第 3 章　面向对象

计算机是一个小世界，而大自然物理世界则是一个大世界，二者有相通之处。本章首先阐述了物理世界与面向对象的思想、方法，然后过渡到计算机面向对象的思想、方法；接下来描述了如何应用 Java 来表达面向对象进行程序设计的思想、方法，具体有：类与对象、对象的三个基本特征、对象之间的关系，最后对面向对象进行程序设计的基本原则作了总结。这是本章的主线，其间穿插讲述了引用类型的类型转换、修饰符、内部类等内容。学完本章读者应能够较深刻地理解并掌握什么是对象、为什么要采用面向对象编程、怎样运用面向对象编程等一系列根本性的问题和相关方法、技术。

3.1　物理世界与面向对象

3.1.1　人与计算机的关系

很多人无形中有一种观念："学习计算机编程就是学习计算机或者向计算机学习"，实际上这种观念是有问题的，正确的应该是"计算机向人们所生活的物理世界学习"这是为什么呢？这涉及人与计算机的关系问题。人们发明计算机是为了让它能够为人类服务，因此人和计算机之间是人处于主体地位，计算机只能处于从属地位。人类是生活在物理世界中的，人们在这个物理世界中要做许许多多的事情，计算机要做到能够较好地为人类服务，它就必须能够模拟这个物理世界，例如计算机售票系统、电子银行、各种办公系统、教学系统，无一不是通过模拟与人相关的物理世界而为人类服务的。再例如，计算机中的 C/S（Client/Server，客户/服务）、P2P（Point to Point，点到点）等计算模式，队列、栈、树等数据结构，TCP/UDP 通信协议等，都源于物理世界。计算机是个小世界，而物理世界是个大世界，小世界来源于大世界，因此要学习计算机编程就需要对物理世界有所了解，学习面向对象编程更是如此。

3.1.2　物理世界的认知

1．事物与对象

怎样来描述物理世界呢？不同的人有不同的描述。我们在这里借用一下哲学上的有关描述。

（1）世界由万物构成。

（2）事物是运动的、发展的、变化的。

（3）事物之间的关系是复杂的、多变的。

第一句话说的是事物是物理世界的一种组成单位，所以事物很重要，把事物纳入到计算机程序设计中就是"对象"，面向对象的意思就是面向事物。许多初学者学习面向对象时总是搞不清楚"对象"是什么，因为"对象"一词很抽象，不是很直观，有些教材就举例子，说汽车是对象，桌子是对象……。这没有错，但有问题，这些例子都是具体的，不能反映出"对象"一词抽象的内涵和广泛的外延，从哲学的层面来认识对象才是合理的。有一本杂志，叫《物件

导向杂志》，就是探讨面向对象的，他们称作“物件”，这样就好理解了。今后我们不再区分事物、对象、物件这些名称，他们指的都是同一个东西。

任何事物都具有两个方面：一个是静的方面，称为属性；另一个是动的方面，称为功能或行为。例如，冰箱的属性有体积、容积、颜色等，功能有制冷、保鲜、解冻等。属性和功能是事物的两个方面，不可分隔。在面向对象程序设计中，使用类成员变量来描述事物的属性，使用方法（也称为函数）来描述事物的功能。

2. 事物的基本特征

事物有三个基本特征：封装、继承和多态。所谓基本特征，就是事物最基本、最根本的特征，也可以认为是任何事物都具有的，而且是最重要的特征，这些特征彼此是独立的。这三个既然是基本特征，那就非常重要了，下面就谈谈这三个基本特征。

（1）封装。

什么是封装？这是首先需要弄懂的问题。很多教材认为，封装就是把事物的属性和方法放在一起，合并到一个类中，这样的说法不太恰当。既然属性和功能是事物的两个不可分隔的方面，如果用类来描述事物，类中自然就应该包含属性和方法，这是其一；其二，“世界由万物构成”的前提是事物能够生存、存在。任何事物要生存至少需要两个东西：一个是保护措施，另一个是相对独立性。例如，超市里卖的食品都要包装，有些还是真空包装。包装具有保护作用，若没有包装，食品很容易坏掉，会变质或被蚊虫侵蚀。同时包装也使食品与外界有了相对独立性。再例如人，人的封装是很复杂的，衣服对人具有保护作用，不穿衣服就很容易生病。人要学习各种东西来包装自己（礼仪、经验、交际、科学文化等）。学的东西越多，懂得越多，受到的制约、伤害就越小，能力就越强，对别人的依赖程度就少，相对独立性就强，生存能力就强。谈的这些都是封装的作用：一个是保护作用，另一个是增强相对独立性。

封装是任何事物都具备的，因为封装的一个基本作用就是保护对象本身。反过来说，若存在的某个事物没有封装性，也就是说该事物没有自我保护措施，那么经过大自然的反复考验，这个事物最终一定就被破坏并且不再存在了。就像现在草原变成沙漠，青山变成秃岭一样，表面的植被被破坏了，唇亡则齿寒。谈到这里，读者朋友们应该明白封装有很多表现形式，不仅仅只是包在事物表面上的那层壳，比如前面谈到人的封装，方法就有很多。谈论封装，关键是看它的两个好处：保护作用和增强自身的独立性。

把封装纳入到面向对象程序设计中，是对封装这一基本特征的应用，当然是要应用封装的这两个好处的。软件的安全性需要保护机制，软件的模块需要相对独立性，用计算机术语说就是模块内需要强内聚、模块之间需要弱耦合，这是软件工程提出的用于衡量软件质量的重要指标。这样回过头来再看“封装就是把属性和方法合到一块”那样的观点，是否就有些不妥了呢？属性表达的是事物静态的一面，方法表达的是事物动态的一面，动与静本来就是一个事物具有的两个不可分隔的方面，本来就在一块。

那么这种观点是否就没有一点道理呢？不是的，它是从为了区分面向对象程序设计与早期的结构化程序设计的不同这个角度而言的。到这里我们已经理解了什么是封装，仅从封装这个角度就知道面向对象程序设计是一种好的程序设计方法，因为封装具有的两个好处。但要想真正弄清楚面向对象程序设计为什么会受到那么大的青睐，受到那么多人的追捧这个问题，还需要了解一下结构化程序设计以及之前人们是怎样开发软件的。我们借这个机会，在这里谈谈这两个问题。

最早的程序设计是没有什么方法可言的，人们编写程序基本上是想到什么就写什么，尤其是流程语句的跳转，可以使用 goto 关键字随意跳转，在一个程序中使用较多的 goto 跳来跳去，最后导致程序流程混乱，外国话形容其乱如一碗面条，中国话就是一团乱麻，所以 Java 禁止使用 goto 关键字。在 20 世纪 60 年代末 70 年代初出现了一些大型的软件系统，如操作系统、数据库系统等，大型系统往往需要花费大量的人力和财力，然而开发出来的软件却常常存在诸多问题：可靠性差、错误多，系统维护也相当困难。当时人们称这种现象为“软件危机”。这种程序曾导致了 20 世纪 60 年代美国登月计划的失败和 IBM 公司投资上千万美元的软件开发计划的失败等严重事件。因此寻找一种科学的程序设计方法就十分必要了。

E.W.Dijikstra 在 1965 年提出了结构化程序设计方法，其主要思想是采用三种基本控制结构（顺序、选择分支和循环）构造程序，自顶向下、逐步求精。这种程序设计方法有人简单概括为：数据结构+算法。从而结构化程序设计方法才有了数据流图和控制流图等手段。数据是对象静的一面，而算法是对象动的一面，即把一个事物本身具有的两个方面给分隔开来了，这是结构化程序设计方法的缺陷，从而导致后来一系列的问题。然而结构化程序设计方法的推出，对软件开发起了巨大的推动作用。其亮点就在于提出了三种基本控制结构，这三种结构被保留下来，Java 中自然也有（详见 2.3 节），而且从理论上证明了任何流程无论有多么复杂，都可以只用这三种基本控制结构等价处理，这个结果是很了不起的。在结构化程序设计中没有对象的概念，只有数据和流程方法（算法）。而对象是宇宙的组成单位，宇宙是一个大系统，对象本身是一个小系统。后来的面向对象方法正是认识到了这一点，才具有如此魅力，正如老子的“道法自然”，面向对象程序设计方法其实就是从大自然中来的。关于封装，Java 是如何表达的，详见 3.3 节。

（2）继承。

继承是事物的另一个基本特征。事物是不断发展的，继承在事物的发展过程中起了重要作用，作用有两个：一个是大大提高了事物发展的速度，另一个是大大提高了事物发展的质量。假如没有继承，人们不可能会在短短的几十年中学完人类几千年积累的知识文化。假如没有继承，科学技术发展的速度不会越来越快、水平不会越来越高。万事万物皆如此。把继承应用到程序设计中，这两个好处自然也有了。运用继承，软件开发的速度大大加快了，软件的质量也大大提高了。关于继承，Java 是如何表达的，详见 3.4 节。

（3）多态。

事物的第三个基本特征是多态。多态就是多种形态、多种表现形式。事物是发展的，发展离不开继承，继承的原则是取其精华，去其糟粕，并加以进化创新，这就导致了事物的多样性。例如汽车，当今世界上的汽车可谓是种类繁多、形式多样。但这些汽车都是在第一辆汽车的基础上经过不断的继承发展而来的。世界由万物构成，要正确认识、把握世界就要从事物入手，但事物又是复杂的、多样的，要把世界上所有的具体事物（个体，具体的表现形态）一一研究完，是做不到的。所以人们才提出了分类研究，每一类事物中找出典型代表进行研究，总结出规律，当再遇到这一类的其他事物时，即使没有研究过，也能够把握它。用哲学的话来说就是：本质是抽象的、不变的、简单的，现象是具体的、可变的、多样的、复杂的，要透过现象看本质，以不变应万变。注意这是一个双向的过程，首先是通过一类事物的各种具体表象形态总结规律，把本质抽出来，然后再运用本质来把握具体的个体事物，做到以本质的不变应对表象的万变。注意这里有几个术语：个体事物、形态、表现形式、表象、现象都指的是一回事，

不再区分。

在这里举个例子来说明这个问题：一月份的时候，某城市根据规划新修了一条路，路的下面设置了自来水管道系统，二月份的时候，根据要求，需要在路的下面增添煤气管道，工程很快展开了，挖开路面，埋入煤气管道，然后再修补路面，三月份的时候，新的需求又来了，需要在路的下面增设 Internet 光纤，于是又重新挖开路面，植入 Internet 光纤，然后又修补路面。接下来又需要新增供热管道……，麻烦的事接踵而来，搞得工程师对道路挖了又挖、改了又改、补了又补，道路最终变得不成样子。如果设计师在设计道路的时候，考虑到需要植入不同具体的管道（现象），然后抽象出一个管道方案（本质），这样设计出的道路就能够适用于各种具体管道的需要了，而不必挖了又挖。

同样，软件是经常变化的，不管是在开发阶段还是在后期维护阶段。假如为某家企业开发软件，企业的业务、工作流程等是经常变化的，这导致了软件的需求会经常发生变化，如果在研发软件时没有考虑到软件对这种变化的适应，也就是没有把这些经常变化的业务抽象出来，形成一个抽象的业务，那么开发出的软件就必然会随着业务的变化而不断地修改，改来改去，最终不成样子，很难满足企业的要求。

把多态纳入到程序设计中，就能够大大增强软件的适应能力，用计算机术语说就是增强软件系统的可伸缩性、弹性。多态并不仅仅谈的是多种表现形态，更重要的是如何通过对多种表现形态的归纳，抽象出其本质，然后再把本质应用到表现形态中。关于多态，Java 语言是如何表达的，详见 3.5 节。

3. 事物间的关系

上面我们了解了什么是对象以及对象的三个基本特征，也就是哲学上的前两个描述：世界由万物构成；事物是运动的、发展的、变化的。下面我们再谈谈事物之间的关系，即哲学上的第三个描述。

不同的事物之间的关系很复杂，粗略地说大致有 5 种：泛化关系（Generalization）、聚合关系（Aggregation）、组合关系（Composition）、依赖关系（Dependency）、关联（Association）。

泛化关系是一种纵向关系，可以表达为“is-a”，主要体现在抽象与具体、本质与现象等范畴，在面向对象编程中涉及到继承和多态。例如：汽车与轿车的关系、鸟与喜鹊的关系等。

聚合关系和组合关系是一种横向关系，可以表达为“has-a”，描述的是整体与部分的关系。在聚合关系中，组成部分和整体（属主）具有各自的生命周期，组成部分可以离开属主而单独存在，而在组合关系中，组成部分和属主具有相同的生命周期，二者不可分隔。例如，学校和学生之间是一种聚合关系，学生可以离开学校而单独存在，例如放假的时候。再例如桌子和桌面是一种组合关系，二者不可分隔，分开了就没有意义了。试想如果桌子没有桌面了还能叫桌子吗？在物理世界中有些对象之间是聚合关系还是组合关系并不是固定的，这要依实际情况而定。例如，汽车和轮胎的关系，对汽车而言，轮胎是一定要组合在汽车中的，因为它离开了汽车就没有意义了，但是在卖轮胎的店铺里，就算轮胎离开了汽车，它也是有意义的，这就可以用聚合了。聚合与组合的区别，关键一点是看组成部分与整体是否具有相同的生命周期。

依赖关系，这个关系比较特殊，因为前面三个关系都是静态的，而依赖关系是动态的。这里所谓静态是指无条件的，所谓动态是指有条件的。也就是说依赖关系是在特定的条件下产生的关系，例如张三某日从青岛去北京要乘火车，这时张三对火车就发生了依赖关系，若不是在

那日，张三和火车就没有产生依赖。所谓依赖就是指当被依赖的对象发生了变化，会对依赖对象产生影响，例如张三订的火车晚点了或者火车发生了故障，自然对张三就有影响了。

最后是关联，一般理解为不同的对象之间不是独立的，而是有关系、有联系，按照这种理解来说，五种关系中的前四种也属于关联关系。对关联还有另一种理解，那就是对象之间不是相互独立的，但又不属于前四种关系中的任何一种。我们这里采用的是后一种理解。关联在物理世界中是很普遍的，例如两个事物之间存在感应，万有引力就是一种感应，两个事物通过万有引力产生了关联；再例如客户和订单之间的关系也属于关联。

我们已经知道，计算机是要为人类服务的，而计算机要能够做到较好地为人类服务，就必须能够模拟人类所生活的物理世界。至此，我们了解了物理世界的三个核心方面：世界由万物构成；事物是运动的、发展的、变化的；事物之间的关系是复杂的、多变的。这三个核心方面展开就是上面的内容。那么计算机是如何模拟这三个核心方面的呢？这就涉及了程序的面向对象方法。

3.1.3 面向对象方法与 UML

开发一个具有一定规模和复杂性的软件系统和编写一个简单的程序大不一样。之间的差别，借用 G.Booch 的比喻，如同建造一座大厦和搭一个狗窝的差别。大型复杂的软件系统的开发是一项工程，必需按照软件工程学的方法组织软件的生产与管理，经过需求、分析、设计、实现、测试、维护等一系列的软件生命周期阶段。这是人们从软件危机中获得的最重要的教训和经验。这一认识促使了软件工程学的诞生。编程仍然是重要的，但是更具有决定意义的是系统建模。只有在分析和设计阶段建立了良好的系统模型，才有可能保证工程的正确实施。正是由于这一原因，许多首先在编程领域出现的新方法和新技术，总是很快地扩展到软件生命周期的分析与设计阶段。

面向对象方法正是经历了这样的发展过程，它首先在编程领域兴起，作为一种崭新的程序设计模型引起了世人的瞩目。继 Smalltalk-80 之后，20 世纪 80 年代又有一大批面向对象的编程语言问世，标志着面向对象方法走向成熟和实用。此时，面向对象方法开始向系统设计阶段延伸，出现了如 Booch86、GOOD（通用面向对象开发）、HOOD（层次式面向对象设计）、OOSD（面向对象结构设计）等一批 OOD（面向对象设计/开发）方法。但是这些早期的 OOD 方法不是以 OOA（面向对象分析）为基础的，而主要是基于结构化分析。到 1989 年之后，面向对象方法的研究重点开始转向软件生命周期的分析阶段，并将 OOA 和 OOD 密切地联系在一起，出现了一大批面向对象的分析与设计（OOA&D）方法。截至 1994 年，公开发表并具有一定影响的 OOA&D 方法已达 50 余种。这表明面向对象方法已经深入到分析与设计领域，并随着面向对象的测试、集成与演化技术的出现而发展为一套贯穿整个软件生命周期的方法体系。目前，大多数较先进的软件开发组织已经在分析、设计、编程、测试阶段全面地采用面向对象方法，使面向对象无可置疑地成为当前软件领域的主流技术。

1997 年，OMG 组织（Object Management Group）发布了统一建模语言（Unified Modeling Language，UML）。UML 的目标之一就是为开发团队提供标准通用的设计语言来开发和构建计算机软件。UML 提出了一套 IT 专业人员期待多年的统一的标准建模符号。通过使用 UML，这些人员能够方便地阅读和交流系统架构和设计规划——就像建筑工人多年来所使用的建筑设计图一样。到 2003 年，UML 已经获得了业界的广泛认同。

本书采用了部分 UML 图来表达面向对象的编程思想和编程方法。绘制 UML 图可以采用工具来实现，开源工具有 StarUML、ArgoUML 等。

3.2　类与对象

世界由万物构成，经由前面的学习，我们知道研究物理世界要用分类的方法。在面向对象程序设计中用类（class）来模拟一类事物，例如“学生”是一类人，用类实例对象（instance，也称为对象实例、对象，这里的对象是具体的，不是“面向对象”中抽象的“对象”）来模拟具体的个体，例如具体的某个学生“张三”。类实例对象是根据类来产生的，这样类就好比是模具，实例对象就好比是由模具生产的工件。

类和实例对象的关系就是本质与现象的关系。类代表的是一个类别，是一类事物，实例对象代表的是个体。在设计类时，是把众多个体的属性、功能抽象出来，形成一个类，这个类就是本质，它是不变的；使用类时，必需把它实例化，产生个体（即表象），而个体是千变万化的。就是使用类这个不变（本质）演绎出千变万化的实例（表象）。例如“学生”这个类可以演绎出“张三”、“李四”、“王五”等不同的个体实例对象。类与实例之间已经蕴涵着多态的思想。本质与现象的问题，在下面介绍完 Java 类的定义以及对象的创建语法后就更清楚了。

3.2.1　类的定义

类的定义需用关键字 class，定义一个简单类的语法如下：

```
class 类名{// 该行称为类头，大括号之间的部分称为类体
    零或多个属性;
    零或多个方法;
}
```

其中属性的定义格式有两种，如下：

（1）类型 属性名 [=属性值];

（2）类型 属性名 1 [=属性值 1],属性名 2 [=属性值 2],...;

例如：

```
1  class Student {
2     String sno; //学号
3     String name, major; //姓名,专业
4     int age; //年龄
5
6     void showInfo(){
7         System.out.println("学号:"+sno+"\t 姓名:"+name+
8             "\t 年龄:"+age+"\t 专业:"+major);
9     }
10
11    Student(){ //第一种构造方法
12    }
13
14    /*第二种构造方法*/
15    Student(String sno1,String name1, String major1,int age1){
```

```
16        sno=sno1;
17        name=name1;
18        major=major1;
19        age=age1;
20    }
21  }
22
```

注意，这里定义的是一个简单的类，完整的类的定义还涉及了许多其他内容，在后续章节中再一一补充。其中的构造方法这里暂不解释，暂时不用管它，详见 3.2.2 节。

Student 类模拟的是一类学生，该类学生都有属性：sno、name、major、age，都有一个方法 showInfo()，用来表明学生自己的信息，这些东西都是从很多具体的学生个体中归纳抽象出来的，由于这个做法比较简单，所以初学者普遍是这么做的，但未必能够意识到这是在归纳抽象，是在完成由表象到本质的一个过程。那么如何来模拟这一类学生中的某个个体呢？由前面的学习我们知道了，可以用类的实例对象来模拟。实例对象可以使用类的构造方法和 new 关键字构造出来。语法为：

类名　对象引用=new 构造方法;

例如：

```
Student t = new Student();
```

等号左边定义了一个变量 t，其类型为 Student 类，类型就是类别的意思，t 指向等号右边通过 new 构造的一个对象实例，变量 t 称为该对象实例的引用（reference），引用存储在栈（stack）内存中，而实例对象本身存储在堆（heap）内存中，如图 3.1 所示。

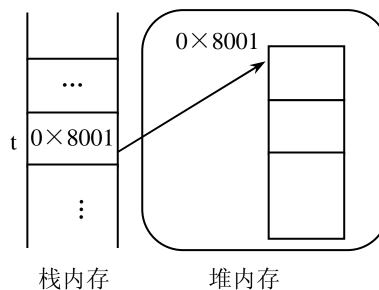

图 3.1　引用和对象

```
1  class ReferenceTest{
2      public static void main(String[] args){
3          Student t=new Student ();
4          //输出引用 t 的值，形如 Student@c17164
5          System.out.println(t);
6      }
7  }
```

由引用的值（上面程序的输出结果）可以看出引用不同于 C/C++中的指针，只是在指向对象方面相似。

1. 成员属性的定义

成员属性也称为成员变量，定义一个成员属性，格式为：

[修饰符]　类型　成员属性名[=成员属性值];

例如：

```
int a=10;
int b;
String s1;
String s2=new String("abc");
```

其中修饰符暂不讲，详见 3.8 节。

2. 成员方法的定义

语法为：

```
[修饰符] 返回类型 方法名([形式参数列表])
    [throws 异常类型列表] {//方法头
    执行语句块; //方法体
}
```

其中大括号前面的部分称为方法头，大括号部分称为方法体，throws 异常类型列表暂不介绍（详见第 5 章）。由于成员方法定义很简单，不再举例。

3. 成员变量的初始化

Java 中的变量分为对象成员变量和局部变量，局部变量是指方法的形式参数以及在方法体内声明的变量，其有效作用域只在方法执行过程中，方法一旦停止执行或者执行完毕，局部变量就失效了，在使用局部变量前必须先赋初始值，而成员变量若不赋初值，则采用默认值（也称缺省值，default），默认值如表 3.1 所示。

表 3.1　成员变量默认值

成员变量类型	默认值	成员变量类型	默认值
boolean	false	char	''（即空字符）
byte，short，int	0	long	0L
float	0.0F	double	0.0D
引用类型	null		

注意：数组属于 Java 对象，假如声明了一个数组变量而还没有分配空间，若该变量是成员变量，则自动初始化为 null，若该变量是局部变量，则在使用之前必须初始化；若刚分配了空间而没有赋初值，则无论是成员数组还是局部数组，则其元素都被自动赋予如表 3.1 所示的缺省值。

最后给出一个较完整的例子：

```
1  class StudentTest{
2    double[] array1;
3    int[] array2=new int[3];
4
5    public static void main(String[] args){
6       //使用了 Student 类的第一个构造方法构造出了一个学生
7       Student s1 = new Student();
8               s1.sno="20080101";
9               s1.name="张三";
```

```
10              s1.major="计算机";
11              s1.age=18;
12              s1.showInfo();
13
14          //使用了 Student 类的第二个构造方法构造出了另一个学生
15          Student s2=new Student("20080102","李四","数学",19);
16              s2.showInfo(); //使用引用访问其成员
17
18          int a=new Student().age;//使用匿名对象
19          System.out.println(a);
20
21          int b; //声明了一个局部变量
22          //System.out.println(b);//错误，使用前没有赋值
23
24          long array3=new long[3];//array3 的初值为{0L,0L,0L}
25          StudentTest t=new StudentTest();
26          //StudentTest 对象创建后 array1 的初值为 null，而 array2 的初值为{0,0,0}
27
28          for(long e:array3){
29             System.out.println(e);//输出 0L,0L,0L
30          }
31          for(long e:t.array2){
32             System.out.println(e);//输出 0,0,0
33          }
34
35          System.out.println(t.array1);//输出 null
36
37      }
38  }
39
```

上面代码中 public 是一种访问修饰符，详见 3.3.1 节。请读者自己分析该程序的执行结果。

3.2.2 构造方法

当使用 new 关键字创建一个类实例对象时，系统（JVM）会自动调用该类的构造方法来完成类实例对象的构建，从而构造方法对类定义来说是必须的。那么我们在类中如何声明一个构造方法呢？如下：

```
[访问修饰符] 类名([参数列表]){
   //......
}
```

说明：

（1）构造方法的访问修饰符可以是 public，protected 或者缺省（即不用访问修饰符），但不允许 private。

（2）构造方法的名称和类名相同，但没有返回类型。

（3）若没有定义任何构造方法，则在编译阶段编译器会在编译后的字节码文件中插入一

个默认的构造方法，默认的构造方法是：无参数列表，访问修饰符和类的访问修饰符相同，方法体内为空。

（4）构造方法本身不能递归调用。

（5）构造方法体的最后一行语句可以是空的 return 语句，即“return;”，也可以不用 return 语句。

（6）可在一个类中定义多个构造方法，其区别在于参数列表不同，这是构造方法的重载。关于重载，详见 3.5.1 节。

3.2.3　对象的创建与使用

创建一个对象要使用关键字 new，访问一个对象的成员属性或成员方法使用句点（.）。例如：

```
Student s=new Student();
s.age=20; //通过引用 s 访问

new Student().showInfo(); //通过对象直接访问
```

像上面代码最后一行那样创建的对象称为匿名对象，使用匿名对象有以下两种情况：

（1）若对一个对象的成员只需访问一次，这时该对象可以采用匿名对象；

（2）把匿名对象作为实际参数传递给调用方法，如 a.func(new Student());。

3.3　封装

在 3.1.2 节中介绍过封装，并谈到封装有两个作用：保护作用和增强对象的独立性。保护作用是通过设置访问修饰符来实现的。因此下面先谈谈访问修饰符。

3.3.1　访问修饰符

访问修饰符有 4 个：public、protected、默认访问修饰符、private。其中默认访问修饰符就是没有修饰符，即不用 public、protected、private 中的任何一个。访问修饰符用于对类、成员属性和成员方法设置访问权限，对它们实施保护作用。

（1）public 可用于修饰类、成员变量和方法。表明该成员变量和方法是共有的，能在任何情况下被访问。Java 应用程序中的 main()方法必须用 public 来修饰（否则能通过编译，但不能运行），就是为了使 JVM 能够访问它。

在一个源代码文件中最多只能定义一个 public 类，且该类名必须和源代码文件同名。

（2）protected 只能用于修饰成员属性或成员方法，不能修饰类。用 protected 修饰的成员可以被同包（package）下其他类方法访问，也可以被不同包下的子类方法访问。

（3）默认访问修饰符修饰的成员只能被同包下的类方法访问。

（4）private 修饰的成员只能被同类中的方法来访问。

这 4 种访问修饰符的访问范围由大到小，或安全保护程度由低到高依次是：public、protected、默认访问修饰符、private。

3.3.2 封装的保护作用

对标识符设置不同的访问修饰符，就对该标识符设置了不同的保护。例如：

```
1     class T{
2        private String s="abc";
3        public String getS(){//称作访问方法
4           return s;
5        }
6     }
7
8     class Test {
9        public void func(){
10           T t=new T();
11           String s1 = t.s; //非法
12           String s2 = t.getS(); //合法
13        }
14     }
```

在 T 类中对 s 设置了 private 保护，所以在 Test 的方法中直接访问 s 就不允许了（11 行代码），只能通过访问方法 getS()来访问，不能直接访问成员变量我们称作成员变量的隔离，就如同把贵重物品放到保险箱中一样。虽然不能直接访问 s，但还可以通过访问方法来访问，这如同保险箱没有锁上，只是把贵重物品放到保险箱中隔离了，安全保护还没有得到体现。对保险箱而言需要设置开箱密码，对程序而言，只需在访问方法的开头（第 3、4 行代码之间）加入一些安全检查代码即可。

3.3.3 增强独立性

类在软件中是最小的模块单元，是一个小系统、小世界，自身具有动和静的两面，是构成软件系统的基本模块，如同事物是构成物理世界的基本单位（指具有一定的系统意义的基本单位）一样。要增强类的独立性，也就是减少类对其他类的依赖，那么办法就是在类中尽可能使用接口，请结合课后习题第 3 题进行理解。

3.4 类的继承

3.4.1 继承

在 3.1.2 节中介绍过继承，并谈到继承的作用：一是提高了软件的开发速度，二是提高了开发软件的质量。继承有广义和狭义之分。广义的继承有些拿来主义的味道，因此在 Java 中分为横向手法和纵向手法。横向手法是指采用 import 语句来继承要使用的类或 Java 接口；纵向手法是指采用 extends 关键字来实现的手法。狭义的继承就是只指纵向手法。无论是横向手法还是纵向手法都不违背继承的两个作用。在面向对象编程中继承概念通常是指狭义的，但我们对继承的理解应该扩展到广义的概念上去。import 在 2.1.11 节已经介绍过，下面

谈谈 extends。

Java 采用关键字 extends 来描述两个类之间的纵向继承关系，例如：

```
1  class Student {
2     public void func() {
3        System.out.println("Student");
4     }
5  }
6
7  class Freshman extends Student {
8     public static void main(String[] args){
9        Freshman b=new Freshman();
10
11      /* 能够调用 Freshman 对象的 func 方法，
12       * 说明 func 来自 Student.
13       */
14      b.func();
15   }
16 }
```

其中 Student 类称为父类，Freshman 类称为子类。Java 不支持多重继承，单继承使 Java 的继承关系很简单，一个类只能有一个父类，易于管理程序，同时一个类可以实现多个接口（详见 3.6 节），从而克服单继承的缺点。

继承描述的是一种 is a 关系，例如上例 Freshman is a Student，是一种特殊与一般的关系。

继承是在两个不同的类之间发生的一种关系，被继承的内容是建立在访问权限（受访问修饰符控制）可访问的基础上的，有以下结论。

（1）子类的访问修饰符不小于父类的访问修饰符（private<默认<protected<public）。

（2）private 修饰的成员不可以被继承。

（3）默认访问修饰符修饰的成员只能被同包下的其他类继承。

（4）构造方法不能被继承。

（5）不管父类是否是 abstract 的，子类可以声明为 abstract 的，同样，父类中的方法不管是否是 abstract 的，在子类中都可以把该方法声明为 abstract 的。

读者可自行编程验证上述结论，也可以参考《Java 程序设计实训》的 6.2 节。abstract 详情参考 3.6 节。

3.4.2　继承的 UML 符号

UML 采用一个矩形框来表示一个类，矩形框内分三个部分：上部是类的名字，中部是类的成员属性，下部是类的方法。访问修饰符用+(public)、#(protected)、-(private)来表示。类的继承关系采用一个空心箭头来表示，箭头指向父类。如图 3.2 所示。

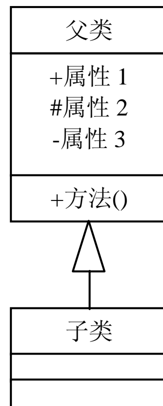

图 3.2　类的继承关系

3.4.3 this 与 super 关键字

this 代表类实例对象自身，super 代表父类实例对象。例如：

```
1  class A {
2     private int k;
3     func1(){
4        System.out.println("k="+k);
5     }
6     func2(int k){
7        this.func1(); //this 代表对象自身，该行代码等价于：func1();
8        this.k=k; //该行中的 this 不能省略。
9     }
10  }
11
12  class B extends A {
13     void func3(){
14        super.fun2(3); //super 代表父类对象
15     }
16  }
```

在一个类中可以定义多个构造方法，若在一个构造方法中调用另一个构造方法时，必须使用 this。例如：

```
1  class C{
2     public C(){
3        this(3);
4        //C(3); //这种用法就错了!
5     }
6     public C(int a){
7        System.out.println(a);
8     }
9     public static void main(String [] args){
10       new C(); //输出结果为 3
11       new C(5); //输出结果为 5
12    }
13  }
```

super 不可以连续使用，例如：

```
1  class A{
2     public void fun1(){
3        System.out.println("A");
4     }
5  }
6
7  class B extends A{
8     public void fun2(){
9        System.out.println("B");
10    }
```

```
11  }
12
13  class C extends B{
14      public void fun3(){
15          System.out.println("C");
16          super.fun2(); //输出结果为 B
17          super.super.fun1();// 错误，不可以连续使用 super
18          //调用 祖父对象的 func1()，应该使用下面的方法
19          ((A)this).func1();
20      }
21      public static void main(String[] args){
22          new C().fun3();
23      }
24  }
```

上例第 19 行代码涉及到了引用类型转换，详见 3.7 节。

3.5　多态

在 3.1.2 节中介绍过多态，Java 表达多态的方式有两种：一是重载（overload），二是覆盖（override）。

3.5.1　重载

重载是指在同一个类中定义多个同名方法，要求他们的参数列表不同。区分重载方法只能通过参数列表，例如：

```
1  class T{
2      public void func(){
3          System.out.println("func()");
4          func(10); //输出结果为 a=10
5      }
6      public void func(int a){
7          System.out.println("a="+a);
8      }
9      protected int func(int a, int b){...}
10     String func(String s){...}
11  }
```

重载方法对修饰符列表、返回类型、抛出异常类型（详见第 5 章）是否相同均不作要求，区别仅仅在于参数列表。

重载方法是程序的一种静态结构，即重载方法之间的关系在编译器编译后就确定了，不会随程序代码的执行来动态确定，这一点和覆盖不同，覆盖是程序的一种动态结构。正是由于重载是一种静态结构，所以完全可以把重载方法看作是各自不同的方法，这一点与不同名的方法之间的关系没有什么区别。

在一个重载方法内可以直接调用另外一个重载方法，但在一个重载的构造方法内却不能直接调用另一个重载的构造方法，必须使用 this，详见 3.4.3 节。

3.5.2 覆盖

覆盖是指在具有继承关系的两个不同类中定义同名方法，是建立在继承的基础上的，要求如下：

（1）private 方法不能被覆盖，因为 private 方法不允许被继承。

（2）方法名、返回类型、参数列表必须相同。

（3）子类方法的访问修饰符≥父类方法的访问修饰符。

（4）子类中的覆盖方法的修饰符不能是 final、static，因为 final 修饰的标识符是不可改变的，而覆盖就是对父类方法的一种修改，若父类方法不需要修改，只采用继承就好了。static 是一种静态行为，而覆盖则是一种动态行为。

（5）子类中的覆盖方法声明的抛出异常不能是父类被覆盖方法声明的抛出异常的祖先类，只能是其子类或同类。子类中的覆盖方法也可以不声明抛出异常，尽管父类中的被覆盖方法声明抛出了异常。详见第 5 章。

读者可自行编程验证上述结论，也可以参考本书配套的实训教材的 7.2 节。读者需要把上述要求与重载的要求做仔细对照，以加深对重载和覆盖的理解。

方法可以被覆盖，成员变量也可以被覆盖，例如下例中的变量 s：

```
1  class T1{
2     public String s="T1";
3     public String getS(){
4        return s;
5     }
6  }
7  class T2 extends T1{
8     public String s="T2";
9     public String getS(){
10       return s;
11    }
12 }
13 class Test{
14     public static void main(String[] args){
15        T1 t1=new T1();
16        T2 t2=new T2();
17
18        System.out.println(t1.s);   //输出  T1
19        System.out.println(t1.getS()); //输出 T1
20        System.out.println(t2.s);   //输出 T2
21        System.out.println(t2.getS()); //输出 T2
22
23        t1=t2;
24        System.out.println(t1.s); //输出 T1，而不是 T2
25        System.out.println(t1.getS()); //输出 T2，而不是 T1
26     }
27 }
```

从上面程序 24、25 两行代码的输出结果，可以得出如下结论：

（1）成员变量可以被覆盖。

（2）若直接访问发生覆盖的成员变量，则只能访问引用类型的成员变量，上例中第 24 行，虽然 t1 指向的对象是 T2 类型的，但引用 t1 的类型是 T1，故输出结果是 T1。

（3）若通过发生覆盖的方法来访问发生覆盖的成员变量，则访问的成员变量和方法属于同一个对象，例如第 25 行，t1 指向的对象是 T2 类型的，方法 getS()是属于 T2 实例对象的，故访问的变量是 T2 实例对象的。

所以，若上例中没有第 8 行代码，则成员变量 s 会被继承，从而第 25 行代码的输出是 T1。若没有第 9～11 行代码，会发生方法的继承，但方法本身还是类 T1 的，不是 T2 的，故第 25 行代码的输出是 T1。

3.6　接口与抽象类

3.6.1　接口

在 Java 中提到接口，有两种含义：一是 Java 接口，Java 语言中存在的结构，和类（class）相似，是一种 Java 引用类型；二是一个成员方法，即 API（Application Programming Interface）。为了区分这两种不同的含义，前者叫做“Java 接口”，后者叫做“接口”，Java 接口有时候也称为接口，读者从上下文中不难辨识。这里介绍的是“Java 接口”。

Java 接口的定义有两部分：接口头和接口体。格式如下：

```
[public | abstract] interface 接口名 [extends 接口列表]{
    常量声明;
    方法声明;
}
```

定义接口必须采用关键字 interface。接口中的成员属性修饰符只能是 static final，即使省略了，接口的成员属性也仍然是 static final 的。而接口的成员方法的修饰符则只能是 public abstract（同时采用这两个修饰符），即使省略了，默认的修饰符仍然是 public abstract，而不能是 static。例如：

```
interface T {
    static final String S1="abc";
    String S2="def"; // S2 的修饰符和 S1 的相同
    public abstract void func1();
    void func2(); //func2 的修饰符和 func1 的相同
}
```

接口中的方法只能有方法头，而不能有方法体。

与类相似，接口之间可以有继承关系，而与类的单继承不同的是，接口的继承允许多重继承。例如，若已经定义了两个接口 A、B，可以定义 C 来继承 A 和 B。如下：

```
interface C extends A, B{ //多个父接口之间用逗号分隔
    ... ...
}
```

接口不能使用关键字 new 来实例化，所以接口没有构造方法，定义好接口后，必需使用

一个具体的类采用 implements 关键字来实现它，例如：

```
class D implements A,B {
    ... ...
}
```

实现类也可以是抽象类，这时是部分实现了接口。

3.6.2 抽象类

类与接口是两个极端的情况，前者是所有方法都得到了实现，后者是所有方法都没有实现。在程序设计过程中，有时还需要一种中间状态，那就是只需要部分方法实现，其余的是抽象方法，这就是抽象类存在的必要性。当然抽象类中的方法可以全部都是抽象的，也可以全部都是实现的，这是两种极端的情况。定义一个抽象类的语法是：

```
[访问修饰符] abstract class 抽象类名 {
    abstract void func();
    ... ...
}
```

说明：

（1）抽象类必须采用修饰符 abstract。

（2）含有抽象方法的类必须声明为抽象类。

（3）抽象类不能使用关键字 new 实例化，但抽象类可以有构造方法。

（4）抽象类是一种特殊的类，遵循类的单继承规则和（单/多）接口实现规则。

一个类可以在继承一个父类的同时，实现一个或多个接口，例如：

```
class D extends C implements A,B{
    ... ...
}
```

其中 A、B 是两个接口，C 是父类。

3.7 引用类型的类型转换

Java 的数据类型有两类：一是基本类型；二是引用类型。在 2.1.6 节介绍过基本类型的转换，现在谈谈引用类型的转换。引用类型的类型转换发生在具有继承关系（extends）或者具有实现关系（implements）的类型之间。同基本类型一样，引用类型也分为自动类型转换和强制类型转换两种。

3.7.1 自动类型转换

在具有继承关系的类与类之间，或者接口与接口之间，或者具有实现关系的接口和类之间，由下层类型向上层类型转换时，发生自动类型转换。自动类型转换也称为类型自动提升。例如：

```
1   interface I{
2       public void fun();
3   }
4
```

```
5    class T1 implements I {
6       public void fun(){}
7    }
8
9    class T2 extends T1{}
10
11   class Test {
12       public static void main(String[] args){
13           I i= new T1();//合法，发生自动类型转换
14           i= new T2(); //合法
15           T1 t1=new T2(); //合法
16       }
17   }
```

3.7.2　强制类型转换

在具有继承关系的类之间或者具有实现关系的接口和类之间，由上层类型向下层类型转换时，发生强制类型转换，注意其前提是对象的类型必须是转换目标类型本身或者是其子孙类型，即 obj instanceof RefType 表达式的值为 true 时，否则不能转换。例如：

```
1     interface I {
2        public void fun();
3     }
4     class T1 {
5        public void fun() {}
6     }
7     class T2 extends T1 {}
8     class T3 extends T2 {}
9
10     class K extends T1 {}
11
12     class Test{
13        public static void main(String[] args){
14            T1 t1= new T3();
15            T2 t2= (T2)t1; //合法，发生强制类型转换
16            T3 t3=(T3)t2; //合法
17            t3 = (T3)t1; //合法
18
19            t1=new T1();
20            t2=(T2)t1; //非法，因为 t1 指向的对象是 T1 类型的
21
22            t1= new K();
23            t2=(T2)t1; //非法，因为虽然 T2,K 都继承了 T1，但 t1
24                     //指向的对象是 K 类型的，不是 T2 类型的
25
26            K a = new K();
27            I b=(I)a; //合法
```

```
28            t1=(T1)b; //合法
29            t2=(T2)b; //非法
30            K c=(K)b; //合法
31            c=(K)t1; //合法
32         }
33     }
```

在发生强制类型转换之前，必然已经发生过类型的自动提升转换。

这里提一下数组的类型转换，数组是引用类型，默认其父类是 Object，且实现了 Java 接口 Cloneable 和 Serializable。例如：

```
/*下面是一个基本类型的数组例子*/
int[] a = {1,2,3};
Object obj = a; //合法
int[] b = (int[]) obj; //合法

Long[] L = a; //不合法，不存在这样的转换
byte[] by=(byte[])a; //不合法，不存在这样的用法

/*下面是一个对象类型的数组例子*/
String [] s={"a","b"};
Object os=s;    //合法
String[] str=(String[])os; //合法
```

数组类型的转换在实际编程中用的较少，读者了解即可。

学习了引用类型的类型转换后，就可以使用前面讲解的继承、多态思想和编程技术来进行这样的程序设计了：透过现象抓本质，以不变应万变。这属于从思想理论到具体上机实践，非常重要。

3.7.3 自动打包/拆包机制

从 JDK 1.5 中引入了一个新的机制，基本类型的自动打包和其对应的引用类型的自动拆包机制。在 JDK 1.5 之前，基本类型与其封装类之间的转换作如下处理（以 int 类型为例）：

```
int i = 123;
Integer ic = new Integer(i); //把基本类型打包为一个对象
i= ic.intValue(); //把一个对象拆包为基本类型
```

而在 JDK 1.5 及以后的版本中，可以如下使用：

```
int i=123;
Integer ic=i; //自动把 i 打包成对象
i=ic;        //自动把 ic 拆包为基本类型
```

这是一个很好的改进，方便了编程。

3.8 其他修饰符

3.8.1 final

final 是最终、最后的意思，即不能再发生变化、终结了。因此，若 final 修饰变量（不管

是局部变量还是成员变量），则变量一旦被赋值后就不能再改变其值了；若 final 修饰方法，则该方法就不能再被覆盖了，因为要动态（即在程序运行过程中）修改方法的功能的办法就是采用覆盖；若采用 final 修饰类，则该类就不能再被继承，当然 final 类的方法也就不能被覆盖了，这是因为动态修改类的定义的办法只能是在继承的基础上修改，而覆盖的前提是能够继承，现在继承都不允许了，当然其方法也就不能被覆盖了。

3.8.2　static

static 修饰符可以修饰：①成员属性；②成员方法；③代码块。

static 不能修饰方法内的局部变量（包括方法体内的局部变量和形式参数）。

1．static 成员属性

static 成员属性可以存在于类中，也可以存在于接口中。对于接口中的 static 成员属性，在声明的同时必须赋初值，因为接口中的成员属性是 final 的。类中的 static 成员属性为该类所有的类实例对象所共享，接口中的 static 成员属性为该接口的所有实现类实例对象所共享。

访问 static 成员可以通过类实例对象来访问，也可以通过类或者接口本身来访问，建议采用后者方式。例如：

```
1  interface I{
2     int a=20;  //默认的修饰符是 public static final
3     public static final int b=15;
4  }
5  class A implements I {}
6  class T{
7     static int k=10;
8     static void func(){
9        System.out.println("k="+k);
10    }
11    public static void main(String[] args){
12       int test;          // 注意：在使用 test 变量前必须首先赋值
13       test=I.a;          // 对接口中的 static 属性通过接口本身来访问
14       test=new A().a;    //通过接口的实现类的实例对象来访问
15       test=A.a;              //通过接口的实现类来访问
16
17       T t=new T();
18       int a=t.k;         // 通过类的实例对象来访问
19       t.func();
20       a=T.k;                 // 通过类本身来访问，建议采用这种访问方式
21       T.func();
22
23       T t1=new T();
24       T t2=new T()
25       System.out.println("T.k="+T.k); //输出结果为 T.k=10
26       t1.k=20; //修改了对象 t1 的属性 k
27
```

```
28          //输出结果为 t2.k=20，说明 k 在 t1, t2 之间共享
29          System.out.println("t2.k="+t2.k);
30          System.out.println("T.k="+T.k); //输出结果为 T.k=20
31      }
32  }
```

请读者自行分析上面程序代码及其输出结果。

2. static 成员方法

static 成员方法只能存在于类中，不能存在于接口中，接口中的方法不能是 static，也不能是 final 的。

访问 static 方法的方式和访问 static 成员属性的方式相同。在使用 static 方法时，注意：

（1）在 static 方法内只能直接访问同类中其他的 static 成员（包括成员属性和方法），不能直接访问非 static 成员。对非 static 成员，只能先使用 new 创建类实例对象，然后通过类实例对象来访问其非 static 成员。

（2）static 方法内不能使用 this 或 super 关键字。

通过对 static 关键字的学习，就明白了 main()方法为什么要设计成 static 的原因了。JVM 可以直接调用 main()方法，而不必事先创建一个对象。

类的 static 成员通常称作类成员，类的 static 属性通常称作类属性，类的 static 方法通常称作类方法。static 属于静态行为，在程序的编译阶段就确定了，而动态行为则是在程序运行时随着程序代码的执行情况而定的，所以 static 方法可以被继承，但不可以被覆盖。例如：

```
1      class A implements I{
2         static int b=9; //可以被继承
3         static void f(){
4            System.out.println("A");
5         }
6         void g(){
7            System.out.println("A.g()");
8         }
9         static void h(){
10            System.out.println("Hello");
11         }
12     }
13     class B extends A{
14         static void f(){
15            System.out.println("B");
16         }
17         void g(){
18            System.out.println("B.g()");
19         }
20     }
21     class T{
22         public static void main(String[] args){
23            A a=new B();
```

```
24              a.f();      //该行说明 static 方法不能被覆盖，
25                          //该情况说明 static 方法属于各自的类，
26                          //故该行和下一行是相同的
27              A.f();
28
29              a.g();      //输出结果为 B.g()，说明方法覆盖成功
30              int b=B.b;  //该行说明 static 成员属性可以被继承
31              B.h();      //该行说明 static 方法可以被继承
32          }
33      }
```

上例程序中，若在类 B 中有成员属性 static b=2；则会覆盖掉从类 A 中继承下来的 b 的值。

3. static 代码块

在类中可以使用 0～n 个 static 代码块，不允许在接口中使用 static 代码块。当类被加载到内存时，这些 static 代码块按照定义的先后顺序被执行且仅被执行一遍。例如：

```
1     class T{
2         static int k;
3         static int b;
4         static {
5             k=10;
6             System.out.println("h1");
7         }
8
9         int a=5;
10
11         static {
12             System.out.println("h2");
13         }
14     }
15
16     class Test {
17         public static void main(String[] args){
18             /* 下面两行输出结果说明执行的顺序是：
19                 先执行 static 成员属性代码行，若没有赋值，则采用默认值；
20                 然后再执行 static 代码块。*/
21             System.out.println(T.k); //输出结果为 10
22             System.out.println(T.b); //输出结果为 0
23
24             T t=new T();  //没有输出内容
25             t=new T();    //没有输出内容。说明 static 代码块是在
26                           //加载时执行的
27         }
28     }
```

static 代码块通常用于对类作初始化工作。虽然 Java 支持在一个类中可以存在多个 static 代码块，但在实际编程中，没有理由这样做，若出现多个 static 代码块，可以把它们合并到一个 static 代码块中，否则会导致程序代码的可读性差。

3.8.3 native

一个 native 方法（也称作本地方法）就是一个 Java 程序调用非 Java 代码的接口。native 方法的实现采用非 Java 语言实现，比如 C、Fortran、汇编语言等。这个特征并非 Java 所特有，很多其他的编程语言都有这一机制，比如在 C＋＋中，可以用 extern"C"告知 C＋＋编译器去调用一个 C 的函数。因为在外部实现了方法，所以在 java 代码中，只需要声明方法头就可以了，类似 Java 抽象方法的声明那样。native 可以和其他一些修饰符连用，但是 abstract 方法和接口方法不能用 native 来修饰。

```
1     public class Test {
2        public native void doMethod();
3
4        static {
5           //native 方法的实现放在本地库 NativeLib 中
6           System.loadLibrary("NativeLib"); //首先加载本地库
7        }
8
9        public static void main(String[] args){
10            Test t=new Test();
11            //使用 native 方法的方式与使用非 native 方法的方式相同
12            t.dMethod();
13        }
14     }
```

Object 类中的很多方法，例如 clone()和 notify()都是 native 方法。在 Java 应用程序中使用 native 方法会打破 Java 的“Write once, run anywhere”的特征。既然这样，为什么 Java 要支持 native 方法呢？

首先，Java 虽然使用起来非常方便，然而有些层次的任务用 Java 实现起来并不容易，或者我们对程序的效率很在意时，例如科学计算程序，问题就来了。

其次，有时 Java 应用需要与 Java 外面的环境交互。这是 native 方法存在的主要原因，例如 Java 需要与一些底层系统，如操作系统或某些硬件交换信息时的情况。native 方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且无需去了解 Java 应用之外繁琐的细节。

最后，与操作系统交互，JDK、JVM 是个中间层系统，它们依赖底层的操作系统，同时它们又为上层的 Java 应用程序提供接口支持，这需要使用 native 方法。

JVM 怎样使 native 方法运行起来呢？我们知道，当一个类第一次被使用时，这个类的字节码会被加载到内存，并且只会加载一次。这时可以在 static 代码块中使用语句 System.loadLibrary()来加载含有 native 方法实现的本地动态库。然后在访问 native 方法时，JVM 就会自动在已加载的本地库中查找并执行 native 方法的实现了。

注意，使用 native 方法是有代价的，它丧失了 Java 的很多好处，尤其是跨平台特征。只有别无选择，才使用 native 方法。

3.8.4 transient

transient 只能修饰类的成员变量，标记为 transient 的变量，在类实例对象被存储（也称为序列化）时，这些变量状态不会被持久化，详情见 7.4.4 节。

3.8.5 strictfp

strictfp（strict float point），意思是精确计算浮点数。在 Java 虚拟机进行浮点运算时，如果没有指定 strictfp 关键字，Java 的编译器以及运行环境在对浮点运算的表达式采取一种近似于我行我素的行为，以致于得到的结果往往无法令人满意。而一旦使用了 strictfp 来声明一个类、接口或者方法时，那么 Java 编译器以及运行环境会依照浮点规范 IEEE-754 来执行精确的浮点计算。因此如果你想让你的浮点运算更加精确，而且不会因为不同的硬件平台所执行的结果不一致的话，那就用关键字 strictfp。

可以将一个类、Java 接口以及方法声明为 strictfp，但是不允许对接口中的方法以及构造函数声明 strictfp 关键字，例如：

```
1     strictfp interface A {}
2
3     interface A {
4        strictfp void f(); //错误的
5     }
6
7     public strictfp class FpDemo {
8        strictfp void f() {}
9        strictfp FpDemo(){} //错误的
10    }
11
```

当某个类、Java 接口或者方法用 strictfp 声明，内部所有的 float 和 double 表达式都会成为 strictfp 的。

3.8.6 volatile

volatile 修饰符用得很少，只修饰成员变量，用在多处理器环境中的线程异步通信，在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

3.8.7 assert

J2SE 1.4 在语言上提供了一个新特性，就是 assertion（断言）功能，它是该版本在 Java 语言方面最大的革新。在软件开发中，assertion 是一种经典的调试、测试方式，很多编程语言中都支持这种机制。

assertion 就是在程序代码中的一条语句，它对一个 boolean 表达式进行检查，一个正确程序必须保证这个 boolean 表达式的值为 true；如果该值为 false，说明程序已经处于不正确的状

态，系统将给出警告或退出。

一般来说，assertion 用于保证程序最基本、关键的正确性。assertion 检查通常在开发和测试时开启。为了提高性能，在软件发布后，assertion 检查通常是关闭的。下面简单介绍一下 Java 中 assertion 的实现。

1．assert 语法

assert 关键字语法很简单，有两种用法：

（1）assert　<boolean 表达式>

如果<boolean 表达式>为 true，则程序继续执行。如果为 false，则程序抛出 AssertionError，并终止执行。

（2）assert　<Boolean 表达式> : <错误信息表达式>

如果<boolean 表达式>为 true，则程序继续执行。如果为 false，则程序抛出 java.lang.AssertionError，并输出<错误信息表达式>。

下面是一些 assert 的例子。

```
1. assert   0 < value;
2. assert   0 < value : "value="+value;
3. assert   ref != null : "ref doesn't equal null";
4. assert   isBalanced();
```

2．编译与运行

编译带有 assert 语句的程序与普通程序没有区别，但默认情况下 JVM 关闭了 assertion 语句的执行，因此，运行带有 assertion 语句的程序，必须向 JVM 中传递开启参数。有以下两类参数：

（1）参数-esa 和-dsa：它们的含义为开启（关闭）系统类的 assertion 功能。由于新版本 Java 的系统类中，也使了 assertion 语句，因此如果用户需要观察它们的运行情况，就需要打开系统类的 assertion 功能，我们可使用-esa 参数打开，使用-dsa 参数关闭。-esa 和-dsa 的全名为-enablesystemassertions 和-disenablesystemassertions，全名和缩写名有同样的作用。

（2）参数-ea 和-da：它们的含义为开启（关闭）用户类的 assertion 功能。通过这个参数，用户可以打开某些类或包的 assertion 功能，同样用户也可以关闭某些类和包的 assertion 功能。打开 assertion 功能参数为-ea；如果不带任何参数，表示打开所有用户类；如果带有包名称或者类名称，表示打开这些类或包；如果包名称后面跟有三个点，代表这个包及其子包；如果只有三个点，代表无名包。关闭 assertion 功能参数为-da，使用方法与-ea 类似。-ea 和-da 的全名为-enableassertions 和-disenableassertions，全名和缩写名有同样的作用。

下面表格表示了参数及其含义，并有例子说明如何使用。

参数	举例	作用
-ea	java -ea	打开所有用户类的 assertion
-da	java -da	关闭所有用户类的 assertion
-ea: <classname>	java -ea:myclass	打开 myclass 的 assertion
-da: <classname>	java -da:myclass	关闭 myclass 的 assertion
-ea: <packagename>	java -ea:pkg	打开 pkg 包的 assertion
-da: <packagename>	java -da:pkg	关闭 pkg 包的 assertion

续表

参数	举例	作用
-ea:...	java -ea:...	打开缺省包（无名包）的 assertion
-da:...	java -da:...	关闭缺省包（无名包）的 assertion
-ea: <packagename>…	java -ea:pkg...	打开 pkg 包和其子包的 assertion
-da: <packagename>…	java -da:pkg...	关闭 pkg 包和其子包的 assertion
-esa	java -esa	打开系统类的 assertion
-dsa	java -dsa	关闭系统类的 assertion
综合使用	java -dsa:myclass:pkg	关闭 myclass 和 pkg 包的 assertion

其中...表示此包及其子包的含义。

assertion 为开发人员提供了一种灵活地调试和测试机制，它的使用也非常简单、方便。今后编写 Java 程序时，不必再使用繁琐的 System.out.println()语句来测试了，可以改用 assertion 语句。

3.9 类实例对象的创建过程

通过下面的例程来说明实例对象的创建过程。

```
1  class A{
2      int a;
3      static {
4          System.out.println("static block in A");
5      }
6      public A(){
7          System.out.println("a="+a);
8          System.out.println("Constructor of A");
9      }
10  }
11  class B extends A{
12      private int k;
13
14      public B(){
15          System.out.println("k="+k);
16          System.out.println("Constructor of B");
17      }
18
19      static {
20          System.out.println("static block in B");
21      }
22
23      public static void main(String[] args){
24          new B();
25      }
26  }
27
```

执行结果为：

```
static block in A
static block in B
a=0
Constructor of A
k=0
Constructor of B
```

执行结果说明 Java 类实例化的先后顺序为：

（1）如果有 static 代码块，也称为 static initializer，则首先运行 static 代码块；若有继承关系，则首先执行父类的，再执行子类的。

（2）若有继承关系，则给父类对象的成员属性赋默认值。

（3）若有继承关系，则调用父类对象的构造方法，完成父类对象的构建。

（4）对对象的成员属性赋默认值。

（5）调用对象的构造方法最终完成对象的构建。

对成员属性赋默认值，如 3.2.1 节中的表 3.1 所示。

3.10 内部类

内部类，顾名思义，就是在一个类中再定义一个类，即类中类。这里介绍内部类是因后文 11.6.3 节的需要。内部类分为 3 种：成员内部类、方法内部类和匿名内部类。

3.10.1 成员内部类

成员内部类与类的成员（变量或方法）的地位是相同的，因而在修饰符和访问空间上相似，在这个意义下，可以把成员内部类当作一个成员来看待。成员内部类按照有无 static 修饰又分为两类：static 成员内部类，非 static 成员内部类。

1. static 成员内部类

static 成员内部类，可以在类或者 Java 接口中声明。例如。在类的内部声明一个内部类：

```
1  class OuterClass{
2      public int k=10;
3      private static String str="abc"; //可直接被内部类访问
4
5      public static class InnerClass {
6         private String s="inner class";
7         public void func(){
8            System.out.println(s);
9
10           //可以直接访问外部类的 static 成员
11           System.out.println(str);
12
13           //也可以这样访问
14           System.out.println(OuterClass.str);
15
16           //只能通过创建对象来访问外部类非 static 成员
```

```
17            int k1=new OuterClass().k;
18            int k2=k; //此行错误，不能这样访问
19            System.out.println("k:="+k1);
20        }
21    }
22
23    //测试
24    public static void main(String[] args){
25        /*注意：
26          声明引用变量的类型；
27          new 后面的构造方法。
28         */
29        OuterClass.InnerClass obj =
30            new OuterClass.InnerClass();
31        obj.func();
32    }
33 }
```

在接口内部声明一个内部类：

```
1  public interface OuterInterface{
2      public static class InnerClass {
3          private String s="inner class";
4          public void func(){
5              System.out.println(s);
6          }
7      }
8  }
9
10 class Test {
11     //测试
12     public static void main(String[] args){
13         /*注意：
14           声明引用变量的类型；
15           new 后面的构造方法。
16           */
17         OuterInterface.InnerClass obj =
18             new OuterInterface.InnerClass();
19         obj.func();
20     }
21 }
```

通过上面两个程序，可归纳 static 成员内部类的定义和使用方式如下：

（1）在类或者接口内部，使用 static 修饰符来声明一个成员内部类，可以有也可以没有访问修饰符。

（2）在使用 new 创建该内部类对象时，外部类或接口相当于一个 Java 包。

（3）可在该内部类中直接访问外部类或者接口的 static 成员，访问非 static 成员只能通过创建外部类对象来实现，注意 Java 接口不能被实例化，从而在接口中定义的成员都是 static 的（接口中的成员的 static 修饰符可省略，省略后仍是 static 的）。

（4）声明该内部类体中的成员的方式和普通类相同。

2. 非 static 成员内部类

非 static 成员内部类自然不能使用修饰符 static，因此只能在类（不能在 Java 接口）中声明该类型的内部类。

```
1   class OuterClass{
2           public int a=10;
3           static String s="abc";
4           private int c=5;
5
6           public class InnerClass{
7               public int b=100;
8               static int sb=1;
9               //上面一行错误，因为此类型的内部类不允许定义 static 成员
10              public void innerFunc(){
11                  System.out.println("inner b:="+b);
12                  //可以直接访问外部类的所有成员
13                  System.out.println("a:="+a);
14                  System.out.println("s:="+s);
15                  System.out.println("c:="+c);
16              }
17          }
18
19          public void outerFunc(){
20              System.out.println("a:="+a);
21          }
22
23          //在外部类方法中可以这样创建内部类对象
24          public void createInner(){
25              InnerClass in=new InnerClass();
26              in.innerFunc();
27          }
28
29          //测试
30          public static void main(String[] args){
31              //可以这样创建内部类对象
32              OuterClass.InnerClass c= new OuterClass().new
33                          InnerClass();
34              c.innerFunc();
35
36              new OuterClass().createInner();
37          }
38    }
```

通过上面的例程可以总结出非 static 成员内部类的定义和实例化方法如下：

（1）只能在类中（不能在接口中）声明非 static 成员内部类，修饰符不能有 static。

（2）能够直接在该内部类中访问外部类的一切成员，甚至是 private 成员。

（3）该内部类中不能声明 static 成员。

（4）只能先创建外部类对象，然后在外部类对象内创建内部类对象（new OuterClass().new InnerClass()）。

3.10.2　方法内部类

方法内部类就是在外部类的方法中声明的内部类，例如：

```
1  class OuterClass{
2      private int oa=1;
3      public void outerFunc(){
4          public class InnerClass{//public 是非法的
5              int b=10;
6              public void innerFunc(int m, final int n){
7                  static int si=10; //static 是非法的
8                  int ia=m+n;
9                  final int ib=m-n;
10                 System.out.println("oa:="+oa);
11                 System.out.println("m:="+m);
12                 System.out.println("ia:="+ia);
13                 System.out.println("ib:="+ib);
14             }
15         }//InnerClass
16
17         InnerClass c=new InnerClass();//实例化内部类
18         c.innerFunc(1,2);
19     }
20
21     //测试
22     public static void main(String[] args){
23         OuterClass out=new OuterClass();
24         out.outerFunc();
25     }
26 }
```

由该例程可以总结出方法内部类的定义和实例化方法如下：

（1）在外部类的成员方法内定义一个内部类，不能使用访问修饰符，也不能用 static 修饰。

（2）方法内部类的作用空间只在方法体内，因此必须在声明内部类后，在结束该方法之前实例化该内部类，并访问其方法。

（3）该内部类可访问一切外部类成员和保护该内部类方法的合法局部变量。

（4）该内部类中不能声明 static 成员。

3.10.3　匿名内部类

匿名内部类就是没有名字（类的标识符）的内部类，它在被声明的同时被实例化。例如：

```
1  //AnonymousInnerClass.java
2
3  interface I {
4      void func1();
5  }
6
7  abstract class B {
8      public abstract void func2();
9  }
```

```
10
11  class C{
12      void func3(){
13          System.out.println("C.func3");
14      }
15  }
16
17  class D{
18      public D(int k){
19          System.out.println("k:="+k);
20      }
21      void funcD(){
22          System.out.println("D.funcD");
23      }
24  }
25
26  class Test{
27      public void test(){
28          I it = new I(){//注意接口是没有构造方法的
29              public void func1(){
30                  System.out.println("func1");
31              }
32          };
33
34          it.func1();
35
36
37          B st = new B(){//抽象类也不会有构造方法的
38              public void func2(){
39                  System.out.println("func2");
40              }
41          };
42
43          st.func2();
44
45          new C(){//执行时，调用了 C 缺省的构造方法
46              public void func3(){
47                  System.out.println("func3");
48              }
49          }.func3();
50
51          new D(44){//执行时，调用了 D 对应的构造方法
52              public void func4(){//与 D 中定义的方法不同
53                  System.out.println("func4");
54              }
55          }.func4();
56      }
57
58      public static void main(String[] args){
59           new Test().test();
```

```
60     }
61 }
62
```

程序代码的第 28～34 行测试的是一个接口 I 的匿名实现类，声明的同时被实例化，该匿名类是接口 I 的实现类，注意第 32 行的分号是不可少的。第 37～43 行测试的是一个抽象类 B 的匿名继承类。第 45～49 行测试的是一个普通类 D 的继承类。第 51～55 行测试的是一个带有非缺省构造方法的普通类的匿名继承类。注意第 52 行代码定义的方法与类 D 中的方法不同。在定义的匿名类的方法必须采用 public 访问修饰符，采用其他访问修饰符都不对。

匿名内部类在 Java GUI 编程中，常常用于实现事件监听器。请读者自己从该例程中总结出匿名内部类的声明和使用方式。

3.11 对象之间的关系

在 3.1.2 节部分描述了物理世界中事物之间的关系，有泛化、聚合、组合、依赖和关联。

3.11.1 泛化

泛化描述的是抽象与具体（或者称为一般与特殊、本质与现象）的关系，是一种 is-a 关系。

本质是抽象的、稳定的、简单的，现象是具体的、多变的、复杂的。例如水、冰、水蒸气是 H_2O 这种事物的现象，而本质是 H_2O。这就指导我们在把握事物的时候，要透过现象抓本质，以不变应万变。即透过各种复杂多变的现象抽象出本质来，然后运用本质来指导对具体现象的认识和把握。在计算机软件设计中就是把具体的事物（模块、类）抽象成接口或者祖先类，接口或者祖先类就是本质，不同模块之间的通信或者方法的调用都是通过接口或者祖先类这个本质完成的。这样当底层具体的模块或者类发生变化时，不会影响到上层接口或者祖先类之间的协作。这一点在软件后期的维护中作用是很重要的。

例如：A 公司是销售台式机电脑的，现在委托软件公司 B 为其开发一套销售系统，系统用了一段时间后，A 公司的业务转向了销售笔记本电脑，这时 A 要求 B 来修改软件系统。若 B 在开发软件系统时，没有考虑到本质与现象的关系并作相关处理，这时做软件维护几乎是不可能的，若 B 考虑了本质与现象的关系，在开发阶段，就会把“销售台式机电脑”这种具体的业务模块（现象）抽象出“销售产品”接口（本质）来，而“销售台式机电脑”业务模块只不过是“销售产品”业务接口的一种表象而已，当 A 的业务由台式机电脑转向了笔记本电脑时，只需把台式机电脑业务模块替换为笔记本电脑业务模块即可，如图 3.3 所示。

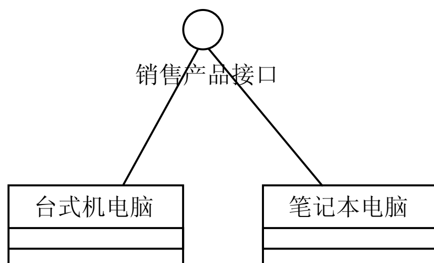

图 3.3 泛化

随着时间的延续，A 的业务可能还在不断的变化，这都没有关系，只需把相应的业务模块替换一下就可以继续使用该软件系统了。通过泛化，大大节省了维护成本，减少了维护时间，延长了软件的生命周期。

泛化包括两个过程：自下而上（由具体到抽象，或由表象到本质）和自上而下（由抽象到具体，或由本质到表象）。其中难点是自下而上这一过程。

在 Java 中综合运用继承（extends）、多态、实现（implements）这三种技术来表示泛化关系。继承包括抽象类之间、非抽象类之间、抽象类和非抽象类之间、接口之间的继承。多态的基础是继承（或者 Java 接口实现）和引用类型的自动提升，如图 3.4 所示。

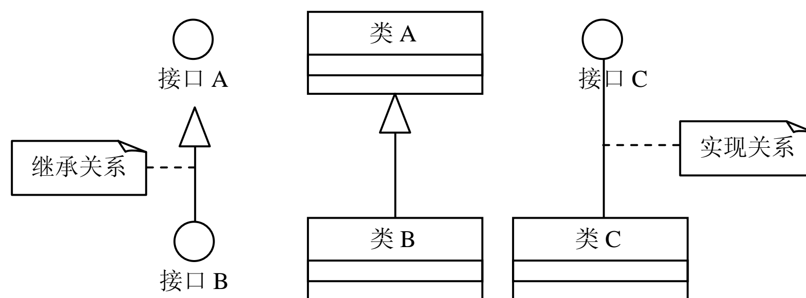

图 3.4 泛化的 java 技术

3.11.2 聚合、组合

聚合、组合描述的是整体与部分之间的关系，是一种 has-a 关系，在 3.1.2 节已有介绍。聚合关系的两个类处于不同的层次，一个是整体，一个是部分。同时，是一种弱的“拥有”关系。体现的是 A 与 B 分离后能够独立存在，而组合则不然。然而，聚合与组合的代码表现形式是一样的，都可以表现为以下的形式，它们仅仅具有语义上的区别。

```
import java.util.*;
class B{ ... }
class A{
    List<B> bList = new ArrayList<B>();
    public A(){
        //至于 b 在什么时候被实例化，则视具体情况而定
        bList.add(new B());
        bList.add(new B());
        ...
    }
}
```

一般地，整体是一个，而部分有多个，例如学校由很多学生聚合而成，故在整体对象中一般采用 Java 的集合类来存放部分个体，Java 的集合类详见第 8 章。聚合关系采用 UML 符号◇—表示，组合关系采用◆—表示，如图 3.5 所示。

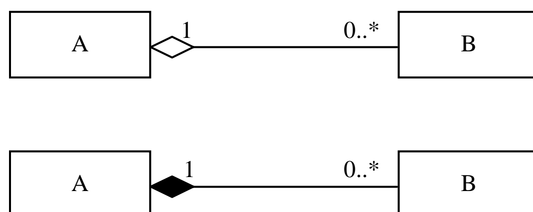

图 3.5　聚合，组合关系 UML 表示

3.11.3　依赖

两个相对独立的对象，当一个对象负责构造另一个对象，或者依赖另一个对象的服务时，这两个对象之间主要体现为依赖关系，例如生产零件的机器和零件，机器负责构造零件对象。再例如充电电池和充电器，充电电池通过充电器来充电。依赖关系是一种 use a 关系。是类与类之间的连接，表示一个类依赖于另一个类的定义，其中一个类的变化将影响另外一个类。例如，如果 A 依赖于 B，则 B 体现为 A 中方法的参数、方法内的局部变量或静态方法的调用，其实例代码如下：

```
class B {
    public static void bFunc(){...}
}
class A {
    aFunc(B b1) { //方法参数
        B b2 = new B(); //局部变量
        ...
        B.bFunc(); //静态方法的调用
    }
}
```

依赖关系在 UML 中采用虚箭头表示，如图 3.6 所示。

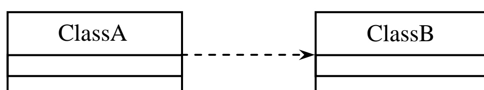

图 3.6　依赖关系 UML 表示

3.11.4　关联

关联关系表示不同对象之间的结构关系，它将多个类的实例永久地连接在一起（这与依赖关系不同，依赖关系表示两个实例之间的临时关联关系）。例如：客户和订单，每个订单对应特定的客户，每个客户对应一些特定的订单。

多数关联关系是二元的(即只存在于两个类之间)。关联关系分为单向的和双向的，在 UML 中分别采用箭头和直线段表示。关联关系的两端为角色，角色规定了类在关联关系中所起的作用。每个角色都必须有名称，而且对应一个类的所有角色名称都必须是唯一的。角色名称应该是一个名词，能够表达被关联关系对象的角色与关联关系对象之间的关系。以一个订单输入系

统中各类之间的关系为例，客户可以使用两种不同地址：一个账单发送地址和一些订货发送地址。结果，客户和地址之间存在两个关联关系，如图 3.7 所示。这些关联关系中标注了关联关系地址为客户担任的角色。

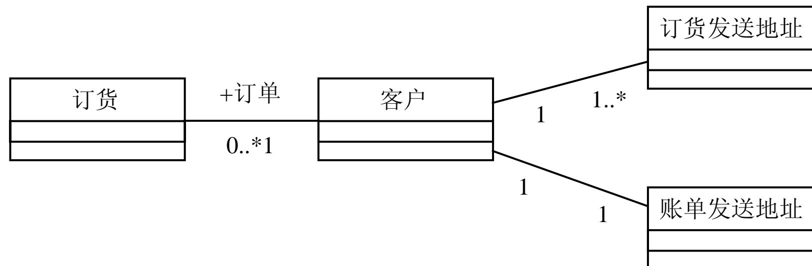

图 3.7　关联关系 UML 表示

3.12　面向对象程序设计的基本原则

在学习和使用面向对象（OO）设计的时候，我们应该明白：OO 的出现使得软件工程师们能够用更接近真实世界的方法描述软件系统。然而，软件毕竟是建立在抽象层次上的东西，再怎么接近真实，也不能替代真实。

OO 设计有五大原则，它们之间并不是相互孤立的。彼此间存在着一定关联，一个可以是另一个原则的加强或是基础。违反其中的某一个，也可能同时违反了其余的原则。因此应该把这些原则融会贯通，牢记在心。

3.12.1　单一职责原则

所谓单一职责（Single Responsibility Principle），就是一个设计元素只做一件事。现实中也是如此，如果要你专心做一件事情，任何人都有信心可以做得很出色。

“单一职责”就是要在设计中为每种职责设计一个类，彼此保持独立，互不干涉。

3.12.2　开闭原则

开闭原则（Open-Closed Principle，OCP）是指在进行面向对象设计（Object Oriented Design，OOD）中，设计类或其他程序单位时，应该遵循如下原则：

（1）对扩展开放（open）；

（2）对修改关闭（closed）。

扩展开放的含义是：若某模块的功能是可扩展的，则该模块是扩展开放的。软件系统功能上的可扩展性要求模块是扩展开放的。修改关闭的含义是：某模块被其他模块调用，如果该模块的源代码不允许修改，则该模块是修改关闭的。

开闭原则是判断面向对象设计是否正确的最基本的原理之一。根据开闭原则，在设计一个软件系统模块的时候，应该可以在不修改原有模块（修改关闭）的基础上，能扩展其功能。软件系统功能上的稳定性、持续性要求是修改关闭的。这也是系统设计需要遵循开闭原则的原因，具体如下：

（1）稳定性。开闭原则要求扩展功能不修改原来的代码，这可以让软件系统在变化中保持稳定。

（2）扩展性。开闭原则要求对扩展开放，通过扩展提供新的或改变原有的功能，让软件系统具有灵活的可扩展性。

遵循开闭原则的系统设计，能让软件系统可复用，并且易于维护。

1. 开闭原则的实现方法

为了满足开闭原则，应该对软件系统中不变的部分加以抽象，方法如下：

（1）可以把那些不变的部分抽象成接口，而接口是稳定的、不变的，这些不变的接口可以应对未来的扩展。

（2）接口的最小功能设计原则。根据这个原则，原有的接口或者可以应对未来的扩展；或者可以通过定义新的接口来实现不足的部分。

（3）模块之间的调用通过抽象接口进行，这样即使实现层发生变化，也无需修改调用方的代码。

接口可以被复用，但接口的实现却不一定能被复用。接口是稳定的、关闭的，但接口的实现是可变的、开放的。可以通过对接口的不同实现以及类的继承行为等为系统增加新的或改变系统原来的功能，实现软件系统的柔性扩展。简单地说，软件系统是否有良好的接口设计是判断软件系统是否满足开闭原则的一种重要的判断基准。现在多把开闭原则等同于面向接口的软件设计。

2. 开闭原则的相对性

软件系统的构建是一个需要不断重构的过程，在这个过程中，对模块的功能进行抽象，以及模块与模块间的关系，都不会从一开始就非常清晰明了，所以构建百分之百满足开闭原则的软件系统是相当困难的，这就是开闭原则的相对性。但在设计过程中，通过对模块功能的抽象（接口定义），模块之间关系的抽象（通过接口调用），抽象与实现的分离（面向接口的程序设计）等，可以尽量接近满足开闭原则。

3.12.3 里氏代换原则

里氏代换原则（Liskov Substitution Principle, LSP）是指：对于具有继承关系的两个非抽象类，子类型必须能够替换它的基类型。对于这个原则，通常的理解是，在一个软件系统中，子类应该可以替换任何基类能够出现的地方，并且经过替换以后，代码还能正常工作。

里氏代换原则目的就是要保证继承关系的正确性。在实际的项目中，大多数情况下按照“is-a”去设计继承关系是没有问题的，只有极少的情况下，如果一个继承类的对象可能会在基类出现的地方出现运行错误，这时该子类不应该从该基类继承，或者说，应该重新设计它们之间的关系。

“正方形不是长方形”是一个理解里氏代换原则的经典案例。在数学概念上，正方形是长方形，它是一个长宽相等的长方形。所以，假如开发一个与几何图形相关的软件系统，让正方形类继承长方形类是很自然的事情。现在，我们选取该系统的一个代码片段进行分析：

```
class Rectangle {
   double length;
   double width;
   public double getLength() {
```

```
            return length;
        }
        public void setLength(double height) {
            this.length = length;
        }
        public double getWidth() {
            return width;
        }
        public void setWidth(double width) {
            this.width = width;
        }
    }

    class Square extends Rectangle {
        public void setWidth(double width) {
            super.setLength(width);
            super.setWidth(width);
        }
        public void setLength(double length) {
            super.setLength(length);
            super.setWidth(length);
        }
    }
```

下面的类 TestRectangle 是软件系统中的一个测试类，它有一个 resize 方法要用到基类 Rectangle，resize 方法的功能是模拟长方形宽度逐步增长的效果：

```
    class TestRectangle {
        public void resize(Rectangle rect) {
            while( rect.getWidth() <= rect.getLength()  ) {
                objRect.setWidth(  rect.getWidth () + 1 );
            }
        }
    }
```

运行一下这段代码就会发现，如果把一个长方形对象作为参数传入 resize 方法，就会看到长方形宽度逐渐增长的效果，当宽度大于长度,代码就会停止，这种行为的结果符合我们的预期；如果再把一个正方形对象作为参数传入 resize 方法后，就会看到正方形的宽度和长度都在不断增长，代码会一直运行下去，陷入死循环，直至系统产生溢出错误。所以，长方形是适合这段代码的，正方形不适合。

由此得出结论：在 resize 方法中，Rectangle 类型的参数是不能被 Square 类型的参数所代替，如果进行了替换就得不到预期结果。因此，Square 类和 Rectangle 类之间的继承关系违反了里氏代换原则，它们之间的继承关系不成立，正方形不是长方形。

那么如果两个类之间的继承关系违反了 LSP 原则，如何修改呢？假设 B 是子类，A 是基类，那么根据具体的情况可以在下面的两种重构方案中选择一种。

（1）创建一个新的抽象类 C，作为两个类的超类，将 A、B 的共同行为移动到 C 中来解决问题。

（2）从 B 到 A 的继承关系改为委派关系。

例如我们用第一种方案重构正方形和长方形的关系，首先构造一个抽象的四边形类，把

长方形和正方形共同的行为放到这个四边形类里面，让长方形和正方形都是它的子类，问题就解决了。对于长方形和正方形，取 width 和 height 是它们共同的行为，但是给 width 和 height 赋值，两者行为不同，因此，这个抽象的四边形的类只实现了取值方法，没有实现赋值方法。代码如下所示：

```
public abstract class Quadrangle {
   double width, length;
   public double getWidth(){
      return width;
   }
   public double getLength(){
      return length;
   }
   public void setWidth(double w);
   public void setLength(double len);
}

public class Rectangle extends Quadrangle {
   public void setWidth(double w){
      width=w;
   }

   public void setLength(double len) {
      length=len;
   }
}

public class Square extends Quadrangle {
   public void setWidth(double w) {
      width=w;
      length=w;
   }

   public void setLength(double len) {
      setWidth(len);
   }
}
```

3.12.4　依赖倒转原则

传统的过程化程序的设计方法倾向于使高层模块依赖于低层次的模块，抽象层次依赖于具体层次。倒转原则就是要把这个依赖关系倒转过来，这就是依赖倒转原则的由来。

在面向对象的系统中，两个类之间通常是三种依赖关系。

（1）零耦合：两个类之间没有耦合关系。

（2）具体耦合：具体耦合关系发生在两个具体的类之间，经由一个类引用另一个类造成。

（3）抽象耦合：抽象耦合发生在一个具体类和一个抽象之间或者 Java 接口之间，使两个必须发生关系的类之间存在最大的灵活性。

通常我们对依赖倒转原则的表述为：抽象不应当依赖于具体实现，具体实现应当依赖于抽象。另一种表述为：针对接口编程，不要针对实现编程。也就是说我们应当使用 Java 接口

和抽象 Java 类进行变量的类型声明、方法的返回类型说明，以及数据类型的转换等。要保证做到这一点，一个具体的 Java 类应当只实现 Java 接口和抽象 Java 类中声明过的方法，而不应当给出多余的方法。

引用对象的抽象类型：在很多情况下，一个 Java 程序需要引用一个对象，这个时候，如果这个对象有一个抽象类型的话，应当使用这个抽象类型作为变量的静态类型，这个就是针对接口编程的含义。比如说，Shape 代表抽象，Circle 代表具体，那么 Circle 应依赖于 Shape，而 Shape 不应当依赖于 Circle，假设 Shape 是 Java 接口或者 Java 抽象类，Circle 是一个具体类，X 是一个变量，那么声明变量应该是：Shape X＝new Circle()，而不应当是 Circle X＝new Circle()。

只要一个被引用的对象存在抽象类型，就应当在任何引用此对象的地方使用抽象类型，包括参量的类型声明、方法的返回类型声明、属性变量的类型声明等。

联合使用 Java 接口和 Java 抽象类：由于抽象类具有提供缺省实现的优点，而接口具有其他的优点，所以联合使用两者是很好的办法。首先，声明类型的工作是由接口承担，但是同时给出的还有一个抽象类，为这个接口提供一个缺省实现。其他同属于这个抽象类型的具体类可以选择实现这个接口，也可以选择继承自这个抽象类。如果需要向接口加入一个方法的话，只要同时向这个抽象类加入这个方法的具体实现就可以了，因为继承自这个抽象类的子类都会从这个抽象类得到这个具体方法，如 Action 接口和 AbstractAction 抽象类。

依赖倒转原则虽然很强大，但是不容易实现，因为依赖关系倒转的缘故，对象的创建可能会使用对象工厂（一种设计模式），以避免对具体类的使用。此外，依赖倒转原则假定所有的具体类都是会变化的，但这也不总是正确的，如果一个具体类是稳定的、不会发生变化的，那么使用这个具体类的客户端完全可以依赖于这个具体类型。

3.12.5 接口隔离原则

接口隔离原则（Interface Segregation Principle，ISP）：使用多个专门的小接口比使用一个大的总接口要好。也就是说，一个类对另外一个类的依赖性应当是建立在最小的接口上的。

这里的“接口”往往有两种不同的含义：一种是指一个类型所具有的方法特征的集合，仅仅是一种逻辑上的抽象；另一种是指某种语言具体的“接口”定义，有严格的定义和结构。比如 Java 语言里面的 interface 结构。对于这两种不同的含义，ISP 的表达方式以及含义都有所不同。

当我们把“接口”理解成一个类所提供的所有方法的特征集合的时候，这就是一种逻辑上的概念。接口的划分就直接带来类的划分。这里可以把接口理解成角色，一个接口就只是代表一个角色，每个角色都有它特定的一个接口，这里的这个原则可以叫做“角色隔离原则”。

如果把“接口”理解成狭义特定语言的接口，那么 ISP 表达的意思是说，对不同的客户端，同一个角色提供宽窄不同的接口，也就是定制服务，个性化服务。就是仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来。

在面向对象设计（OOD）时，一个重要的工作就是恰当的划分角色和角色对应的接口。将没有关系的接口合并在一起，是对角色和接口的污染。如果将一些看上去差不多的接口合并，并认为这是一种代码优化，这是错误的。不同的角色应该交给不同的接口，而不能都交给一个接口。

对于定制服务，这样做最大的好处就是系统的可维护性。向客户端提供接口是一种承诺，发布接口后是不能改变的，因此不必要的承诺就不要做出，承诺越少越好。

3.12.6 合成复用原则

合成复用原则也称为合成、聚合复用原则，就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部份，新的对象通过向这些对象的委派达到复用已有功能的目的。原则是：要尽量使用合成、聚合，尽量不要使用继承。

Has-a：代表一个类是另外一个类的一个组成部分或角色。Is-a：代表一个类是另外一个类的一种。如果两个类是“Has-a”关系那么应使用合成/聚合，如果是“Is-a”关系那么可使用继承。

3.12.7 迪米特法则

又叫最少知识原则、不要和“陌生人”讲话原则，要求一个类（软件实体）应该尽可能少的和其他类（软件实体）发生作用。不要向间接对象（陌生人）发送消息（讲话）。应该只向以下直接对象（朋友）发送消息：

（1）this 对象（自身）。

（2）方法的参数对象。

（3）this 属性直接引用的对象。

（4）作为 this 属性集合中的元素对象。

（5）在方法中创建的对象。

其意图是避免客户与间接对象和对象之间的对象连接产生耦合。利用一个“中间人”是“迪米特法则”解决问题的办法。迪米特法则的主要用意是控制信息的过载，在将其运用到系统设计中应注意以下几点：

（1）在类的划分上，应当创建有弱耦合的类。类之间的耦合越弱，就越有利于复用。

（2）在类的结构设计上，每一个类都应当尽量降低成员的访问权限。一个类不应当 public 自己的属性，而应当提供取值和赋值的方法让外界间接访问自己的属性。

（3）在类的设计上，只要有可能，一个类应当设计成不变类。

（4）在对其他对象的引用上，一个类对其他对象的引用应该降到最低。

（5）限制局部变量的有效范围，只有当需要一个变量的时候才声明它，这样可以有效地限制局部变量的有效范围。

本章小结

面向对象编程对于初学者是较难的，其思想、概念往往较难领会理解，初学者往往只从计算机这个角度来考虑，如果多联系一下我们的物理世界、我们身边的事物，那么对学习面向对象编程来说就容易多了。本章采用取类比象的方法介绍了面向对象编程的思想、基本特征以及它们在 Java 中的表达，为初学者学习面向对象提供了一条较好的道路，同学们在学习本章的时候，要多思考，多联系实际，多进行上机实训。深入理解面向对象，既要理解普遍意义上的面向对象编程，又要知道这些思想、概念、方法在 Java 上的具体表达应用。本章最后给出的面向对象编程的基本原则是较难的，是对如何运用面向对象进行编程的经验总结和指导，当开发经验积累到一定程度时，自然就彻底理解这些原则了。

面向对象的三个基本特征是自然界事物的法则，面向对象的计算机编程仅仅是该法则的

一种应用，读者完全可以把这三个基本特征作为方法应用到自己的学习、工作、生活中，使自己终生受益。通过不断地学习、实践来封装自己，保护好自己避免受到伤害、增强自己的独立性就是尽量减少对别人的依赖，自己能够独当一面；学会继承的方法，使自己少走弯路，提高自身发展的速度和质量；多态性就是在不同的场合下自己的言行和做事的方法要合理的应变，以使自己与周围的环境相和谐。若做到这三点中的任意一点，未来的成功就会属于你。

习题 3

1．根据教材第一节的内容，谈谈你对人与计算机关系的认识。

2．请简述面向对象的基本特征及其带来的好处。

3．请从面向对象的基本特征分析下面设计的类存在哪些问题：

```
1   public class School {
2     private int index;
3     private String [] studentNames;
4
5     public School() {
6       studentNames = new String[10000];
7       index=0;
8     }
9
10    public void addStudent(String stuName){
11      if(index < studentNames.length－1){
12         studentNames[index++]=stuName;
13      }
14    }
15  }
```

4．电子商务在现代商务中占有重要地位，请同学们根据下面的描述设计一个简单的购物系统：这个系统有商品和购物车，每件商品有名称和价格，每个购物车用以记录将要购买的不同的商品名称及其数量。最后要能从外部使用购物车并获得购物车中商品的总价格。

5．仔细观察如下代码，分析其不妥之处：

```
1   public class Automobile {//汽车类
2      int width;
3      int length;
4      public int area(){
5         return width * length;
6      }
7
8      public Automobile(int w, int len){
9         width=w;
10        length=len;
11     }
12  }
```

```
13
14  public class Window extends Automobile{//汽车窗口类
15      public Window(int w, int len){
16          super(w,len);
17      }
18  }
```

6．设计题目：一个农场，专门种植销售各类水果，在这个系统中需要描述下列水果，葡萄（Grape），草莓（Strawberry），苹果（Apple）。水果与其他的植物有很大的不同，水果最终是可以采摘食用的。那么一个自然的做法就是建立一个各种水果都适用的接口，以便与农场里的其他植物区分开。水果接口规定出所有的水果都必须实现的接口，包括任何水果必须具备的方法：种植 plant()、生长 grow()、收获 harvest()。Apple 类是水果中的一种，因此它实现了水果接口所声明的所有方法。另外，由于苹果是多年生植物，因此多出一个 treeAge 性质，描述苹果树的树龄。Grape 类是水果类的一种，也实现 Fruit 接口中所声明的所有方法。但由于葡萄分为有籽和无籽的两种，因此比通常的水果多出一个 seedless 性质。Strawberry 类也是水果的一种，也实现了 Fruit 接口。农场的园丁也是系统的一部分，自然要由一个合适的类来代表。这个类就是 FruitGardener，它会根据农场老板的要求，使用 factory()方法创建出不同的水果对象，比如 Apple、Grape、Strawberry 的实例。而如果接到不合法的要求，会提示错误。农场的市场调查员也是系统的一部分，也需要一个类代表，这个类是 MarketInquirer，它通过 inquire()方法调查今年市场上哪一种水果热销。农场的老板也是系统的一部分，仍需要一个类来代表，这个类是 FruitBoss，他会根据市场调查员的反馈信息，通知农场的园丁今年种植哪种水果。

要求：请根据上述系统需求，用 Java 语言采用面向对象方法设计这个农场系统。

7．物理世界是个大世界、大系统，在这个大系统中包含了许多的小世界、小系统，而这些小系统中又包含了更小的系统……，例如宇宙中包含了银河系统，银河系统又包含了太阳系统，太阳系统又包含了地球系统，地球系统又包含了……。而世界由万物构成，大的系统可以看作是事物或对象，小的系统也可以看作是事物或对象，更小的系统同样仍然可以看作是事物或对象。所以世界由万物构成，不是从大小方面来说的（例如有些观点认为世界是由分子、原子构成的），而是从属性方面来来说的，即只要具备动（功能方法）与静（成员变量属性）两个方面的属性，具有三个基本特征就应该看作是一个对象。计算机是一个模拟系统，它能够模拟物理世界，因此计算机是物理世界的一个缩影，这正是面向对象程序设计中采用取类比象原则的依据。请同学们结合着这段描述以及本章内容，谈谈你对物理世界的认识，计算机与物理世界的相通之处，以及面向对象编程中的对象如何来理解、如何来面向？

8．简述引用类型的类型转换规则。

9．描述一下类实例对象的创建过程。

10．总结一下内部类有几种，如何定义以及实例化它们？

11．谈谈对象之间的关系。

12．谈谈你对面向对象程序设计基本原则的理解。