第 3 章　面向对象编程

面向对象编程（Object-Oriented Programming，OOP），简单地说，就是将一系列数据及其操作进行封装，让数据操作更直观、代码维护更高效。另外，通过继承重复使用的代码，还可以进一步简化软件的开发工作。

传统的过程式开发（如 C 语言）中，定义一个汽车移动到指定坐标的代码可能如下所示。

```
auto_move_to(auto, x, y);
```

在面向对象编程中，代码可能如下所示。

```
auto.moveTo(x, y);
```

不同的开发方式并没有绝对的好与不好，主要还是看软件类型、技术要求和各种因素的综合考虑。

本章将讨论面向对象编程在 Java 中的具体应用，主要内容包括：

- ❑ 类与对象
- ❑ 方法
- ❑ 继承
- ❑ 数据类型处理
- ❑ java.lang.Math 类
- ❑ java.util.Random 类

3.1　类与对象

第 2 章介绍了基本数据类型的使用，这些类型可以处理整数、浮点数、字符和布尔类型的数据。而类（class）则是一种更加复杂的数据类型，它主要包括两种成员，即字段（field）和方法（method）。其中，字段用来存储数据，方法则定义数据的一系列操作。

在 Java 中，定义类需要使用 class 关键字。下面通过项目资源管理器中的“源包”右键菜单“新建”→“Java 类”项添加一个新的类，如图 3-1 所示。

本例中，将新建的类命名为 CAuto。然后，修改 CAuto.java 文件的内容，如下所示。

```
package com.caohuayu.javademo;

public class CAuto {
    //
    public String model = " ";
    public int doors = 4;
```

```
    // moveTo() 方法
    public void moveTo(int x, int y) {
        String s = String.format("%s 移动到 (%d,%d)",model,x,y);
        System.out.println(s);
    }
    //
}
```

图 3-1　添加 Java 类

代码中，CAuto 类定义了两个字段和一个方法，分别是：

- model 字段，表示车的型号，定义为 String 类型，默认为空字符串。
- doors 字段，表示车门数量，定义为 int 类型，默认为 4。
- moveTo() 方法，用于显示车的移动信息。

String 是什么？它也是一个类，这里暂时当作简单的字符串类型使用就可以了。字符串又是什么？可以把它视为文本内容，还要使用一对双引号定义。

代码中使用了 String.format() 方法，它的功能是将各种类型的数据组合为字符串形式，先照样子敲代码就可以了，第 6 章会详细讨论字符串的应用。

回到 CAuto 类，应该如何使用它呢？首先，CAuto 是一个类型，可以定义此类型的“变量”，也就是 CAuto 类的实例（instance）。下面的代码定义了 CAuto 类型的变量 auto。

```
CAuto auto;
```

这里，auto 称为 CAuto 类的一个实例，或者 CAuto 类型的对象，可以简称为 auto 对象。不过，auto 对象暂时还不能使用，因为它还没有实例化，其值默认为 null。

实例化一个对象时，需要使用 new 关键字，如下面的代码所示。

```
CAuto auto = new CAuto();
```

接下来，就可以使用 auto 对象了，如下面的代码所示。

```
public static void main(String[] args) {
CAuto auto = new CAuto();
auto.model = "X9";
auto.moveTo(10, 99);
}
```

代码中，通过圆点运算符（.）调用对象的字段和方法，首先将 model 字段的值设置为 X9，然后调用 moveTo() 方法。执行代码，可以看到如图 3-2 所示的结果。

图 3-2　使用 CAuto 类的实例

此外，注意，main() 是程序的入口方法，它定义在应用的主类中。

3.1.1　构造函数与对象释放

再看一下 auto 对象的实例化代码。

```
CAuto auto = new CAuto();
```

代码中的 CAuto() 是方法吗？好像是，不过，这可不是一般的方法，而是在调用 CAuto 类的构造函数，但并没有定义这个构造函数。

实际上，如果没有在类中没有定义构造函数，则会包含一个空的构造函数。当然，也可以自己创建构造函数。下面的代码在 CAuto 类中添加一个构造函数。

```
package com.caohuayu.javademo;

public class CAuto {
    //
    public String model = "";
    public int doors = 4;
    // 构造函数
public CAuto() {
        System.out.println(" 正在创建汽车对象 ...");
    }
    // moveTo() 方法
    public void moveTo(int x, int y) {
        String s = String.format("%s 移动到 (%d,%d)",model,x,y);
        System.out.println(s);
    }
    //
}
```

构造函数虽然看上去和方法差不多，但它的名称与类名相同，而且不需要定义返回值类型，如 moveTo() 方法中关于 void 关键字的部分。关于返回值的更多内容，3.2 节会详细讨论。

实际开发中，一个类还可以有多个构造函数，只要它们的参数设置能够有效区分就可以。下面的代码在 CAuto 类中创建三个构造函数。

```
// 构造函数
public CAuto() {
    System.out.println(" 正在创建汽车对象 ...");
}
//
public CAuto(String m){
    model = m;
```

```
    }
    //
    public CAuto(String m , int d) {
        model = m;
        doors = d;
    }
```

细心的读者可能会发现一些小问题，例如，这三个构造函数之间并没有什么联系，而且在两个构造函数中出现了重复的代码。这也许不是什么大问题，但还有机会改进代码。下面的代码就是 CAuto.java 文件修改后的全部代码。

```
package com.caohuayu.javademo;

public class CAuto {
    //
    public String model = "";
    public int doors = 4;
// 构造函数
public CAuto(String m , int d) {
        model = m;
        doors = d;
        //
        System.out.println(" 正在创建 " + m +" 汽车 ");
    }
    //
    public CAuto(String m){
        this(m, 4);
    }
    //
    public CAuto() {
        this("");
    }
    // moveTo() 方法
    public void moveTo(int x, int y) {
        String s =
                String.format("%s 移动到 (%d,%d)",model,x,y);
        System.out.println(s);
    }
    //
}
```

第一个构造函数中使用了两个参数，分别指定 model 和 doors 字段的值。重点在接下来的两个构造函数中，首先看下面的版本。

```
public CAuto(String m){
this(m, 4);
}
```

当看到 this 关键字时，应该想到当前实例（对象），而这里就是在调用 CAuto(String m, int doors) 构造函数，其中将车门数设置为 4。最后构造函数就比较好理解了，它调用 CAuto(String m) 版本的构造函数。

实际上，这三个构造函数组成一个构造函数链，通过这种方法可以减少重复代码，提高

代码维护效率。

下面的代码分别使用这三个构造函数创建对象。

```
public static void main(String[] args) {
    CAuto auto1 = new CAuto();
    CAuto auto2 = new CAuto("X9");
    CAuto auto3 = new CAuto("XX", 2);
    System.out.println(auto3.doors);
}
```

代码执行结果如图 3-3 所示。

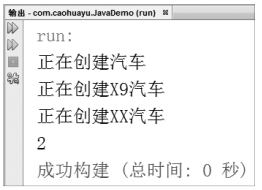

图 3-3　调用不同的构造函数

通过构造函数，可以进行对象的初始化操作，那么，当对象不再使用时应该怎么做呢？实际上，Java 运行环境可以自动回收不再使用的对象，大多情况下并不需要开发者编写代码进行处理。不过，当对象中使用了一些外部资源时，就应该保证这些资源能够正确地关闭，例如，打开文件并进行读写操作后，就应该及时关闭文件。

3.1.2　getter() 和 setter() 方法

这里并不是要创建名为 getter() 和 setter() 的方法，而是通过这两种方法控制字段数据的读取和设置操作。

还以 CAuto 类为例，看一下 doors 字段的使用，如下面的代码所示。

```
CAuto auto = new CAuto("X9");
auto.doors = -2;
System.out.println(auto.doors);
```

本例中的车门数量设置为 –2，难道是在平行宇宙中？还是回到现实世界中，要控制车门数量的设置操作。

对于这样的问题，可以使用 getter() 和 setter() 方法来解决。首先，将字段设置为私有的（private），这样就不能在类的外部访问它。然后，使用 setXXX() 方法设置字段数据，使用 getXXX() 方法返回字段数据。

下面的代码展示在 CAuto 类中修改 doors 字段的方式。

```
package com.caohuayu.javademo;

public class CAuto {
    //
    public String model = "";
    private int doors = 4;
    //设置车门数量
    public void setDoors(int d) {
        if(d >=2 && d <= 5)
            doors = d;
        else
            doors = 4;
```

```
        }
        // 获取车门数量
        public int getDoors() {
            return doors;
        }
        // 构造函数
        public CAuto(String m , int d) {
            model = m;
            setDoors(d);
            //
            System.out.println(" 正在创建 " + m +" 汽车 ");
        }
        //  其他代码
    }
```

代码中所做的修改包括以下几个。

- 将 doors 字段的 public 修饰符更改为 private，稍后会讨论这两个修饰符的区别。
- 添加 setDoors() 方法来设置车门数量，其中，当指定的数据在 2 到 5 之间时，就修改 doors 字段的值，否则使用默认的 4 门。
- 添加 getDoors() 方法来获取车门数量，其中使用 return 语句返回 doors 字段的值即可。
- 构造函数中，对于车门数量的设置，改用 setDoors() 方法来实现。

下面的代码测试与车门数量相关的操作。

```
public static void main(String[] args) {
    CAuto auto = new CAuto("X9", 6);
    System.out.println(auto.getDoors());
    auto.setDoors(2);
    System.out.println(auto.getDoors());
}
```

代码执行结果如图 3-4 所示。

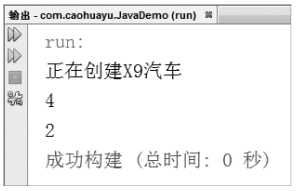

图 3-4　使用 setter 和 getter 方法

示例中，首先使用构造函数设置车门数量为 6，可以看到，实际上 doors 设置为 4。当使用 setDoors() 方法设置车门数量为 2 时，doors 字段的值才会正确设置。

实际应用中，可以通过 setter() 方法控制数据的正确性，设置的数据有问题时，可以使用一个默认值，如前面的示例中那样。当然，如果数据无效，也可以抛出一个异常（Exception），让对象的使用者来处理，第 5 章会讨论异常处理的相关内容。

此外，如果一些数据不需要在类的外部设置，只允许读取，可以只定义 getter() 方法。

3.1.3　静态成员与静态初始化

前面，在 CAuto 类中创建的字段和方法，都必须使用 CAuto 类的实例（对象）来访问，它们称为类的实例成员。开发中，使用 static 关键字，还可以将成员定义为静态成员，静态成员可以使用类的名称直接访问。

下面的代码（CAutoFactory.java 文件）创建了一个汽车工厂类。

```
package com.caohuayu.javademo;

public class CAutoFactory {
    private static int counter = 0;
    //
    public static int getCounter(){
        return counter;
    }
    //
    public static CAuto createSuv() {
        counter++;
        return new CAuto("SUV", 5);
    }
}
```

在 CAutoFactory 类中，定义了三个静态成员，分别如下所示。

- counter 字段，生产计数器，表示工厂生产了多少汽车，它定义为私有的，只能在类的内部自动处理。
- getCounter() 方法，以只读方式返回生产计数器的值。
- createSuv() 方法，用于创建 SUV 车型。

下面的代码测试 CAutoFactory 类的使用。

```
public static void main(String[] args) {
    CAuto suv1 = CAutoFactory.createSuv();
    CAuto suv2 = CAutoFactory.createSuv();
    System.out.println("汽车生产数量为 " + CAutoFactory.getCounter());
}
```

代码执行结果如图 3-5 所示。

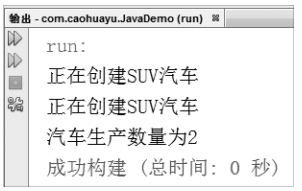

图 3-5　使用静态成员

示例中，使用 CAutoFactory 类直接调用 createSuv() 方法，每一次执行后，counter 的值都会加 1。所以，当创建两辆 SUV 汽车对象后，CAutoFactory.getCounter() 方法返回的数据就是 2。

如果需要对静态成员进行初始化，还可以在类中使用 static 语句定义一个结构来完成，结构中的代码会在第一次调用静态成员时执行一次。下面的代码在 CAutoFactory 类中添加一个静态初始化结构。

```
package com.caohuayu.javademo;

public class CAutoFactory {
    // 静态初始化结构
    static {
        System.out.println("汽车工厂开工了");
    }
    //  其他代码
}
```

再次执行代码，可以看到如图 3-6 所示的结果。

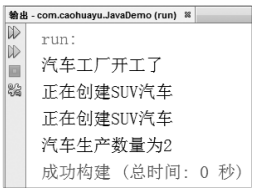

图 3-6　调用静态初始化结构

本例中，虽然代码中多次调用了 CAutoFactory 类中的静态成员，但静态初始化代码只会执行一次。

此外，使用汽车工厂生产汽车的方法是否使代码更加直观呢？实际上，这里使用了一种比较常用的代码结构，它的名称正是“工厂方法”。

3.2　方法

前面的内容中已经使用了不少方法，如 setter 方法、getter 方法、静态方法、工厂方法。现在，讨论方法。

Java 中，定义一个方法的格式如下。

```
< 修饰符 >< 返回值类型 >< 方法名 >(< 参数列表 >) {
< 方法体 >
}
```

其中：

- ❑ < 修饰符 >，确定方法的访问形式（如静态方法）与访问级别（如私有的、公共的等），稍后会有关于修饰符的更多讨论。
- ❑ < 返回值类型 >，指定方法返回数据的类型，如果方法不需要返回数据，则指定为 void。
- ❑ < 方法名 >，指定方法的名称，一般会使用首字母小写，然后每个单词首字母大写的形式，如 getCounter()、moveTo() 等。
- ❑ < 参数列表 >，代入方法的数据，如果没有可以空着。参数可以有一个或多个，多个参数使用逗号分隔，每一个参数都应用包含数据类型和参数变量。
- ❑ < 方法体 >，作为方法的主体部分，是方法完成工作的地方。如果在方法体中需要返回数据，则使用 return 语句来完成。实际上，即使方法不需要返回值，也可以使用空的 return 语句随时终止方法的执行。

下面的代码在 CAuto 类中添加一个静态方法，用于计算百公里的油耗。

```
public class CAuto {
    // 其他代码
    // 百公里油耗
    public static double lphkm(double km, double litre) {
        return litre / (km / 100.0);
    }
}
```

代码中，定义 lphkm() 方法为公共的静态方法，这样就可以使用 CAuto 类的名称访问。两个参数分别指定行驶的里程和耗油量，类型也都定义为 double。返回值类型定义为 double，方法中使用 return 语句返回百公里油耗。

下面的代码测试 lphkm() 方法的使用。

```
public static void main(String[] args) {
    double l = CAuto.lphkm(1000d, 98d);
    String s = String.format("百公里油耗为%.2f升",l);
    System.out.println(s);
}
```

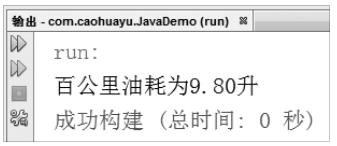

图 3-7　调用方法

代码执行结果如图 3-7 所示。

3.2.1　可变长参数

如果在方法中需要使用零个或多个相同类型的参数，可以通过可变长（variable-length）参数简化参数的定义。

定义可变长参数时，需要在参数类型后加上 ... 运算符。下面的代码在 CAuto 类中添加 join() 实例方法，用于向车中添加乘员。

```
package com.caohuayu.javademo;

public class CAuto {
    // 其他代码
    // 添加乘员
    public void join(String... names) {
        for(String s : names) {
            System.out.println(s + "上车");
        }
    }
    //
}
```

join() 方法的参数看上去只有一个，但是，在 String 后面使用了 ... 运算符，这样，调用 join() 方法时就可以使用零个或多个 String 类型的参数。

下面的代码演示了 join() 方法的使用。

```
public static void main(String[] args) {
    CAuto suv = CAutoFactory.createSuv();
    suv.join("Tom","Jerry","John");
}
```

图 3-8　使用可变长参数

代码执行结果如图 3-8 所示。

可修改 suv.join() 方法中的参数数量（零个或多个），并观察执行结果。

3.2.2　重载

方法的重载是指，多个方法具有相同的名称，但不同的参数定义能够明显地区分方法的版本。调用方法时，可以根据代入的参数自动调用最匹配的版本。实际上，CAuto 类的构造函数已经使用了重载。

下面的代码在 CAuto 类中再添加三个 moveTo() 方法。

```
// moveTo() 方法
public void moveTo(int x, int y) {
    String s = String.format("%s 移动到 (%d,%d)",model,x,y);
    System.out.println(s);
}
//
public void moveTo(float x, float y) {
    String s = String.format("%s 移动到 (%.2f,%.2f)",model,x,y);
    System.out.println(s);
}
//
public void moveTo(String target) {
    System.out.println(" 移动到 " + target);
}
//
public void moveTo(double longitude, double latitude) {
    String s = String.format(" 移动到经度 %.4f, 纬度 %.4f",
                    longitude,latitude);
    System.out.println(s);
}
```

代码中的 moveTo() 方法中，第一个版本是前面创建的，包括两个 int 类型的参数；第二个版本包括两个 float 类型参数；第三个版本使用一个 String 类型的参数，用于指定目的地名称；第四个版本使用经纬度指定坐标，两个参数定义为 double 类型。

下面的代码分别调用这四个版本的 moveTo() 方法。

```
public static void main(String[] args) {
    CAuto aerocar = new CAuto("ZX");
    aerocar.moveTo(99, 11);
    aerocar.moveTo(99f, 11f);
    aerocar.moveTo(" 那啥地方 ");
    aerocar.moveTo(99.0, 11.0);
}
```

第一个 moveTo() 方法中，因为默认的整数是 int 类型，所以会调用参数类型为 int 的版本。第二个 moveTo() 方法中，指定参数为 float 类型，所以调用的是参数为 float 类型的版本。第三个 moveTo() 方法中，使用 String 类型的参数。第四个 moveTO 方法中，因为默认的浮点数是 double 类型，所以会调用参数为 double 的版本。

代码执行结果如图 3-9 所示。

图 3-9　方法的重载

3.3　继承

前面创建的 CAuto 类和 CAutoFactory 类已经定义了不少代码，而且由这两个类生产的 SUV 车型还不错。接下来，还要给 SUV 装上武器，用于开发军用车型。面向对象编程

中，这些工作并不需要完全重新开始，而是在现有类的基础上进行改造和扩展。下面的代码（CAssaultVehicle.java 文件）创建 CAssaultVehicle 类。

```
package com.caohuayu.javademo;

public class CAssaultVehicle extends CAuto {

}
```

这里使用了 extends 关键字，其含义是扩展，但在面向对象编程概念中，更多情况下会说 CAssaultVehicle 类继承于 CAuto 类，即 CAssaultVehicle 类是 CAuto 类的子类，而 CAuto 类称为 CAssaultVehicle 类的超类（或基类、父类）。

CAssaultVehicle 类中，只是让它继承了 CAuto 类，并没有定义任何内容。CAssaultVehiche 类有什么功能呢？不如测试一下，如下面的代码所示。

```
public static void main(String[] args) {
    CAssaultVehicle av = new CAssaultVehicle();
    av.model = "突击者";
    av.setDoors(5);
    av.moveTo(10, 99);
}
```

代码执行结果如图 3-10 所示。

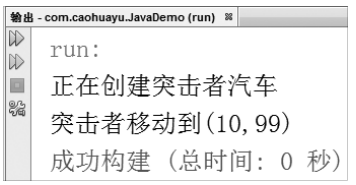

图 3-10　类的继承

示例中，虽然 CAssaultVehicle 类中没有定义任何成员，但它已经从 CAuto 类中继承了不少东西，主要包括无参数的构造函数和非私有的成员（非 private 定义的成员），如 model 字段、setDoors() 方法、getDoors() 方法、moveTo() 方法等。可以发现，继承的作用还是挺大的。

进一步讨论继承之前，需要注意一个问题，如果一个类不希望被继承，可以在定义时使用 final 关键字。下面是一个简单的示例。

```
public final class C1 {
//
}
```

这样，C1 类就不能被继承了，例如，下面的代码就会提示错误。

```
public class C2 extends C1 {
    //
}
```

另外一个需要注意的问题是，在 Java 中，不像在 C++ 中那样，子类可以同时继承多个超类。也就是说，一个类同时只能有一个直接超类。

了解了这些，接下来将讨论关于继承的更多内容。

3.3.1 java.lang.Object 类

定义在 java.lang 包的 Object 类有什么特殊之处？它可是 Java 中其他类的终极超类，也是唯一一个没有超类的类型。如果一个类没有明确指定超类，则默认继承于 Object 类。

对于前面创建的 CAuto 类、CAssaultVehicle 类，以及 Object 类，它们的继承关系如图 3-11 所示。

那么，是不是在所有类中都可以使用 Object 类的非私有成员呢？答案是肯定的，例如，下面的代码使用了一些 CAuto 类中没有定义的成员。

```
public static void main(String[] args) {
CAuto auto = new CAuto();
    CAssaultVehicle av = new CAssaultVehicle();
    System.out.println(auto.toString());
    System.out.println(av.getClass().getSuperclass().toString());
}
```

第一个输出语句使用 toString() 方法显示了 auto 对象的信息。第二个输出语句显示了 av 对象所属类型的超类信息。代码执行结果如图 3-12 所示。

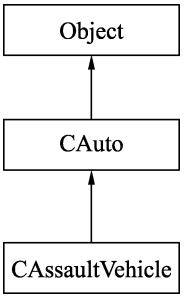

图 3-11　类继承的层次

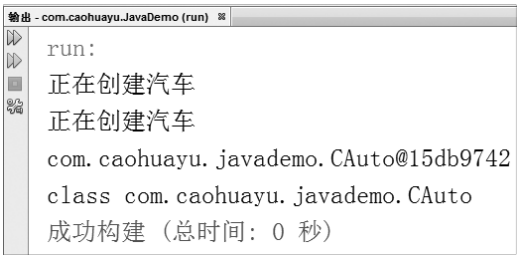

图 3-12　继承 Object 类成员

可以看到，在 Java 代码中动态处理对象和类的信息也是比较方便的，稍后还会讨论相关内容。下面先回到 CAssaultVehicle 类，前面提到要在车上安装武器。

3.3.2 扩展与重写

如果 CAssaultVehicle 类只是简单地继承 CAuto 类，继承的意义就不大了。实际上，在子类中可以扩展超类功能，或者对超类的功能进行重写。

首先考虑 CAssaultVehicle 类的构造函数。如果在子类中没有定义构造函数，默认会继承超类中的无参数构造函数；如果在子类中定义了一个构造函数，就不能直接使用超类的构造函数创建对象了。

那么，在 CAuto 类中创建的构造函数就无用武之地了吗？当然不是，只不过需要在 CAssaultVehicle 类中加个“外壳”而已。例如，下面的代码在 CAssaultVehicle 类中添加了一个无参数的构造函数。

```
package com.caohuayu.javademo;

public class CAssaultVehicle extends CAuto {
    // 构造函数
    public CAssaultVehicle() {
        super("突击者", 4);
    }
    //
}
```

代码中，使用 super 关键字调用超类的构造函数，分别指定型号和车门数量，这里调用的就是 CAuto 类中的 CAuto(String m, int d) 构造函数。

下面的代码测试 CAssaultVehicle 对象的创建。

```
public static void main(String[] args) {
    CAssaultVehicle av = new CAssaultVehicle();
    av.moveTo("9 号地区");
}
```

代码执行结果如图 3-13 所示。

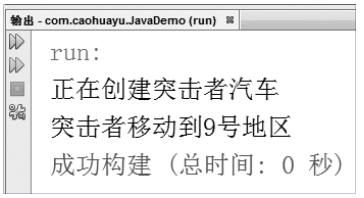

图 3-13　调用超类构造函数

通过以上示例可以看到，创建构造函数的过程中，通过 this、super 关键字，可以合理地重用当前类或超类中的构造函数，使用灵活的方式来构建对象。

如果需要扩展 CAssaultVehicle 类的功能，直接写出来即可。下面的代码在 CAssaultVehicle 类中添加一个字段和一个方法。

```
package com.caohuayu.javademo;

public class CAssaultVehicle extends CAuto {
    // 构造函数
    public CAssaultVehicle() {
        super("突击者", 4);
    }
    // 武器字段
    public String weapon = "";
    // 攻击方法
    public void attack(String target) {
        String s = String.format("使用 %s 攻击 %s",weapon,target);
        System.out.println(s);
    }
}
```

代码中，创建了 weapon 字段和 attack() 方法。下面测试这两个新成员的使用。

```
public static void main(String[] args) {
    CAssaultVehicle av = new CAssaultVehicle();
    av.weapon = "12.7mm 机枪";
    av.attack("靶标");
}
```

代码执行结果如图 3-14 所示。

此外，子类中如果需要重新实现超类中的成员，也可以直接定义。然后，还可以使用 super 关键字访问超类中的成员。下面的代码在 CAssaultVehicle 类中重写 moveTo(String target) 方法。

```
package com.caohuayu.javademo;

public class CAssaultVehicle extends CAuto {
    // 其他代码
    //
    public void moveTo(String target) {
        super.moveTo(target);
        System.out.println(model + "快速行驶到" + target);
    }
}
```

这里使用 super 关键字调用了超类（CAuto 类）中的同名方法。下面的代码演示了新方法的使用。

```
public static void main(String[] args) {
    CAssaultVehicle av = new CAssaultVehicle();
    av.moveTo("X 地区");
}
```

代码执行结果如图 3-15 所示。

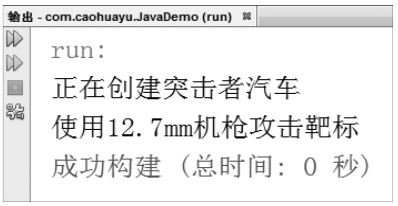

图 3-14　扩展类成员

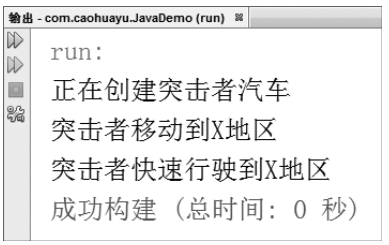

图 3-15　重写超类方法

3.3.3　访问级别

前面的示例中已经多次使用了访问级别的控制，这里简单总结一下 Java 中的常用访问级别。

- private，定义私有成员，即成员只能在其定义的类中访问。
- protected，受保护的成员，它可以在定义的类或子类中访问。
- public，公共成员，它可以供类的外部代码调用。

此外，当成员不使用访问控制关键字时，称为默认（default）访问级别。默认访问级别的成员与 public 有些相似，可以在类的外部调用，但是默认访问级别的成员只能在其定义的包中使用。

一般情况下，出于数据的安全性，类成员的访问级别应遵循最小原则，即优先使用

private 级别。然后，根据需要定义为 protected 或 public 级别。对于默认访问级别，它看上去并不直观，容易让人感到困惑，所以需要熟悉其含义，并在开发中合理使用。

3.3.4　instanceof 运算符

instanceof 运算符用于判断一个对象是否为某个类的实例。在继承关系中，需要注意它的灵活使用。

下面的代码创建一个 CAuto 对象和一个 CAssaultVehicle 对象。

```
public static void main(String[] args) {
    CAuto auto = new CAuto("X9");
    CAssaultVehicle av = new CAssaultVehicle();
    //
    System.out.println(auto instanceof CAuto);                // true
    System.out.println(av instanceof CAuto);                  // true
    System.out.println(av instanceof Object);                 // true
    System.out.println(auto instanceof CAssaultVehicle);      // false
}
```

分别来看四个输出语句。

第一个输出语句中，auto 对象定义为 CAuto 类的实例，所以显示为 true，这个比较容易理解。

第二个输出语句中，av 对象定义为 CAssaultVehicle 类的实例，但 CAssaultVehiclee 类定义为 CAuto 类的子类，所以 av 对象完全可以按 CAuto 对象的方式进行操作。

第三个输出语句中，实际上，所有对象在此都会显示为 true，因为 Object 类是终极超类。

第四个输出语句中，auto 对象不能使用 CAssaultVehicle 类中的新增成员，不能按 CAssaultVehicle 对象的方式进行工作，所以显示为 false。

通过以上示例可以看到 instanceof 运算符的一些应用特点。

- ❑ 所有对象与 Object 类的运算结果都是 true。
- ❑ 对象与其类型或其超类的运算结果为 true。

3.3.5　抽象类与抽象方法

定义方法时使用 abstract 关键字，方法就定义为抽象方法。抽象方法并不需要包含方法体，它必须由类的子类来实现。同时，当一个类中包含抽象方法时，这个类应该定义为抽象类。

例如，下面的代码（CPlaneBase.java 文件）创建一个名为 CPlaneBase 的抽象类。

```
package com.caohuayu.javademo;

public abstract class CPlaneBase {
    public String model;
    // 构造函数
    public CPlaneBase(String m) {
```

```
            model = m;
        }
        //
        public abstract int getMaxSpeed();
        public abstract String getWeapon();
    }
```

这里，在 CPlaneBase 类中定义一个字段、一个构造函数和两个抽象方法，其中，抽象方法中并没有使用“{”和“}”符号定义方法体，而是直接以分号结束。

请注意，抽象类是不能创建实例的，例如，下面的代码就不能正确执行。

```
CPlaneBase plane = new CPlaneBase();                    // 错误
```

接下来，创建一个 CPlaneBase 类的子类，如下面的代码（CFighter.java 文件）所示。

```
package com.caohuayu.javademo;

public class CFighter extends CPlaneBase {
    // 构造函数
    public CFighter() {
        super("战斗机");
    }
    //
    public String getWeapon() {
        return "导弹和机炮";
    }
    //
    public int getMaxSpeed() {
        return 3000;
    }
}
```

下面的代码测试 CFighter 类的使用。

```
public static void main(String[] args) {
    CFighter f = new CFighter();
    System.out.println(f.model);
    System.out.println(f.getWeapon());
  System.out.println(f.getMaxSpeed());
}
```

代码执行结果如图 3-16 所示。

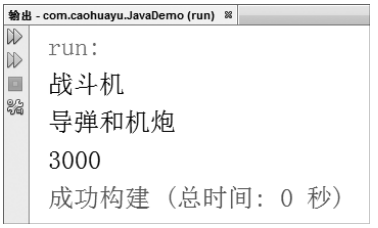

图 3-16　继承抽象类

实际应用中，抽象类更像是标准制定者，它可以定义一系列抽象方法，然后让其子类去具体实现，从而创建具有相同成员但实现各有不同的类型。

下面的代码（CConveyor.java 文件）再创建一个 CConveyor 类，同样，它定义为 CPlaneBase 类的子类。

```
package com.caohuayu.javademo;

public class CConveyor extends CPlaneBase {
    // 构造函数
```

```
    public CConveyor() {
        super("运输机");
    }
    //
    public String getWeapon() {
        return "运输机不用安装武器";
    }
    //
    public int getMaxSpeed() {
        return 1000;
    }
}
```

下面的代码来测试这几个类的使用。

```
public static void main(String[] args) {
    CPlaneBase plane = new CFighter();
    System.out.println(plane.model);
    System.out.println(plane.getWeapon());
    System.out.println(plane.getMaxSpeed());
    //
    System.out.println("*** 飞机变形 ***");
    //
    plane = new CConveyor();
    System.out.println(plane.model);
    System.out.println(plane.getWeapon());
    System.out.println(plane.getMaxSpeed());
}
```

代码中，plane 对象定义为 CPlaneBase 类型，但它不能实例化为 CPlaneBase 类的实例。首先，将 plane 实例化为 CFighter 类的对象，显示信息后，又将 plane 对象实例化为 CConveyor 类的对象并显示信息。代码执行结果如图 3-17 所示。

实际上，对于标准的制定者，接口（interface）会更加纯粹，第 4 章将讨论相关内容。

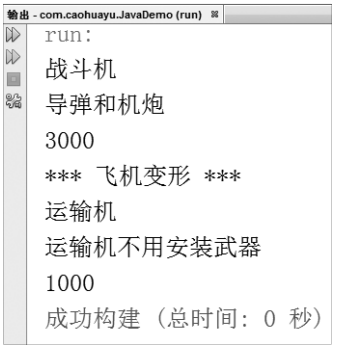

图 3-17　抽象类的综合测试

3.4　数据类型处理

第 2 章讨论了基本数据类型的应用，如整数、浮点数、字符和布尔类型，以及它们的定义、运算、转换等操作。本章讨论面向对象编程的内容。那么，在应用开发中，如何合理地处理这些数据类型呢？

本节就讨论相关内容，首先来看基本数据类型及其包装类的使用。

3.4.1 基本数据类型与包装类

第 2 章讨论过的基本数据类型在 java.lang 包中都提供了一个面向对象的包装类，其对应关系如表 3-1 所示。

表 3-1 基本数据类型与包装类

基本数据类型	包 装 类
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

开发中，应该如何使用基本数据类型和包装类呢？当基本类型的数据需要面向对象操作时，就可以将其转换为对象来使用，如下面的代码所示。

```
public static void main(String[] args) {
    Integer objX = new Integer(10);
    System.out.println(objX.toString());        // 10
}
```

代码中，以 int 类型为例，使用 Integer 类的构造函数代入一个整数，从而创建一个值为 10 的 Integer 对象。然后，通过 toString() 方法显示其内容。实际上，还可以使用更加简单的方式创建 Integer 对象，如下面的代码所示。

```
Integer objY = 99;
System.out.println(objY.toString());            // 99
```

这里，直接将整数赋值给 Integer 类型的 objY 对象，此时，编译器会自动完成转换工作。

示例中，objX 和 objY 就是 Integer 类型的对象，可以使用 Integer 类中定义的成员来处理数据。接下来，还可以使用 Integer 类中的一系列方法将数据转换为所需要的类型，如下面的代码所示。

```
public static void main(String[] args) {
    Integer objX = new Integer(10);
    System.out.println(objX.intValue());
    System.out.println(objX.floatValue());
}
```

代码中，分别使用 intValue() 和 floatValue() 方法将 objX 对象中的数据转换为整数和浮点数，执行结果如图 3-18 所示。

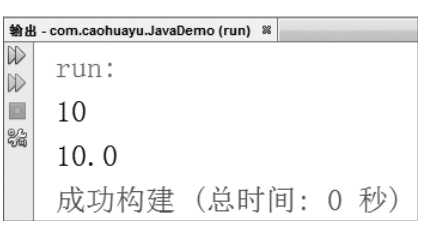

图 3-18 使用基本数据类型与包装类

3.4.2　数据的传递

基本数据类型称为值类型，而所有的类称为引用类型。它们在使用中有什么区别呢？接下来着重讨论值类型与引用类型在数据传递过程中的一些特点。

下面的代码在 JavaDemo 类中创建三个数据改装方法，它们定义在 main() 方法的外面。

```
package com.caohuayu.javademo;

public class JavaDemo {
    public static void main(String[] args) {
        // 测试代码
    }

    // int 数据改装
    static void intModification(int x) {
        x = 99;
    }

    // String 对象改装
    static void stringModification(String s) {
        s = " 新字符串 ";
    }

    // CAuto 对象改装
    static void autoModification(CAuto auto) {
        auto.model = " 改装车 ";
        auto.setDoors(2);
    }
}
```

代码中创建的三方法分别如下所示。

- ❑ intModification() 方法，其中将参数（int 类型）的值修改为 99。
- ❑ stringModification() 方法，其中会修改 String 类型参数的内容。
- ❑ autoModification() 方法，其中会对 CAuto 对象进行修改。

首先，测试 int 类型的改装，如下面的代码所示。

```
public static void main(String[] args) {
    int x = 10;
    intModification(x);
    System.out.println(x);                    //  10
}
```

代码执行结果如图 3-19 所示。

现实可能和你想象的不一样，代码执行结果是，intModification() 方法并没有改变 x 变量的值，这是为什么呢？原因很简单，因为 int 是值类型，而值类型的参数在传递时会产生一个副本。也就是说，在 intModification() 方法中处理的并不是

图 3-19　值类型参数的传递

main() 方法中定义的 x 变量，而是 x 变量的副本，而修改副本的值并不会影响原变量中的数据。

下面的代码改装汽车。

```
public static void main(String[] args) {
CAuto auto = new CAuto("X9", 4);
    System.out.println(auto.model);
    System.out.println(auto.getDoors());
    System.out.println("*** 对汽车进行改装 ***");
    autoModification(auto);
    System.out.println(auto.model);
    System.out.println(auto.getDoors());
}
```

代码执行结果如图 3-20 所示。

代码中，在 autoModification() 方法中成功地对 auto 对象进行了改造，将型号（model）修改为“改装车”，车门数量变成 2。为什么这个操作会成功呢？因为 CAuto 是一个类类型，也就是一个引用类型，引用类型在传递时，会直接传递其对象位于内存中的位置，所以在 autoModification() 方法中实际操作的就是 main() 方法中的 auto 对象。

最后看引用类型中的异类，即 String 类型的改装测试，如下面的代码所示。

```
public static void main(String[] args) {
    String s = "abc";
    stringModification(s);
    System.out.println(s);
}
```

代码执行结果如图 3-21 所示。

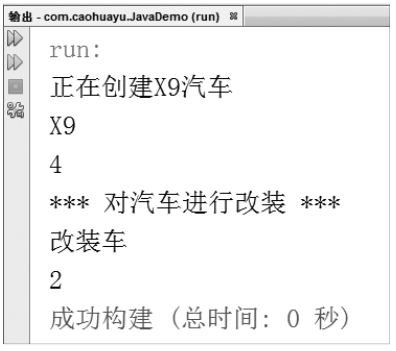

图 3-20　引用类型参数的传递

图 3-21　String 类型参数的传递

String 类型是引用类型。那为什么和 CAuto 对象的表现不一样呢？实际上，不止是在 Java 中，在 C# 中也是这样，String 类用于处理不可变字符串类型。也就是说，String 对象的内容一旦确定就不能改变了，对于字符串内容的任何操作，都会生成一个新的字符串对象。所以，在 stringModification() 方法中修改字符串对象的内容时，实际上已经生成了一个新的字符串对象，而不是 s 所指向的字符串对象。

前面的示例中，使用 + 运算符来连接字符串，这一操作实际上会生成多个字符串对象。

对于需要大量拼接字符串的操作来说，其效率是非常低的。解决方案是使用 StringBuffer 或 StringBuilder 类来操作字符串，第 6 章会讨论相关主题。

3.4.3　类型的动态处理

应用开发过程中，为了简化代码，经常会使用 Object 类或其他通用类型来传递对象，但对象会保留原始类型的相关信息。此时，如何获取对象的真正类型、如何判断对象可以进行什么操作就是一项非常重要的工作。

动态处理对象时，Object 类中的一系列成员，以及 Class 等类型的使用将扮演非常重要的角色。

在讨论继承的过程中，已经使用了 Object 类中的一些方法。下面再来看一看 Object 类中的其他常用成员。

- ❑ equals() 方法，与参数指定的对象进行比较，当两个对象是同一引用时返回 true，否则返回 false。
- ❑ toString() 方法，返回对象的文本描述。
- ❑ getClass() 方法，返回一个 Class 类型的对象，即对象类型的描述对象。

接下来测试 Class 类的使用，如下面的代码所示。

```
package com.caohuayu.javademo;

import java.lang.reflect.Method;

public class JavaDemo {
    public static void main(String[] args) {
Object av = new CAssaultVehicle();
        Class c = av.getClass();
        //
        System.out.println(c.getPackage());
        System.out.println(c.getName());
        System.out.println(c.getSuperclass().getName());
        // 显示方法列表
        Method[] ms = c.getMethods();
        for(Method m : ms) {
            System.out.println(m.getName());
        }
    }
}
```

代码中，首先定义了 CAssaultVechicle 类的 av 对象。然后，使用 av 对象的 getClass() 方法获取对象的类型信息，它会返回一个 Class 类型的对象。

接下来，使用 Class 类中的 getPackage() 方法返回类型所在包的名称；使用 getName() 方法返回类的名称，这是包含包名的完整类名；使用 getSuperclass() 返回类型的超类信息，同样是 Class 对象，同样使用 getName() 显示超类的名称。

最后，使用 Class 对象的 getMethods() 方法返回 CAssaultVechicle 类的所有方法，包括

自定义方法和继承的方法。请注意，Method 类定义在 java.lang.reflect 包中，在代码文件的开始处，package 语句的下面，需要使用 import 语句引用这个类，如下面的代码所示。

```
import java.lang.reflect.Method;
```

如果需要引用 java.lang.reflect 包中的所有资源，可以使用 * 通配符，如下面的代码所示。

```
import java.lang.reflect.*;
```

此外，关于代码中的 for 语句结构，会在第 5 章详细讨论。

3.5 java.lang.Math 类

JDK 中包含了大量的开发资源，其中，java.lang.Math 类定义了很多与数学计算相关的资源。

首先，在 Math 类中定义了一些数学常量，如圆周率。下面的代码将会计算圆的周长和面积。

```
package com.caohuayu.javademo;

import java.lang.Math;

public class JavaDemo {
    public static void main(String[] args) {
        double r = 5.6;
        System.out.println("周长：" + (r * 2 * Math.PI));
        System.out.println("面积：" + (r * r * Math.PI));
    }
}
```

代码显示结果如图 3-22 所示。

图 3-22 使用 Math 类中的常量

查看文档，可以看到，Math 类中 PI 和 E 常量的定义如下。

```
public static final double PI =3.141592653589793;
public static final double E =2.718281828459045;
```

这里使用了 public、static 和 final 关键字，这样就在类中定义了一个静态的最终字段，也就是定义在类中的常量。

接下来，再来看 Math 类中的一些常用方法。

- abs() 方法，获取参数的绝对值，包括各种基本数据类型的重载版本，如 Math.abs (−9) 返回 9。
- hypot(x ,y) 方法将返回 x^2+y^2 的算术平方根（double），如 Math.hypot(3, 4) 返回 5.0。
- sqrt() 方法用于计算参数（double）的算术平方根（double），如 Math.sqrt(16) 返回 4.0。

- pow(x, y) 方法用于计算 x^y 的值，参数类型与结果类型都为 double，如 Math.pwd(2,3) 返回 8.0。
- min() 方法返回两个参数中较小的那一个。
- max() 方法返回两个参数中较大的那一个。
- floor() 方法返回小于或等于参数的最大整数。
- ceil() 方法返回大于等于参数的最小整数。

此外，在 Math 类中还包含了一系列的三角函数计算方法，相信需要的读者很快就能上手。完整的 Math 类定义可以参考官方文档，网址是 http://docs.oracle.com/javase/8/docs/api/index.html。

3.6　java.util.Random 类

很明显，Random 类用于产生随机数。不过，在讨论 Random 类之前，先了解一下 Math.random() 方法。

Math.random() 方法会返回一个大于等于 0.0 但小于 1.0 的随机数（double）。如果要求其他类型的随机数，就需要进一步计算，例如，需要 0 ~ 9 之间的一个随机整数，可以使用如下代码。

```
public static void main(String[] args) {
    int rnd = (int)(Math.random() * 10);
    System.out.println(rnd);
}
```

使用 Random 类会让代码更加清晰，下面的代码同样获取 0 ~ 9 之间的一个随机数。

```
public static void main(String[] args) {
    Random rand = new Random();
    int rnd = rand.nextInt(10);
    System.out.println(rnd);
}
```

代码中，必须创建 Random 类的实例才能来创建随机数，其中使用了 nextInt() 方法的一个重载版本，其参数为一个整数。该方法会返回一个 int 类型的随机数，其值大于等于 0，且小于参数。

如果需要创建指定范围的随机数，可以使用如下代码。

```
public static void main(String[] args) {
    Random rand = new Random();
    int min = 5, max = 10;
    int rnd = rand.nextInt(max - min + 1) + min;
    System.out.println(rnd);
}
```

代码会生成一个大于等于 5 而且小于等于 10 的随机数。