第 3 章

Java 面向对象

面向对象程序设计（OOP）是现在被程序员所熟知的一个名词，面向对象并不指的是哪一种语言，而是思维方式的转变。虽然本书以 Java 语言为基础，但是本章会先引领读者树立一种面向对象的编程思想，再进一步了解 Java 语言面向对象的世界。

通过本章的学习，可以掌握以下几点：

- 了解面向对象编程思想
- 掌握如何定义类、类的成员
- 掌握类与对象之间的关系
- 掌握构造方法和方法重载
- 掌握 this 和 static 关键字
- 掌握权限修饰
- 掌握继承、重写、对象转型与多态的关系
- 掌握抽象类、接口

3.1 面向对象概述

本节主要介绍什么是对象、什么是类、对象与对象之间的关系、类与类之间的关系、类与对象之间的关系。本节主要强调的是面向对象的基本概念。

3.1.1 对象

对象是一种现实存在或者是可以描述有具体行为的事物。例如，“一张桌子”“一个人”“一只猫”等，扩大范围到世界的任何一个事物，都可以称做对象。

对象是简单的又是复杂的。例如，一个人去开车，简单来说人和车都是一个对象，但这句话中也包含了其他的对象，如车包含车轮、车灯等，人包含手、脚等。当细化时可以发现对象中还包含其他的对象。

3.1.2　类

类是一系列相似事物的模板，如“人类”“汽车类”“动物类”这些都是类。在“人类”这个概念里不难理解到有一些无法或者长期不改变的属性，如性别、年龄、姓名等。Java 把这些定义成为静态行为，又称做属性或成员变量。在“人类”这个概念里不仅存在静态行为也存在动态行为，如跑步、吃饭、学习等。动态行为又称做方法或成员方法。

在上述的描述中，类是由属性和方法组成的。属性中定义类中的信息，在程序中可以把属性看做一个特殊的变量。而对于方法是一些操作的行为，在程序中是按照具体的语法要求来完成的。

3.1.3　抽象

读者对抽象这个名词一定不陌生，无论是理工科的学生还是文科的学生都会使用到抽象这个名词。抽象并不单单是面向对象程序设计中存在的，在很多书籍中都不会单独讲解抽象作用，但是在面向对象的学习中首先要学会的就是如何从众多的相同或者相似的对象中抽取共同的特征来形成一个模板，这个模板就是这些对象的类。

从具体事物抽出、概括出它们共同的方面、本质属性与关系等，而将个别的、非本质的方面、属性与关系舍弃，这种思维过程称为抽象。抽象就如许多人抽象出人类这个概念一样，如图 3-1 所示。

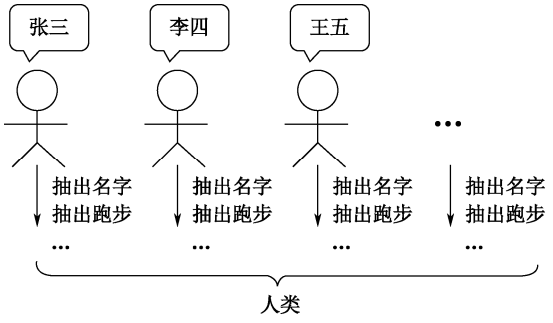

图 3-1　抽象

✻知识拓展

✧ 对于刚把抽象的思维运用到面向对象上的读者可能觉着有点难，在抽象上还有另一种解释，如一个人想去造一辆车，但是他没有头绪，这时最好的解决方案就是先去看看别人已经可以开动的车，并总结出规律或者相同的特征。这个思考的过程就是面向对象中的抽象，而其他人的车是对象，总结出来的规律或特征构成一个车的类。

3.1.4　封装

封装是面向对象的三大特性之一，从字面的意思看像是将一个东西装在盒子里封存起来。封装是面向对象最基本的原则，若没有封装，面向对象中的继承和多态就无法实现。封装有两层含义：第一是将类实例化的对象中的属性和方法看成一个密不可分的整体，将这两者“封装”到一个对象中；第二是一种信息隐藏，有些信息是不能被外界知道的，如果被外界知道可能会

使程序遭到破坏，所以尽可能不让外界知道内部的结构，如用到封装的属性，只需要给出行为即可。

第一层含义在面向对象的程序设计中是通过类抽象出对象间的共有特性。在实例化对象时，对象的状态都是单独的，对于其他的成员是屏蔽的。

第二层含义在面向对象中，为封装的不同等级可以设置访问权限。

3.1.5 继承

继承也是面向对象的三大特性之一，继承可以高效地复用代码。面向对象中的继承和日常生活中的继承有一定的区别，也有一定的相似性。

在日常生活中，继承是继承物件、血统等，对于面向对象中的继承是复制父类的功能存放在子类中，而表现形式却是对象表现出来的。无论是日常生活中的继承还是面向对象中的继承，都是发生在父类与子类之间的。

面向对象首先由若干对象抽象出类，将类书写出来后，该类拥有一般特性，如果发现在该类的基础上又有特殊一类对象存在时，继承就出现了。综合上述，可以发现面向对象中的继承首先需要一个特性的类，然后在其基础上派生子类。例如，现在有人类这个概念，该类只能反映一般人拥有的行为和属性，在进一步学生这个概念，学生类是继承了人类的，学生不仅拥有人类所有的属性和行为，还增加了自己的属性和行为。

继承机制大大增强了代码的复用性，提高了软件的开发效率，而且降低了程序产生错误的概率，为调试 Bug 和程序的修改、扩充提供了便利。

3.1.6 多态

多态同样是面向对象的特性之一，多态是面向对象的设计精华所在。在程序中，某一个类或该类的父类和子类中可能会出现同名的方法，前者称为重载，后者称为重写。无论是哪一种多态形式，都是以最小的代价为程序扩展功能。

方法重载：在一个类中，方法名相同、参数不同，根据参数传递值形成不同的执行的流程不一样，最终结果也不一样。

方法重写：子类对象可以与父类对象之间相互转换，根据具体的要求和场景，调用不同的方法流程，完成的功能也不一样。

❋知识拓展

✧ 上述是 Java 中存在的两种多态，在面向对象中存在四种多态，即参数多态、包含多态、过载多态和强制多态。其中包含多态和过载多态是上述的重载与重写。参数多态：采用参数化模板，通过给出不同类型的参数，使一个结构有多种类型。强制多态：编译程序通过语义操作，把操作对象的类型强行加以转换，以符合函数或操作符的要求。

3.2 类、对象的创建与使用

在 3.1 节中主要讲述的是面向对象的基本概念，而本节将用 Java 语言来描述类、对象和引用，理解本节是深入理解 Java 面向对象的基础。

3.2.1　类的书写格式

在概述中讲到过类是一系列相似事物的模板。类是一个很抽象的概念，如“人类”这个名词我们很是熟悉，但却不知道如何定义。这是一个看似很简单却很难回答的问题。在概述中也提到了可以将类分为两种行为：第一种是静态行为；第二种是动态行为。而 Java 也以同样的方式定义和书写类的格式。

语法格式：

```
权限修饰 class 类名{
        静态行为;
        动态行为;
}
```

静态行为又称为成员变量，动态行为又称为成员方法。在 Java 中成员变量和成员方法都存在权限修饰和值类型，类的具体定义如下：

```
权限修饰 class 类名{
    权限修饰 数据类型 成员变量名称;
    …;
    权限修饰 返回值 成员方法名称 (参数类型 参数名称, …){
          程序块;
          [return 表达式];
    }
    …;
}
```

成员变量

成员方法

格式解释：

class：声明一个类。

类名：类的名称。

成员变量：

权限修饰有四种：public、default、protected 和 private。

数据类型：基本数据类型、引用数据类型。

成员变量名称：成员变量的表示。

…：有多个成员变量。

成员方法：

权限修饰有四种：public、default、protected 和 private。

返回值类型：基本数据类型、引用数据类型和 void。

成员方法名称：成员方法的表示。

参数类型 参数名称：表示形参。

程序块：要执行的程序。

return：返回的值，值的类型要与返回值类型相同。

…：有多个方法。

示例【C03_01】定义一个 Person 类。代码如下：

```
public class Person {
    public String name;//定义姓名
    public int age;//定义年龄
    public void showInfo(){//显示信息
```

```
            System.out.println("姓名为："+name);
            System.out.println("年龄为："+age);
        }
    }
```

以上程序在 Person 类定义 name、age 两个属性，分别表示人的姓名和年龄，在之后定义一个 showInfo 方法，此方法的功能就是打印这两个属性。

3.2.2 对象的创建与使用

如果在创建 Person 类后，不能直接使用类，则需要对类进行实例化使其成为对象才可以使用。实例化后的对象才能调用属性和方法。

1．对象的创建

语法格式：

```
类名 对象名 = null;//对象的声明
对象名 = new 类名([数据类型 参数]);//对象的实例化
```

这两个步骤也可以合在一起。

```
类名 对象名 = new 类名([数据类型 参数]);//声明和实例化一步完成
```

格式解释：

类名：对象的类型。

对象名：对象的引用。

new：关键字表示创建了一个新的对象。

类名()：利用这个类的某一个构造函数创建的对象。

[数据类型 参数]：对象创建时需要参数。

执行流程：

在执行语法“类名 对象名 = new 类名()”时，首先要在栈内存中开辟一小块空间存储对象名，然后在堆内存中开辟一块空间存储对象。

执行内存分析：

执行“类名 对象名 = new 类名()”内存，如图 3-2 所示。

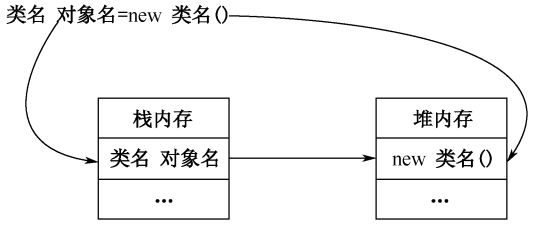

图 3-2 创建对象内存图

示例【C03_02】定义一个 Person 类，并实例化一个对象。代码如下：

```
public class Person {
        public String name;//定义姓名
        public int age;//定义年龄
        public void showInfo(){//显示信息
            System.out.println("姓名为："+name);
            System.out.println("年龄为："+age);
```

```
        }
    public static void main(String[] args) {//实例化对象
        Person p = new Person();
    }
}
```

2．使用对象

在对象被实例化后就可以使用对象的成员变量和成员方法了，具体的使用格式如下。

语法格式：

访问属性：对象名.成员变量名
访问方法：对象名.成员方法名

格式解释：

对象名：被实例化的对象。

.：使用的意思。

成员变量名：声明在类体中的成员变量名称。

成员方法名：声明在类体中的成员方法名称。

执行流程:

在创建对象时，成员变量是在对象的内部创建的，与此同时将成员变量设置为设定的值（如果没有设定，系统将会给成员变量设置默认值）。而成员方法在代码区也就是第 1 章中用 Javac 编译的 class 文件中，在使用方法时，对象会根据内存存放方法的地址去代码区寻找相应的方法。

执行内存分析：

对象在使用属性和方法的内存，如图 3-3 所示。

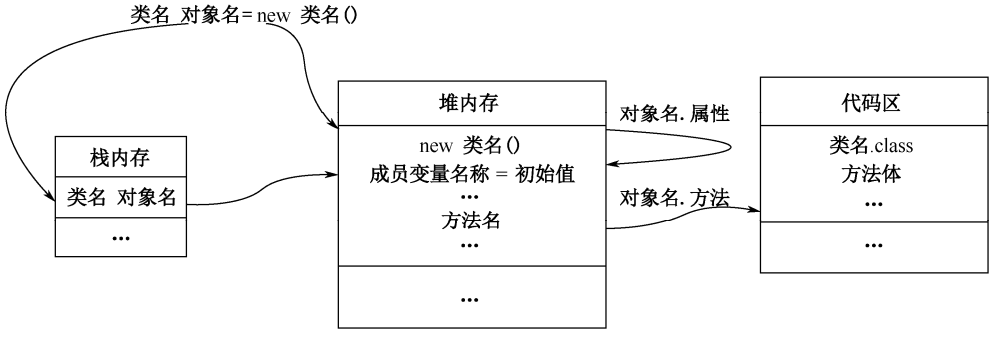

图 3-3　使用对象内存

示例【C03_03】定义一个 Person 类，实例化一个对象，并使用这个对象对成员变量和成员方法进行访问。代码如下：

```
public class Person {
        public String name;//定义姓名
        public int age;//定义年龄
        public void showInfo(){//显示信息
            System.out.println("姓名为："+name);
            System.out.println("年龄为："+age);
        }
    public static void main(String[] args) {
```

```
        Person p = new Person();//实例化对象
        p.name = "张三";//访问name属性
        p.age = 18;//访问age属性
        p.showInfo();//访问showInfo方法
    }
}
```

运行结果：

```
姓名为：张三
年龄为：18
```

❋知识拓展

✧ 内存中共有四个区域，分别是：栈区、堆区、方法区和数据区。栈区的作用是存放基本数据类型和引用数据类型的对象名；堆区主要存放的是对象；方法区主要存放的是方法；数据区主要存放的是常量、静态变量。

✧ Java 中使用“.”形式调用成员变量和成员方法。

3．使用多个对象

可以使用上述的方法创建并使用多个对象，具体方式见示例【C03_04】。

示例【C03_04】定义一个 Person 类，实例化多个对象，并使用这些对象对成员变量和成员方法进行访问。代码如下：

```
public class Person {
        public String name;//定义姓名
        public int age;//定义年龄
        public void showInfo(){//显示信息
            System.out.println("姓名为："+name);
            System.out.println("年龄为："+age);
        }
    public static void main(String[] args) {
        Person p = new Person();//实例化第一个对象
        Person p1 = new Person();//实例化第二个对象
        p.name = "张三";//访问张三的name成员变量
        p.age = 18;//访问张三的age成员变量
        p1.name = "李四";//访问李四的name成员变量
        p1.age = 29;//访问李四的age成员变量
        p.showInfo();//访问张三的showInfo方法
        p1.showInfo();//访问李四的showInfo方法
    }
}
```

运行结果：

```
姓名为：张三
年龄为：18
姓名为：李四
年龄为：29
```

在该示例中不难发现创建了两个对象，这两个对象分别调用自身的成员变量与成员方法。相同类型不同对象的内存分析如图 3-4 所示。

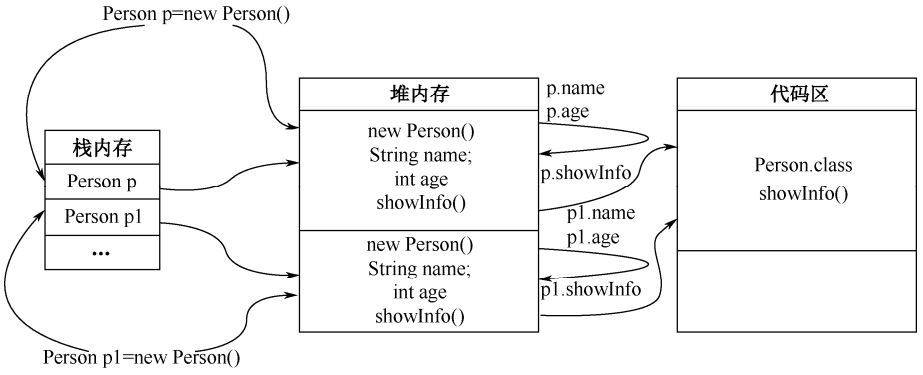

图 3-4　相同类型不同对象的内存分析

由图 3-4 可以发现，每次创建一个新的对象都是独立的一块。每一个对象中都包含成员变量和成员方法在堆中的“注册”，对象通过“.”就可以调用相应的成员变量和成员方法。

3.3　类的构成

类的基本构成有成员变量和成员方法，其中类还包含构造方法、构造代码块等。本节主要讲解类中的成员变量、成员方法和构造方法的使用，以及类中 this 与 static 的使用。

3.3.1　成员变量

成员变量是类的基本组成之一，通常用于承载数据，其格式定义如下。

语法格式：

```
权限修饰 数据类型 成员变量名称；
```

形式如：

```
public String name = "张三";
public int age = 18;
public char gender ;
```

格式解释：

权限修饰：public、default、protected 和 private。

数据类型：基本数据类型、引用数据类型。

成员变量名称：成员变量名与变量的书写相同。

在成员变量中可以有初始值，也可以没有初始值。在没有给出初始值的情况下，系统会给出默认的初始值。

示例【C03_05】定义一个 Person 类，添加三个成员变量，并创建对象对成员变量进行赋值和访问。代码如下：

```
public class Person {
    public String name = "张三";//定义姓名
    public int age ;//定义年龄
    public char gender;//定义性别
    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(p.name);//打印姓名
```

```
            System.out.println(p.age);//打印年龄
            System.out.println(p.gender);//打印性别
            p.name = "李四";//修改姓名
            p.age = 18;//添加年龄
            p.gender = '女';//添加性别
            System.out.println(p.name);//修改之后打印姓名
            System.out.println(p.age);//修改之后打印年龄
            System.out.println(p.gender);//修改之后打印性别
        }
    }
```

运行结果：

```
张三
0

李四
18
女
```

结合示例和结果可以看出，在定义类并且没有给出 age 和 gender 的初始值时，访问的是系统给出的初始值。Java 中引用数据类型给出的默认值都是 null。基本数据类型成员变量的初始值见表 3-1。

表 3-1　基本数据类型成员变量的初始值

数 据 类 型	初　始　值
byte	0
short	0
int	0
long	0L
double	0.0d
float	0.0f
char	'\u0000'
boolean	false

❋知识拓展

✧成员变量：很多地方看到的静态行为、字段、数据域、属性和数据成员在含义上属性相同，但是与属性在详细的定义中有一些区别。

✧成员变量与变量的区别：变量在不给出初始值时是无法使用的，而成员变量不给出初始值时系统默认会给出初始值，是可以使用的。

3.3.2　成员方法

方法又称为成员方法，同样是 Java 类中的基本组成之一，通常使用控制或者流程实现一个业务。

语法格式：

```
权限修饰 返回值 方法名称([数据类型 参数],…){
  程序块;
```

```
[return 表达式];
}
```

格式解释：

权限修饰：public、default、protected 和 private。

返回值：void、基本数据类型、引用数据类型。

方法名称：方法的名字。

[数据类型 参数]：形参。

…：可以有多个参数。

程序块：要执行的程序。

[return 表达式]：与返回值的类型相同，当返回值是 void 时这个表达式不存在。

在格式()中的参数是可以存在也可以不存在的，如果有则表示有参方法，没有则代表无参方法。return 是根据返回值的类型给出值的，如果是 void 类型则没有返回值。这样就形成了方法的四种形式：

- 有参有返；
- 有参无返；
- 无参有返；
- 无参无返。

示例【C03_06】定义一个 Person 类，添加四个成员变量，使用方法的四种形式构建不同的方法，并创建对象对成员变量进行赋值和访问。代码如下：

```
public class Person {
    public long id = 4115211887564215241;//定义身份证号
    public String name = "张三";//定义姓名
    public int age = 18 ;//定义年龄
    public char gender = "女";//定义性别
    /*修改身份证号*/
    public long modifierId(long id){//有参有返
        this.id = id;
        return id;
    }
    /*修改姓名*/
    public void modifierName(String name){//有参无返
        this.name = name;
        System.out.println(name);
    }
    /*输出年龄*/
    public int receiveAge(){//无参有返
        return age;
    }
    /*获得性别*/
    public void receiveGender(){//无参无返
        System.out.println(gender);
    }
    public static void main(String[] args) {//测试类
        Person p = new Person();//新建对象
        /*使用四种类型的方法*/
```

```
            System.out.println(p.modifierId(452187945545115441));
            p.modifierName("李四");
            System.out.println(p.receiveAge());
            p.receiveGender();
        }
    }
```

运行结果：

```
45218794554511544
李四
18
女
```

方法的四种形式适用于不同的场景，有需要输入的就使用有参数的方法，有需要返回一个结果的可以使用有返回的方法。读者需要灵活地根据场景使用不同的方法类型。

❋知识拓展

✧方法与成员变量一样，在方法名和成员变量名前面的修饰成分还可以使用 static、final 等关键字，后续会讲到这些关键字的使用方式和修饰的作用。

3.3.3 重载

重载是一个很重要的概念，它是使用方法执行同类型功能，根据输出的参数不同，执行的过程也不同。重载是一种多态，是同一个类面对不同的输入可以执行不同的过程。

示例【C03_07】定义一个 Tools 类，类实例化的对象可以与整数、浮点数、字符 ASCII 码和字符串长度比较大小。代码如下：

```
public class Tools {
    public void compareMax(int a ,int b){//比较整型数据的大小
        if(a >= b){
            System.out.println(a);
        }
        System.out.println(b);
    }
    public double compareMax(double a ,double b){//比较两个浮点型数据的大小
        if(a >= b){
            return a;
        }
        return b;
    }
    public void compareMax(double a,double b,double c){//比较三个浮点型数据的大小
        if(a >= c && a >= b){
            System.out.println(a);
        }else if(b >= c && b >= a ){
            System.out.println(b);
        }else{
            System.out.println(c);
        }
    }
    public void compareMax(char a ,char b){//比较字符的ASCII码
```

```
            if(a >= b){
                System.out.println(a);
            }
            System.out.println(b);
        }
        public void compareMax(String a ,String b){//比较字符串的长度
            if(a.length() >= b.length()){
                System.out.println("字符串较长的是" + a +"长度为"+a.length());
            }
            System.out.println("字符串较长的是" + b +"长度为"+b.length());

        }
        public static void main(String[] args) {
            Tools t = new Tools();
            t.compareMax(5,3);
        }
    }
```

在该示例中会获取最大值，根据传入值类型或个数不同，展现不同的执行过程。

示例一：

```
t.compareMax(5,3);
```

运行结果：

```
5
```

示例二：

```
t.compareMax('a','b');
```

运行结果：

```
b
```

示例三：

```
t.compareMax("hello","java");
```

运行结果：

```
字符串较长的是hello长度为5
```

示例四：

```
t.compareMax(5.3,5.4,6.1);
```

运行结果：

```
6.1
```

由示例和运行结果可以看出，“方法名相同，参数不同”就可以构成方法的重载，参数包括参数的个数和参数的类型，与方法的返回值没有关系。

3.3.4　构造方法

构造方法同样是方法的一种，只不过使用构造方法是用实例化对象的。构造方法除返回值外与普通的方法相同，构造方法也可以重载。

语法格式：

```
权限修饰 构造方法名([数据类型 参数],…){
    程序块;
}
```

格式解释：

权限修饰：public、default、protected 和 private。

构造方法名：与类名相同。

[数据类型 参数]：形参。

…：可以有多个形参。

程序块：要执行的代码。

需要注意的是构造方法的方法名与类的名称要一致，在程序没有给出构造方法时，该类会自动给出一个无参构造方法。如果显示写出的构造方法，则该类不会再自动给出默认的构造方法，想使用默认无参构造方法，需要将无参构造函数书写出来，否则在实例化对象时，无法使用无参的构造方法创建对象。

示例【C03_08】创建一个 Person 类，定义成员变量和成员方法，并书写测试程序。代码如下：

```
public class Person {
        Person(){//构造方法
        }
        public String name;//定义姓名
        public int age;//定义年龄
        public void showInfo(){//显示信息
            System.out.println("姓名为："+name);
            System.out.println("年龄为："+age);
        }
    public static void main(String[] args) {
        Person p = new Person();//实例化对象
        p.showInfo();
    }
}
```

与类名相同，在不显示书写时，类会自动给出

构造方法功能实例化对象

运行结果：

```
姓名为：null
年龄为：0
```

示例【C03_09】创建一个 Person 类，定义成员变量和成员方法，在构造方法中初始化成员变量，并书写测试程序。代码如下：

```
public class Person {
      Person(String name,int age){//构造方法
          this.name = name;
          this.age = age;
      }
        public String name;//定义成员变量
        public int age;//定义年龄
        public void showInfo(){//显示信息
            System.out.println("姓名为："+name);
            System.out.println("年龄为："+age);
        }
    public static void main(String[] args) {
        Person p = new Person("张三",18);//实例化对象
        p.showInfo();
    }
}
```

含有参数的构造方法，书写之后，类给出的无参构造方法会消失

不能调用无参的构造方法

运行结果：

```
姓名为：张三
年龄为：18
```

示例【C03_10】创建一个 Person 类，定义成员变量和成员方法，在构造方法中初始化成员变量，并可以使用无参构造方法创建对象。书写测试类测试程序，代码如下：

```
public class Person {
        Person(){//无参构造方法
        }
        Person(String name,int age){//有参构造方法
            this.name = name;
            this.age = age;
        }
        public String name;//定义成员变量
        public int age;//定义年龄
        public void showInfo(){//显示信息
            System.out.println("姓名为："+name);
            System.out.println("年龄为："+age);
        }
    public static void main(String[] args) {
        Person p = new Person("张三",18);//利用有参实例化对象
        Person p1 = new Person();//利用无参实例化对象
        p.showInfo();
        p1.showInfo();
    }
}
```

两个构造方法构成重载

运行结果：

```
姓名为：张三
年龄为：18
姓名为：null
年龄为：0
```

由示例【C03_08】到示例【C03_10】可知对象的实例化依据的是构造方法，一个类可以拥有多个构造方法。含有参数的构造方法在创建对象的一开始就能使用初始化数值，可以在实例化对象时传入对象需要的数值。

3.3.5　this 与 static 关键字

this 与 static 关键字是 Java 语言中必须掌握的两个关键字，这两个关键字使用得比较频繁。在很多地方 this 与 static 比较难理解，本节将会详细讲述这两个关键字。

1．this 关键字

this 关键字在 Java 类与对象中、类体中指代表自身的关键字。主要用途有以下四个方面：

（1）使用 this 关键字在自身构造方法内部引用其他构造方法；

（2）使用 this 关键字代表自身类的对象；

（3）使用 this 关键字引用成员变量；

（4）使用 this 关键字引用成员方法。

示例【C03_11】定义一个 Student 类，测试 this 关键字用途的四个方面。代码如下：

```
public class Student {
    public int id = 2018140452;//学生学号
    public String name = "张三";//学生姓名
    Student(){
        System.out.println("学生类的无参构造方法");
    }
    Student(int id){
        this();//this调用无参构造函数
        this.id = id;
        System.out.println("学生类的有参构造方法");
    }
    public void setName(String name){
        this.name = name;//引用成员变量
    }
    public String getName(){
        return name;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public void showInfo(){
        System.out.println("学生的学号为："+this.getId());//引用成员方法
        System.out.println("学生的姓名为："+this.getName());
    }
    public Student createObject(){
        return this;//代表自身对象
    }
    public static void main(String[] args) {
        Student s = new Student();//无参创建s
        Student s1 = new Student(2018140453);//有参创建s1
        Student s2 = s1.createObject();
        s2.showInfo();
    }
}
```

运行结果：

```
学生类的无参构造方法
学生类的无参构造方法
学生类的有参构造方法
学生的学号为：2018140453
学生的姓名为：张三
```

s 利用无参构造方法创建了一个对象，在 s1 创建时是根据值传递创建对象。当一个类内部的构造方法比较多时，可以只书写一个构造方法的内部功能代码，然后其他的构造方法都通过调用该构造方法来实现，这样既保证了所有的构造是统一的，也降低了代码的重复。s1 调用的 createObject()返回的是自身，事实上 s1 和 s2 指的是同一个对象（形式如同 s2 = s1）。

2．static 关键字

static 关键字表示静态，被 static 修饰的成员一般称做静态成员。由于静态成员不依赖于任何对象就可以进行访问，因此对于静态方法来说是没有 this 的，因为它不依附于任何对象，并且由于这个特性，在静态方法中不能访问类的非静态成员变量和非静态成员方法，所以非静态成员方法和非静态成员变量都是必须依赖具体的对象才能够被调用的。static 可以修饰的成分有块、成员变量、成员方法等。

示例【C03_12】定义一个 Student 类，测试 static 关键字修饰的类成员与非 static 关键字修饰的类成员的区别。代码如下：

```
public class Student {
    public static String school = "**学院";
    public static int phoneNumber =18388886666;
    public int id = 2018140452;//学生学号
    public String name = "张三";//学生姓名
    Student(){
        this.say(); //使用静态方法
    }
    public static void say(){//静态方法
        System.out.println("我是一名**学院的学生！");
    }
    public void showInfo(){
        System.out.println("学生的学号为："+this.id);//非静态成员
        System.out.println("学生的姓名为："+this.name);
    }
    public static void main(String[] args){
        System.out.println(Student.school);//通过类名可以直接调用静态成员
        System.out.println(Student.phoneNumber);
        Student s = new Student();
        s.showInfo();
    }
}
```

运行结果：

```
**学院
18388886666
我是一名**学院的学生！
学生的学号为：2018140452
学生的姓名为：张三
```

静态成员在数据区中，资源是被所有的对象共享的，如一个对象对其进行修改，那么所有的对象在访问到这个资源时其都会被改变。静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，在类初次加载时会被初始化；而非静态变量是对象所拥有的，在创建对象时被初始化，存在多个副本，各个对象拥有的副本互不影响。示例【C03_12】内存分析如图 3-5 所示。

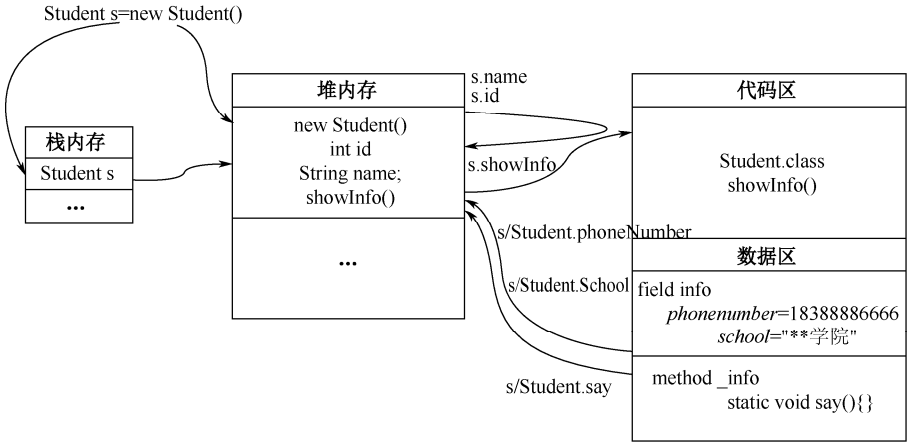

图 3-5　示例【C03_12】内存分析

改变以下代码：

```
System.out.println(Student.School);
System.out.println(Student.phoneNumber);
Student s = new Student();
s.phoneNumber = 18388886667;
s.showInfo();
System.out.println(Student.phoneNumber);
```

运行结果：

```
**学院
18388886666
我是一名**学院的学生！
学生的学号为：2018140452
学生的姓名为：张三
18388886667
```

虽然在静态方法中不能访问非静态成员方法和非静态成员变量，但在非静态成员方法中是可以访问静态成员方法和静态成员变量的。用静态修饰的成员可直接被类名访问，同样也可被对象访问。

3.4　继　　承

继承是面向对象重要的特性之一，也是 Java 语言中较为重要的知识点。继承是代码复用的一种体现。

3.4.1　继承的基本概念

继承就是子类继承父类的特征和行为，使子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使子类具有与父类相同的行为。交通工具的继承实例如图 3-6 所示。

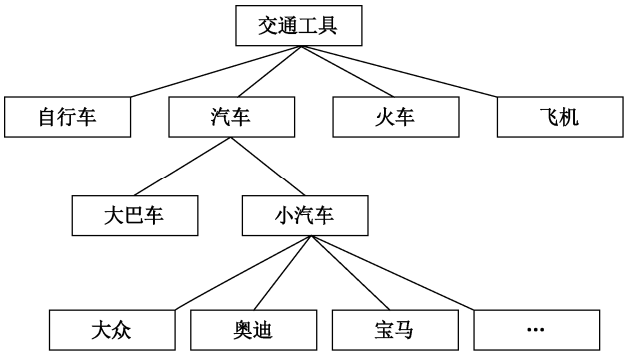

图 3-6　交通工具的继承实例

在 Java 语言中继承的语法格式如下。

语法格式：

```
权限修饰 class 父类名{
    …
}
权限修饰 class 子类名 extends 父类名{
    …
}
```

格式解释：

权限修饰：可以使用权限修饰符。

class：类的关键字。

父类名：被继承的类名。

子类名：继承的类名。

extends：发生继承的关键字。

发生继承关系需要两个或者两个以上的类，在类中被直接或间接继承的称为父类、超类、基类，直接或间接继承的类称为子类、派生类。

示例【C03_13】创建四个类，即 Person 类、Teacher 类、Student 类和 MiddleStudent 类。Person 类中有成员变量 name、age、gender 和成员方法（无参无返回的）eat()、sleep()方法。Teacher 类继承 Person 类，额外添加成员变量 jobNumber 和成员方法 teach()。Student 类继承 Person 类，额外添加成员变量 studentNumber 和成员方法 Study()。MiddleStudent 类继承 Student 类，额外添加成员变量 school 和成员方法 homeWork()。书写测试程序测试各个类的信息，代码如下：

```
public class Person{//其他类的父类
    public String name;
    public int age;
    public char gender;
    public void eat(){
        System.out.println("Person is eating…");
    }
    public void sleep(){
        System.out.println("Person is sleeping…");
    }
    public static void main(String[] args) {
```

```
        Person p =  new Person();//实例化Person对象
        Teacher t = new Teacher();//实例化Teacher对象
        Student s = new Student();//实例化Student对象
        MiddleStudent ms = new MiddleStudent();//实例化MiddleStudent对象
        p.eat();//Person的对象调用Person中的eat()方法
        p.sleep();//Person的对象调用Person中的sleep()方法
        t.eat();//Teacher的对象调用Person中的eat()方法
        t.sleep();//Teacher的对象调用Person中的sleep()方法
        t.teach();//Teacher的对象调用Teacher中的teach()方法
        s.eat();//Student的对象调用Person中的eat()方法
        s.sleep();//Student的对象调用Person中的sleep()方法
        s.study();//Student的对象调用Student中的study()方法
        ms.eat();//MiddleStudent的对象调用Person中的eat()方法
        ms.sleep();//MiddleStudent的对象调用Person中的sleep()方法
        ms.study();//MiddleStudent的对象调用Student中的study()方法
        ms.homeWork();//MiddleStudent的对象调用Person中的homeWork()方法
    }
}
class Teacher extends Person{//Person类的子类
    public int jobNumber;
    public void teach(){
        System.out.println("Teacher is teaching…");
    }
}
class Student extends Person{//MiddleStudent的父类
    public int studentNumber;
    public void study(){
        System.out.println("Student is studying…");
    }
}
class MiddleStudent extends Person{
    public int school;
    public void homeWork(){
        System.out.println("MiddleStudent is doing homework…");
    }
}
```

运行结果：

```
Person is eating…
Person is sleeping…
Person is eating…
Person is sleeping…
Teacher is teaching…
Person is eating…
Person is sleeping…
Student is studying…
Person is eating…
Person is sleeping…
Student is studying…
MiddleStudent is doing homework…
```

在示例【C03_13】中可以看出即使子类没有定义父类的方法也可以直接调用父类中拥有的方法，而且子类中可以调用添加额外的成员方法和成员变量。MiddleStudent 即使没有直接继承 Person 类，但是依然可以使用 Person 中的方法。

Java 中的继承是单根继承，没有明确指出父类则都认为从 Object 继承。

❋知识拓展

✧子类中没有定义的成员变量，父类中定义了，子类也可以使用。无论是父类定义的成员变量还是成员方法，在权限修饰范围允许的情况下子类都可以显式地调用。

✧无论能不能显式地调用父类的成员变量，子类都会在实例化自身对象之前先实例化父类对象。

3.4.2　super 和 final 关键字

1．super 关键字

在 3.3.5 节中讲述过 this 关键字指向的是自身，而对于 super 关键字指向的是父类。super 有两种通用形式：

（1）访问被子类成员隐藏的父类成员；

（2）可以调用父类的构造函数。

示例【C03_14】定义一个 Student 类，测试 super 关键字的两种形式。代码如下：

```
public class Student {
    private String name ;//设置成员变量为私有
    private int StudentId;
    private char gerder;
    Student( String name,int StudentId){//父类的构造方法
        this.StudentId =StudentId;
        this.name = name;
        System.out.println("Student的构造方法");
    }
    /*成员变量的访问器和修改器*/
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getStudentId() {
        return StudentId;
    }
    public void setStudentId(int studentId) {
        StudentId = studentId;
    }
    public char getGerder() {
        return gerder;
    }
    public void setGerder(char gerder) {
        this.gerder = gerder;
```

```
    }
    public void showInfo(){
        System.out.println("这个学生的姓名："+name);
        System.out.println("这个学生的年龄："+StudentId);
    }
    public static void main(String[] args) {
        MiddleStudent ms = new MiddleStudent("张三",20181452);
        //创建MiddleStudent的对象
        ms.showInfo();//ms调用父类的方法
        ms.showGerder('男');//ms调用自身方法
    }
}
class MiddleStudent extends Student{
    MiddleStudent(String name, int StudentId) {//子类的构造方法
        super(name, StudentId);//必须先构建父类
        System.out.println("MiddleStudent的构造方法");
    }
    public void showGerder(char gerder){
        super.setGerder(gerder);//通过super调用父类的方法
        System.out.println("这个学生性别为："+super.getGerder());
    }
}
```

运行结果：

```
Student的构造方法
MiddleStudent的构造方法
这个学生的姓名：张三
这个学生的年龄：20181452
这个学生性别为：男
```

示例【C03_14】的内存分析如图 3-7 所示。

图 3-7　示例【C03_14】的内存分析

通过示例【C03_14】可以看出 super 在子类中相当于父类的一个对象，其中成员变量被设置成私有的，访问和修改要通过 get 和 set 方法，这样做的目的是为了不破坏类的封装性原则。权限修饰详情参见 3.5.2 节。

❋知识拓展

✧成员变量与属性的区别：在很多地方属性与成员变量都不作区分，属性是 JavaBean 中的概念，成员变量拥有修改器和访问器之后称为属性。

2．final 关键字

Java 中的 final 关键字非常重要，它可以应用于类、方法及变量。一旦将引用声明做 final，

将不能再改变这个引用，因为编译器会检查代码，如果试图将变量再次初始化，编译器就会报编译错误。

示例【C03_15】定义一个 Person 类，测试由 final 关键字修饰的类成员与普通类成员的区别。代码如下：

```
public class Person {
    public String name ;//姓名
    public final String NATIONALITY = "China" ;//国籍
    public int IDNumber ;//身份证号
    public final void eat(){
        System.out.println("正在吃饭…");
    }
    public void eat(String s){//eat()重载
        System.out.println("使用"+s+"正在吃饭…");
    }
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "张三";
        p.IDNumber = 4152165;
        //p.NATIONALITY = "美国";//被final修饰不能被重新定义
        System.out.println(p.IDNumber);
        System.out.println(p.name);
        System.out.println(p.NATIONALITY );
        p.eat();
        p.eat("筷子");
    }
}
```

运行结果：

```
4152165
张三
China
正在吃饭…
使用筷子正在吃饭…
```

由示例【C03_15】可以看出，被 final 修饰的成员变量使用时是不能再次进行修改的。同样被 final 修饰的成员方法也是不能被重写的，但被 final 修饰的成员方法是可以被重载的。

```
public class Person {
public final void eat(){
    }
}
class student extends Person{
public void eat(){
}
}
```

如果重写 final 修饰的成员方法会出现以下错误：

```
Cannot override the final method from Person
```

被 final 修饰的类也不能被继承，代码如下：

```
public final class Person {
}
```

```
class student extends Person{
}
```

如果继承 final 修饰的类会出现以下错误：

```
The type student cannot subclass the final class Person
```

✻知识拓展

✧重写（override）是构成多态的基本条件之一。

✧按照 Java 代码惯例，final 变量就是常量，而且通常常量名要大写。

3.5 控 制 访 问

在开发一些程序时，尤其是较大的程序时，存在一些类不能被另一些类访问的情况或是一些类不能直接访问到另一些类的类成员。但在 Java 程序设计中，可以通过包和权限修饰做到访问。

3.5.1 包的概念

为了更好地组织类，Java 提供了包机制，用于区别类名的命名空间，它是类的一种文件组织和管理方式，是一组功能相似或相关的类或接口的集合。Java package 提供了访问权限和命名的管理机制，它是 Java 中基础的却又非常重要的一个概念。包的作用有以下几点：

（1）把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。

（2）如同文件夹一样，包也采用了树形目录的存储方式。同一个包中的类名字是不同的，不同包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名来加以区别。因此，包可以避免名字冲突。

（3）包也限定了访问权限，只有拥有包访问权限的类，才能访问某个包中的类。

语法格式：

```
package pkg1[. pkg2[. pkg3…]];
如：java.util.Scanner;
```

格式解释：

package：书写包的关键字。

pkg1：第一层包。

“ . ”：该路径下。

pkg2：第二层包。

…：可以有多层包。

在实际开发中，包结合 import 将关键字和权限修饰一同使用，可达到控制访问的目的。在程序中给包命名时，通常是将域名倒过来写，如“ www.abc.com ”，则包名应该是“ com.abc ”。

示例【C03_16】定义一个 Student 类，将类放到“ com.abc ”包中，再新建不同的包 Test 类并输出学生信息。代码如下：

```
package com.abc;//包名
public class Student {
    public int id = 2018140452;//学生学号
    public String name = "张三";//学生姓名
```

```
    public void showInfo(){
        System.out.println("学生的学号为："+id);//引用成员方法
        System.out.println("学生的姓名为："+name);
    }
}
package three.com;
import com.abc.Student;//引入包
public class Test {
    public static void main(String[] args) {
        Student s = new Student();
        s.showInfo();//打印com.zzdl.Student中的学生信息
    }
}
```

运行结果：

```
学生的学号为：2018140452
学生的姓名为：张三
```

在使用 package 关键字新建包并在包中定义类后，如果再使用这个类则需要使用 import 关键字导入。

✻知识拓展

✧使用包应结合权限修饰来对文件进行管理。

3.5.2　权限修饰

权限修饰是 Java 类与对象中必须掌握的知识点，权限修饰包含四种：private、default、protected 和 public。权限修饰可以修饰类、类成员，每一个权限修饰的作用范围都要结合实际的意义对实际程序进行访问控制。权限修饰的作用范围见表 3-2。

表 3-2　权限修饰的作用范围

	private	default	protected	public
自身类	√	√	√	√
相同包的其他类	×	√	√	√
不同包的其他类	×	×	×	√
相同包的子类	×	√	√	√
不同包的子类	×	×	√	√

“√”表示可见的意思，“×”表示不可见的意思。

public 修饰内外部类、成员变量、成员方法，对项目中的类是可见的，但在不同的包中访问是需要导入包的。

default 修饰内外部类、成员变量、成员方法，对项目中同一个包的类是可见的。

protected 修饰内部类、成员变量、成员方法，对项目中同一个包中的类可见或子类可见。

private 修饰内部类、成员变量、成员方法，对项目中除自身外的其他类都不可见。

示例【C03_17】定义一个 Student 类，设置四个成员变量，使用四种权限修饰这四个成员变量，并书写两个测试类分别进行测试：在不同和相同 Student 类的包中访问成员变量的情况。代码如下：

```
package com.abc;//包名
public class Student {
```

```
    public int id = 2018140452;//学生学号
    String name = "张三";//学生姓名
    protected char gender = "女";//学生性别
    private int age = 18;//学生年龄
}
```

同一个包的测试：

```
package com.abc;//包名
public class Test {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s.id);//尝试打印id
        System.out.println(s.name);//尝试打印name
        System.out.println(s.gender);//尝试打印gender
        System.out.println(s.age);//尝试打印age
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    The field Student.age is not visible
```

在同一个包中 age 是不可见的。说明 private 修饰类的成分对同一个包中其他的类是不可见的。

修改代码为：

```
package com.abc;//包名
public class Test {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s.id);//尝试打印id
        System.out.println(s.name);//尝试打印name
        System.out.println(s.gender);//尝试打印gender
    }
}
```

运行结果：

```
2018140452
张三
女
```

说明 public、default 和 protected 修饰的成分对同一个包是可见的。

不同的包测试：

```
package three.com;//包名
public class Test {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s.id);//尝试打印id
        System.out.println(s.name);//尝试打印name
        System.out.println(s.gender);//尝试打印gender
        System.out.println(s.age);//尝试打印age
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
    Student cannot be resolved to a type
    Student cannot be resolved to a type
```

即使 public 对不同的包可见但也需要导入包。

修改代码为：

```
package three.com;//包名
import com.abc.Student;
public class Test {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s.id);//尝试打印id
    }
}
```

运行结果：

```
2018140452
```

说明在不同包中 default、protected 和 private 修饰的成分对其他包中的成分是不可见的。

3.5.3　内部类

内部类是指在一个类的内部定义一个类。内部类作为外部类的一个成员，依附于外部类而存在。内部类可以是静态的，也可以用四种权限修饰符进行修饰。内部类有四种形式：成员内部类、局部内部类、静态内部类和匿名内部类。

1．成员内部类

成员内部类可以与成员变量一样，属于类的成员变量。

语法格式：

```
权限修饰class 外部类名{
    权限修饰 数据类型 成员变量名;
    权限修饰 class 内部类名{
    …
  }
  …
}
```

格式解释：

外部类：

权限修饰：public、default。

class：类的关键字。

外部类名：外部类的名称。

…：可以额外定义成员变量和成员方法。

权限修饰：public、default、protected 和 private。

数据类型：基本数据类型、引用数据类型。

成员变量名：成员变量名称。

内部类：

权限修饰：public、default、protected 和 private。

class：类的关键字。

内部类名：内部类的名称。

…：可以定义成员变量和成员方法。

如果使用内部类就需要一个外部类实例化一个对象。例如：

```
Outer o = new Outer();
Outer.Inner oi = o.new Inner();
```

只有实例化对象后才能实例化成员内部类的成员变量和成员方法。

示例【C03_18】创建一个 ClassRoom 类，类体中包含成员变量 size、floor，成员内部类 Chair 类还包含打印成员变量的成员方法 showInfo(), Chair 类中包含 length、seats 和 showInfo(), 并书写 Test 类测试程序。代码如下：

```
public class ClassRoom {
    public int size;//成员变量大小
    public int floor;//成员变量楼层
    public class Chair{//成员内部类
        public int length;
        public int seats;
        public void showInfo(){//内部类的成员方法
            System.out.println("椅子的个数为"+seats);
            System.out.println("椅子的长度为"+length);
        }
    }
    public void showInfo(Chair c){//内部类的成员方法
        System.out.println("班级的大小为"+size);
        System.out.println("班级的楼层为"+floor);
        System.out.println("班级的椅子的个数为"+c.seats);
        System.out.println("班级的椅子的长度为"+c.length);
    }
}
```

测试类：

```
public class Test {
    public static void main(String[] args) {
        ClassRoom cr = new ClassRoom();//创建外部类
        ClassRoom.Chair crc= cr.new Chair();//创建内部类
        cr.floor = 4;//楼层赋值4
        cr.size = 50;//大小50米
        crc.length = 2;//长度2米
        crc.seats = 88;//座位个数88个
        crc.showInfo();//内部类打印信息
        cr.showInfo(crc);//外部类打印信息
    }
}
```

运行结果：

```
椅子的个数为88
椅子的长度为2
班级的大小为50
班级的楼层为4
```

```
班级的椅子的个数为88
班级的椅子的长度为2
```

✻知识拓展

✧内部类的权限修饰不能大于外部类的权限修饰，否则程序会出错。

由示例【C03_18】可知内部类可以使用与外部类相同的成员变量或成员方法名，某种意义上内部类与外部类的功能是一样的，只不过内部类在外部类的类体中。

2．局部内部类

局部内部类声明在成员方法体内，作用范围只能是该方法体。

语法格式：

```
权限修饰class 外部类名{
    权限修饰 数据类型 成员变量名;
    权限修饰 返回值 成员方法名([数据类型 参数],…){
        权限修饰 class 类名(){
            …
        }
        …
        [return 表达式];
    }
    …
}
```

格式解释：

外部类：

权限修饰：public、default。

class：类的关键字。

外部类名：外部类的名称。

…：可以额外定义成员变量和成员方法。

成员变量：

权限修饰：public、default、protected 和 private。

数据类型：基本数据类型、引用数据类型。

成员变量名：成员变量名称。

成员方法：

权限修饰：public、default、protected 和 private。

返回值：void、基本数据类型、引用数据类型。

成员方法名：成员方法的名字。

[数据类型 参数]：形参。

…：可以有多个参数。

程序块：要执行的程序。

[return 表达式]：与返回值的类型相同，当返回值是 void 时这个表达式不存在。

内部类：

权限修饰：public、default、protected 和 private。

class：类的关键字。

类名：内部类的名称。

…：可以定义成员变量和成员方法。

局部内部类类似方法的局部变量，所以在类外或类的其他方法中不能访问这个内部类，但这并不代表局部内部类的实例和定义了它的方法中的局部变量具有相同的生命周期。

示例【C03_19】创建一个外部类 Outer，创建局部内部类 LocalClass 并测试局部内部类。代码如下：

```
public class Outer {
    public static void showOuter(){//静态方法
        System.out.println("Outer的静态方法");
    }
    public void printOuter(){//普通方法
        System.out.println("Outer的普通方法");
    }
    public void testOuter(){
        class LocalClass{//局部内部类
            public LocalClass lc;
            public void printLocalClass(){
                System.out.println("局部内部类");
            }
        }
        LocalClass lc = new LocalClass();
        lc.printLocalClass();
    }
    public static void main(String[] args) {
        Outer o = new Outer();
        o.showOuter();
        o.printOuter();
        o.testOuter();//打印局部内部类
    }
}
```

运行结果：

```
Outer的静态方法
Outer的普通方法
局部内部类
```

只能在方法内部类（局部内部类）定义后使用，不存在外部可见性问题，因此没有访问修饰符。不能在局部内部类中使用可变的局部变量，可以访问外围类的成员变量。如果是 static 方法，则只能访问 static 修饰的成员变量，可以使用 final 或 abstract 修饰。

3．静态内部类

静态内部类和静态成员变量类似，都是使用 static 修饰的。

语法格式：

```
权限修饰class 外部类名{
    权限修饰 数据类型 成员变量名;
    权限修饰 static class 内部类名{
        …
    }
```

```
}
```

格式解释：

外部类：

权限修饰：public、default。

class：类的关键字。

外部类名：外部类的名称。

…：可以额外定义成员变量和成员方法。

成员变量：

权限修饰：public、default、protected 和 private。

数据类型：基本数据类型、引用数据类型。

成员变量名：成员变量名称。

内部类：

权限修饰：public、default、protected 和 private。

static：静态。

class：类的关键字。

内部类名：内部类的名称。

…：可以定义成员变量和成员方法。

在创建静态内部类时，不需要将静态内部类的实例绑定在外部类的实例上。普通非静态内部类的对象是依附在外部类对象之中的，要在一个外部类中定义一个静态的内部类，不需要利用关键字 new 来创建内部类的实例。静态类和方法只属于类本身，并不属于该类的对象，更不属于其他外部类的对象。例如：

```
Outer.Inner oi = new Outer.Inner();
```

示例【C03_20】创建一个 ClassRoom 类，类体中包含成员变量 size、floor，静态内部类 Chair 类还包含打印成员变量的成员方法 showInfo(), Chair 类中包含 length、seats 和 showInfo(), 书写 Test 类测试程序。代码如下：

```
public class ClassRoom {
    public int size;//成员变量大小
    public int floor;//成员变量楼层
    public static class Chair{//成员内部类
        public int length;
        public int seats;
        public void showInfo(){//内部类的成员方法
            System.out.println("椅子的个数为"+seats);
            System.out.println("椅子的长度为"+length);
        }
    }
    public void showInfo(Chair c){//内部类的成员方法
        System.out.println("班级的大小为"+size);
        System.out.println("班级的楼层为"+floor);
        System.out.println("班级的椅子的个数为"+c.seats);
        System.out.println("班级的椅子的长度为"+c.length);
        }
```

```
}
```

测试类：

```
public class Test {
    public static void main(String[] args) {
        ClassRoom cr = new ClassRoom();//创建外部类
        ClassRoom.Chair crc= new ClassRoom.Chair();//创建静态内部类
        cr.floor = 4;//楼层赋值4
        cr.size = 50;//大小50
        crc.length = 2;//长度2
        crc.seats = 88;//座位个数88
        crc.showInfo();//内部类打印信息
        cr.showInfo(crc);//外部类打印信息
    }
}
```

创建方式与局部内部类不同

运行结果：

```
椅子的个数为88
椅子的长度为2
班级的大小为50
班级的楼层为4
班级的椅子的个数为88
班级的椅子的长度为2
```

要创建静态内部类的对象，并不需要其外部类的对象，不能从静态内部类的对象中访问非静态的外部类对象。

4．匿名内部类

一个局部内部类只被使用一次（只用它构建一个对象），就可以不用对其命名了，这种没有名字的类称为匿名内部类。

语法格式：

```
权限修饰class 外部类名{
    权限修饰 数据类型 成员变量名;
    权限修饰 返回值 成员方法名([数据类型 参数],…){
            new 匿名内部类(){
            }
            程序块
            [return 表达式]
        }
    }
}
```

格式解释：

外部类：

权限修饰：public、default。

class：类的关键字。

外部类名：外部类的名称。

…：可以额外定义成员变量和成员方法。

成员变量：

权限修饰：public、default、protected 和 private。

数据类型：基本数据类型、引用数据类型。
成员变量名：成员变量名称。
成员方法：
权限修饰：public、default、protected 和 private。
返回值：void、基本数据类型、引用数据类型。
成员方法名：成员方法的名字。
[数据类型 参数]：形参。
…：可以有多个参数。
程序块：要执行的程序。
[return 表达式]：与返回值的类型相同，当返回值是 void 时这个表达式不存在。
内部类：
new：创建对象的关键字。
匿名内部类：父类的名称。
示例【C03_21】匿名内部类的示例。代码如下：

```
public class Outer {
    public void printOuter(){//普通方法
        System.out.println("Outer的普通方法");
    }
    public void testOuter(){
      new LocalClass(){//匿名内部类
            public void printOuter(){
                System.out.println("匿名内部类");
            }
        }.printOuter();
    }
    public static void main(String[] args) {
        Outer o = new Outer();
        o.printOuter();
        o.testOuter();//打印匿名内部类
    }
}
class LocalClass extends Outer{
}
```

该类没有构造器，依靠父类

运行结果：

```
Outer的普通方法
匿名内部类
```

使用匿名内部类的规则如下：

（1）使用匿名内部类时，必须是继承一个类或实现一个接口，但是两者不可兼得，同时也只能继承一个类或实现一个接口。

（2）匿名内部类中是不能定义构造函数的。

（3）匿名内部类中不能存在任何的静态成员变量和静态方法。

（4）匿名内部类为局部内部类，所以局部内部类的所有限制同样对匿名内部类生效。

（5）匿名内部类不能是抽象的，它必须要实现继承的类或实现接口的所有抽象方法。

❋知识拓展

✧ 接口是一组抽象操作的集合，在本章的 3.8 节中会详细讲解。

3.6 多　　态

多态（Polymorphism）按字面的意思就是“多种状态”。在面向对象语言中，接口的多种不同的实现方式即为多态。在 Java 中多态有两种：第一种是重载形成的多态；第二种是重写形成的多态。前者是在编译期间就形成了多态，称为静态多态；后者是在运行时根据父类接收的对象确定运行的方法，在运行期间形成的多态又称为动态多态。本节讲述的是动态多态。形成动态多态有继承、重写和向上转型三个必需的条件。继承在本章 3.4 节中详细讲述过了，此处不再赘述，本节主要讲述重写和向上转型。

3.6.1 重写

子类可继承父类中的方法，而不需要重新编写相同的方法。但有时子类并不想原封不动地继承父类的方法，而是想做一定的修改，这就需要采用方法的重写。方法重写又称方法覆盖。

语法格式：

```
权限修饰 class 父类名{
            成员变量;
            …
        权限修饰 返回值 method([数值类型 参数],…){
            父类method程序段;
        }
            …;
}
权限修饰 class 子类名 extends 父类名{
            成员变量;
…
        权限修饰 返回值 method([数值类型 参数],…){
            子类method程序段;
        }
            …;
}
```

格式解释：

权限修饰：可以使用权限修饰符。

class：类的关键字。

权限修饰 返回值 method([数值类型 参数],…)：要被重写的成员方法。

父类名：被继承的类名。

子类名：继承的类名。

extends：发生继承的关键字。

权限修饰 返回值 method([数值类型 参数],…)：重写的成员方法。

由上面的定义可以看出重写必须要有继承，而且需要子类中的成员方法名与父类相同，同时子类使用相同的方法执行体，在不同的情况下才能构成重写。

示例【C03_22】创建一个 Animal 动物类，Animal 有成员变量重量（weight）、皮毛颜色（furColor），成员方法 eat()、sleep()；再创建 Cat 猫类继承 Animal 动物类，增加成员变量品种

（variety），增加成员方法 scream()，并重写 eat()方法。代码如下：

```
public class Animal {
    public double weight;//定义成员变量
    public String furColor;
    public void eat(){//定义eat()方法
        System.out.println("Animal eat food");
    }
    public void sleep(){
        System.out.println("Animal is sleeping");
    }
    public static void main(String[] args) {
        Cat c = new Cat();//Cat类实例化对象c
        c.sleep();//c调用sleep方法
        c.eat();//c调用eat()方法
    }
}
class Cat extends Animal{
    public String variety;
    public void eat(){//重写eat()方法
        System.out.println("Cat eat fish");
public void scream() {//自身定义一个额外的方法
        System.out.println("Cat scream");
    }
    }
}
```

运行结果：

```
Animal is sleeping
Cat eat fish
```

由结果可以看出，如果子类中没有重写父类的方法，子类会自动调用父类的方法，如果子类重写了父类的方法，子类的对象在运行时会执行自身重写之后的方法。示例【C03_22】内存分析如图 3-8 所示。

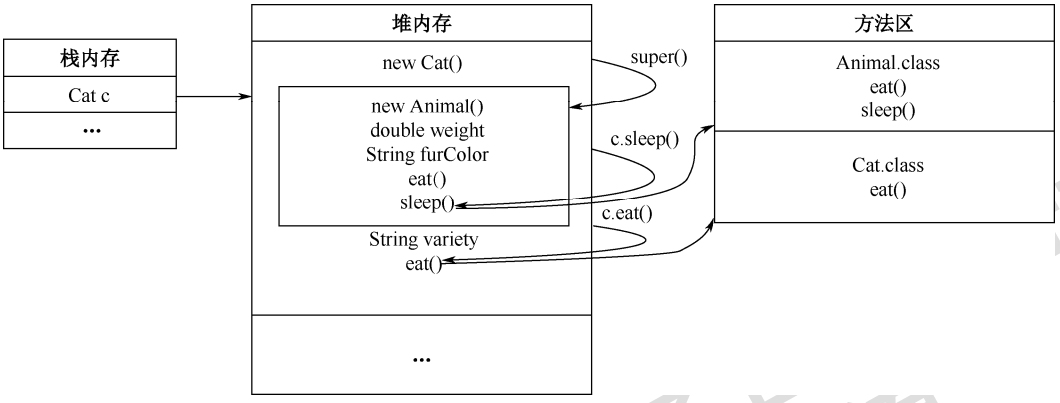

图 3-8　示例【C03_22】内存分析

❋知识拓展

✧ 重载（overload）与重写（overwrite）有很多读者在初学 Java 时分不清。重载是对自身

的方法进行重新加载，方法名相同、参数不同就可以达到重载。重写又称为覆盖（override），需要有继承，子类对父类进行重写需要方法名、返回值和参数都一致才能进行。

3.6.2 对象转型与多态

对象转型与多态是 Java 中核心的知识点，多态的前提是要有继承、重写和对象转型。本节主要讲述对象转型与多态。

1．对象转型

对象转型中的向上转型动态是多态的必要条件，对象转型分为两种：一种称为向上转型（父类对象的引用或者称为基类对象的引用指向子类对象，这就是向上转型），另一种称为向下转型。

（1）向上转型。

向上转型一定是安全的，从小范围转换成大范围。父类的变量接收子类的对象。例如：

```
Parent p = new Children();
```

示例【C03_23】创建一个 Animal 动物类，Animal 有成员变量重量（weight）、皮毛颜色（furColor），成员方法 eat()、sleep()；创建 Dog 狗类继承 Animal 动物类，增加成员变量品种（variety），增加成员方法 lookDoor()，并测试向上转型。代码如下：

```
public class Animal {
    public double weight;// 定义成员变量
    public String furColor;
    public void eat() {// 定义eat()方法
        System.out.println("Animal eat food");
    }
    public void sleep() {
        System.out.println("Animal is sleeping");
    }
    public static void main(String[] args) {
        Animal a = new Dog();//父类的变量接收子类的对象
        a.sleep();// a调用sleep方法
        a.eat();// a调用eat()方法
        //a.lookDoor()//无法使用自身定义的方法
    }
}
class Dog extends Animal {
    public String variety;
    public void lookDoor() {//自身定义一个额外的方法
        System.out.println("Dog lookDoor");
    }
}
```

运行结果：

```
Animal is sleeping
Animal eat food
```

由示例【C03_23】可知向上转型是用父类的变量接收子类的对象，而在运行时子类把自身当成一个父类的对象看待，所以不能调用自身的方法。示例【C03_23】的内存分析如图 3-9 所示。

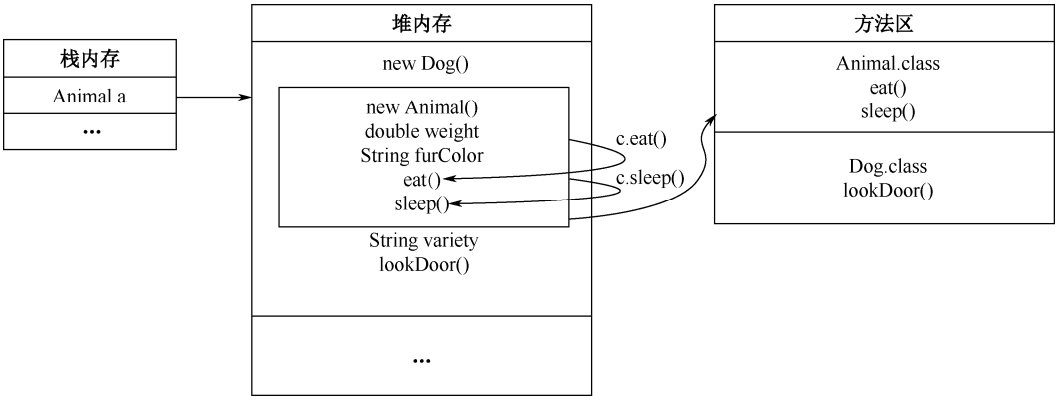

图 3-9　示例【C03_23】的内存分析

从图 3-9 中可以看出 Animal a 指向的是 new Dog 体创建的 Animal 对象。Animal 类型指向了 Dog 这个对象，在程序中会把这只 Dog 当成一只普通的 Animal，既然是把 Dog 当成一只普通的 Animal，那么 Dog 类里面声明的成员变量 variety 就不能访问了，因为 Animal 类里面没有这个成员变量。同样这个变量指向的这个对象是无法识别 lookDoor()方法的。

（2）向下转型。

向下转型不一定是安全的，从大范围转换成小范围。子类类型的变量接收父类的对象。例如：

```
Parent p = new Children();
Children c = (Children )p;
```

示例【C03_24】创建一个 Animal 动物类，Animal 有成员变量重量（weight）、皮毛颜色（furColor），成员方法 eat()、sleep()；创建 Dog 狗类继承 Animal 动物类，增加成员变量品种（variety），增加成员方法 lookDoor()，并测试向下转型。代码如下：

```
public class Animal {
    public double weight;// 定义成员变量
    public String furColor;
    public void eat() {// 定义eat()方法
        System.out.println("Animal eat food");
    }
    public void sleep() {
        System.out.println("Animal is sleeping");
    }
    public static void main(String[] args) {
        Animal a = new Dog();//父类的变量接收子类的对象
        Dog d = (Dog)a;//强制转换
        a.sleep();// a调用sleep方法
        a.eat();// a调用eat()方法
        d.sleep();
        d.eat();
        d.lookDoor();//可以使用自身方法
    }
}
class Dog extends Animal {
    public String variety;
```

```
        public void lookDoor() {//自身定义一个额外的方法
            System.out.println("Dog lookDoor");
        }
    }
```

运行结果：

```
Animal is sleeping
Animal eat food
Animal is sleeping
Animal eat food
Dog lookDoor
```

程序中的 a 与 d 指向的都是 new Dog()这个对象，将 a 强制转换成一个 Dog 类有可能会发生错误。示例【C03_24】的内存分析如图 3-10 所示。

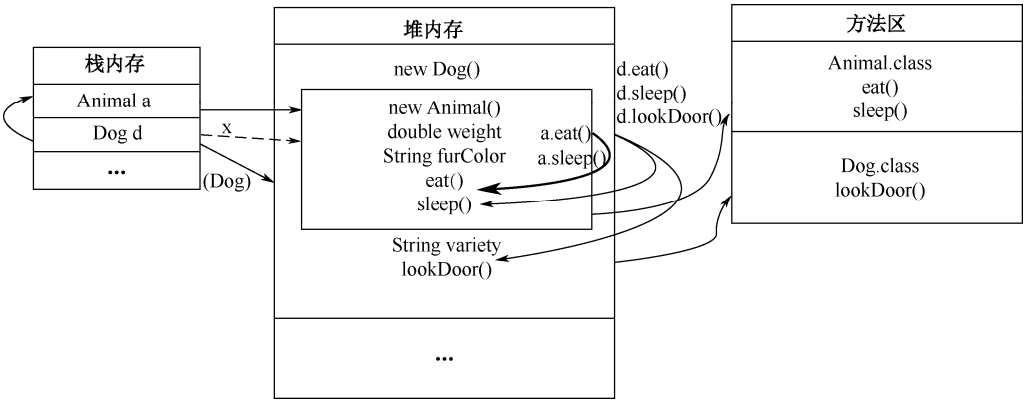

图 3-10　示例【C03_24】的内存分析

在程序的开始 Animal a 的变量会指向 Animal 的对象，而 Dog d 指向的是 a 的指向，在强制转换后 d 指向了 new Dog()，此时 d 可以调用 Dog 类中的成员变量和成员方法。a 强制转换成一个 Dog 类有可能会出现下列错误：

```
        public static void main(String[] args) {
            Animal a = new Animal();//不是子类的对象
            Dog d = (Dog)a;//强制转换
            a.sleep();// a调用sleep方法
            a.eat();// a调用eat()方法
            d.sleep();
            d.eat();
            d.lookDoor();//可以使用自身方法
        }
```

运行结果：

```
Exception in thread "main" java.lang.ClassCastException: three.student.Animal
cannot be cast to three.student.Dog
```

✻知识拓展

✧java.lang.ClassCastException 是个类型不能转换的异常，详情请查阅第 4 章。

可以修改代码为：

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

```
        Animal a = new Animal();//不是子类的对象
        if(c instanceof Dog){
            Dog d = (Dog)a;//强制转换
            d.sleep();
            d.eat();
            d.lookDoor();//可以使用自身方法
        }else{
            System.out.println("error");
        }
        a.sleep();// a调用sleep方法
        a.eat();// a调用eat()方法
    }
```

运行结果：

```
error
Animal is sleeping
Animal eat food
```

instanceof 可以判别当前对象是否属于某一种类型。

2．多态

在学习继承、重写和向上转型后，多态就很容易理解了。Java 运用了一种动态地址绑定的机制实现了多态。运行期间判断对象的类型，并分别调用适当的方法。也就是说，编译器此时依然不知道对象的类型，但方法调用机制能自我调查，找到正确的方法主体。

语法格式：

```
权限修饰 class 父类名{
    成员变量;
      …
        权限修饰 返回值 method([数值类型 参数],…){
            父类method程序段;
        }
            …;
}
权限修饰 class 子类名 extends 父类名{
        成员变量;
      …
        权限修饰 返回值 method([数值类型 参数],…){
            子类method程序段;
        }
            …;
}
父类名 变量 = new 子类名();
变量.method([数值类型 参数],…);
```

格式解释：

权限修饰：可以使用权限修饰符。

class：类的关键字。

权限修饰 返回值 method([数值类型 参数],…)：要被重写的成员方法。

父类名：被继承的类名。

子类名：继承的类名。

extends：发生继承的关键字。

权限修饰 返回值 method([数值类型 参数],…)：重写的成员方法。

父类名 变量 = new 子类名()：向上转型。

变量.method([数值类型 参数],…)：调用类重写的方法。

多态实际上利用向上转型和运行期间判断对象的类型，从而调用子类重写过的方法。

示例【C03_25】创建一个 Animal 动物类，Animal 有成员变量重量（weight）、皮毛颜色（furColor），成员方法 eat()、sleep() ；创建 Dog 狗类继承 Animal 动物类，增加成员变量体型（size），增加成员方法 lookDoor()，并重写 eat()方法；创建 Cat 猫类继承 Animal 动物类，增加成员变量品种（variety），增加成员方法 scream()，并重写 eat()方法，测试三个不同类的 eat()方法。代码如下：

```
public class Animal {
    public double weight;// 定义成员变量
    public String furColor;
    public void eat() {// 定义eat()方法
        System.out.println("Animal eat food");
    }
    public void sleep() {
        System.out.println("Animal is sleeping");
    }
    public static void main(String[] args) {
        Animal a = new Animal();
        //接收子类的对象
        Animal a1 = new Dog();
        Animal a2 = new Cat();
        a.eat();
        //展现不同形态
        a1.eat();
        a2.eat();
    }
}
class Dog extends Animal {
    public String size;
    public void lookDoor() {//自身定义一个额外的方法
        System.out.println("Dog lookDoor");
    }
    public void eat() {// 狗类重写eat()方法
        System.out.println("Dog eat meat");
    }
}
class Cat extends Animal {
    public String variety;
    public void scream() {//自身定义一个额外的方法
        System.out.println("Cat scream");
    }
    public void eat() {// 猫类重写eat()方法
```

```
            System.out.println("Cat eat fish");
        }
    }
```

运行结果：

```
Animal eat food
Dog eat meat
Cat eat fish
```

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则出现编译错误；如果有，再去调用子类的同名方法。多态的好处：可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。示例【C03_25】的内存分析如图 3-11 所示。

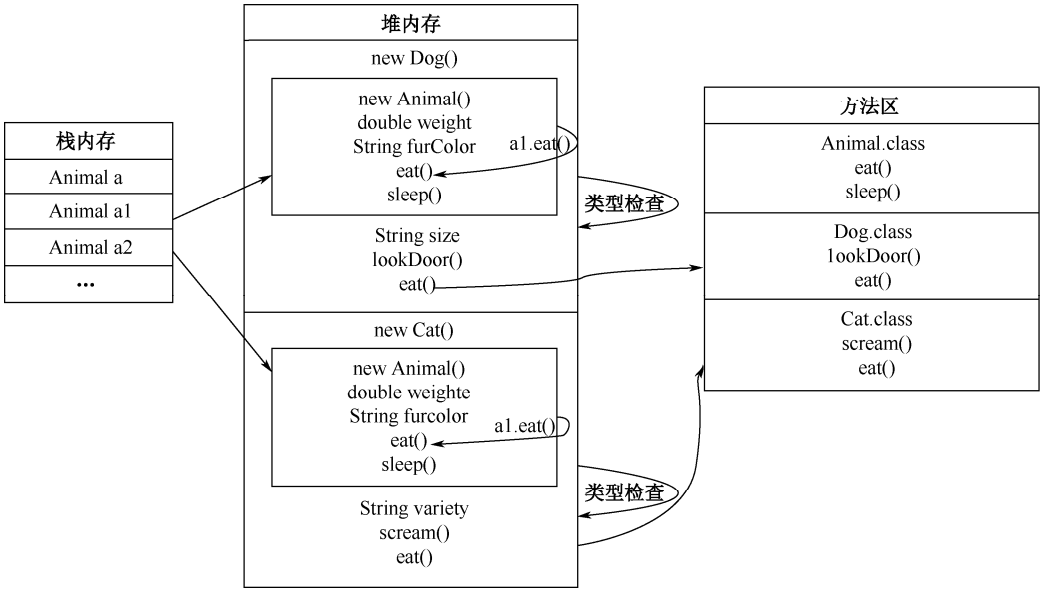

图 3-11　示例【C03_25】的内存分析

示例【C03_26】在示例【C03_25】的基础上添加一个 Person 类，类中有喂养动物 feedPet()，有两个类 Man 和 Women 继承 Person 类，并重写 feedPet()方法，Man 类有额外的 doWork()方法，Women 类有 doHouseWork 的方法，Women 类只能养 Dog 类。测试多态代码如下：

```
public  class Person {
    public void feedPet(Animal a){
    }
    public static void main(String[] args) {
        //多态
        Person p = new Men();
        Person p1 = new Women();
        p.feedPet(new Dog());
        p1.feedPet(new Cat());
    }
}
class Men extends Person{
    @Override
    public void feedPet(Animal a) {//重写方法
```

```
            super.feedPet(a);//父类方法
            System.out.println("man feed"+a.getClass().getName());
        }
        public void doWork(){
            System.out.println("man is working");
        }
    }
    class Women extends Person{
        @Override
        public void feedPet(Animal a) {
            super.feedPet(a);
            if(a instanceof Dog){//判别是否是Dog类
                System.out.println("Women feed "+a.getClass().getName());
            }else{
                System.out.println("a isn't a dog");
            }

        }
        public void doHouseWork(){
            System.out.println("Women is working");
        }
    }
```

运行结果：

```
man feed Dog
a isn't a dog
```

有时可以将父类类型作为另外一个类的参数类型，根据传参的调用相同方法名执行的结果有可能不同。上述例子是一个简单的依赖关系的体现。依赖关系是类与类之间最简单的关系。其他的关系还有关联、泛化和实现等。

3.7 抽 象 类

语法格式：

```
权限修饰 abstract class 抽象类名{
        成员变量;
        成员方法;
        [抽象方法];
}
```

格式解释：

权限修饰：可以使用权限修饰符。

abstract：抽象关键字。

class：定义类关键字。

成员变量：成员变量。

成员方法：成员方法。

[抽象方法]：可以有抽象方法。

示例【C03_27】抽象类的示例。代码如下：

```
public abstract class Employee {
    private String name;
    private String address;
    private int number;
    public Employee(String name, String address, int number) {//构造方法
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }
    public double computePay() {
        System.out.println("Inside Employee computePay");
        return 0.0;
    }
    public void mailCheck() {
        System.out.println("Mailing a check to " + this.name + " "
                + this.address);
    }
    public String toString() {
        return name + " " + address + " " + number;
    }
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String newAddress) {
        address = newAddress;
    }
    public int getNumber() {
        return number;
    }
}
public class AbstractDemo {
    public static void main(String[] args) {
        /* 以下是不允许的，会引发错误 */
        Employee e = new Employee("zby.", "baidu, bd", 22);
        System.out.println("邮件");
        e.mailCheck();
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    Cannot instantiate the type Employee
```

不能实例化对象的异常，说明即使在有构造方法时也不能创建抽象类的对象。一般是通过继承中抽象类的子类实例化子类对象对抽象类中的方法进行调用的。代码如下：

```
public class Staff extends Employee {
```

```
    private double Staff; // 设置工资
    public Staff(String name, String address, int number, double Staff) {
        super(name, address, number);//调用父类的构造方法
        setStaff(Staff);
    }
    public void mailCheck() {
        System.out.println("Within mailCheck of Staff class ");
        System.out.println("Mailing check to " + getName() + " with Staff "
                + Staff);
    }
    public double getStaff() {
        return Staff;
    }
    public void setStaff(double newStaff) {
        if (newStaff >= 0.0) {
            Staff = newStaff;
        }
    }
    public double computePay() {
        System.out.println("Computing Staff pay for " + getName());
        return Staff / 30;
    }
}
public class AbstractDemo {
    public static void main(String[] args) {
        Staff s = new Staff("hhx", "bj", 3, 3600.00);//构建自身对象
        Employee e = new Staff("zzb", "zz", 2, 2400.00);//向上转型
        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();
        System.out.println("Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}
```

运行结果：

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Staff class
Mailing check to hhx with Staff 3600.0
Call mailCheck using Employee reference--
Within mailCheck of Staff class
Mailing check to zzb with Staff 2400.0
```

书写抽象类的格式包含了抽象方法，含有抽象方法的类一定是一个抽象类，但是抽象类中不一定有抽象方法。抽象方法一定要被重写。

示例【C03_28】抽象方法的使用。代码如下：

```
public abstract class Employee {
    private String name;
    private String address;
```

```
    private int number;
    public abstract double computePay();//抽象方法
}
public class Staff extends Employee{
   private double Staff; // Annual Staff
   public double computePay(){
      System.out.println("Computing Staff pay for " );
      return Staff/30;
   }
}
```

抽象类的总结：

（1）抽象类不能被实例化（初学者很容易犯的错误），如果被实例化，就会报错，编译无法通过。只有抽象类的非抽象子类可以创建对象。

（2）抽象类中不一定包含抽象方法，但是有抽象方法的类必定是抽象类。

（3）抽象类中的抽象方法只是声明，不包含方法体。

（4）构造方法，类方法（用 static 修饰的方法）不能声明为抽象方法。

（5）抽象类的子类必须给出抽象类中抽象方法的具体实现，除非该子类也是抽象类。

3.8　接　　口

Java 接口是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现，因此这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为。

语法格式：

```
public interface接口{
    public final 数据类型 成员变量;
        抽象方法;
}
public interface Door{
         public void open();
         public void close();
}
```

接口实现和类继承的规则不同，为了数据的安全，继承时一个类只有一个直接父类，也就是单继承，但是一个类可以实现多个接口，接口弥补了类不能多继承的缺点，继承和接口的双重设计既保证了类的数据安全又变相实现了多继承。

书写格式：

```
权限修饰 class 类名 extends 父类implements 接口,…{
}
```

示例【C03_29】定义一个接口 Door，有两个方法 open()和 close()，定义一个 ElectronicDoor 类实现这个接口。代码如下：

```
public interface Door {//定义接口
    public void open();
    public void close();
}
class ElectronicDoor implements Door{
```

```
    @Override
    public void open() {//重写方法
        System.out.println("open door");
    }
    @Override
    public void close() {//重写方法
        System.out.println("close door");
    }
}
public class Test {
    public static void main(String[] args) {
        Door d = new ElectronicDoor();//用接口接收实现类的对象
        d.close();
        d.open();
    }
}
```

运行结果：

```
close door
open door
```

用接口的变量可以接收实现类的对象，这也是多态现象。接口中可以含有变量和方法。但是要注意，接口中的变量会被隐式地指定为 public static final 变量（并且只能是 public static final 变量，用 private 修饰会报编译错误），而方法会被隐式地指定为 public abstract 方法且只能是 public abstract 方法（用其他关键字，如 private、protected、static、final 等修饰会报编译错误），并且接口中所有的方法不能有具体的实现，也就是说，接口中的方法必须都是抽象方法。从这里可以看出接口和抽象类的区别，接口是一种极度抽象的类型，它比抽象类更加“抽象”，并且一般情况下不在接口中定义变量。

在 Java 中，类的多继承是不合法的，但接口允许多继承。在接口的多继承中 extends 关键字只需要使用一次，在其后跟着继承接口即可。

书写格式：

```
权限修饰 interface 子接口extends 接口,…{
}
```

示例【C03_30】定义两个接口 Door 和 Bell，Door 接口有两个方法 open()和 close()，Bell 中有一个响铃 bell()方法，定义一个 ElectronicDoor 接口继承这两个接口。代码如下：

```
public interface Door {//定义接口
    public void open();
    public void close();
}
interface Bell{//定义接口
    public void bell();
}
interface ElectronicDoor extends Door,Bell{//接口可以多重继承
}
```

在重写接口中声明的方法时，需要注意以下规则：

（1）类在实现接口的方法时，不能抛出强制性异常，只能在接口中或者继承接口的抽象类中抛出该强制性异常。

（2）类在重写方法时要保持一致的方法名，并且应该保持相同或相兼容的返回值类型。

（3）如果实现接口的类是抽象类，那么就没必要实现该接口的方法。

在实现接口时，也要注意一些规则：

（1）一个类可以同时实现多个接口。

（2）一个类只能继承一个类，但是能实现多个接口。

（3）一个接口能继承另一个接口，这和类之间的继承比较相似。

小　结

本章是学习 Java 的重中之重，面向对象的思想贯穿了 Java 所有的知识。初学者应当着重了解面向对象的构成、思想和设计。

由对象出发，很多对象抽象形成文中叙述的类，类通过实例化形成对象。从对象中总结出成员方法和成员变量两个构成成分。这两个成分分别记录了一个类的静态行为和动态行为。成员方法的方法体可以控制对象的行为。如果只关注类的方法，那么可以将这个类书写成抽象类或者接口。在类与类、接口与接口之间有继承关系，类与接口之间有着实现关系。与此同时还应该了解一些关键字的使用，如 Package、import 等。

Java 有大量的程序接口，蕴含着面向对象的思想，应熟练掌握本章内容，为理解 Java 自带 API 奠定坚实的基础。

课 后 练 习

1．什么是抽象？什么是类？什么是对象？什么是封装、继承和多态？

2．编写 Java 应用程序，该程序中有梯形类和主类。要求如下：梯形类具有属性上底、下底、高和面积，具有返回面积的功能，在构造方法中对上底、下底和高进行初始化；主类用来测试梯形类的功能。

3．按要求编写 Java 应用程序：定义描述学生的 Student 类，有一个构造方法对属性进行初始化，一个 outPut 方法用于输出学生的信息；定义主类，创建两个 Student 类的对象，测试其功能。

4．定义 Point 类，表示二维坐标中的一个点，有属性横坐标 x 和纵坐标 y，还有用来获取和设置坐标值，以及计算到原点距离平方值的方法。定义一个构造方法初始化 x 和 y，在主类中创建两个点对象，分别使用两个对象调用相应方法，输出 x 和 y 的值，以及到原点距离的平方。

5．定义一个名为 Vehicles（交通工具）的基类，该类中应包含 String 类型的成员属性 brand（商标）和 color（颜色），还应包含成员方法 run（行驶，在控制台显示“我已经开动了”）和 showInfo（显示信息，在控制台显示商标和颜色），并编写构造方法初始化其成员属性。编写 Car（小汽车）类继承 Vehicles 类，增加 int 型成员属性 seats（座位），还应增加成员方法 showCar（在控制台显示小汽车的信息），并编写构造方法。编写 Truck（卡车）类继承 Vehicles 类，增加 float 型成员属性 load（载重），还应增加成员方法 showTruck（在控制台显示卡车的信息），并编写构造方法，在 main 方法中测试以上各类。

6．编写一个类 Calculate1，实现加、减两种运算，然后编写另一个派生类 Calculate2，实现乘、除两种运算。

7．建立三个类：居民、成人和官员。居民包含身份证号、姓名、出生日期，而成人继承自居民，多包含学历、职业两项数据；官员则继承自成人，多包含党派、职务两项数据。在测试类中创建对象，并打印数据。

8．定义立方体类 Cube，具有属性边长和颜色，还应具有方法设置颜色和计算体积，在该类的主方法中创建一个立方体对象，将该对象边长设置为“ 3 ”，颜色设置为“ green ”，输出该立方体的体积和颜色。

9．设计一个名为 Account 的类：

（1）一个名为 id 的 int 类型私有数据域（默认值为 0）。

（2）一个名为 balance 的 double 类型私有数据域（默认值为 0）。

（3）一个名为 annualInterestRate 的 double 类型私有数据域存储当前利率（默认值为 0）。假设所有的账户都有相同的利率。

（4）一个用于创建默认账户的无参构造方法。

（5）一个用于创建带特定的 id 和初始余额账户的构造方法。

（6）id、balance 和 annualInterestRate 的访问器和修改器。

（7）一个名为 getMonthlyInterestRate()的方法，返回月利率。

（8）一个名为 withDraw 的方法，从账户提取特定数额。

（9）一个名为 deposit 的方法，向账户存储特定金额。

10．在第 9 题的基础上，创建 Account 类的一个子类 CheckAccount 代表可透支的账户，在该账户中定义一个属性 overdraft 代表可透支限额。在 CheckAccount 类中重写 withdraw 方法，其算法如下：如果（取款金额<账户余额），可直接取款；如果（取款金额>账户余额），计算需要透支的额度。判断可透支额 overdraft 是否足够支付本次透支需要，如果可以，将账户余额修改为 0，减去可透支金额；如果不可以，提示用户超过可透支额的限额。

11. 定义一个宠物类（Pet），它有两个方法 :叫 cry()、吃东西 eat()。定义宠物的子类狗（Dog）和猫（Cat），覆盖父类的 cry()和 eat()方法，里面写 System.out.println（“ 猫吃了鱼 ”）这样的打印语句，另外狗有自己的方法看门 guardEntrance()，猫有自己独有的方法捉老鼠 huntMice()。

（1）定义一个 Test 类，在 main 中定义两个 Pet 变量 pet1、pet2，采用引用转型实例化 Dog、Cat，分别调用 Pet 的 cry()、eat()。

（2）将 Pet 强制转换为具体的 Dog、Cat，再调用 Dog 的 guardEntrance()、Cat 的 huntMice()（提示：先用 instanceof 进行类型判断）。

（3）修改 Test 类，添加喂养宠物 feedPet(Pet pet)的方法，在 feedPet 中调用 cry()、eat()方法，实例化 Test 类，再实例化狗 Dog dog = new Dog()、猫 Pet cat = new Cat()，Test 调用 feedPet()方法分别传参数 cat、dog。思考这两种方式的异同，深入理解引入转型和多态。

12. 创建一个名称为 Vehicle 的接口，在接口中添加两个带有一个参数的方法 start()和 stop()。在两个名称分别为 Bike 和 Bus 的类中实现 Vehicle 接口。创建一个名称为 interfaceDemo 的类，在 interfaceDemo 的 main()方法中创建 Bike 和 Bus 对象，并访问 start()和 stop()方法。

13．综合习题：

定义一个抽象的 Role 类，有姓名、年龄、性别等成员变量。

（1）要求尽可能隐藏所有变量（能够私有就私有，能够保护就不要公有），再通过 GetXXX()

和 SetXXX()方法对各变量进行读/写。具有一个抽象的 play()方法，该方法不返回任何值，同时至少定义两个构造方法。Role 类中要体现出 this 的几种用法。

（2）从 Role 类派生出一个 Employee 类，该类具有 Role 类的所有成员，除构造方法外，并扩展 salary 成员变量，同时增加一个静态成员变量“职工编号 ID”。同样至少要有两个构造方法，要体现出 this 和 super 的几种用法，还要求覆盖 play()方法，并提供 final sing()方法。

（3）Manager 类继承 Employee 类，有一个 final 成员变量 vehicle。

（4）在 main()方法中制造 Manager 和 Employee 对象，并测试这些对象的方法。