利用这种思想进行编程时，程序员思考问题的方式，是“第一步干 AAA；第二步干 BBB”这种方式。这种方式，也就是考虑实现某个需求的步骤和过程。首先能够把一个需求，细化成很多步骤，然后把一些比较复杂的步骤，又细化成更小的步骤。这样，就能够把一个比较大的需求，逐步细化和求精，最终形成具体的代码。这也就是我们在介绍函数这一章时介绍的编程思想：自顶向下，逐步求精。

但是，面向过程的思想，在进行一些复杂问题的开发时，就会显得力不从心。例如，如果我们要创建一个电子商务网站，让网站的用户能够在网上开店和进行交易。如果用面向过程的思想，则首先要分析这个网站有哪些过程。在这个电子商务网站上，有用户登录的过程，有用户开店的过程，有用户买东西的过程，有用户进行搜索的过程，还有一些例如上传图片、修改价格、修改描述……等等。这么一个复杂的网站中，有非常多的步骤和过程。如果还是使用面向过程的编程方式的话，首先会有太多的过程需要操作；其次，会有很多过程相互之间有关系。例如，用户修改商品价格这个过程，就要保证用户是否是这个商品的拥有者，因此还要有一个验证的过程；而如果之前用户没有进行登录，这在验证之前还要有一个用户登录的过程，等等……

我们可以看到，如果使用面向过程的方式解决问题，当问题变复杂时，会让过程变得庞大而复杂，因此面向过程的编程思想，并不适合用来解决一些比较复杂的问题。

由于面向过程的思想无法解决一些复杂的问题，为此，人们需要寻找新的编程思想，希望用一种新的思维方式来指导程序员编程。很快的，面向对象的编程思想被提出。由于这种思想能够很好的应对复杂的需求，解决一些面向过程很难解决甚至根本无法解决的问题，因此，这种编程思想逐渐成为了计算机行业中的主流。

下面，我们就开始介绍面向对象的思想。

1.2 对象的基本概念

面向对象的思想，其根源是来源于现实世界。因此，我们首先介绍现实生活中，对象的概念是什么。

在现实生活中，凡是客观存在的事物都能称之为对象。例如汽车、台灯、电脑、书本、网站，这些都是现实生活中的对象。

根据现实生活的经验，我们可以总结出，现实生活中所有对象都具有两个主要的要素：“对象有什么”以及“对象能干什么”。其中，对象“有什么”称之为对象的属性；而对象“能干什么”称之为对象的方法。例如，对于一个汽车对象，这个对象有颜色、品牌、价格、最高时速等属性，有启动、加速、刹车等方法。

此外，现实中的对象一定不是孤立存在的。对象和对象之间会通过某种方式产生联系。

一种方式是方法调用的关系。一个对象可以调用另一个对象的方法，从而在这两个对象中产生关联。例如，学生对象，可以调用老师对象的“讲课”方法，司机对象可以调用汽车对象的“行驶”方法，顾客对象可以调用厨师的“做饭”方法，等等。

另一种方式是组合的方式：把小对象组合成为大对象。例如，把主机、键盘、鼠标等等小对象，组合成电脑这个大对象；把若干个学生对象，组合成班级对象；把若干个书本对象，组合成图书馆对象等等。这种方式也可以理解为，大对象的属性还是对象。例如，电脑的属性包括主机、显示器、键盘、鼠标等，这些属性同样也是对象。

通过这两种联系方式，我们可以把一些功能相对简单的对象组合在一起，形成复杂的系统。例如，现实生活中，企业这个对象，往往是一个非常复杂的系统，一些大规模、跨行业的企业，更是有着非常复杂的组织架构。然而，企业中每一个对象，每一个员工，要解决的问题总是相对简单的。例如，在一个软件企业中，程序员要解决的问题就是编程，会计解决的问题就是算账，销售解决的问题就是获得订单，等等。但是，虽然每个员工要解决的问题都相对简单，但是当大量功能简单的对象组合在一起之后，就能形成一个非常复杂的企业系统。

有上面的分析可知，我们生活的客观世界，就是由对象组成的世界。我们身边任何事物都是对象，而且这些对象之间，还通过各种方式产生联系，从而形成了我们这个复杂而多姿多彩的现实世界。

而我们在生活中，遇到一些问题时，也会采用面向对象而不是面向过程的方式来解决。例如，假设我们要从北京出发到上海。为了完成这个任务，首先要做的，是应当确定，用什么方式去上海，是坐飞机？坐火车？还是自己开车？等等。

这是一个选择交通工具的过程，也就是我们在选择合适的对象来解决问题的过程。在这个例子中，我们希望能够选择一个对象，这个对象能有一个“交通运输”的方法。很显然，汽车、火车、飞机，都存在这个“交通运输”方法。我们可以根据实际的情况，来选择不同的交通工具，并调用其“交通运输”方法，从而完成从北京到上海这个目标。

我们可以看到，由于现实世界是一个面向对象的世界，因此，在现实生活中我们解决问题的思路，往往就是面向对象的思路：首先找到符合要求的对象，然后调用这个对象的方法，达到我们的目的。

1.3 计算机中的对象

说完了现实世界，接下来我们来看看计算机世界。

计算机行业的问题，与现实世界中的问题总是相通的。这是因为，计算机就是一种人类发明的工具，这种工具的发明，就是为了给人提供帮助，帮助人更好的解决现实中的问题。

例如，在没有计算机的时候，要想买东西，就必须要去商场。为了解决很多人购物的需求，我们就利用网络和计算机，设计出了电子商务网站。这样，就通过网络部分解决了人们购物的需求。

设计电子商务网站，这是一个计算机领域的问题。但是，我们可以看到，这个问题的提出，是与现实世界紧密联系的。也就是说，计算机编程要解决的问题，往往也都是从现实生活中来的。例如，软件行业中有 word 软件，是为了解决人们处理文档的问题；有 excel 软件，是为了解决人们统计数据以及做报表的需求；有 QQ 软件，是为了解决人们交流和沟通的需求……等等。

既然计算机世界的问题都来源于现实世界，那最好的办法，就应当是让计算机来模拟现实世界。由于现实世界是一个面向对象的世界，因此，很自然的，我们就希望在计算机世界中，也引入面向对象的思想，这样，计算机就可以来更好的模拟现实世界，从而利用现实生活中的经验，更好的解决计算机的问题。

例如，如果要做一个电子商务系统，则可以采用现实生活中的经验，创建管理员对象、顾客对象、卖主对象等。每一个对象都有各自不同的属性和方法，例如，管理员对象就好像是现实生活中的工商管理人員，他们具有批准开店、查封等方法；顾客对象具有浏览商品、下订单和付款的方法；而卖家对象，则有上货、修改价格等方法。这些方法的设计和实现，完全可以参考现实生活中的交易流程来实现。

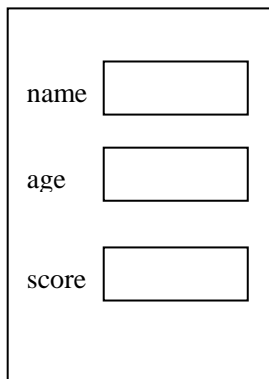
为了让计算机中也有面向对象的思想，首先要解决的问题，就是计算机中如何表示现实生活中的一个对象。

由于现实生活中的对象都具有属性和方法，因此，在计算机中，也应当让对象具有属性和方法。

然而，现实生活中对象的属性和方法非常丰富，在我们对现实世界的对象建模，创建对应的计算机中的对象时，需要对现实生活中对象的属性和方法有所取舍。例如，现实生活中，学生对象的属性非常丰富，学生有姓名、性别、身高、体重、视力、恋爱次数……等等属性，有学习、娱乐、唱歌、跳舞、吃饭、睡觉……等等方法。但是如果要设计一个学生信息管理系统，则需要对学生的属性和方法进行取舍，为学生对象保留姓名、性别、年龄等属性，保留学习、考试、练习等方法。这种取舍的过程称之为“抽象”。通过“抽象”，我们可以把现实生活中的对象用计算机模拟，可以把现实生活中活生生的对象用计算机中的一些数据表示。

通过抽象，我们就把现实中的对象，变成了计算机中的对象。那么，本质上，在计算机中的对象，是怎么来表示现实中的对象呢？

例如，我们对现实生活中的学生对象进行抽象，形成了计算机中的学生对象。抽象的时候，保留了学生对象的这样几个属性：姓名、年龄和学生成绩。因此，对于一个计算机中的学生对象而言，实际上就是在内存中的一块数据区域。这块区域中，有一个字符串，用来表示姓名，有一个整数用来表示年龄，有一个 double 类型的数，用来表示学生的成绩。在内存中的表示如下：



在内存中，我们分配出了一块数据区域，在这块区域中，包含了抽象出的学生对象的属性。这样，我们在计算机中，就模拟出了一个学生对象。因此，从本质上说，计算机中的对象，就是在内存中的一块数据区域。

需要注意的是，计算机中的对象，能够把多个相关的数据，放在同一个对象内进行操作。例如上面的例子，有了面向对象的思想之后，我们操作的是学生对象，而不是操作字符串、整数和 `double` 类型。我们通过对象的概念，把这些互相之间有关联的数据，放到一个区域中进行管理。

通过把现实生活中的对象抽象成计算机中的对象，我们就可以在计算机世界中模拟出跟现实世界中一样的面向对象的世界。这样，我们在进行编程的时候，就可以利用这种思想来解决问题。

例如，我们就可以使用面向对象的方法，来模拟从北京到上海的过程。在现实生活中，我们首先会寻找一个具有“交通运输”方法的对象。如果我们选择了“飞机”对象，那么就需要调用“机票代理人”对象的“卖机票”方法，来获得一个“机票”对象。再调用“出租车”对象的“开车”方法，到达机场。利用手中的机票，调用“飞机”对象的“登机”方法，“飞行”方法。从而到达目的地。示例代码如下：

```
//Ticket 表示机票
class Ticket{
}

//机票代理人对象
class Agent{
    //sellTicket 方法，表示机票代理人卖机票
    //返回值是一张机票
    public Ticket sellTicket(double money){
        ...
    }
}

//出租车对象
class Taxi{
    //drive 方法，表示出租车开往某地
    public void drive(String location){
        ...
    }
}

//飞机对象
```

```

class Plane{
    //登机方法，接受一个机票对象作为参数，返回值表示是否登机成功
    public boolean checkIn(Ticket t){
        ...
    }
    //表示飞往某地
    public void fly(String location){
        ...
    }
}

public class TestTravel{
    public static void main(String args[]){
        Agent a = new Agent(); //创建机票代理人对象
        Plane p = new Plane(); //创建飞机对象
        Taxi taxi = new Taxi(); //创建出租车对象

        //调用代理人的 sellTicket 方法，获得机票对象
        Ticket t = a.sellTicket(1000);
        taxi.drive("Airport"); //调用出租车的开车方法，开往机场
        if (p.checkIn()){ //调用飞机的登机方法，判断登机是否成功
            p.fly("Shang Hai"); //登机成功则飞往上海
        }

    }
}

```

可以看到，上面的代码和我们在现实生活中解决问题的思路是一样的：先获得相应的对象，再调用对象的方法。这与我们现实生活中先选择交通工具，再利用选定的交通工具出行，解决问题的思路是非常吻合的。

有了面向对象的思想之后，与面向过程不同，我们编程的方式发生了非常大的改变。当我们遇到一个复杂需求的时候，我们不应该把这个需求分解成一个一个的步骤，而是应该先设计在系统中有哪些对象。在编程的时候，我们应当先准备好系统中需要使用的对象，然后，通过组合对象以及让对象之间产生方法调用，从而把对象组合在一起。这样，我们通过多个对象的操作，让对象之间产生联系，从而完成复杂的系统和功能，最终完成需求。

1.4 面向对象的特点

面向对象的思想之所以能够流行，原因在于，相对于面向过程，他有以下一些特点：

一、各司其职

这个特点可以说是面向对象最大的优点。面向过程之所以无法应对大型系统，就在于当问题比较复杂的时候，过程会变得繁琐而容易出错。例如，在进行电子商务的设计中，一个简单的修改价格的功能，在面向过程的实现中，需要考虑的因素非常复杂。可能在设计这个函数的时候，这个程序员既要考虑用户是否有资格进行修改，也要考虑修改的数值是否在合理的区间范围之内，还要考虑如果修改失败应当怎么处理错误……等

等一系列的问题。

而如果使用面向对象的话，我们在设计系统的时候，就可以专门设计一个对象，用来完成资格验证的功能；一个对象专门用来进行数据方面的验证，另一个对象负责处理错误……这样，我们就通过多个对象的协作，共同完成了一个功能。这样，每一个对象所要处理的问题都相对简单，从而把问题的复杂性降低了。

从现实生活中来说，各司其职也更符合现实生活中的情况。例如，就像我们之前提到的例子，现实生活中，公司就是一个复杂的系统。而在一个大公司中，完成一个过程，可能需要很多的手续和步骤。例如，如果公司要进行采购的话，首先要让老板进行审核和预算的批准，然后是财务入账，让出纳拿钱；再然后采购到的物品要进行公司资产的登记……这个步骤是非常繁琐而复杂的。但是，在这个过程中，涉及到了多个对象，而每个对象完成的功能又是相对简单的。例如，老板就负责签字，财务就负责走账，出纳就负责拿钱，等等。这样，每个对象的工作和功能都相对简单，但是通过配合，能够完成非常复杂的功能。

在我们的代码中，面向对象的思想把复杂的系统分解成相对简单的对象，使得每个对象处理的事情相对简单和集中。这样，让不同的对象能够“各司其职”，只完成一些相对简单的功能，然后通过各个对象之间的配合，完成整个系统的功能。

可以说，“各司其职”的思想，是整个面向对象思想的核心。

二、可重用性

所谓的可重用性，指的是对于类似的功能，不同的系统可以重复使用相同的代码。这样，有些通用的功能，程序员写了一遍之后，在遇到类似的功能之后，不需要从头开始开发，只需要对这些通用的功能进行使用就可以了。

可重用性并不是面向对象特有的概念。在面向过程的编码中，同样有重用的思想。面向过程的重用是函数的重用，我们可以把通用的过程写成函数，然后在程序中多次调用这个函数，这样就能够实现重用。但是，这种重用比较困难。一方面，对于不同的系统来说，虽然都有类似的功能，但是很难完全不变的直接使用原有的过程。例如，虽然在不同的公司中，都有采购、审批这样的过程，但是，每个公司都有自己过程的特殊性，每个公司的审批流程都是不一样的。因此，在现实生活中，很少有相同的过程，对于编程来说，也很难写出一个函数，能够满足所有的要求。

相对而言，面向对象重用起来很容易。例如，虽然每个公司采购的审批流程不同，但是都会有一个出纳，完成最后的工作。因此，一个出纳，既可以在 A 公司工作，适应 A 公司的流程，也可以在 B 公司工作，适应 B 公司的流程。这说明，按照面向对象的方式，可重用性是比较高的。对于类似的过程而言，只要把对象的组合方式和联系方式进行一些调整即可，对象本身的功能是不需要改变的。

因此，面向对象相对于面向过程而言，有着更好的可重用性。

三、可扩展性

可扩展性指的是，在不修改原有系统的前提下，能够进行扩展。在面向过程的编程方式里，如果要增加系统的功能，则大部分的过程就都必须改变。例如，在我们的电子商务系统中，我们想要增加一个 VIP 会员的功能。这种会员，在开店和购物上面，会有更多的优惠以及便利的功能。如果我们按照面向过程的编程思路，增加了这种功能之后，就必须修改原有的开店和购物函数，在这两个函数中增加判断，如果是 VIP 会员的话，就执行一些新的代码。这样，增加了新的功能之后，势必会影响到原有功能的代码。造成的结果是，一旦系统的功能持续的增加的话，需要进行的修改和维护就变

得越来越复杂，最终导致的是增加一个新功能过于复杂，让系统无法维护，最终让系统无法增加新的功能，从而失去了可扩展性。

而如果是使用面向对象的编程方式呢？如果我们需要增加一个 VIP 会员功能，则只需要创建一个 VIP 会员对象，这个对象中包含了一个 VIP 会员应有的功能。这样，原有的普通会员的代码与 VIP 会员的代码互相独立，增加的 VIP 会员的代码不会影响到原有的功能。这样，新增加功能不会影响系统中原有的功能，这样，让系统增加功能就变得十分简单，从而让系统能够不断的进步和扩展。这就是面向对象的可扩展性。

四、弱耦合性

弱耦合性指的是，让对象和周围环境之间的联系尽可能的弱。那这样有什么优点呢？如果一个对象跟周围的联系比较弱，那么这个对象就可以很容易的被替换。

例如，台式机的显示器和整个系统之间，是通过与主机上的一个接口相连。而笔记本电脑的显示器和整个计算机系统之间，是通过内嵌在模具中联系的。这两种联系相比，台式机显示器的耦合较弱，而笔记本显示器的耦合较强。

弱耦合性能够提高可重用性和可扩展性。例如，如果想要换一个更大的显示器，扩展系统功能，对于台式机而言，相对简单；而对于笔记本电脑来说，这个需求几乎不可能完成。这就是由于台式机的显示器对象与系统之间是弱耦合的关系，而笔记本显示器与系统之间的耦合程度太强，因此不容易替换。

我们在写代码的时候，也会遇到这样的情况。我们希望我们的代码能够具有更好的弱耦合性，这样，如果有一个更好的模块的话，可以很方便的替换现有的模块，不需要对代码进行大量的修改。这就是弱耦合性为我们带来的好处。

1.5 类的概念

接下来要介绍的是“类”的概念。

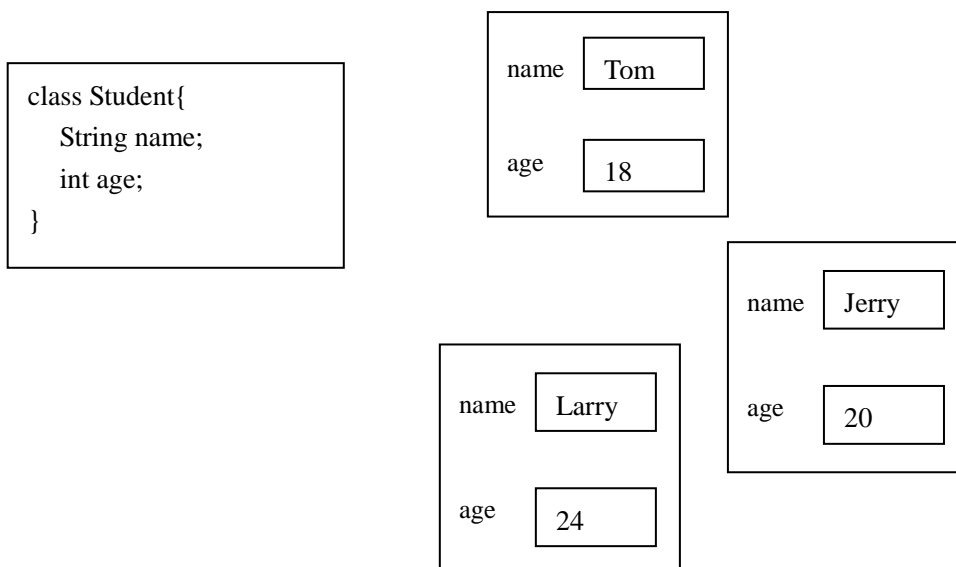
“对象”指的是客观存在的事物，因此每一个对象都意味着客观世界的一个事物。而“类”不同，它是对一“类”事物的总结，可以把类理解为：对大量对象共性的抽象。

例如，现实生活中，存在许许多多的狗“对象”（小白、大黄、旺财、来福……）。对于一个不认识“狗”的婴儿来说，当他见到越来越多的狗对象之后，慢慢的就能逐渐总结出，这一类对象的特点：四条腿，吃肉，摇尾巴……于是，慢慢的，他就认识了狗。也就是说，在他的脑子中，形成了“狗”这个概念，狗这个“类”就包含在他的脑子中了。

也就是说，当人面对大量的对象之后，慢慢的会把对象的共性进行抽象，从而形成了“类”。换句话说，类是对客观事物的总结和抽象。

与此同时，我们注意到，“类”是在人的脑子中形成的一个概念，也就是说，类是在人脑子里的，而不是一种客观事物。我们可以认为，类就是客观事物在人脑中的反映。

而在计算机世界中，类除了现实生活中的含义之外，还有另外一层含义：类能够作为创建对象的模板。也就是说，在计算机中，如果定义了一个类，则可以利用这个类创建多个对象。



1.6 小结

对象是客观事物。对象有两个主要元素：属性和方法，其中属性表示对象“有什么”，方法表示对象“能干什么”。

计算机中的对象本质上是一块数据，是对现实生活中对象的抽象。在编程时使用面向对象的思想，能够更好的模拟现实世界，更适合解决复杂的问题。

面向对象具有各司其职、可重用、可扩展、弱耦合等特性。

类是客观事物在人脑中的反映，是对对象共性的抽象，在计算机中，类是对象的模板。

2 类的定义

在上一节中我们主要讲述了面向对象的大量概念。在这一节中，我们将给大家讲述面向对象的一些基本语法。

2.1 编写一个类

我们讲过，类是对象的模板。如果我们希望创建一些对象，就必须把类写好。

定义类的语法很简单，相信我们也比较熟悉。

```
class 类名{
    类的内容
}
```

事实上，我们以前就写过很多类。简单回忆一下类的语法：定义类的关键字是“class”，后面跟一个代码块，代码块中定义了类的内容。

一个类在编译之后会生成一个名为“类名.class”的字节码文件。在一个Java源文件中，可以定义多个类，但是只能定义一个公开类。公开类的类名必须和文件名相同。

例如，我们来编写一个“学生”类，用来记录一个学生对象的属性和方法：

```
class Student{
}
```


2.2 定义类的属性

我们可以在类中定义类的属性。例如，学生类包含两个属性，姓名和年龄。则代码如下：

```
class Student{
    int age;
    String name;
}
```

我们可以看到，定义属性实际上就是定义变量。在 `Student` 类中，我们定义了两个变量“age”和“name”。这两个变量和我们以前讲过的“局部变量”的概念是不同的。局部变量指的是在方法内部定义的变量，而这两个变量却是定义在任何方法之外的，这种变量我们称之为：实例变量。下面我们通过具体的代码，来介绍实例变量的用法：

考虑下面的代码：

```
public class TestStudent{
    public static void main(String args[]){
        Student stu = new Student(); // 创建 Student 类的对象 stu
        System.out.println(stu.age); // 打印 stu 对象的 age 属性
    }
}
```

`stu` 对象的 `age` 属性在使用前没有被赋值，但是上述代码能够编译通过，输出结果为 0。

由此可见，实例变量和局部变量不同，局部变量必须先赋值后使用，而对于实例变量，系统会为其分配一个默认值。

实例变量的默认值规则与数组元素默认值一样，对于对象类型的属性，默认值为 `null` 值，对于数值类型的属性，默认值为 0，而对于 `boolean` 类型属性，默认值为 `false`。

我们再来看下面的代码：

```
01: class MyClass{
02:     public void print(){
03:         System.out.println(value);
04:     }
05:     int value = 10;
06: }
```

程序运行会打印出“10”。

尽管实例变量 `value` 的定义是在程序的第 5 行，但是我们依然可以在第 3 行来访问这个属性。因为 `value` 属性是定义在 `MyClass` 类中的，因此在 `MyClass` 类中任何位置，都可以访问这个属性。由此我们可以得出结论：实例变量的访问范围是在整个类的内部。

回忆一下局部变量，局部变量的访问范围是从定义的位置开始，到包含它的代码块结束。显然，实例变量的访问范围要大于局部变量。

我们再来考察一下局部变量和实例变量命名冲突的问题：

```
class MyClass{
    public void print(){
        int value = 20;
        System.out.println(value);
    }
    int value = 10;
```

```
}
```

上述代码能够编译通过。在 `print` 方法内部，我们定义了局部变量 `value`，同时，由于 `print` 方法在实例变量 `value` 作用范围之内，因此在 `print` 方法内部，局部变量和实例变量存在命名冲突。很显然，程序编译通过，说明这种命名冲突是允许的。

如果我们来调用 `print` 方法，会发现，程序输出的结果是“20”。也就是说，当实例变量和局部变量发生命名冲突时，以局部变量优先。

2.3 定义类的方法

从概念上说，一个类的方法指的是这个类能干什么。从语法上说，定义一个方法需要定义“方法声明”和“方法实现”。事实上，方法的概念和前面我们介绍的“函数”概念是非常类似的。

2.3.1 方法声明和方法实现

方法声明分为五个部分：

修饰符 返回值类型 方法名 参数表 抛出的异常

细心的读者可以发现，“返回值类型 方法名 参数表”实际上就是我们以前讲过的“函数三要素”。

方法的修饰符比较灵活，有的方法可能没有修饰符，有的方法可能有很多修饰符，如果一个方法具有多个修饰符，那么修饰符的次序是无关紧要的。例如：我们熟知的主方法的声明如下：

```
public static void main(String[] args)
```

这其中，“`public`”和“`static`”都是方法的修饰符，它们的次序并不重要，因此，下面的写法也是正确的：

```
static public void main(String[] args)
```

抛出的异常部分设计到“异常处理”的知识点，本书将在后续章节中对此做专门介绍。

在方法声明之后，紧跟一个代码块，这个代码块称之为方法的实现。

方法声明描述了调用方法时的一些信息，例如调用方法应该传递哪些参数，调用方法有哪些限制，调用方法会返回什么类型的返回值，等等。我们可以这样理解：方法的声明代表着“对象能做什么？”。

而方法的实现，则定义了对象对于一个方法的实现细节，这往往代表了“对象怎么做？”例如，`Student` 类的 `study` 方法声明，定义了 `Student` 对象能够“学习”，而 `study` 方法的实现则表明了 `Student` 对象是如何“学习”的。

我们为 `Student` 类添加一个方法，代码如下：

```
class Student{
    int age;
    String name;
    public void study(){
        System.out.println("Student study for 8 hours");
    }
}
```

2.3.2 方法的重载

在一个类中，我们可以定义一系列方法，这些方法的方法名相同，参数表不同，这种语法被称为“方法的重载”。例如，我们可以为 `Student` 类定义两个 `study` 方法：

```
class Student{
    public void study(){
        System.out.println("study()");
    }
    public void study(int n){
        System.out.println("study(int)");
    }
}
```

上面这段代码中，在 `Student` 类中定义了两个 `study` 方法，一个无参，另一个带一个字符串参数。程序运行时，根据不同的参数，会调用不同的方法。例如：

```
public class TestStudent{
    public static void main(String args[]){
        Student stu = new Student();
        stu.study(); //调用无参的 study 方法，打印 study()
        stu.study(10); //调用 int 参数的 study 方法，打印 study(int)
    }
}
```

需要注意的是，当程序被编译时，如果出现方法的重载，Java 编译器会根据实参的类型，来匹配一个合适的方法调用。因此，方法的重载又被称为“编译时多态”。请记住，这里格外强调“编译时”的概念，因为，哪一个重载的方法会被调用，这个问题在程序的编译时就已经决定了。

我们强调了，方法的重载的关键在于“方法名相同，参数表不同”。那究竟怎样的参数表算是“不同”呢？以下几种情况都可以认为是参数表不同：

- 参数个数不同。例如 `void study()`和 `void study(int n)`
- 参数类型不同。例如 `void study(int n)`和 `void study(double d)`
- 参数类型的排列不同。例如 `void study(int n, double d)`和 `void study(double d, int n)`

特别要提示的是，如果两个方法仅仅是形参名不同，这不算重载！即下面的两个方法不构成重载：

`void study(int a)`与 `void study(int b)`

此外，方法的重载对方法声明的其他部分没有要求。也就是说，对返回值类型、修饰符、抛出的异常这三个部分，方法重载并不要求它们相同或是不相同。

为什么要设计重载的语法呢？这是为了让方法的调用者，不用关心由于类型不同所造成的差异。举例而言，`System.out` 是一个对象，在这个对象中，有多个重载的 `println` 方法。例如，我们之前写过这样的代码：

```
int i = 10;
System.out.println(i);
System.out.println("Hello World");
System.out.println();
```

在上面的代码中，我们调用了三次 `println` 方法。注意，这三次调用 `println` 方法的时候，

我们为这三个方法传递了不同的参数：第一次参数为 `int` 类型，第二次参数为字符串类型，第三次参数为空。

想这样，对于程序员而言，要打印一某个东西，不需要考虑参数不同的类型，而只需要交给 `System.out.println` 方法打印即可。因此，不论是打印整数和打印字符串，我们调用 `System.out.println` 方法的方式是一样的，这就是重载为我们带来的好处。

在生活中，也有重载的例子。例如，人是一个对象，而人具有“吃”这个方法。但是，人具有多个不同的吃方法。例如，吃药时，含药品拿水往下灌；吃西餐时，用刀叉；吃中餐时，用筷子；吃肯德基麦当劳时，用手……等等。对于人来说，吃不同的东西，具有不同的方法。

但是，我们在日常生活中，并不需要对不同的吃方法加以区分。例如，请朋友吃饭时，只要把东西端上来，自然客人会调用自己合适的吃()方法。这样，请客的主人，就好像是方法的调用者。主人的主要工作，就是为客人的吃()方法，准备合适的参数。例如，主人端上一盘牛排，只要对客人说：“请吃”，调用客人的吃方法就可以了。而客人呢，根据主人拿上来的牛排，就用刀叉来吃；这就相当于根据不同的参数，来选择相应的方法。

因此，我们可以看到，重载的作用在于：对方法的调用者，屏蔽由于不同参数带来的方法实现上的差异。例如，在 `System.out.println` 这种情况下，程序员就是方法的调用者，实现上的差异不需要程序员关心。

2.4 构造方法

构造方法（也被翻译成构造器）是面向对象特有的概念，是一种比较特殊的方法。这个方法与创建对象有关，也是面向对象当中非常重要的一个方法。

我们先看一个代码的例子，为 `Student` 类添加一个构造方法：

```
class Student{
    public Student(){
        System.out.println("Student()");
    }
}
```

2.4.1 构造方法的基本概念

构造方法是一个比较特殊的方法。相比一般方法，构造方法有如下特点：

- 构造方法的没有返回值类型。

参考上面的 `Student` 构造方法。要注意，没有返回值类型是说，省略返回值类型这个部分。要注意没有返回值类型和返回 `void` 之间的区别。类似 `void Student()` 这种方法不是构造方法。

- 构造方法的方法名与类名相同。

例如，给定 `Student` 类，则其构造方法的名字必须是 `Student`。

要注意的是，构造方法对方法名有要求，对参数表没有要求。因此，一个类中可以利用方法重载，添加多个构造方法。例如：

```
class Student{
    public Student(){
```

```

        System.out.println("Student()");
    }
    public Student(int n){
        System.out.println("Student(int)");
    }
}

```

上面的代码为 **Student** 定义了两个构造方法，这两个构造方法方法名相同参数表不同，从而构成重载。

- 构造方法不能手动调用，只能在对象创建时自动调用一次。

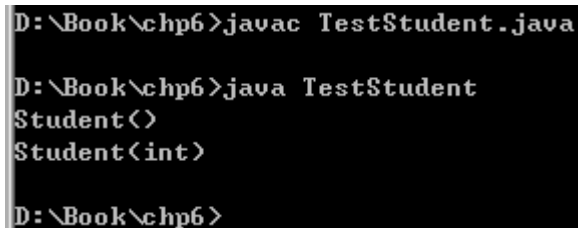
构造方法由 **JVM** 来自动调用。在使用 **new** 关键字创建对象时，会自动调用对象的构造方法。例如，有如下代码：

```

public class TestStudent{
    public static void main(String args[]){
        //创建一个 Student 对象，并在创建时自动调用无参构造方法
        Student stu1 = new Student();
        //创建一个 Student 对象，并在创建时调用有参构造方法
        Student stu2 = new Student(10);
    }
}

```

运行结果如下：



```

D:\Book\chp6>javac TestStudent.java
D:\Book\chp6>java TestStudent
Student()
Student(int)
D:\Book\chp6>

```

可以看出，上面的代码创建了两个对象，在创建这两个对象的时候会调用这两个对象的构造方法。要注意的是，要想给构造方法传递参数，参数应该写在“**new 类名()**”后的圆括号中。这样，编译器就会根据圆括号中的参数来决定，究竟应该调用哪一个构造方法。

2.4.2 默认构造方法

考虑下面的代码：

```

class MyClass{
    int a;
    int b = 100;
}

public class TestMyClass{
    public static void main(String args[]){
        MyClass mc = new MyClass();//编译正常
    }
}

```

```
    }  
}
```

在这段代码中，程序员没有写任何的构造方法。但是，在主方法中，由于创建对象时，使用了 `new MyClass()`；在圆括号中没有加入任何的参数，因此，这就意味着在创建对象时，需要调用 `MyClass` 类的无参构造方法。然后，虽然我们没有在 `MyClass` 中定义无参构造方法，但是上述代码也能够编译通过。这是为什么呢？

这是 `Java` 编译器在编译程序时，为我们做的事情。在 `java` 中，如果类中没有定义任何的构造方法，则编译器会自动生成一个公开的、无参的空构造方法。例如，我们的 `MyClass` 类没有定义任何构造方法，于是，在使用 `javac` 进行编译的时候，此时，编译器会自动生成一个构造方法：`public MyClass(){}` 。这个构造方法修饰符为 `public`，没有参数，方法的实现为空。这是编译器为我们生成的默认构造方法。

有了默认构造方法之后，在创建 `MyClass()` 对象时，就能够调用无参的构造方法，因此，下面的代码就能够编译通过。

```
class MyClass{  
    int a;  
    int b = 100;  
    //MyClass 中没有定义任何构造方法  
    //因此，编译器会自动加上一个构造方法：  
    //public MyClass(){}  
}  
  
public class TestMyClass{  
    public static void main(String args[]){  
        MyClass mc = new MyClass(); //MyClass 中存在无参构造方法  
    }  
}
```

需要注意的是，如果为 `MyClass` 类增加了任何一个构造方法，则编译器就不会自动生成默认构造方法。例如，把 `MyClass` 改成：

```
class MyClass{  
    int a;  
    int b = 100;  
    //增加一个有参构造方法，则编译器就不生成默认构造方法  
    public MyClass(int n){}  
}
```

则在创建 `MyClass` 对象时，不能调用无参构造方法。例如：

```
public class TestMyClass{  
    public static void main(String args[]){  
        MyClass mc = new MyClass(); //! 编译出错，找不到无参构造方法  
    }  
}
```

编译上述代码，出错信息如下：

```
D:\Book\chp6>javac TestMyClass.java
TestMyClass.java:10: cannot find symbol
symbol   : constructor MyClass()
location: class MyClass
        MyClass mc = new MyClass();
                        ^
1 error
D:\Book\chp6>
```

在这里，我们建议各位读者，作为一个良好的编程习惯，我们应当尽量为每一个类，都提供一个无参的构造方法。这样，无论编译器是否自动生成，我们都能够保证每一个类都有一个无参构造方法。这种保证能够给我们未来的 Java 编程带来很大的好处。

2.4.3 对象创建的过程

我们说，构造方法在对象创建的时候调用。那么，对象究竟是怎么创建的呢？分为几个步骤？在哪一个步骤调用构造方法呢？这一小节，我们就来为大家介绍对象创建的过程。

创建一个对象，在不涉及继承（这是下一章的内容）的情况下，基本步骤为 3 步：

1、分配空间

我们曾经介绍过，计算机中的对象，本质上是内存中的一块数据。因此，在计算机中创建一个对象，就意味着必须在内存中为一个对象分配一块数据区域。

此外，在分配空间的同时，会把对象所有的属性设为默认值。这也是为什么实例变量

2、初始化属性

如果直接对属性进行了赋值操作，则在这一步完成。

3、调用构造方法

当分配空间和初始化属性完成之后，最后一步是对该对象调用构造方法。

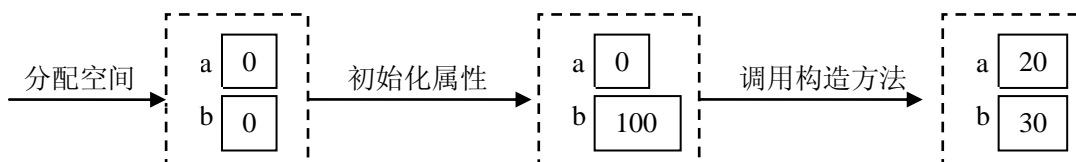
例如：有如下代码

```
class MyClass{
    int a;
    int b = 100;
    public MyClass(){
        a = 20;
        b = 30;
    }
}

public class TestMyClass{
    public static void main(String args[]){
        MyClass mc = new MyClass();
        System.out.println(mc.a);
        System.out.println(mc.b);
    }
}
```

```
}  
}
```

分析上述 `MyClass` 对象创建的过程。首先，为对象分配空间。分配的空间中，有一块区域用来保存实例变量 `a`，有一块区域用来保存实例变量 `b`。这两块区域在分配完之后，获得默认值 0。如下图所示：



分配空间之后，属性被赋值为默认值。之后，执行初始化属性的步骤。由于 `a` 属性没有被直接赋值，因此初始化属性时只需要为 `b` 赋值即可。之后，调用 `MyClass` 的无参构造方法。由于在构造方法内部为 `a` 属性和 `b` 属性赋值为 20 和 30，因此最后在程序中输出时，输出的是 20 和 30。完整的过程如上图所示。

在创建这个对象的过程中，`a` 属性被赋值了两次：一次是分配空间时的默认值，另一次是构造方法中的赋值。而 `b` 属性被赋值了三次：分配空间时获得了默认值 0；初始化属性时获得了初始值 100，调用构造方法时被赋值 30。

我们可以看到，构造方法，是创建对象的最后一个步骤。举例来说，如果说人是一个对象的话，那人的构造过程是什么呢？首先是十月怀胎，胎儿在母体内逐渐长大，并且各种器官逐渐形成，这就是在为新对象分配空间和初始化属性。然后，在分娩之后，医院的护士，会把胎儿和母亲之间的脐带剪断。这个“剪脐带”的动作，就可以认为是人的构造方法。首先，剪脐带这件事情，是生孩子的最后一个步骤。当剪断这根脐带之后，就意味着胎儿和母体相分离，从此，这个胎儿就成了一个独立的对象。其次，剪脐带这个动作，在人的一生中只会进行一次，当一个人出生之后，就再也不用进行第二次剪脐带的动作。第三，这个动作，是在孩子出生时，由父母、护士来完成的，当孩子长大之后，不会自己去调用自己的剪脐带方法。绝对不会有一个人，觉得自己肚脐眼的形状不好看，到医院要求重新剪一次的。

综上所述，对象的创建过程分为三步：分配空间、初始化属性、调用构造方法。其中，调用构造方法是对象创建中最后一个步骤。当构造方法完成之后，意味着对象的创建彻底完成了。

由于构造方法总是在最后一步调用，因此构造方法有一个常见的用法：用来为对象的属性赋值。这样，创建对象的时候就能够直接传递一些属性值，从而减少冗余的代码。典型的例子如下：

```
class Student{  
    String name;  
    int age;  
    public Student() { }  
  
    public Student(String stuName, int stuAge) {
```



```
        name = stuName;
        age = stuAge;
    }
}
```

我们定义的 `Student` 类中，有一个无参的构造方法。这个构造方法什么都不做，因此创建对象时，如果调用无参构造方法，则对象的属性都是默认值。同时，`Student` 类还有一个带参数的构造方法，在这个构造方法中，为对象的属性进行了赋值。

3 对象的创建和使用

3.1 对象的创建

在介绍构造方法时，我们介绍了对象创建的过程。在这一小节中，我们来讲一下创建对象的语法。在介绍之前，我们首先来定义一个 `Student` 类：

```
class Student{
    String name;
    int age;
    public Student() { }
    public Student(String stuName, int stuAge) {
        name = stuName;
        age = stuAge;
    }
    public void study(){
        System.out.println("Student study for 8 hours");
    }
    public void study(int n){
        System.out.println("Student study for" + n + "hours");
    }
}
```

下面我们来介绍如何创建对象。例如下面的两行代码：

```
Student stu;
stu = new Student();
```

这两行代码中，第一行代码创建了一个 `stu` 变量。但是要注意的是，创建这个变量的时候，并没有真正创建一个对象。也就是说，在内存中没有为某一个对象分配一块空间。那么 `stu` 变量用来干什么呢？这个变量叫做引用，可以用来操作对象。具体内容在下一小结会有详细的介绍。

第二行代码，才真正创建了一个对象。在创建对象时，必须要使用到一个关键字：`new`。有了 `new` 关键字之后，后面写上要创建的对象的类型。例如，我们这里要创建一个 `Student` 类型的对象，因此，在 `new` 关键字后面写上 `Student`。

然后，在类名后面，跟上一对圆括号。圆括号中可以写上参数，用来表示调用构造方法时，为方法传递的参数。例如，上面的例子中，在圆括号中没有给出任何参数，所以创建对象时会调用无参的构造方法。

我们也可以在圆括号中给出参数。例如下面的代码：

```
Student stu2 = new Student("Tom", 18);
```

这样，我们定义了一个 `stu2` 变量的同时，还创建了一个新的学生对象。这个学生对象的 `name` 属性和 `age` 属性，我们在构造方法中就给出，让这个学生的姓名为“Tom”，年龄为 18 岁。

3.2 引用

3.2.1 引用的概念

引用是 java 中很重要的概念。这个概念是后面学习面向对象各种特性的基础。

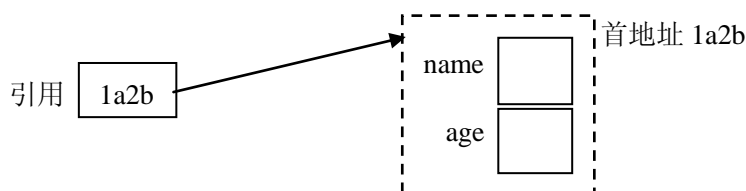
什么是引用呢？考虑下面的代码：

```
Student stu;  
stu = new Student();
```

第一行代码定义了一个对象类型的变量 `stu`，这个对象类型的变量就是一个引用。之前提过，定义了一个对象类型变量并没有创建一个真正的对象，事实上这只是声明了一个引用。

在计算机中，对象本质上是一块内存区域。当在 java 中创建一个对象时，本质上就是在内存中分配了一块数据区域。每一个内存中的数据区域，都会有它的内存首地址（参考数组一节中对内存首地址的描述）。因此，每一个对象，都会有一个首地址。

在引用类型中，保存的就是内存中的首地址。下面的示意图就是引用和对象在内存中的关系。



可以看到，引用就相当于一个指针，指向了内存中的对象。

我们要明确的是，引用本身和对象之间，还是有区别的。引用保存的只是对象的地址，与真正的对象是有本质的区别的。但是，通过引用，我们可以来操作一个对象。我们可以通过引用来使用对象的属性或者调用对象的方法。

拿现实生活举例，如果把家用的空调当做对象的话，则空调的引用就是空调的遥控器。首先，遥控器和空调本身有本质的差别，遥控器不等于空调。如果家里面没有安装空调的话，即使有了空调的遥控器也没用。

其次，要想操作空调，必须通过遥控器才能操作。如果一个家用的挂壁式空调没有了遥控器的话，那就无法对这个空调进行任何操作。

例如下面的代码：

```
public class TestStudent {  
    public static void main(String[] args) {  
        Student stu1 = new Student();  
        stu1.age = 18;  
        System.out.println(stu1.age);  
    }  
}
```

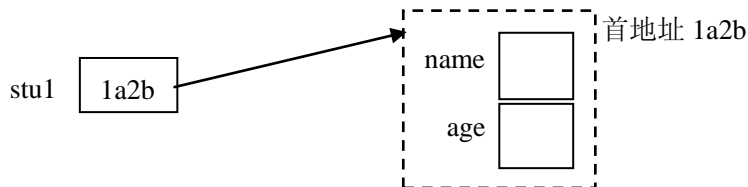
```

        stu1.study();
    }
}

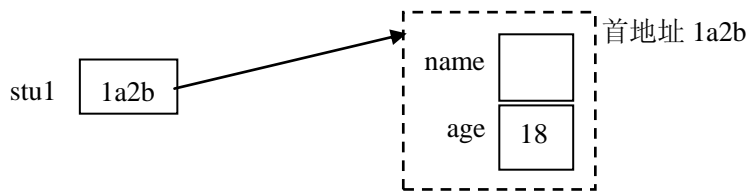
```

上面的代码演示了如何创建对象，以及如何利用引用来设置、获得对象的参数，以及调用对象的方法。

首先，在程序中有 `stu1 = new Student()` 这行代码。这行代码定义了一个 `stu1` 引用，并且创建了一个新的 `Student` 对象。在内存中的情况如下：



然后，`stu1.age = 18`。通过在引用后面加“.”，可以操作引用所指向对象的属性。这句代码的含义是，把 `stu1` 这个引用所指向对象的 `age` 属性，设为 18。执行完这句代码之后，内存中的情况如下：



然后，打印 `stu1.age`，这样获取 `age` 属性的值，输出结果为 18。

然后，调用 `stu1.study()`。这句代码的含义是：对 `stu1` 引用所指向的对象调用 `study()` 方法。程序运行结果如下：

```

D:\Book\chp6>javac TestStudent.java

D:\Book\chp6>java TestStudent
18
Student study for 8 hours

D:\Book\chp6>

```

3.2.2 多个引用指向同个对象

有了引用的基本概念之后，下面是对引用的进一步探讨。例如，有如下代码：

```

class MyValue{
    int value;
}

public class TestMyValue{
    public static void main(String args[]){
        MyValue mv1 = new MyValue();
        mv1.value = 100;
        MyValue mv2 = mv1;
    }
}

```

```

        mv2.value = 200;
        System.out.println(mv1.value);
    }
}

```

代码运行的结果如下：

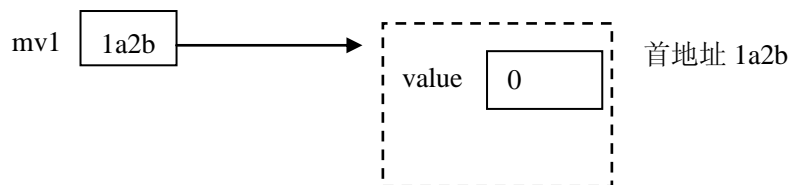
```

D:\Book\chp6>javac TestMyValue.java
D:\Book\chp6>java TestMyValue
200
D:\Book\chp6>

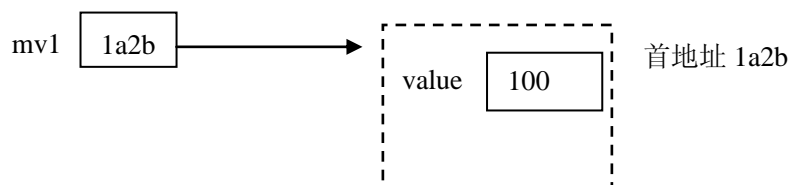
```

注意，输出 mv1 的 value 属性时，结果是 200。为什么修改了 mv2 的 value 属性之后，mv1 的 value 属性也跟着改变了呢？

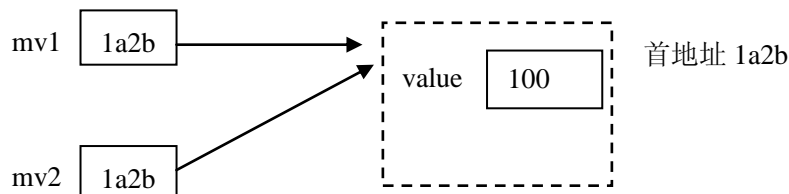
我们来分析一下代码。首先，我们创建了一个 MyValue 对象，并把这个对象的首地址赋值给 mv1 引用。内存中的图如下



注意到此时，对象的 value 属性是默认值 0。然后，执行 mv1.value = 100 这一行代码。在这行代码的意思是，把 mv1 引用所指向的对象的 value 属性设为 100，因此内存中的结果如下：

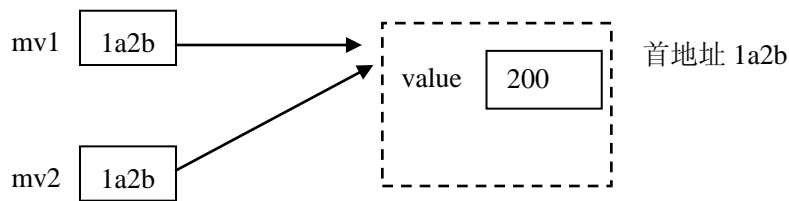


接下来，执行的是 MyValue mv2 = mv1 这一行。这一行把 mv1 的值赋值给 mv2。由于 mv1 是一个引用，而 mv1 的值，就是对象的首地址。因此。这行代码把引用 mv1 中保存的地址，赋值给另外一个引用 mv2。这样赋值之后，mv2 引用和 mv1 引用中保存的地址相同，换言之，这两个引用指向同一个对象。在内存中的情况如下：



这样，mv1.value 指的是：“mv1 引用所指向的对象的 value 属性”，而 mv2.value 指的是“mv2 引用所指向的对象的 value 属性”。由于 mv1 和 mv2 所指向的对象相同，因此，实际上 mv1.value 和 mv2.value 指的是内存中的同一块数据。

因此，`mv2.value` 属性进行了修改成 200，就是指的把 `mv2` 引用所指向的对象的 `value` 属性设为 200。此时，在内存中的情况如下：



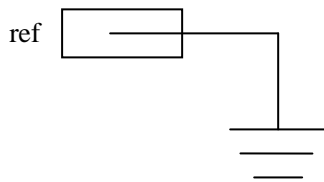
最后，打印 `mv1.value` 值时，打印的实际上是 `mv1` 引用所指向对象的 `value` 属性。由于这个属性在之前已经被修改成了 200，因此，此时打印的值就是 200。

上面的这个例子，给我们演示了两个引用指向同一个对象的例子。

3.2.3 对象类型的 null 值

我们在介绍默认值的时候，曾经介绍过，对于简单类型来说，默认值是各种各样的 0；而对于对象类型来说，默认值为“null”。而对象类型，也就是我们现在所说的引用。那么，引用是 null 值，表示的是什么呢？

所谓的 null 值，指的是：一个引用不指向任何对象。此时，这个引用中保存的地址为空，我们往往用下面的图形来表示：



我们往往用“接地”来表示引用的 null 值。

要注意的是，如果对一个值为 null 的引用调用方法或使用属性，则会产生一个错误。例如下面的代码：

```
MyValue mv1 = null;
System.out.println(mv1.value);
```

上面的代码，把 `mv1` 这个引用的值设为 null 值。而在打印语句中，要输出 `mv1.value`。这表示，要输出 `mv1` 这个引用所指向的对象的 `value` 属性。由于 `mv1` 引用没有指向任何对象，因此，通过 `mv1` 引用只能找到一个 null 而无法找到 `value` 属性。这样，就产生了一个错误。错误如下：

```
D:\Book\chp6>javac TestMyValue.java

D:\Book\chp6>java TestMyValue
Exception in thread "main" java.lang.NullPointerException
    at TestMyValue.main<TestMyValue.java:7>

D:\Book\chp6>
```

可以看到，在编译时，程序能够正常编译通过。但是运行时，由于使用了一个 `null` 引用的属性，所以产生了 `NullPointerException`，这个错误叫做空指针异常，这是 `java` 中非常常见的一个错误。产生这个错误的原因很简单，一定是对 `null` 引用调用了方法或者使用了属性。

2.5.2 方法参数传递

引用类型保存的是对象的地址，这个特点，在方法参数传递上表现的最为明显。本小节要介绍的就是 `Java` 中的方法参数传递规则。

看如下代码：

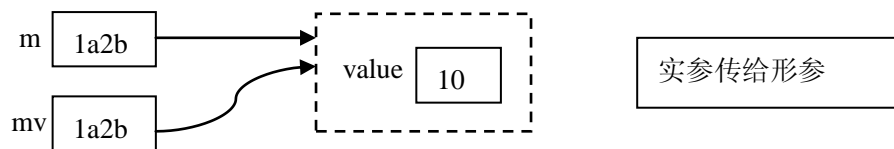
```
class MyValue{
    int value;
}
public class TestMyValue{
    public static void main(String args[]){
        int a = 10;
        changeInt(a);
        System.out.println(a);

        MyValue m = new MyValue();
        m.value = 10;
        changeObject(m);
        System.out.println(m.value);
    }
    public static void changeInt(int n){
        n++;
    }
    public static void changeObject(MyValue mv){
        mv.value++;
    }
}
```

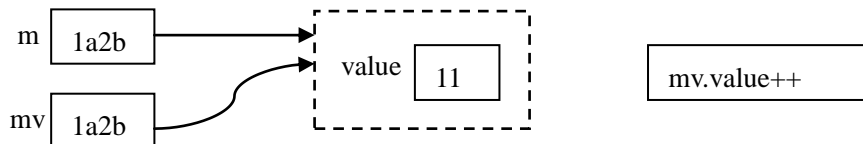
上面的程序展示了 `Java` 方法参数传递的问题。首先，在主方法中定义了一个 `int` 变量 `a`，并赋初值为 10，调用 `changeInt` 之后，输出 `a` 的结果，依然是 10。（这部分内容是上一章“函数”中的内容）。

之后，创建了一个 `MyValue` 对象，并把首地址赋值给 `m`。之后，为 `m.value` 赋值为 10，并调用 `changeObject` 函数。调用时，实参传给形参，传递的是对象的地址。函数调用过程如下。

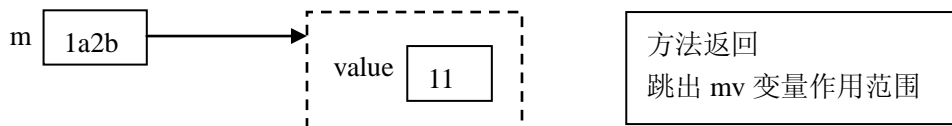
首先，在主方法中，调用了 `changeObject()` 方法，并且把 `m` 作为实参，传递给形参 `mv`。要注意的是，由于 `m` 是一个引用，保存的是一个对象的地址，因此进行传递时，传递给 `mv` 的值就是一个对象的地址。这样，实参 `m` 引用和形参 `mv` 引用中保存的内存地址相同，也就是说，这两个引用指向同一个对象。示意图如下：



然后，在 `changeObject` 方法内部，执行了 `mv.value++` 的操作。这个操作的含义是：对 `mv` 引用所指向的对象的 `value` 属性，进行+1 操作。修改之后，内存中的情况如下：



然后，`changeObject` 返回，这样 `mv` 这个形参就失去了作用范围，因此 `mv` 引用就不存在了。此时，内存中的情况如下：



此时，输出 `m.value` 的值，指的是 `m` 所指向对象的 `value` 值。这是的 `value` 值，就是修改之后的 11。

注意，对象类型当做方法的参数时，与基本类型不同。在实参传给形参时，基本类型传递的值，就是本身的数值；而对象类型传递的值，是实参的地址。因此，在使用上，这两者有很大的不同。因此，请记住这个结论：

在 Java 中的方法参数传递中，基本类型传值，对象类型传引用。

4 this

`this` 是一个很特殊的关键字。这个关键字在不同的用法下有不同的含义。

4.1 this 表示引用

首先我们来看下面这段代码：

```
class MyClass{
    int value = 10;
    public void print(){
        int value = 20;
        System.out.println(value);
    }
}

public class TestMyClass{
    public static void main(String args[]){
        MyClass mc = new MyClass();
        mc.print();
    }
}
```

```
}
```

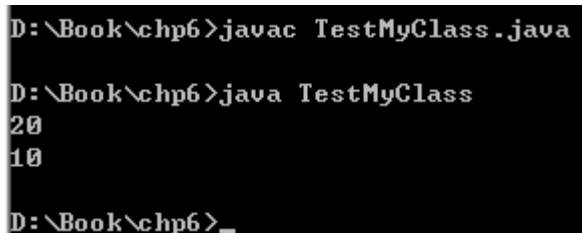
在这段代码中，调用 `print` 方法时，打印的是 20。原因也很简单。我们在 `MyClass` 类中定义了一个实例变量 `value`，这个变量的值 10。但是，我们在 `print` 方法中，同样定义了一个局部变量 `value`，这个 `value` 的值是 20。根据实例变量的特点，当实例变量和局部变量发生命名冲突时，以局部变量优先。因此，此时输出的 `value` 值，就是实例变量的值 20。

那有没有办法，在局部变量和实例变量冲突的时候，明确的指明实例变量呢？这个时候，我们就可以用 `this` 关键字。修改后的代码如下：

```
class MyClass{
    int value = 10;
    public void print(){
        int value = 20;
        System.out.println(value);
        System.out.println(this.value);
    }
}

public class TestMyClass{
    public static void main(String args[]){
        MyClass mc = new MyClass();
        mc.print();
    }
}
```

我们在程序中加入了一行新代码，输出 “`this.value`”。这样，通过 “`this.`”，来明确的指明实例变量。运行结果如下：



```
D:\Book\chp6>javac TestMyClass.java
D:\Book\chp6>java TestMyClass
20
10
D:\Book\chp6>
```

这样，我们就输出了实例变量的值：10。

那么 “`this.`” 究竟是什么含义呢？从概念上说，在这种使用 `this` 的用法中，`this` 关键字表示的是 “当前对象的引用”。那么什么是 “当前” 对象呢？所谓当前对象，指的是：对哪个对象调用的方法，哪个对象就是当前对象。例如下面的例子：

```
class MyClass{
    int value;
    public MyClass(){}
    public void m(){
        System.out.println(this.value);
    }
}

public class TestMyClass{
    public static void main(String args[]){
        MyClass mc1 = new MyClass();
        mc1.value = 100;
```

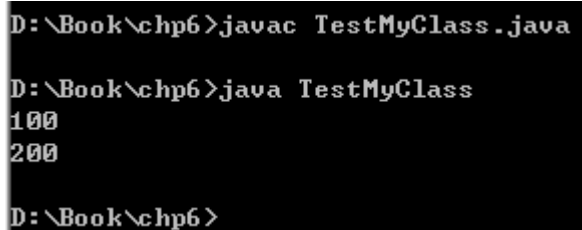


```

        MyClass mc2 = new MyClass();
        mc2.value = 200;
        mc1.m();
        mc2.m();
    }
}

```

对 m1 和 m2 对象分别调用 m 方法。对 m1 调用 m 方法时，m 方法中的 this 指的就是 m1 对象，因此输出的值，就是 m1 对象的 value 值 100。而对 m2 调用 m 方法时，this 指的就是 m2 对象，此时输出的值就是 200。运行结果如下：



```

D:\Book\chp6>javac TestMyClass.java

D:\Book\chp6>java TestMyClass
100
200

D:\Book\chp6>

```

当然，在上面的代码中，由于 value 没有命名冲突，所以如果不加 this，直接输出 value 值，同样输出的是当前对象的实例变量。下面的代码运行结果相同：

```

class MyClass{
    int value;
    public MyClass(){}
    public void m(){
        System.out.println(value);
    }
}

```

因此，当局部变量和实例变量没有命名冲突时，表示实例变量时，可以不用 this。

另外，“this.”这种用法，在构造方法中也非常常见。例如下面的代码：

```

class MyClass{
    int value;
    public MyClass(){}
    public MyClass(int value){
        this.value = value;
    }
}

```

在第二个有参构造方法中，参数名为 value。由于参数是特殊的局部变量，这个参数就跟实例变量 value 产生了命名冲突。于是，在构造方法中，我们就是用了 this 关键字，这句话：

```
this.value = value;
```

“=” 左边的 this.value 表示的是实例变量，而 “=” 右边的 value 指的是局部变量，也就是构造方法的参数。这句话的含义是，把构造方法的参数赋值给实例变量。

这种赋值的方法，在构造方法中非常常见。我们常常可看到这样的代码：

```

class Student{
    String name;
    int age;
}

```

```

    public Student() { }
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

```

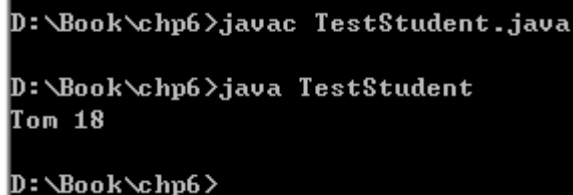
在这个 **Student** 类中，我们定义了带两个参数的构造方法。在这个构造方法内部，把构造方法的两个参数，赋值给了对象的属性。于是在创建对象时，就可以写出这样的代码：

```

public class TestStudent {
    public static void main(String[] args) {
        Student stu = new Student("Tom", 18);
        System.out.println(stu.name + " " + stu.age);
    }
}

```

这段代码在创建学生对象时，调用了其有参构造方法，并把学生对象的 **name** 属性设为 Tom，age 属性设为 18。程序运行结果如下：



```

D:\Book\chp6>javac TestStudent.java

D:\Book\chp6>java TestStudent
Tom 18

D:\Book\chp6>

```

4.2 this 用在构造方法中

除了把 **this** 当做当前对象的引用之外，**this** 还有一种其他的用法。

假设一个对象的构造比较复杂，例如：

```

class MyClass{
    int value;
    public MyClass(){
        //非常复杂的构造过程...
    }
    public MyClass(int value){
        //复杂的构造过程...
        ...
        this.value = value;
    }
}

```

假设有参数的构造方法 **MyClass(int value)** 相对 **MyClass()** 构造方法，只是多出一个语句：**this.value = value**。除此之外的构造过程完全相同。既然这部分代码完全相同，那么最合理的想法，应当是重用 **MyClass()** 这个无参构造函数。为此，我们希望能在 **MyClass(int value)** 这个有参构造方法中，调用无参的构造方法。

如果在一个类的构造方法中，要调用本类的其他构造方法，就必须使用 **this** 关键字。语法是 **this** 加上一个圆括号，圆括号中加上调用构造方法时传递的参数。例如，在这个例子中，

我们要调用 **MyClass** 类的无参构造方法，于是，就应当写成：

```
this();
```

所以，上面的 **MyClass(int value)**，就可以写成：

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

这样，就实现了我们所说的，在 **MyClass(int value)**这个构造方法中，调用 **MyClass()**这个无参构造方法。

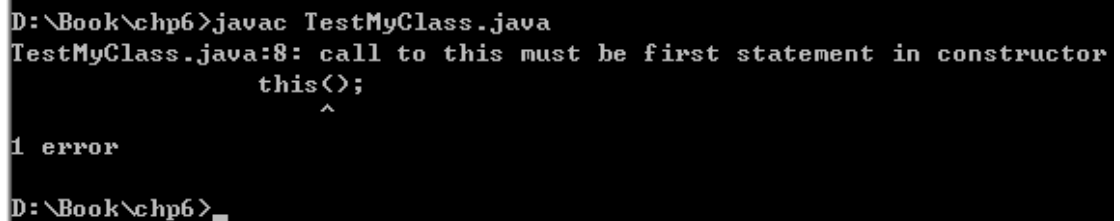
在使用 **this()**这种语法时，要注意三个要点：

1、只能在构造方法中使用，并且只能调用本类的其他构造方法

例如，下面的代码会编译出错：

```
class MyClass{  
    int value;  
    public MyClass(){  
        System.out.println("MyClass()");  
    }  
  
    public void m(){  
        this();  
        System.out.println("m()");  
    }  
}
```

在这个程序中，我们在一个普通的 **m()**方法里面，使用 **this()**调用无参构造方法。这回产生一个编译时错误，出错信息如下：

A screenshot of a command prompt window showing a Java compilation error. The text is as follows:
D:\Book\chp6>javac TestMyClass.java
TestMyClass.java:8: call to this must be first statement in constructor
 this();
 ^
1 error
D:\Book\chp6>

2、在使用时，**this()**必须作为构造方法的第一个语句，否则编译出错。

例如下面的代码：

```
class MyClass{  
    int value;  
    public MyClass(){  
        System.out.println("MyClass()");  
    }  
  
    public MyClass(int value){  
        System.out.println("MyClass(int)");  
        this();  
    }  
}
```

```
}  
}
```

在这段代码中，在 `MyClass(int value)` 这个有参构造方法中，利用 `this()` 调用了无参构造方法。但是，由于 `this()` 不是构造方法的第一个语句，因此编译出错，出错信息如下：

```
D:\Book\chp6>javac TestMyClass.java  
TestMyClass.java:9: call to this must be first statement in constructor  
    this();  
    ^  
1 error  
D:\Book\chp6>_
```

需要强调的是，对 `this()` 的调用必须是构造方法中的第一个语句，这个限制，指的是对 “`this + ()`” 的调用。而 “`this + .`” 的语法，不受这个限制。例如，下面的代码能够顺利的编译通过。

```
class MyClass{  
    int value;  
    public MyClass(){  
        System.out.println("MyClass()");  
    }  
  
    public MyClass(int value){  
        this(); //对 this() 的调用必须是构造方法的第一个语句  
        this.value = value; //对 this. 的调用没有限制  
    }  
}
```

3、`this()` 不能够递归调用。

在介绍函数时，我们曾经给大家介绍过递归的概念：如果在一个函数的内部，调用这个函数本身，则这种情况称之为递归。而 `this()` 不允许递归调用，也就是说，构造方法不能调用自身。例如下面的代码：

```
class MyClass{  
    int value;  
    public MyClass(){  
        this();  
        System.out.println("MyClass()");  
    }  
}
```

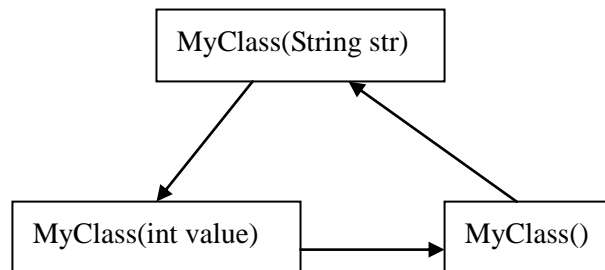
这个类，在 `MyClass()` 这个构造方法中，利用 `this()` 调用了本身，从而形成了递归调用。这样会产生一个编译时错误：

```
D:\Book\chp6>javac TestMyClass.java  
TestMyClass.java:3: recursive constructor invocation  
    public MyClass(){  
        ^  
1 error  
D:\Book\chp6>_
```

这种构造方法内部，直接调用自身，被称为直接递归。除了这种方式之外，还有间接递归。例如：

```
class MyClass{
    public MyClass(){
        this("hello");
    }
    public MyClass(int value){
        this();
    }
    public MyClass(String str){
        this(10);
    }
}
```

在这个程序中，MyClass(String str)调用了 MyClass(int value)，而 MyClass(int value)调用了 MyClass()，最后 MyClass()又调用了 MyClass(String str)。示意图如下：



这样，三个方法之间的调用形成一个循环，这样也构成递归调用。因此，这也会产生一个编译错误。错误信息与直接递归的错误信息相同，在此不再重复。