第 3 章

Java 面向对象编程机制

计算机程序告诉计算机应该做什么。计算机执行的任何操作都是由程序控制的。程序设计是将计算机要执行的操作或者计算机要解决的问题转变成程序的过程。

程序设计的过程主要分为面向过程编程和面向对象编程两种风格，本书讲解的 Java 语言编程属于面向对象编程技术。

本章要点

- 面向对象编程的基本思想。
- 类与对象。
- 抽象类与接口。
- Java 简单程序。

学习目标

- 理解面向对象编程的基本思想。
- 掌握 Java 运行环境的安装和环境变量的配置。
- 掌握类与对象的概念及创建方式。
- 掌握接口与抽象类的声明及创建方法。
- 掌握 Java 程序的分类与运行步骤。

3.1 面向对象编程的基本思想

程序员可以用各种程序语言编写指令，有些语言是计算机直接能理解的，有些则需经过中间的“翻译”步骤。无论如何，程序设计的核心是数据的处理。基于数据处理方式的不同，可将程序设计过程分为面向过程的程序设计和面向对象的程序设计。其中，传统的面向过程的编程设计思路是先设计一组函数，用来解决一个问题，然后再确定函数中需要处理数据的相应存储位置，即“算法+数据结构=程序”。而面向对象编程(Object Oriented Programming，OOP)的思路恰好相反，是先确定要处理的数据，然后再设计处理数据的算法，最后将数据和算法封装在一起，构成类与对象。

面向对象技术是一种设计和构造软件的全新技术，它使计算机解决问题的方式更符合人类的思维方式，更能直接地描述客观世界，通过增加代码的可重用性、可扩充性和程序自动生成功能来提高编程效率，并且能够大大减少软件维护的开销，所以被越来越多的软件设计人员接受。

面向对象技术是一种新的软件技术，从 20 世纪 60 年代提出面向对象的概念，到现在已发展成为一种比较成熟的编程思想，并且逐步成为目前软件开发领域的主流技术。同时，它并不局限于程序设计方面，也已经成为软件开发领域的一种方法论。它对信息科学、软件工程、人工智能和认知科学等都产生了重大影响，尤其在计算机科学与技术的各个方面，影响深远。通过面向对象技术，可以将客观世界直接映射到面向对象求解空间，从而为软件设计和系统开发带来革命性影响。

在面向对象程序设计方法出现之前，程序员用面向过程的方法开发程序。面向过程的方法把密切相关、相互依赖的数据和对数据的操作相互分离，这种实质上的依赖与形式上

的分离使得大型程序不但难于编写，而且难于调试和修改。在多人合作中，程序员之间很难读懂对方的代码，更谈不上代码的重用。由于现代应用程序规模越来越大，对代码的可重用性与易维护性的要求也相应提高。面向对象技术便应运而生了。

面向对象技术是一种以对象为基础，以事件或消息来驱动对象执行处理的程序设计技术。它以数据为中心，而不是以功能为中心来描述系统，数据相对于功能而言，具有更强的稳定性。它将数据和对数据的操作封装在一起，作为一个整体来处理，采用数据抽象和信息隐蔽技术，将这个整体抽象成一种新的数据类型，称为类，并且考虑不同类之间的联系和类的重用性。类的集成度越高，就越适合大型应用程序的开发。另一方面，面向对象程序的控制流程由运行时各种事件的实际发生来触发，而不再由预定顺序来决定，这更符合实际。事件驱动程序的执行围绕消息的产生与处理，靠消息循环机制来实现。更重要的是，利用不断扩充的框架产品 MFC(Microsoft Foundation Classes)，在实际编程时可以采用搭积木的方式来组织程序，站在“巨人”肩上实现自己的愿望。面向对象的程序设计方法使得程序结构清晰、简单，提高了代码的重用性，有效地减少了程序的维护量，提高了软件的开发效率。

例如，可以用面向对象技术来解决学生管理方面的问题。此时重点应该放在学生上，要了解在管理工作中学生的主要属性、要对学生做些什么操作等，并且把它们作为一个整体来对待，形成一个类，称为学生类。作为学生类的实例，可以建立许多具体的学生，而每一个具体的学生就是学生类的一个对象。学生类中的数据和操作可以提供给相应的应用程序共享，还可以在学生类的基础上派生出大学生类、中学生类或小学生类等，实现代码的高度重用。

在结构上，面向对象程序与面向过程程序有很大不同。面向对象程序由类的定义和类的使用两部分组成，在主程序中定义各对象，并规定它们之间传递消息的规律，程序中的一切操作都是通过向对象发送消息来实现的，对象接到消息后，启动消息处理函数，完成相应的操作。

类与对象是面向对象程序设计中最基本且最重要的两个概念，在 3.2 节中，将详细介绍类与对象。

下面使用三种例子，来比较两种方法学的区别，各例程结果基本相同。

【例 3.1】用面向过程的 C 语言实现两数相加：

```
#include <stdio.h>
int sum(int x, int y) {
    return x+y;
};
void main() {
    int a=3,b=4,c=5,d=6;
    printf("a+b=%d\n", sum(a,b));
    printf("c+d=%d\n", sum(c,d));
}
```

【例 3.2】用过程化的面向对象 C++语言实现：

```
#include <iostream.h>
class Calculate {
```

```
public:
    int sum(int x, int y) {
        return x+y;
    }
};
void main() {
    int a=3,b=4,c=5,d=6;
    Calculate obj;
    cout<<obj.sum(a,b)<<endl;
    cout<<obj.sum(c,d)<<endl;
}
```

【例 3.3】用纯面向对象的 Java 语言实现：

```
class Calculate {
    int sum(int x, int y) {
        return x+y;
    }
    public static void main(String[] args) {
        Calculate obj = new Calculate();
        int a=3, b=4, c=5, d=6;
        System.out.println(obj.sum(a,b));
        System.out.println(obj.sum(c,d));
    }
}
```

3.2 类 与 对 象

3.2.1 类与对象

与人们认识客观世界的规律一样，面向对象技术认为，客观世界是由各种各样的对象组成的，每种对象都有各自的内部状态和运动规律，不同对象间的相互作用和联系就构成了各种不同的系统，构成了客观世界。在面向对象程序中，客观世界被描绘成一系列完全自治、封装的对象，这些对象通过外部接口访问其他对象。可见，对象是组成一个系统的基本逻辑单元，是一个有组织形式的含有信息的实体。而类是创建对象的样板，在整体上代表一组对象，编程时设计类而不是设计对象，这样可以避免重复编码。因为类只需要编码一次，就可以创建本类的所有对象。

对象(Object)由属性(Attribute)和行为(Action)两部分组成。对象只有在具有属性和行为的情况下才有意义，属性是用来描述对象静态特征的一个数据项，行为是用来描述对象动态特征的一个操作。对象是包含客观事物特征的抽象实体，是属性和行为的封装体，在程序设计领域，可以用“对象=数据+作用于这些数据上的操作”这一公式来表达。

类(Class)是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分，类是对象集合的再抽象。

类与对象的关系如同一个模具与用这个模具铸造出来的铸件之间的关系。类给出了属于该类的全部对象的抽象定义，而对象则是符合这种定义的一个实体。所以，一个对象又

称作类的一个实例(Instance)。

在面向对象程序设计中，类的确定与划分非常重要，是软件开发中关键的一步，划分的结果会直接影响到软件系统的质量。如果划分得当，既有利于程序进行扩充，又可以提高代码的可重用性。因此，在解决实际问题时，需要正确地进行分“类”。理解一个类究竟表示哪一组对象，如何把实际问题中的事物汇聚成一个个“类”，而不是一组数据。这是面向对象程序设计中的一个难点。类的确定和划分并没有统一的标准和固定的方法，基本上依赖设计人员的经验、技巧以及对实际问题的把握。但有一个基本原则：寻求一个大系统中事物的共性，将具有共性的系统成分确定为一个类。确定类的步骤包括：第一步，要判断该事物是否有多个实例，如果有，则它是一个类；第二步，要判断类的实例中有没有绝对的不同点，如果没有，则它是一个类。

另外，还要知道什么事物不能被划分为类。不能把一组函数组合在一起构成类，也就是说，不能把一个面向过程的模块直接变成类。

类和对象是 OOP 中最基本的两个概念，其实它们是比较容易理解的，简而言之，类是对象的模板，对象是类的具体实现。

对象创建即是实例化，实例化是将类的属性设定为确定值的过程，是“一般”到“具体”的过程；类的定义即是抽象，抽象是从特定的实例中抽取共同的性质以形成一般化概念的过程，是“具体”到“一般”的过程。

1．类的定义

关于“类”，具有四个方面的含义：

- 类是具有共同属性和行为的对象的抽象。
- 类可以定义为数据和方法的集合。
- 类也称为模板，因为它们提供了对象的基本框架。
- 类是对象的类型，在语句中相当于数据类型使用。

Java 中，类定义的一般格式为：

```
[类修饰符] class <类名称> [extends <父类名>] [implements <接口名>] {
    [static {  }] //静态块
    [成员修饰符]  数据类型 成员变量 1;
    [成员修饰符]  数据类型 成员变量 2;
    ……  //其他成员变量
    [成员修饰符] 返回值类型 成员方法 1(参数列表)
    {    }
    [成员修饰符] 返回值类型 成员方法 2(参数列表)
    {    }
    ……//其他成员方法
}
```

【例 3.4】举例说明类的定义方法，定义一个描述圆的类，并能根据给定的半径计算和显示圆的面积。代码如下：

```
public class Circle { //类开始
    private float fRadius;     //成员变量
    final float PI = 3.14f;      //定义常变量 PI
```

```
    void setRadius(float fR) {    //设置圆半径
        fRadius = fR;
    }
    void showArea() {    //显示圆面积
        System.out.println("The area of circle is " + fRadius*fRadius*PI);
    }
    public static void main(String[ ]args) {   //主方法，即程序入口
        Circle circle = new Circle();   //创建圆类的对象
        circle.setRadius(5);   //引用对象方法，设置圆的半径
        circle.showArea();     //引用对象方法，显示圆的面积
    }
}
```

运行结果：

```
The area of circle is 78.5
```

在 Java 源程序代码中，除类模块之外，不包含任何游离于类模块之外的成分，如文件级变量和函数等。

2．对象的创建

关于“对象”，具有两方面的含义：

- 在现实世界中，对象是客观世界中的一个实体。
- 在计算机世界中，对象是一个可标识的存储区域对象，相当于变量。

Java 中，对象创建的一般格式为：

```
<类名称>  <对象名称> = new <类名称>(<参数列表>);
```

例如，上面例子中，创建圆 Circle 的对象 circle 的代码为：

```
Circle circle = new Circle()。
```

在 Java 语言中，规定类中的任何资源都是封装在类内部的，只有类内部成员才有权力调用和使用，其他外部类或者函数，包括主函数 Main，均无权直接访问，因此有了类相关对象的机制，只有指向某类的对象才有权力调用类中的成员变量和成员函数，其调用格式如下：

```
对象名.成员变量
对象名.成员方法()
```

例如，上面例子中的两句代码：

```
circle.setRadius(5);     //引用对象方法，设置圆的半径
circle.showArea();     //引用对象方法，显示圆的面积
```

3．成员变量

成员变量声明或定义的完整格式为：

```
[成员访问修饰符]  [成员存储类型修饰符] 数据类型 成员变量[ = 初值];
```

格式说明如下。

(1) 类的成员变量在使用前必须加以声明，除了声明变量的数据类型外，还需要说明变量的访问属性和存储方式。访问修饰符包括 public、protected、private 和默认(即不带访问修饰符)，存储类型修饰符包括 static、final、volatile、transient。Java 中允许为成员变量赋初值。

(2) 成员变量根据在内存中的存储方式和作用范围，可以分为类变量和实例变量。

(3) 访问成员变量的格式为“对象.成员变量”，如果考虑 static 关键字，那么访问格式可细化为：

```
对象.实例成员变量
```

或者：

```
类名.静态成员变量
```

【例 3.5】使用成员变量的例子：

```
public class Circle {  //类开始
    private float fRadius;  //成员变量
    void showRadius() {  //显示半径
        //int fRadius;  //局部变量必须初始化
        System.out.println("The Radius of circle is " + fRadius);
          //输出半径值
    }
    public static void main(String[]args) {  //主方法，即程序入口
        Circle circle = new Circle();        //创建圆类的对象
        circle.showRadius();     //引用对象方法，显示圆的半径
    }
}
```

运行结果：

```
The Radius of circle is 0.0
```

如果去掉代码第 4 行的注释，运行结果为：

```
variable fRadius might not have been initialized(提示变量 fRadius 尚未初始化)
```

4．成员方法

成员方法声明的完整格式为：

```
[成员访问修饰符][成员存储类型修饰符] 数据类型 成员方法([参数列表])
  [throws Exception];
```

成员方法定义的完整格式为：

```
[成员访问修饰符][成员存储类型修饰符] 数据类型 成员方法([参数列表])
  [throws Exception] {
    [<类型> <局部变量>;]
    方法体语句;
}
```

成员方法格式的说明如下。

(1) 类的成员方法的声明或定义前，除了声明方法的返回值数据类型外，还需要说明方法的访问属性和存储方式。

- 访问修饰符包括 public、protected、private 和缺省。
- 存储类型修饰符包括 static、final、abstract、native、synchronized。

(2) 如果类体中一个方法只有声明，没有定义(即没有方法体)，则此方法是抽象的，且类体和方法体前都须加 abstract。

(3) 访问成员方法的格式为“对象.成员方法”，如果考虑 static 关键字，那么访问格式可细化为：

```
对象.实例成员方法
```

或者：

```
类名.静态成员方法
```

(4) 方法按返回值，可分为无返回值的方法和有返回值的方法。

【例 3.6】方法返回值的作用：

```
public class Circle {  //类开始
    private float fRadius;   //成员变量
    final float PI = 3.14f;     //定义常量 PI
    void setRadius(float fR) {  //设置圆的半径
        fRadius = fR;
    }
    float getArea() {   //这里增加了带返回值的方法 getArea()
        return fRadius*fRadius*PI;
    }
    void showArea() {  //显示圆的面积
        //输出 getArea()值
        System.out.println("The area of circle is " + getArea());
    }
    public static void main(String[] args) { //主方法，即程序入口
        Circle circle = new Circle();    //创建圆类的对象
        circle.setRadius(5); //引用对象方法，设置圆的半径
        circle.showArea();   //引用对象方法，显示圆的面积
    }
}
```

(5) 方法的参数列表：在声明方法中的参数时，需要说明参数的类型和个数。参数之间用逗号隔开，参数的数据类型可以是 Java 认可的任何数据类型。参数名称在它的作用范围内是唯一的，即同一个方法中的参数名称不能相同。

【例 3.7】对象作为方法的参数：

```
class Complex {
    int real, virtual;
    public Complex(int r, int v) {
        real = r;
        virtual = v;
    }
```

```
    void showValue() {
        System.out.println("复数值为: " + real + "." + virtual);
    }
    static Complex addComplex(Complex c1, Complex c2) {
        Complex c = new Complex(0, 0);
        c.real = c1.real+c2.real;
        c.virtual = c1.virtual + c2.virtual;
        return c;
    }
}
public class ObjectParaDemo {

    public static void main(String[] args) {
        Complex c, c1, c2;
        c1 = new Complex(2,3);
        c2 = new Complex(4,5);
        c = Complex.addComplex(c1,c2);
        c.showValue();
    }
}
```

【例 3.8】通过引用改变对象的值：

```
class Timer {
    int minute, second;
    public Timer(int m, int s) {
        minute = m;
        second = s;
    }
    void showTime() {
        System.out.println("现在的时间是: " + minute + "分" + second + "秒");
    }
    static void swapTime(Timer t1, Timer t2) {
        Timer t = t1; //定义了局部变量 t，利用 t 交换 t1 和 t2 的值
        t1=t2; t2=t;
    }
}
public class ReferenceDemo {
    public static void main(String[] args) {
        Timer t1 = new Timer(9,10);
        Timer t2 = new Timer(11,12);
        t1.showTime();
        t2.showTime();
        System.out.println("使用 swapTime 方法交换 Timer 实例后: ");
        Timer.swapTime(t1,t2);
        t1.showTime();
        t2.showTime();
    }
}
```

【例 3.9】方法的重载：

```
public class Sum {
    static int add(int x, int y) {
        return x+y;
    }
    static int add(int x, int y, int z) {
        return x+y+z;
    }
    static float add(float x, float y) {
        return x+y;
    }
    public static void main(String[] args) {
        System.out.println(add(2,3));
        System.out.println(add(2,3,4));
        System.out.println(add(2.1f,3.2f));
    }
}
```

5．构造方法

构造方法是一种特殊的方法，用来创建类的实例，用于给对象进行初始化，它具有针对性，是一种特殊的成员函数。一个类在定义时，如果没有定义过构造函数，那么该类中会自动生成一个空参数的构造函数，完成初始化。如果在类中自定义了构造函数，那么默认的构造函数就没有了。

一个类中，可以有多个构造函数，因为它们的函数名称都相同，所以只能通过参数列表来区分。所以，一个类中如果出现多个构造函数，它们的存在是以重载体现的。

声明构造方法时，可以附加访问修饰符，但没有返回值，不能指定返回类型。

构造方法名必须与类同名。调用构造方法创建实例时，用 new 运算符加构造方法名的形式创建，下面通过几个例子说明构造方法的使用。

【例 3.10】使用普通成员方法来为对象的成员属性赋值：

```
public class Timer {
    int hour, minute, second;
    void setTime(int h, int m, int s) { //用来赋值的成员方法
        hour = h;
        minute = m;
        second = s;
    }
    void showTime() {
        System.out.println(
          "现在的时间是: " + hour + ":" + minute + ":" + second);
    }
    public static void main(String[] args) {
        Timer t1 = new Timer();
        t1.setTime(10, 11, 12);
        t1.showTime();
    }
}
```

【例 3.11】使用构造方法的例子：

```
public class Timer {
    int hour, minute, second;
    public Timer(int h, int m, int s) {   //自定义的构造方法
        hour = h;
        minute = m;
        second = s;
    }
    void showTime() {
        System.out.println(
          "现在时间是: " + hour + ":" + minute + ":" + second);
    }
    public static void main(String[] args) {
        Timer t1 = new Timer(10, 11, 12);
        t1.showTime();
    }
}
```

【例 3.12】使用默认构造方法的例子：

```
public class Timer {
    int hour, minute, second;
    //public Timer() {}      //默认构造方法，此方法可以省略
    public static void main(String[] args) {
        Timer t1 = new Timer();
    }
}
```

【例 3.13】默认构造方法与自定义构造方法的例子：

```
public class Timer {
    int hour, minute, second;
    public Timer() {         //默认构造方法，此例中不能省略
        //下面的赋值语句不写也可，创建实例时，会按照成员变量的默认值规则赋值
        hour = 0;
        minute = 0;
        second = 0;
    }
    public Timer(int h, int m, int s) {    //自定义的构造方法
        hour = h;
        minute = m;
        second = s;
    }
    void showTime() {
        System.out.println(
          "现在时间是: " + hour + ":" + minute + ":" + second);
    }
    public static void main(String[] args) {
        Timer t1 = new Timer(10, 11, 12);
        t1.showTime();
        Timer t2 = new Timer();         //调用了默认构造方法创建实例
```

```
        t2.showTime();
    }
}
```

6．Java 访问控制修饰符

面向对象“封装”的基本思路是，将成员属性设为 private，使用户不能直接访问；将成员方法设为 public，然后通过 public 成员方法访问 private 成员属性。封装虽然最大限度地保证了数据访问的安全，但 private 成员仅可供本类内的方法访问，限制太严格。为了能够灵活控制这些访问需求，面向对象编程提供了其他一些访问控制修饰符，来控制在类、包内外部的访问权限。

在 Java 中，访问控制共有四个等级，分别是：public、protected、private、默认。

按照访问控制权限的高低排序为：public→protected→默认→private。

功能说明如下：

- 当成员被声明为 public 时，该成员可以被程序的任何代码模块访问。
- 当成员被声明为 protected 时，该成员只可以被该类子类的程序代码模块访问。
- 当缺省，即没有访问修饰符时，该成员只可以被该类所在包的类的程序代码模块访问。
- 当成员被声明为 private 时，该成员只可以被本类程序代码模块访问。

【例 3.14】private 访问控制修饰符的作用：

```
class Access {
    private int data;  //private 属性成员只能被本类访问
    void show() {
        System.out.println("data's value is : " + data);
    }
}
public class AccessDemo {
    public static void main(String[] args) {
        Access obj = new Access();
        obj.show();
        System.out.println("object's data is :" + obj.data);
    }
}
```

程序无法编译，会提示如下错误信息：

```
AccessDemo.java: data has private access in Access
```

错误提示表明，代码中访问了一个 Access 类的私有属性 data，原因是 private 属性成员只能被本类访问，将第 2 行代码 data 成员变量的 private 修饰符去掉，即可正常运行。

【例 3.15】理解 main()方法在类中的地位：

```
public class AccessDemo {
    private int data;
    void show() {
        System.out.println("data's value is : " + data);
    }
```

```
    public static void main(String[] args) {
        AccessDemo obj = new AccessDemo();
        obj.show();
        System.out.println("object's data is:" + obj.data); //调用私有成员
    }
}
```

运行结果：

```
data's value is: 0
object's data is:0
```

虽然 data 成员变量仍然是 private，但因为 main()方法与 data 在同一类体中，可以访问。这说明了一个事实：main()方法也是类的成员方法，只不过 main()方法和类的构造方法不参与继承。

7．this 和 super 关键字

this 是对象的别名，是当前类的当前实例的引用，而 super 是当前类父类实例的引用。

(1)　this 关键字。

this 用于类的成员方法的内部，用于代替调用这个方法的实例。this 本质上是一个指针，指向操作当前方法的那个实例，这与 C++的 this 指针完全相同，但因为 Java 中没有显式指针，我们可以把它理解成引用(即隐式指针)。如果方法中的成员调用前没有操作实例名，实际上默认省略了 this。

【例 3.16】this 关键字的使用：

```
class Children {
    int age;    //age 是成员变量，属于全局变量
    void setAge(int age) {  //age 是局部变量
        this.age = age;     //此处的 this.age 代表成员变量
        //this 代表代码第 8 行的实例 children
    }
    public static void main(String args[]) {
        Children children = new Children();
        children.setAge(9); //实例 children 操作了 setAge 方法
    }
}
```

(2)　super 关键字。

super 有两种调用形式：

```
super    //代表父类的实例
super()    //代表父类的构造方法
```

【例 3.17】super 关键字的使用：

```
class Children extends Parent {
    String cName;    //子女姓名
    char cSex;       //子女性别
    Children(String pName, String cName, char cSex) {
        super(pName);        //利用 super()调用父类构造方法
```

```
        this.cName = cName;  //this.strName 表示当前类成员
        this.cSex = cSex;    //this.cSex 表示当前类成员
    }

    /*--- 定义布尔型方法，判断 this 是否为 Children 的对象的引用 ---*/
    boolean isChildren() {
        if(this instanceof Children)
            return true;
        else
            return false;
    }
    void showFamilyInfo() {
        System.out.println("父母名称：" + super.pName);//用 super 引用父类实例
        System.out.println("子女名称：" + cName);
        System.out.println("子女性别：" + cSex);
    }

    public static void main(String[] args) {
        Children children = new Children("刘强，王丽", "刘华", 'F');
        System.out.println("this 为当前类的实例吗？"+children.isChildren());
        children.showFamilyInfo();
    }
}
```

运行结果：

```
this 为当前类的实例吗？True
父母名称：刘强，王丽
子女名称：刘华
子女性别：F
```

3.2.2 面向对象技术的基本特征

面向对象技术强调在软件开发过程中面向客观世界或问题域中的事物，采用人类在认识客观世界的过程中普遍运用的思维方法，直观、自然地描述客观世界中的有关事物。

面向对象技术的基本特征主要有抽象性、封装性、继承性和多态性等。

1．抽象性

把众多的事物进行归纳、分类，是人们在认识客观世界时经常采用的思维方法，“物以类聚，人以群分”就是分类的意思，分类所依据的原则是抽象。抽象(Abstract)就是忽略事物中与当前目标无关的非本质特征，更注意与当前目标有关的本质特征。从而找出事物的共性，并把具有共性的事物划为一类，得到一个抽象的概念。例如，在设计一个学生成绩管理系统的过程中，考察学生张华这个对象时，就只关心他的班级、学号、成绩等，而忽略他的身高、体重等信息。因此，抽象性是对事物的抽象概括描述，实现了客观世界向计算机世界的转化。将客观事物抽象成对象及类，是比较难的过程，也是面向对象方法的第一步。例如，将学生抽象成对象及类的过程，如图 3-1 所示。

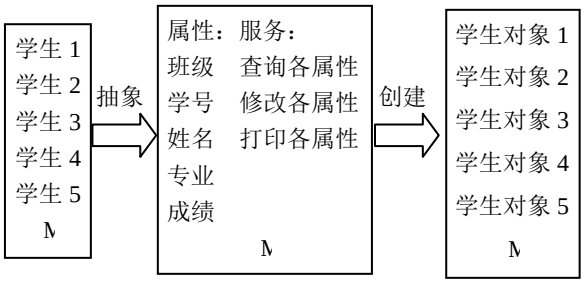

图 3-1　抽象过程

2．封装性

封装(Encapsulation)就是把对象的属性和行为结合成一个独立的单位，并尽可能隐蔽对象的内部细节。图 3-1 中的学生类也反映了封装性。

封装有两个含义：一是把对象的全部属性和行为结合在一起，形成一个不可分割的独立单位，对象的属性值(除了公有的属性值)只能由这个对象的行为来读取和修改；二是尽可能隐蔽对象的内部细节，对外形成一道屏障，与外部的联系只能通过外部接口实现。

封装的信息隐蔽作用反映了事物的相对独立性，可以只关心它对外所提供的接口，即能做什么，而不注意其内部细节，即怎么提供这些服务等。

例如，用陶瓷封装起来的一块集成电路芯片，其内部电路是不可见的，而且使用者也不关心它的内部结构，只关心芯片引脚的个数、引脚的电气参数及引脚提供的功能，利用这些引脚，使用者将各种不同的芯片连接起来，就能组装成具有一定功能的模块。

封装的结果，使对象以外的部分不能随意存取对象的内部属性，从而有效地避免了外部错误对它的影响，大大减小了查错和排错的难度。另一方面，当对象内部进行修改时，由于它只通过少量的外部接口对外提供服务，因此，同样减小了内部的修改对外部的影响。同时，如果一味地强调封装，则对象的任何属性都不允许外部直接存取，要增加许多没有其他意义、只负责读或写的行为，这为编程工作增加了负担，增加了运行开销，并且使得程序显得臃肿。为了避免这一问题，在语言的具体实现过程中，应使对象有不同程度的可见性，进而与客观世界的具体情况相符合。

封装机制将对象的使用者与设计者分开，使用者不必知道对象行为实现的细节，只需要用设计者提供的外部接口，让对象去做。封装的结果，实际上隐蔽了复杂性，并提供了代码重用性，从而降低了软件开发的难度。

3．继承性

客观事物既有共性，也有特性。如果只考虑事物的共性，而不考虑事物的特性，就不能反映出客观世界中事物之间的层次关系，不能完整、正确地对客观世界进行抽象描述。运用抽象的原则就是舍弃对象的特性，提取其共性，从而得到适合一个对象集的类。如果在这个类的基础上，再考虑抽象过程中被舍弃的一部分对象的特性，则可形成一个新的类，这个类具有前一个类的全部特征，是前一个类的子集，形成一种层次结构，即继承结构，如图 3-2 所示。

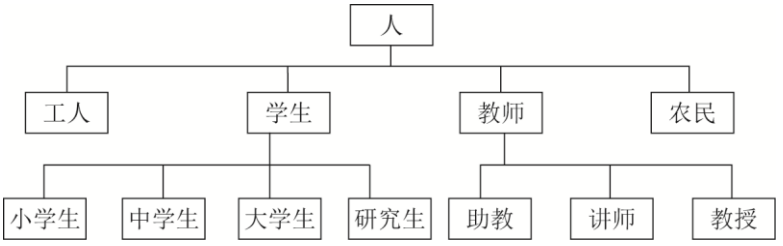

图 3-2　类的继承结构

继承(Inheritance)是一种联结类与类的层次模型。继承性是指特殊类的对象拥有其一般类的属性和行为。继承意味着“自动地拥有”，即特殊类中不必重新定义已在一般类中定义过的属性和行为，而它却自动地、隐含地拥有其一般类的属性与行为。继承允许和鼓励类的重用，提供了一种明确表述共性的方法。一个特殊类既有自己新定义的属性和行为，又有继承下来的属性和行为。尽管继承下来的属性和行为是隐式的，但无论在概念上还是在实际效果上，都是这个类的属性及行为。当这个特殊类又被它更下层的特殊类继承时，它继承来的和自己定义的属性及行为又被下一层的特殊类继承下去。因此，继承是传递的，体现了大自然中特殊与一般的关系。下面举例说明类继承机制的定义方法。

【例 3.18】类继承的作用：

```
class Children extends Parent {
    int age;     //子类自定义的成员变量
    Children(String cName, int cAge) {   //构造方法
        //super();       //默认省略了此语句
        name = cName;       //name 属性继承自父类 Parent
        age = cAge;
    }
    public static void main(String[] args) {
        Children children = new Children("王强", 10);
        children.showInfo();   //showInfo()方法继承自父类 Parent
    }
}
```

运行结果：

```
姓名：王强
```

由此可见，继承是对客观世界的直接反映，通过类的继承，能够实现对问题的深入抽象描述，反映出人类认识问题的发展过程。

在软件开发过程中，继承性实现了软件模块的可重用性、独立性，缩短了开发周期，提高了软件开发的效率，同时，使软件易于维护和修改。

4．多态性

面向对象设计借鉴了客观世界的多态性，体现在不同的对象收到相同的消息时，会产生多种不同的行为方式。例如，在一般类“几何图形”中定义了一个行为“绘图”，但并不确定执行时到底画一个什么图形。特殊类“椭圆”和“多边形”都继承了几何图形类的

绘图行为，但其功能却不同，一个是要画出一个椭圆，另一个是要画出一个多边形。这样，一个绘图消息发出后，椭圆、多边形等类的对象接收到这个消息后，各自执行不同的绘图函数，如图 3-3 所示，这就是多态性的表现。

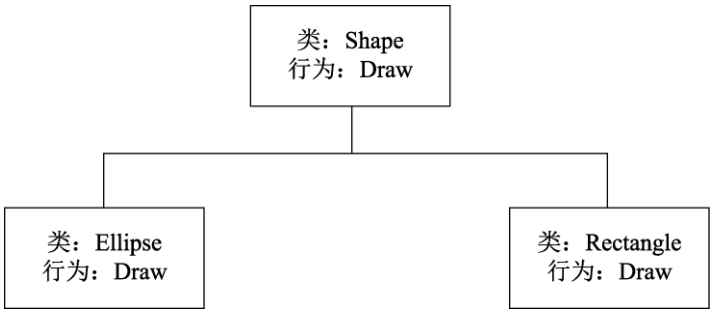

图 3-3　多态性示意

具体地说，多态性(Polymorphism)是指类中同一函数名对应多个具有相似功能的不同函数，可以使用相同的调用方式来调用这些具有不同功能的同名函数。通过类的继承，子类可以使用父类的成员变量和成员方法。但当子类重新定义了与父类同名的方法时，子类方法的功能将会覆盖父类同名方法的功能，这叫作方法“重写”。同样，当子类的成员变量与父类的成员变量同名时，在子类中将隐藏父类同名变量的值，这叫作变量覆盖。另外，还须注意下面的两条限制。

(1)　重写的方法不能比被重写的方法拥有更严格的访问权限。

(2)　重写的方法不能比被重写的方法产生更多的异常。

【例 3.19】成员方法重写：

```
class Children extends Parent {
    String name;      //子女姓名
    int age;
    Children(String cName, char cSex, int cAge) {  //构造方法
        //super();      //默认省略了此语句
        name=cName; sex=cSex; age=cAge;
    }
    void showInfo() {    //显示子类实例信息，重写了父类的 showInfo()方法
        System.out.println("孩子的姓名: " + name);
        System.out.println("孩子的性别: " + sex);
        System.out.println("孩子的年龄: " + age);
    }
    public static void main(String[] args) {
        Children children = new Children("王强", 'M', 10);
        //System.out.println("子类信息如下: ");
        children.showInfo();
    }
}
```

运行结果：

```
孩子的姓名: 王强
```

```
孩子的性别：M
孩子的年龄：10
```

继承性和多态性的结合，可以生成一系列虽类似但却独一无二的对象。由于继承性，这些对象共享许多相似的特征；由于多态性，针对相同的消息，不同对象可以有独特的表现方式，实现特性化的设计。

面向对象技术四大特征的充分运用，为提高软件开发效率起着重要的作用。面向对象技术可使程序员不必反复地编写类似的程序，通过继承机制进行特殊类化的过程，使得程序设计变成仅对特殊类与一般类的差异进行编程的过程。当高质量的代码可重复使用时，复杂性就得以降低，效率则得到提高。不断扩充的 MFC 类库和继承机制能很大程度地提高开发人员建立、扩充和维护系统的能力。面向对象技术将数据与操作封装在一起，简化了调用过程，方便了维护，并减少了程序设计过程中出错的可能性。继承性和封装性使得应用程序的修改带来的影响更加局部化，而且类中的操作是易于修改的，因为它们被放在唯一的地方。因此，采用面向对象技术进行程序设计具有开发时间短、效率高、可靠性好、所开发的程序更强壮等优点。

3.3 接口和抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是，并不是所有的类都是用来描绘对象的，也有一种类，它没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类往往用来表征我们在对问题领域进行分析、设计时得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象，我们不能把它们实例化，因此称为抽象。比如，我们要描述“水果”，它就是一个抽象，它有质量、体积等一些共性，但又缺乏特性，我们拿不出唯一一种能代表水果的东西，因为苹果、橘子都不能代表水果，此时可用抽象类来描述它，所以抽象类是不能够实例化的。

当我们用某个类来具体描述“苹果”时，这个类就可以继承描述“水果”的抽象类，从而推导出“苹果”是一种“水果”。

在面向对象领域，抽象类主要用来进行类型隐藏。我们可以构造出一组固定行为的抽象描述，但是，这组行为却能够有任意个可能的具体实现方式。这个抽象描述就是抽象类，而这一组任意个可能的具体实现方式，则表现为这个抽象类的所有派生类。

接口和抽象类中的所有抽象方法不能有具体实现，而应在它们的子类中实现所有的抽象方法，Java 的设计者为抽象方法的灵活性考虑，让每个子类可根据自己的需要来实现抽象方法，即要求必须有函数体。

1. 抽象类的定义

抽象类(Abstract Class)的定义方式如下：

```
public abstract class AbstractClass { //里面至少有一个抽象方法
    public int t; //普通数据成员
    public abstract void method1();
      //抽象方法，抽象类的子类在类中必须实现抽象类中的抽象方法
```

```
    public abstract void method2();
    public void method3(); //非抽象方法
    public int method4();
    public int method4() {
        ... //抽象类中可以赋予非抽象方法默认的行为，即方法的具体实现
    }
    public void method3() {
        ... //抽象类中可以赋予非抽象方法的行为，即方法的具体实现
    }
}
```

【例 3.20】抽象类举例：

```
public abstract class  State { //抽象类 State
    public abstract void startUp(Vehicle vehicle);
    public abstract void stop(Vehicle vehicle);
}
//继承抽象类 State
public class VehicleMoveState extends State {
    public void startUp(Vehicle vehicle) {
        System.out.println(vehicle.getName() + "已经在运动状态了");
    }
    public void stop(Vehicle vehicle) {
        System.out.println(vehicle.getName() + "停止运动");
        vehicle.setState(vehicle.getRestState());
    }
}
//继承抽象类 State
public class VehicleRestState extends State {
    public void startUp(Vehicle vehicle) {
        System.out.println(vehicle.getName() + "开始运动");
        vehicle.setState(vehicle.getMoveState());
    }
    public void stop(Vehicle vehicle) {
        System.out.println(vehicle.getName() + "已经是静止状态了");
    }
}
public class Vehicle {
    static State moveState, restState;
    static State state;
    String name;
    Vehicle(String name) {
        this.name = name;
        moveState = new VehicleMoveState();
        restState = new VehicleRestState();
        state = restState;                  //车辆的默认状态是静止状态
    }
    public void  startUp() {
        state.startUp(this);
    }
    public void stop() {
```

```
        state.stop(this);
    }
    public void setState(State state) {
        this.state = state;
    }
    public State getMoveState() {
        return moveState;
    }
    public State getRestState() {
        return restState;
    }
    public String getName() {
        return name;
    }
}
public class Application {
    public static void main(String args[]) {
        Vehicle carOne = new Vehicle("卧铺车厢");
        Vehicle carTwo = new Vehicle("普通车厢");
        carOne.startUp();
        carTwo.startUp();
        carTwo.stop();
        carOne.stop();
    }
}
```

2．接口的定义

接口(interface)的定义方式如下：

```
public interface Interface {
    static final int i = 1;
     //接口中不能有普通数据成员，只能够有静态的不能被修改的数据成员，static 表示全局，
     //final 表示不可修改，可以不用 static final 修饰，会隐式声明为 static 和 final
    public void method1(); //接口中的方法一定是抽象方法，所以不用 abstract 修饰
    public void method2(); //接口中不能赋予方法的默认行为，即不能有方法的具体实现
}
```

【例 3.21】接口举例：

```
public interface TemperatureState { //创建接口 TemperatureState
    public void showTemperature();
}
//实现接口 TemperatureState
public class HeightState implements TemperatureState {
    double n = 26;
    HeightState(int n) {
        if(n > 26)
            this.n = n;
    }
    public void showTemperature() {
        System.out.println("现在温度是" + n + "属于高温度");
```

```
    }
}
//实现接口 TemperatureState
public class MiddleState implements TemperatureState {
    double n = 10;
    MiddleState(int n) {
        if(n>0 && n<=26)
            this.n = n;
    }
    public void showTemperature() {
        System.out.println("现在温度是" + n + "属于正常温度");
    }
}
//实现接口 TemperatureState
public class LowState implements TemperatureState {
    double n = 0;
    LowState(double n) {
        if(n <= 0)
            this.n = n;
    }
    public void showTemperature() {
        System.out.println("现在温度是" + n + "属于低温度");
    }
}
public class  Thermometer {
    TemperatureState state;
    public void showMessage() {
        System.out.println("***********");
        state.showTemperature();
        System.out.println("***********");
    }
    public void setState(TemperatureState  state) {
        this.state = state;
    }
}
public class Application {
    public static void main(String args[]) {
        TemperatureState state = new LowState(-12);
        Thermometer thermometer = new Thermometer();
        thermometer.setState(state);
        thermometer.showMessage();
        state = new MiddleState(20);
        thermometer.setState(state);
        thermometer.showMessage();
        state = new HeightState(39);
        thermometer.setState(state);
        thermometer.showMessage();
    }
}
```

3. 接口与抽象类的区别

(1) 接口的缺点：如果向一个 Java 接口加入一个新的方法，所有实现这个接口的类都得编写具体的实现。

(2) 接口的优点：一个类可以实现多个接口，接口可以让这个类不仅具有主类型的行为，而且具有其他的次要行为，比如 HashMap 接口的主要类型是 Map，而 Cloneable 接口使它具有一个次要类型，这个类型说明它可以安全地克隆。

(3) 抽象类的缺点：一个类只能由一个超类继承，所以抽象类作为类型定义工具的效能大打折扣。

(4) 抽象类的优点：具体类可从抽象类自动得到这些方法的默认实现。

根据接口和抽象类各自的优缺点，总结出抽象类与接口的区别如下：

- 抽象类可以包含部分方法的实现，这是抽象类优于接口的一个主要地方。
- Java 是单继承的，每个类只能从一个抽象类继承，但是每个类可以实现多个接口。使用接口还可以实现混合类型的类。接口可以继承多个接口，即接口间可以多重继承。
- 将类抽取出通用部分作为接口容易，要作为抽象类则不太方便，因为这个类有可能已经继承自另一个类。

简而言之，抽象类是一种功能不全的类，接口只是一个抽象方法声明和静态不能被修改的数据的集合，两者都不能被实例化。从某种意义上说，接口是一种特殊形式的抽象类，在 Java 语言中，抽象类表示的是一种继承关系，一个类只能继承一个抽象类，而一个类却可以实现多个接口。在许多情况下，如果不需要刻意表达属性上的继承的话，接口确实可以代替抽象类。

本 章 小 结

本章主要介绍面向对象编程的重要机制——类、对象、抽象类和接口等，更加深入地体会 Java 语言的面向对象特征。

习　　题

(1) 上机练习题。请使用 Java 中的面向对象编程机制实现下列编程题目。

题目一：某公司采用公用电话传递数据，数据是四位的整数，在传递过程中是加密的，加密规则如下——每位数字都加上 5，然后用和除以 10 的余数代替该数字，再将第一位和第四位交换，第二位和第三位交换。

题目二：海滩上有一堆桃子，五只猴子来分。第一只猴子把这堆桃子平均分为五份，多了一个，这只猴子把多的一个扔入海中，拿走了一份。第二只猴子把剩下的桃子又平均分成五份，又多了一个，它同样把多的一个扔入海中，拿走了一份。第三、第四、第五只猴子都是这样做的，问海滩上原来最少有多少个桃子？

(2) 简述 Java 中抽象类与接口的区别。