第5章 类和对象

学习目标

- 了解面向过程编程思想与面向对象编程思想的区别。
- 重点掌握类与对象的概念。
- 重点掌握类的属性(成员变量)和方法。
- 了解匿名对象。

本章将深入探讨 Java 中的类和对象,学习如何定义类、如何创建对象,以及如何通过这些对象来交互和执行操作,从基础概念到高级特性,逐步揭示使用它们来设计和实现各种各样的 Web 和 App 应用。

5.1 面向对象概述

在当前的编程领域,面向对象编程(Object-Oriented Programming,OOP)已经成为一种广受欢迎的程序设计方法。

在过去,程序员在编写程序时往往需要按照计算机的逻辑来思考问题,这可能与人类的自然思维方式存在差异。如果完全按照计算机的逻辑来编程,可能会牺牲编程的愉悦感,特别是在面对复杂的系统开发时,这种差异会使代码变得难以管理和维护。因此,如何使用程序来描述和解决复杂的系统问题,成为每位程序员必须面对的问题。

20 世纪 70 年代,一种新的编程范式——面向对象编程(OOP)诞生了。在 OOP 的框架下,数据和操作数据的函数被统一封装成对象。这些对象能够响应消息,而解决问题的方法则是创建对象并向其发送各种消息。通过这种方式,程序中的多个对象可以相互协作,共同构建出复杂的系统,以解决现实世界的问题。

5.2 类

类是现实世界实体的抽象,定义了一组具有相同属性和行为的对象结构。对象则是类的实例,每个实例可以拥有不同的属性值,但共享相同的行为集合。例如,风景优美或有古代遗迹的著名地方统称为名胜古迹,那么读者最喜欢的名胜古迹是哪里?故宫。名胜古迹就是类,故宫就是实例。

随着人工智能和机器人技术的飞速发展,仿生机器人逐渐成为科技创新的热点,而在众多仿生机器人中,机器狗以其独特的设计和功能特别引人注目,比如波士顿动力(Boston Dynamics)的 Spot 机器狗、小米公司的 CyberDog 机器狗等。Spot 仿生狗,如图 5-1 所示。这些充满未来感的仿生狗集成了多种传感器和执行器,还拥有高度自主的运动和交互能力,

因此，本章将借用小米的 CyberDog 仿生狗，向读者介绍 Java 面向对象编程的核心概念。

图 5-1　Spot 仿生狗（图片来自 unsplash.com）

5.2.1　类声明

在 Java 中类声明是定义类的基础。声明一个类需要通过关键字 class。那么，首先声明一个 CyberDog 类代表仿生狗，如例 5-1 所示。

【例 5-1】 仿生狗 CyberDog 类的定义。

```
1   class CyberDog {
2
3   }
```

关键字 class，表示后面正在定义一个类，类的名字叫 CyberDog，这第 1 行就叫作类声明。在 Java 编程中，类名是一个很重要的概念。标识一个 Java 类的名称，命名规则请参考 2.2.2 节。

5.2.2　类体

在类声明之后，花括号（{}）及花括号之间的内容就叫作类体。类体分为两部分：变量的声明和方法的定义。

```
class [类名称]{
    成员变量
    成员方法
}
```

5.2.3　成员变量

成员变量（有时也叫“字段”）用来描述类的具体属性。如例 5-1 声明中，name（名字）、batteryLevel（电池电量）都是 CyberDog 类的属性。

成员变量的类型可以是基本类型，如 int、double、String 等，也可以是指向对象的一个引用，但是该引用必须已经初始化。若是基本类型，则可在类定义位置直接初始化，如例 5-2 所示。

【例 5-2】 仿生狗 CyberDog 成员变量的定义。

```
1   class CyberDog {
2       String name = "旺财";                    //仿生狗的名字
3       int batteryLevel = 100;                  //电池电量
4   }
```

5.2.4 成员方法

成员方法(通常叫作“方法”)就是类所具有的动态功能。如例 5-1 中 CyberDog 声明的 run、rollOver、jump 都是其所具有的功能。成员方法的声明格式如下。

```
[修饰符] [方法返回类型] [方法名](方法参数列表) {
    若干方法语句;
    return 方法返回值;
}
```

(1) 方法返回类型：指定方法执行完毕后返回的数据类型。

(2) 方法名：标识方法的名称,在调用该方法时使用。

(3) 方法参数列表：定义方法调用时需要传递的参数,括号内列出参数类型和名称。

(4) 修饰符：将在 5.2.6 节进行详细阐述。

5.2.5 对象的创建

类是现实世界中某些具有共同特征和行为的事物的抽象。每个对象都是其类的一个实例,拥有自己的状态和行为。

状态指对象的属性,即对象的数据。例如,对于 CyberDog 类,状态可能包括 name(名字)、batteryLevel(电量)等属性。

行为指对象可以执行的操作,通常通过类中定义的方法来实现。对于 CyberDog 类,行为可能包括 run(奔跑)、rollOver(打滚)等方法。

应用程序想要完成具体的功能,仅有类是远远不够的,还需要根据类创建实例对象。在 Java 程序中可以使用 new 关键字创建对象,具体格式如下。

```
[类名] [对象名称] = new 类名();
```

在创建对象后,可以通过对象的引用来访问对象所有的成员,具体格式如下。

```
对象引用.对象成员
```

根据例 5-2 可以创建一个名为“旺财”的 CyberDog,并访问其提供的“奔跑”“打滚”等方法,例 5-3 展示了具体的应用。

【例 5-3】 CyberDog 对象的创建。

```
1   class Example03 {
2       public static void main(String[] args){
3           CyberDog myDog = new CyberDog();
4           myDog.run();
5           myDog.rollOver();
6       }
7   }
```

运行结果如下所示。

```
旺财正在奔跑,消耗 15 % 电量.
旺财正在打滚,消耗 8 % 电量.
```

每个 CyberDog 对象都是独一无二的,即使它们都是根据同一个类创建的。例如,一个 CyberDog 对象可能被命名为“旺财”,电量为 75%,而另一个可能被命名为“富贵”,电量为

50%。至于如何为 CyberDog 在初始时赋予不同的参数，将在 5.3 节中详细讲解。

5.2.6 类的封装

封装是面向对象编程(OOP)中的一个核心概念，通过封装可以隐藏对象的内部状态和实现细节，只暴露出一个可以被外界访问和操作的接口。这种信息隐藏提高了代码的安全性和可维护性，同时也促进了代码的重用。

假设需要给上述的 CyberDog 增加一个低电量告警的功能，那么要在类 CyberDog 中定义一个方法 checkBatteryStatus()表示检测当前电量状态，例 5-4 展示了具体的应用。

【例 5-4】 利用定义方法，检测当前电量状态。

```
1   class CyberDog {
2       String name;            //仿生狗的名字
3       int batteryLevel;       //电池电量
4       void checkBatteryStatus() {
5           if ( batteryLevel < 20) {
6               System.out.println("当前电量为" + batteryLevel + "%,请注意充电!");
7           } else {
8               System.out.println("当前电量为" + batteryLevel + "%,请放心使用!");
9           }
10      }
11  }
12  针对设计的 CyberDog 类创建对象,并访问该对象的成员
13  class Example04 {
14      public static void main(string[] args){
15          CyberDog myDog = new CyberDog();
16          myDog.batteryLevel = -10;
17          myDog.checkBatteryStatus();
18      }
19  }
```

运行例 5-4，运行结果如下所示。

```
当前电量为 -10% ,请注意充电!
```

在例 5-4 的 main()方法中，将电量赋值为一个负数－10，这在程序中不会有任何问题，但在现实生活中明显是不合理的。为了解决电量不能为负数的问题，在设计一个类时，应该对成员变量的访问做出一些限定，不允许外界随意访问。这就需要实现类的封装。

所谓类的封装是指在定义一个类时，将类中的属性私有化，即使用 private 关键字来修饰，私有属性只能在它所在类中被访问。因此，需要对 CyberDog 成员变量的声明重新修改，如例 5-5 所示。

【例 5-5】 成员变量的私有化。

```
1   class CyberDog {
2       private String name;                //仿生狗的名字
3       private int batteryLevel;           //电池电量
4   }
```

由此可以看出，完整的成员变量声明应该由三部分组成，依次为访问修饰符、变量类型、变量名称。

访问修饰符用于控制哪些类可以访问该成员变量或方法。Java 共支持 4 种不同的访

问权限。

(1) default(即默认,无访问修饰符):在同一包内可见,不使用任何修饰符。

(2) private:在同一类内可见。

(3) public:对所有类可见。

(4) protected:对同一包内的类和所有子类可见。

目前,本小节只考虑 public 和 private,其他访问修饰符将在 5.8 节及第 6 章再进行讨论。因此,根据封装的原则,一般会将成员变量私有化,即设为 private。这意味着 name、batteryLevel 这些字段只能从 CyberDog 类的内部直接访问。

同成员变量一样,方法也需要设定访问修饰符,一般会将方法设为 public 允许外部代码直接调用。例如,为了能让外界访问私有属性,需要提供一些使用 public 修饰的公有方法,其中包括用于获取属性值的 getXxx()方法和设置属性值的 setXxx()方法。

下面将通过例 5-6 中 CyberDog 类低电量告警的案例展示类的封装的应用。

【例 5-6】 CyberDog 低电量告警功能。

```
1   class CyberDog {
2       private String name = "旺财";                    //仿生狗的名字
3       private int batteryLevel = 10;                   //电池电量
4       //公有的 getXxx()和 setXxx()方法
5       public String getName() {
6           return name;
7       }
8       public void setName(String dogName) {
9           name = dogName;
10      }
11      public int getBatteryLevel() {
12          return batteryLevel;
13      }
14      public void setBatteryLevel(int newBatteryLevel) {
15          //下面是对传入的参数进行检查
16          if (newBatteryLevel < 0 || newBatteryLevel > 100) {
17              System.out.println("设置电量不合法,将保持原值");
18          } else {
19              batteryLevel = newBatteryLevel;          //对属性赋值
20              System.out.println("设置电量成功");
21          }
22      }
23      public void checkBatteryStatus() {
24          if ( batteryLevel < 20) {
25              System.out.println(name + "当前电量为" + batteryLevel + "%,请注意
                充电!");
26          } else {
27              System.out.println(name + "当前电量为" + batteryLevel + "%,请放心
                使用!");
28          }
29      }
30  }
31  public class Example06 {
32      public static void main(String[] args) {
33          CyberDog myDog = new CyberDog();
34          myDog.setBatteryLevel(-10);
35          myDog.checkBatteryStatus();
36          myDog.setBatteryLevel(80);
```

```
37            myDog.checkBatteryStatus();
38        }
39    }
```


运行例 5-6，运行结果如下所示。

```
设置电量不合法,将保持原值
旺财当前电量为 10%,请注意充电!
设置电量成功
旺财当前电量为 80%,请放心使用!
```

在例 5-6 中，使用 private 关键字将属性 name 和 batteryLevel 声明为私有变量，并向外界提供了几个公有的方法，其中，getName 方法用于获取 name 属性的值，setName 方法用于设置 name 属性的值。同理，getBatteryLevel 和 setBatteryLevel 方法用于获取和设置 batteryLevel 属性的值。在 main 方法中创建 CyberDog 对象，并调用 setBatteryLevel 方法传入一个负数－10，在 setBatteryLevel 方法中对参数 batteryLevel 的值进行检查，由于当前传入的值小于 0，因此会打印“设置电量不合法”的信息，batteryLevel 属性没有被赋值，仍为默认初始值 10，最终显示“当前电量为 10%，请注意充电!”。同理，当调用 setBatteryLevel 方法传入 80 时，判断传入的值为非负数，则显示“设置电量成功”，表示充电成功。在 setBatteryLevel 方法中进行电量数值的校验，这样就保证了设置电量的合理性。

5.3 构造方法与对象的创建

从前面所学到的知识中可以发现，实例化一个类的对象后，如果要为这个对象中的属性赋值，则必须通过直接访问对象的属性或调用 setXxx 方法的方式才可以。如果需要在实例化对象的同时就为这个对象的属性进行赋值，可以通过构造方法来实现。构造方法是类的一个特殊成员，它会在类实例化对象时被自动调用。接下来学习构造方法的具体用法。

5.3.1 构造方法

在 Java 中，构造方法是一种特殊的方法，用于在创建对象时初始化对象的状态。在一个类中定义的方法如果同时满足以下三个条件，该方法称为构造方法。

(1) 方法名与类名相同。

(2) 在方法名的前面没有返回值类型的声明。

(3) 在方法中不能使用 return 语句返回一个值。

构造方法分为无参构造方法和有参构造方法。无参构造方法没有参数，通常用于提供类的默认配置。如果开发者没有为类定义任何构造方法，Java 编译器会自动提供一个无参构造方法。有参构造方法允许在创建对象时传递参数，用于自定义对象的初始状态。

下面将通过例 5-7 实现给 CyberDog 初始化参数来演示如何在类中定义构造方法。

【例 5-7】 构造方法的定义。

```
1   class CyberDog {
2       private String name;              //初始姓名
3       private int batteryLevel;         //电量
4       //无参构造方法,提供默认配置
```

```
5       public CyberDog() {
6           //初始化属性为默认值
7           name = "旺财";
8           batteryLevel = 100; //假设电量以百分比表示
9           System.out.println(name + "初始化完成," + "此时电量为" + batteryLevel);
10      }
11  }
12  public class Example07 {
13      public static void main(String[] args) {
14          CyberDog myDog = new CyberDog();
15      }
16  }
```

运行例 5-7,运行结果如下所示。

```
旺财初始化完成,此时电量为 100
```

在例 5-7 的 CyberDog 类中定义了一个无参的构造方法 CyberDog()。从运行结果可以看出,CyberDog 类中无参的构造方法被调用了。这是因为第 14 行代码在实例化 CyberDog 对象时会自动调用类的构造方法,new CyberDog()语句的作用除了会实例化 CyberDog 对象,还会调用构造方法 CyberDog()。

在一个类中除了定义无参的构造方法,还可以定义有参的构造方法,通过有参的构造方法可以实现对属性的赋值。下面将对例 5-7 进行改写,如例 5-8 所示。

【例 5-8】 有参构造方法和无参构造方法的定义。

```
1   class CyberDog {
2       private String name;                    //初始姓名
3       private int batteryLevel;               //电量
4       //无参构造方法,提供默认配置
5       public CyberDog() {
6           //初始化属性为默认值
7           name = "旺财";
8           batteryLevel = 100;                 //假设电量以百分比表示
9          System.out.println(name + "初始化完成," + "初始电量为" + batteryLevel);
10      }
11      //有参构造方法,使用自定义配置
12      public CyberDog(String cName, int cBatteryLevel) {
13          name = cName;
14          batteryLevel = cBatteryLevel;
15          System.out.println(name + "初始化完成," + "设置电量为" + batteryLevel);
16      }
17  }
18  public class Example08 {
19      public static void main(String[] args) {
20          CyberDog myDog = new CyberDog("富贵", 80);
21      }
22  }
```

运行例 5-8,运行结果如下所示。

```
富贵初始化完成,设置电量为 80
```

例 5-8 的 CyberDog 类中定义了有参的构造方法 CyberDog(String cName,int cBatteryLevel)。第 20 行代码中的 new CyberDog 会在实例化对象的同时调用有参的构造方法,并传入了名称(name)、电量(batteryLevel)。通过运行结果可以看出,CyberDog 对象在初始化后,其

name 属性已经被赋值为"富贵"。

5.3.2 对象的内存布局

5.2.5 节中介绍了对象如何创建，然而对象的创建并不仅限于编写几行代码并看到一个结果。在对象的生命周期中，内存管理是一个至关重要的环节，它影响着程序的性能和稳定性。本小节将会对 Java 中的对象创建进行更深层次的讲解，包括对象是如何在内存中布局的，以及对象是如何在这些内存区域中被创建和销毁的。

在对象创建和销毁时，主要关注以下两种内存空间。

(1) 栈(Stack)：局部变量存储的地方，每个线程都有自己的栈。

(2) 堆(Heap)：所有线程共享的内存区域，用于存储对象和数组。

下面通过创建 CyberDog 的对象，详细描述在 Java 中对象创建的过程。

```
1   CyberDog myDog = new CyberDog();
```

① new 创建对象：new CyberDog()用于创建 CyberDog 类的一个实例对象，此为对象创建的开始操作。

② 堆分配：Java 会在堆上为新 CyberDog 实例分配必要的内存空间。

③ 调用构造方法：CyberDog 的构造方法被调用，同时在堆上分配的内存空间里初始化 name 和 batteryLevel 等属性。

④ 返回引用：构造方法执行完成后，它会返回一个指向该对象的引用，这个引用实际上是对象在堆内存中的地址，可以把这个引用想象成一个指向 CyberDog 实例的指针。

⑤ 栈分配：CyberDog myDog 声明了一个 CyberDog 类型的变量 myDog，Java 会在栈上为 myDog 分配一个空间，用于存放④中返回的 CyberDog 实例的地址。

在内存中变量 myDog 和对象之间的引用关系，Java 对象内存分布如图 5-2 所示。

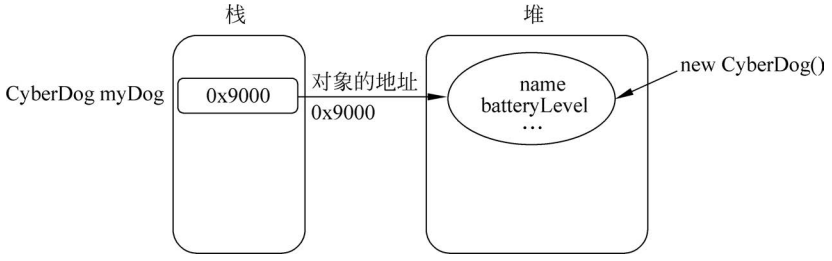

图 5-2 Java 对象内存分布

下面通过例 5-9 展示从内存分布的角度学习访问对象成员时的内存情况。

【例 5-9】 CyberDog 对象的创建。

```
1   public class Example09 {
2       public static void main(String[] args) {
3           CyberDog dog1 = new CyberDog("旺财", 20);      //创建第一个 CyberDog 对象
4           CyberDog dog2 = new CyberDog("富贵", 80);      //创建第二个 CyberDog 对象
5       }
6   }
```

运行例 5-9，运行结果如下所示。

```
旺财初始化完成,设置电量为 20
富贵初始化完成,设置电量为 80
```

例 5-9 中，dog1、dog2 分别引用了 CyberDog 类的两个实例对象。从运行结果可以看出，dog1 和 dog2 对象打印的 name 值不相同。这是因为 dog1 对象和 dog2 对象是两个完全独立的个体，它们分别拥有各自的 name 属性。程序运行期间 dog1、dog2 引用的对象在内存中的状态如图 5-3 所示。

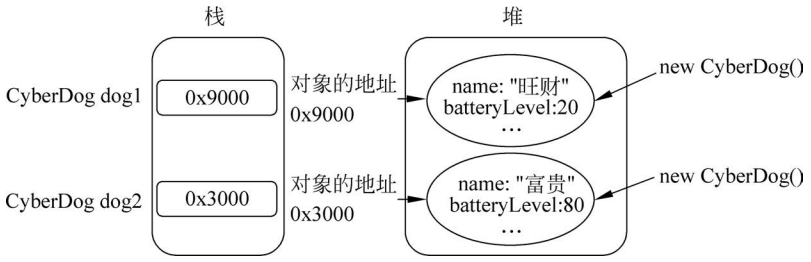

图 5-3　例 5-9 中对象在内存中的状态

当对象被实例化后，在程序中可以通过对象的引用变量来访问该对象的成员。需要注意的是，当没有任何变量引用这个对象时，它将成为垃圾对象，不能再被使用。接下来通过两段程序代码来分析对象是如何成为垃圾对象的。

第一段程序代码：

```
1   {
2       CyberDog dog1 = new CyberDog();      //创建第一个 CyberDog 对象
3       //...
4   }
```

上面的代码中使用变量 dog1 引用了一个 CyberDog 类型的对象，当这段代码运行完毕时，变量 dog1 就会超出其作用域而被销毁，这时 CyberDog 类型的对象没有被任何变量引用，而变成垃圾对象。

第二段程序代码如例 5-10 所示。

【例 5-10】 对象的销毁。

```
1   public class Example10 {
2       public static void main(String[] args) {
3           CyberDog dog = new CyberDog();       //创建 CyberDog 对象
4           dog.run();                           //调用 run()方法
5           dog = null;                          //将 CyberDog 对象置为 nu11
6           dog.run();
7       }
8   }
```

运行例 5-10，运行结果如下所示。

```
旺财初始化完成,设置电量为 80 %
正在奔跑,消耗 15 % 电量
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "CyberDog.run()"
because "dog" is null
        at Example10.main(Example10.java:6)
```

在例 5-10 中，创建了一个 CyberDog 类的实例对象，并两次调用了该对象的 run()方法。第一次调用 run()方法时可以正常打印，但在第 5 行代码中将变量 dog 的值置为 null，当再次调用 run()方法时抛出了空指针异常（在第 7 章中会详细解释，此处理解为错误即可）。在 Java 中，null 是一种特殊的常量，当一个变量的值为 null 时，则表示该变量不指向

任何一个对象。当把变量 dog 置为 null 时，被 dog 所引用的 CyberDog 对象就会失去引用，成为垃圾对象。

5.4 参数传递

函数的参数传递有两种，按值传递和按引用传递。按值传递表示函数接收的是调用者提供的值，按引用传递表示函数接收的是调用者提供的变量地址。需要注意的是，函数可以修改按引用传递的参数对应的变量值，但不可以修改按值传递的参数对应的变量值，这是两者的最大区别。

5.4.1 基本数据类型参数的传值

对于 Java 的 8 种基本数据类型（如 int、double、char 等），参数传递是按值进行的。这意味着当一个基本数据类型的变量作为参数传递给方法时，方法接收到的是原始值的一个复制，方法内部对参数的修改不会影响到原始变量。

下面通过例 5-11 定义 CyberDog 类实现添加充电的功能。

【例 5-11】 基本数据类型参数的传值。

```
1   public class Example11 {
2       private static void charge(int bLevel, int delta) {
3           if (bLevel + delta <= 100) {
4               bLevel += delta;
5               System.out.println("已充入" + delta + "%电量");
6           } else {
7               bLevel = 100;
8               System.out.println("充电已满.");
9           }
10      }
11      public static void main(String[] args) {
12          CyberDog myDog = new CyberDog("旺财", 20);
13          System.out.println("充电前电量: " + myDog.getBatteryLevel() + "%.");
14          charge(myDog.getBatteryLevel(), 30); //尝试给 CyberDog 充电 30%
15          System.out.println("充电后电量: " + myDog.getBatteryLevel() + "%.");
16      }
17  }
```

例 5-11 中的代码定义了一个 charge 方法，方法内部将传入的参数 batteryLevel 的值增加 delta 数值的电量。

运行例 5-11，运行结果如下所示。

```
充电前电量: 20%
已充入 30%电量
充电后电量: 20%
```

从上面的运行结果来看，charge 方法中 delta 的值是 30，charge 方法执行结束后，CyberDog 的电量依然是 20。

由上可以看出，charge 方法里的 bLevel，并不是 main 方法里 myDog 的成员变量 batteryLevel，charge 方法内部对 bLevel 的值的修改并没有改变实际参数 myDog.batteryLevel 的值，改变的只是 charge 方法中 bLevel 的值，因为 charge 方法中的 bLevel 只

是 myDog.batteryLevel 的复制品。

因此，Java 中基本数据类型的参数在进行传递时，是按值传递的。

5.4.2 引用数据类型参数的传值

前面看到的只是基本数据类型的参数，那如果参数是一个对象，结果又是怎样的？仍然是为 CyberDog 类添加充电的功能，如例 5-12 所示。

【例 5-12】 引用数据类型参数的传值。

```
1   public class Example12 {
2       private static void charge(CyberDog innerDog, int delta) {
3           if (innerDog.getBatteryLevel() + delta <= 100) {
4               innerDog.setBatteryLevel(innerDog.getBatteryLevel() + delta);
5               System.out.println("已充入" + delta + "%电量");
6           } else {
7               innerDog.setBatteryLevel(100);
8               System.out.println("充电已满.");
9           }
10      }
11      public static void main(String[] args) {
12          CyberDog myDog = new CyberDog("旺财", 20);
13          System.out.println("充电前电量: " + myDog.getBatteryLevel() + "%.");
14          charge(myDog, 30); //尝试给 CyberDog 充电 30%
15          System.out.println("充电后电量: " + myDog.getBatteryLevel() + "%.");
16      }
17  }
```

在例 5-12 的第 14 行代码中，charge 方法传入了 myDog 对象。运行例 5-12，运行结果如下所示。

```
充电前电量: 20%
设置电量成功
已充入 30%电量
充电后电量: 50%
```

经过 charge 方法执行后，innerDog 的电量 batteryLevel 被改变了。分析上述代码，main 方法中的 myDog 是一个引用，它保存了 CyberDog 对象的地址值，当把 myDog 的值赋给 charge 方法的 innerDog 形参后，即让 charge 方法的 innerDog 形参也保存了这个地址值，因此也会引用到堆内存中的 CyberDog 对象。程序运行期间 myDog、innerDog 引用的对象在内存中的状态，如图 5-4 所示。

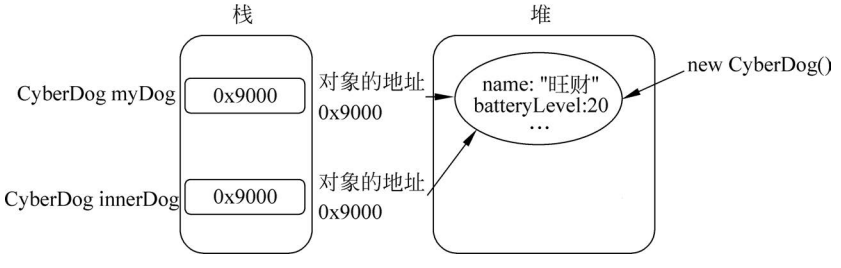

图 5-4 例 5-11 中对象在内存中的状态

【小提示】 结合生活中的场景，深入理解值传递和引用传递。

你有一个房子，当你的朋友想要去你家的时候，你给他一个你的房子的复制品（或者一个房子的模型）。你的朋友可以进入这个复制的房子，但他们对复制品做的任何改变都不会影响到你原来的那个房子，这就是值传递。

你有一个房子，当你的朋友想要去你家的时候，你直接把钥匙给他了，这就是引用传递。这种情况下你的朋友可以进入你家，看电视（查询操作）、改变你的家具位置（修改操作）、变卖你的家电（删除操作）等。

综上所述，在 Java 的方法中，传递基本数据类型的时候是值传递，这意味着传递的是值的复制。在传递引用数据类型的时候是引用传递，这意味着传递的是引用值（地址），所有引用都指向同一个对象。

5.5 方法的重载

在 Java 编程语言中，重载（Overloading）是一种允许根据输入参数的数量、类型或顺序来定义多个同名方法的机制，它可以为同一个行为提供多种不同的实现方式。重载方法可以对不同的输入数据执行相同的操作，但具体的行为会根据输入参数的不同而有所区别，这不仅提高了代码的可读性，还增加了方法的灵活性。

5.5.1 重载的特点

1. 方法名相同

被重载的方法必须具有相同的方法名。例如，bark 方法可以被重载，以适应不同的使用场景，但是方法的名字必须一致，如例 5-13 所示。

【例 5-13】 方法的重载。

```
1   class CyberDog {
2       private String name;                    //仿生狗的名字
3       private int batteryLevel;               //电池电量
4       //其他成员变量及方法...
5       public void bark() {
6           System.out.println("woof!");
7       }
8       public void bark(int numberOfTimes) {
9           for (int i = 0; i < numberOfTimes; i++) {
10              System.out.println("汪!");
11          }
12      }
13  }
14  public class Example13 {
15      public static void main(String[] args) {
16          CyberDog myDog = new CyberDog();
17          myDog.bark();
18          myDog.bark(3);
19      }
20  }
```

运行例 5-13，运行结果如下所示。

```
旺财初始化完成,初始电量为 100
woof!
汪!
汪!
汪!
```

2. 参数列表不同

可以是参数的数量不同、参数的类型不同,或者参数的排列顺序不同。例如,move 方法被重载了三次,每次接收不同的参数组合,以满足不同的移动需求。这样,当需要调用移动方法时,可以根据实际情况选择最合适的方法重载,如例 5-14 所示。

【例 5-14】 利用方法重载实现不同方法的调用。

```
1   class CyberDog {
2       private String name;                    //仿生狗的名字
3       private int batteryLevel;               //电池电量
4       //其他成员变量及方法...
5       //第一种重载:根据方向和速度移动
6       public void move(String direction, double speed) {
7           System.out.println("向" + direction + "方向,以每小时" + speed + "公里的速度
            移动.");
8       }
9       //第二种重载:根据方向、速度和持续时间移动
10      public void move(String direction, double speed, int duration) {
11          System.out.println("向" + direction + "方向,以每小时" + speed + "公里的速
            度,持续移动" + duration + "秒");
12      }
13      //第三种重载:直接移动到指定坐标
14      public void move(double newX, double newY) {
15          //实现坐标移动逻辑
16          System.out.println("移动到新坐标: (" + newX + ", " + newY + ")");
17      }
18  }
19  public class Example14 {
20      public static void main(String[] args) {
21          CyberDog myDog = new CyberDog();
22          myDog.move("东", 2.0);
23          myDog.move("南", 5.0, 5);
24          myDog.move(10.0, 20.0);
25      }
26  }
```

运行例 5-14,运行结果如下所示。

```
旺财初始化完成,初始电量为 100
向东方向,以每小时 2.0 公里的速度移动.
向南方向,以每小时 5.0 公里的速度,持续移动 5 秒
移动到新坐标: (10.0, 20.0)
```

5.5.2 重载的注意事项

1. 返回类型不作为区分条件

返回类型并不是方法重载的条件之一,如下面的例子,尽管方法名 fetch 相同,返回类型不同,会导致编译错误,因为编译器无法根据返回类型来区分这两个方法。

```
class CyberDog {
    //其他成员变量及方法…
    public String fetch() {
        return "Fetched item details";
    }
    public int fetch() {
        return 42;      //返回一个整数值
    }
}
```

2. 访问修饰符不作为区分条件

访问修饰符定义了方法的可见性，而不是用来区分重载的方法。如下面的例子，尽管访问控制符不同，但 Java 编译器仍会把这两个 displayInfo 方法作为相同的方法，导致编译错误。

```
class CyberDog {
    //其他成员变量及方法…
    //公共访问修饰符
    public void displayInfo() {
    }
    //受保护的访问修饰符
    protected void displayInfo() {
    }
}
```

3. 构造方法的重载

与普通方法一样，构造方法也可以重载，在一个类中可以定义多个构造方法，只要每个构造方法的参数类型或参数个数不同即可。如 5.3.1 节中例 5-8 所示，在创建对象时，可以通过调用不同的构造方法为不同的属性赋值。

5.6 this 关键字

在 Java 编程语言中，this 关键字是一个特殊的关键字，每当创建一个对象时，Java 都会自动为该对象生成一个 this 关键字引用，用于访问该对象的状态和行为。this 关键字的使用非常广泛，它不仅可以访问当前对象的成员变量和方法，还可以在构造方法中区分同名的局部变量和成员变量，或者在方法重载时明确指定调用哪个对象的方法。

5.6.1 this 关键字调用成员变量

在例 5-8 中，有参构造方法中使用的是 cName、cBatteryLevel，而成员变量使用的是 name、batteryLevel，从而导致程序的可读性很差。因此需要将一个类中表示姓名、电量的变量进行统一的命名，如都声明为 name。但是，这样会导致成员变量和局部变量的名称冲突，在方法中将无法访问成员变量 name。

为了解决这个问题，Java 中提供了一个 this 关键字，用于在方法中访问对象的其他成员，从而解决与局部变量名称冲突问题。下面将例 5-8 的有参构造方法进行重写，具体示例如例 5-15 所示。

【例 5-15】 利用关键字 this 调用成员变量。

```
1   class CyberDog {
2       private String name;            //仿生狗的名字
3       private int batteryLevel;       //电池电量
4       //其他成员变量及方法...
5       //有参构造方法,使用自定义配置
6       public CyberDog(String name, int batteryLevel) {
7           this.name = name;
8           this.batteryLevel = batteryLevel;
9           System.out.println(name + "初始化完成," + "设置电量为" + batteryLevel);
10      }
11      //其他方法
12  }
```

在例 5-15 的代码中,构造方法的参数被定义为 name,它是一个局部变量,在类中还定义了一个成员变量,名称也是 name。在构造方法中如果使用 name,则是访问局部变量,但如果使用 this.name 则是访问成员变量。

5.6.2 this 关键字调用成员方法

this 关键字不仅是对当前对象的引用,还可以用来调用当前对象的成员方法。假设需要给 CyberDog 增加一个助跑跳跃的功能,那么要在类 CyberDog 中定义一个方法 runAndJump,其具体的实现可以直接调用已有的 run()和 jump()方法,如例 5-16 所示。

【例 5-16】 利用关键字 this 调用成员方法。

```
1   class CyberDog {
2       private String name;                //仿生狗的名字
3       private int batteryLevel;           //电池电量
4       //其他成员变量及方法...
5       //助跑跳跃
6       public void runAndJump() {
7           this.run();
8           this.jump();
9           System.out.println(name + "完成助跑跳跃...");
10      }
11  }
12  public class Example16 {
13      public static void main(String[] args) {
14          CyberDog myDog = new CyberDog();
15          myDog.runAndJump();
16      }
17  }
```

运行例 5-16,运行结果如下所示。

```
旺财初始化完成,初始电量为 100
旺财正在奔跑,消耗 15 % 电量.
旺财正在跳跃,消耗 15 % 电量...
旺财完成助跑跳跃...
```

在上面的 runAndJump()方法中,使用 this 关键字调用 run()和 jump()方法。注意,此处的 this 关键字可以省略,也就是说例 5-16 中第 7 行代码写成 this.run()和 run(),效果是完全一样的。

5.6.3 this 关键字调用构造方法

构造方法是在实例化对象时被 Java 虚拟机自动调用的，在程序中不能像调用其他方法一样去调用构造方法，如 this.CyberDog()，这就是错误的。但是可以在一个构造方法中直接使用 this([参数 1,参数 2,...])的形式来调用其他的构造方法。接下来通过例 5-17 改写构造方法 CyberDog()来演示 this 关键字的用法。

【例 5-17】 利用关键字 this 调用构造方法。

```
1   class CyberDog {
2       private String name;                    //仿生狗的名字
3       private int batteryLevel;               //电池电量
4       //其他成员变量及方法...
5       //无参构造方法,提供默认配置
6       public CyberDog() {
7           this("旺财", 100);
8           System.out.println(name + "初始化完成," + "初始电量为" + batteryLevel);
9       }
10      //有参构造方法,使用自定义配置
11      public CyberDog(String name, int batteryLevel) {
12          this.name = name;
13          this.batteryLevel = batteryLevel;
14          System.out.println(name + "初始化完成," + "设置电量为" + batteryLevel);
15      }
16  }
17  public class Example17 {
18      public static void main(String[] args) {
19          CyberDog myDog = new CyberDog();
20      }
21  }
```

运行例 5-17，运行结果如下所示。

```
旺财初始化完成,设置电量为 100
```

例 5-17 中第 19 行代码在实例化 CyberDog 对象时，调用了无参的构造方法，在该方法中通过 this 关键字调用了有参的构造方法，因此运行结果中显示有参构造方法被调用了。

在使用 this 关键字调用类的构造方法时，应注意以下两点。

(1) 只能在构造方法中使用 this 关键字调用其他的构造方法，不能在成员方法中使用。

(2) this 关键字调用其他的构造方法时，这条语句必须作为构造方法的第一行代码出现，此外，每个构造方法中只能包含一次 this 关键字调用，不能多次使用。

下面的写法是非法的。

```
1   public class CyberDog {
2       private String name;
3       private int batteryLevel;
4       //正确的 this 关键字调用方式
5       public CyberDog() {
6           this("DefaultName", 100);       //调用带参数的构造方法
7       }
8       //带参数的构造方法
9       public CyberDog(String name, int batteryLevel) {
10          this.name = name;
```

```
        this.batteryLevel = batteryLevel;
        //其他初始化代码...
    }
    //错误的 this 关键字调用方式:不是构造方法的第一行语句
    public CyberDog(String name) {
        System.out.println("Initializing with name only.");
        this(name, 100); //非法:this 关键字调用不是第一行
    }
    //错误的 this 关键字调用方式:不能在成员方法中调用 this
    public void run() {
        this(name, 100);
    }
}
```

不能在一个类的两个构造方法中使用 this 关键字互相调用,下面的写法编译也会报错。

```
class CyberDog {
    public CyberDog() {
        this("DefaultName", 100);       //调用带参数的构造方法
    }
    public CyberDog(String name, int batteryLevel) {
        this();
    }
}
```

5.7 static 关键字

在 Java 中,定义了一个 static 关键字,它为类和方法赋予了特殊的属性和行为。static,意味着“静态的”或“全局的”,当一个成员(方法或变量)被声明为 static 时,它就与类本身绑定在一起,而不是与类的任何特定实例绑定。这使得 static 成员可以在没有创建类的实例的情况下被访问和使用,为程序设计提供了极大的灵活性和便利。

5.7.1 静态变量

类的定义不仅是对事物特征和行为的描述,更是一种创建具体实例的模板。在 Java 中,使用 new 关键字创建类的实例时,系统会为每个新对象分配独立的内存空间,以存储其独特的数据。然而在某些情况下,内存中只需要为某些特定的数据保留一份副本即可,类的所有实例都共享这份数据。这种做法不仅节省内存,还能保证数据的一致性。例如在 CyberDog 的设计中,硬件规格基本上是统一的,如型号、电机数量、传感器类型等,因此可以将这些属性定义为静态属性,所有实例共享同一组属性,如例 5-18 所示。

【例 5-18】 静态变量的定义与使用。

```
class CyberDog {
    private String name;                                    //仿生狗的名字
    private int batteryLevel;                               //电池电量
    private static String model = "CyberDog V1.0";          //型号
    private static int motorCount = 4;                      //电机数量
    private static String sensorType = "Infrared";          //传感器类型
    //其他成员变量及方法...
}
```

```
9   public class Example18 {
10      public static void main(String[] args) {
11          CyberDog myDog1 = new CyberDog();
12          CyberDog myDog2 = new CyberDog("富贵", 100);
13          //访问静态属性,获取硬件规格信息
14          System.out.println("CyberDog 型号: " + CyberDog.model);
15          System.out.println("Dog1 的型号: " + myDog1.model);
16          System.out.println("Dog2 的型号: " + myDog2.model);
17          CyberDog.model = "CyberDog V2.0";
18          System.out.println("Dog1 的新型号: " + myDog1.model);
19          System.out.println("Dog2 的新型号: " + myDog2.model);
20      }
21  }
```

运行例 5-18,运行结果如下所示。

```
旺财初始化完成,设置电量为 100
富贵初始化完成,设置电量为 100
CyberDog 型号: CyberDog V1.0
Dog1 的型号: CyberDog V1.0
Dog2 的型号: CyberDog V1.0
Dog1 的新型号: CyberDog V2.0
Dog2 的新型号: CyberDog V2.0
```

在 CyberDog 类中,第 4～6 行代码使用 static 关键字来修饰成员变量,则这些变量就被称作静态变量。静态变量被所有实例共享,使用“类名.变量名”的形式来访问。

例如,model 变量用于表示 CyberDog 的型号,它被所有的实例所共享。由于 model 是静态变量,因此可以直接使用 CyberDog.model 的方式进行调用,也可以通过 CyberDog 的实例对象进行调用,如 myDog1.model。第 17 行代码将变量 model 修改为“CyberDog V2.0”,通过运行结果可以看出,对象 myDog1 和 myDog2 的 model 属性均变为“CyberDog V2.0”。具体内存中的分配情况如图 5-5 所示。

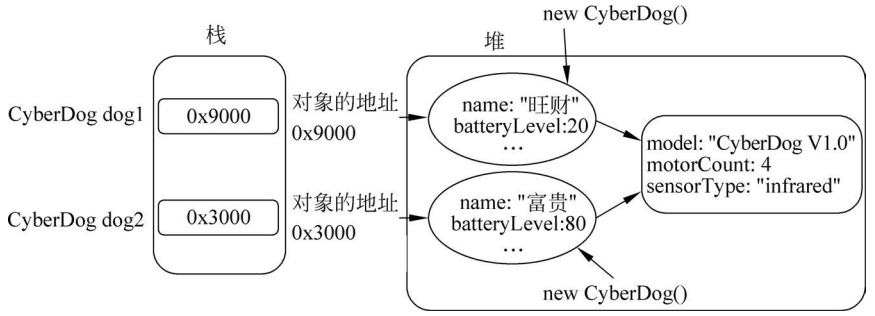

图 5-5　内存中的分配情况

注意:static 关键字只能用于修饰成员变量,不能用于修饰局部变量,否则编译会报错,例如下面的代码是非法的。

```
1   class CyberDog {
2       public run() {
3           static double speed = 10.0;          //这行代码是非法的,编译会报错
4       }
5   }
```

5.7.2 静态方法

静态方法，也称为类方法，是与类本身相关联的，而不是与类的任何特定实例相关联。这意味着静态方法可以在没有创建类的实例的情况下被调用，它们通过类名直接访问。同样，静态方法也是使用 static 关键字声明，并且它们不能访问类的非静态成员，因为非静态成员需要实例化后才能使用，所以静态方法通常用于实现与类实例状态无关的功能。

例如，可以将 CyberDog 类的欢迎语和唤醒词设计成静态方法，如例 5-19 所示。

【例 5-19】 静态方法的定义与使用。

```
class CyberDog {
    private String name;                //仿生狗的名字
    private int batteryLevel;           //电池电量
    //静态方法:CyberDog 的欢迎语
    public static void welcome() {
        System.out.println("主人你好,欢迎使用 CyberDog!");
    }
    //静态方法:CyberDog 的唤醒词
    public static void wakeUp(String word) {
        if (word.equals("小汪同学")) {
            System.out.println("主人,我在!");
        }
    }
    //其他成员变量及方法...
}
public class Example19 {
    public static void main(String[] args) {
        CyberDog.welcome();
        CyberDog.wakeUp("小汪同学");
    }
}
```

运行例 5-19，运行结果如下所示。

```
主人你好,欢迎使用 CyberDog!
主人,我在!
```

5.8 包与权限访问

在 Java 编程中，随着项目规模的扩大，管理越来越多的类成为一个挑战。这时，“包”(Package)的概念就显得尤为重要。包是 Java 中的一个命名空间，用于逻辑上组织类和接口，使得大型项目的维护和导航变得更加容易。

5.8.1 包的声明

包提供了一个命名空间，可以避免类的名称冲突。在 Java 中，包的声明是通过 package 关键字完成的，通常位于 Java 源文件的最顶部。声明包时，通常使用点(.)分隔的路径，这与文件系统的目录结构相对应。

假设 CyberDog 类，它属于一个名为 com.example.robots 的包，意味着它是 robots 模块的一部分，可以按照如下方式定义：

```
package com.example.robots;        //声明包
public class CyberDog {
    //类定义...
}
```


5.8.2 类的导入

要在 Java 程序中使用其他包中的类，可以使用 import 语句，具体示例如下。

```
import com.example.robots.CyberDog;     //导入 CyberDog 类
public class Main {
    public static void main(String[] args) {
        CyberDog myDog = new CyberDog();
    }
}
```

在上面的例子中，import 语句允许 Main 类使用 CyberDog 类，而无须每次都指定完整的包路径。

5.8.3 包的命名规范

包的名称采用反域名命名法。例如，QQ 邮箱的域名是 mail.qq.com，那么 QQ 邮箱的研发工程师在开发过程中使用的包名就是 com.qq.mail。而且，包名称通常使用小写字母，以避免与 Java 关键字或类名冲突

5.8.4 包的作用域

在 5.2.6 节中，介绍了 Java 中 4 种不同的访问修饰符 default、private、public、protected，并对 private、public 做了详细介绍，而 default（即默认，无访问修饰符）的访问级别，也称为包访问级别（package-private），是当类成员没有指定访问修饰符时的默认行为。这意味着这些成员仅在它们所属的包内可见，对于包外的类则是不可见的。

```
package com.example.robots;                //定义包
class CyberDog {
    String breed;                          //默认访问权限(package-private)
    void bark() {
        System.out.println("Woof! Woof!");
    }
}
```

在上述例子中，CyberDog 类的 breed 变量和 bark 方法具有 default 访问权限，它们可以在 com.example.robots 包内的其他类中被访问，但对其他包不可见。下面的写法尝试在 com.example.other 包中的一个类 OtherClass 访问 CyberDog 类的 breed 成员，将会导致编译错误，因为 breed 只在 com.example.robots 包内可见。

```
//文件位置:com/example/other/OtherClass.java
package com.example.other;                 //定义另一个包
import com.example.robots.CyberDog;        //尝试导入 CyberDog 类
public class OtherClass {
    public void interactWithDog() {
        CyberDog dog = new CyberDog();     //可以实例化 CyberDog
        //下面的代码将导致编译错误,因为 breed 是 package-private
```

```
8            System.out.println(dog.breed);
9        }
10   }
```

在这个 OtherClass 类中，尽管能够导入并实例化 CyberDog 类，但由于 breed 成员具有默认访问权限，尝试访问 dog.breed 将导致编译时错误，原因就是 breed 在 OtherClass 包中并不可见。

5.9 示例学习

以下通过设计一个 CyberDog 的充电站，深入了解类与对象的概念，并将本章的知识点进行融会贯通。

【例 5-20】 设计充电站类 ChargingStation，要求如下：

(1) 成员变量：充电端口数量、当前正在充电的 CyberDog 对象列表。

(2) 静态成员变量：所有充电站的总充电次数。

(3) 方法：充电(可以选择充满自停或充入任意合法电量)、停止充电。

(4) 静态方法：获取当前所有充电站的总充电次数。

```
1    class ChargingStation {
2        private int portCount;                                    //充电端口数量
3        private List < CyberDog > chargingDogs;                   //当前正在充电的 CyberDog 列表
4        private static int totalCharges;                          //静态变量存储总充电次数
5        public ChargingStation() {
6            this.portCount = 5;                                   //默认充电端口数量为 5
7            this.chargingDogs = new ArrayList <>();
8        }
9        public ChargingStation(int portCount) {
10           this.portCount = portCount;
11           this.chargingDogs = new ArrayList <>();
12       }
13       //为 CyberDog 充电，充满自停
14       public void charge(CyberDog dog) {
15           if (dog == null) {
16               System.out.println("CyberDog 不存在,无法为其充电.");
17               return;
18           }
19           if (chargingDogs.size() < portCount) {
20               chargingDogs.add(dog);
21               System.out.println(dog.getName() + "开始充电...");
22               incrementTotalCharges();
23               //TODO: 充电逻辑
24               dog.setBatteryLevel(100); //充满电
25           } else {
26              System.out.println("充电站已满,无法为" + dog.getName() + "充电.");
27           }
28       }
29       //CyberDog 充电，设置充电量
30       public void charge(CyberDog dog, int chargeAmount) {
31           if (dog == null) {
32               System.out.println("CyberDog 不存在,无法为其充电.");
33               return;
34           }
```

```
35          if (chargingDogs.size() < portCount) {
36              chargingDogs.add(dog);
37              System.out.println(dog.getName() + "开始充电...");
38              incrementTotalCharges();
39              dog.setBatteryLevel(dog.getBatteryLevel() + chargeAmount <= 100 ? dog.
                getBatteryLevel() + chargeAmount : 100); //充电
40          } else {
41              System.out.println("充电站已满,无法为" + dog.getName() + "充电.");
42          }
43      }
44      //CyberDog 停止充电
45      public void stop(CyberDog dog) {
46          if (chargingDogs.contains(dog)) {
47              chargingDogs.remove(dog);
48              System.out.println(dog.getName() + "停止充电.");
49          } else {
50              System.out.println(dog.getName() + "未在充电站充电.");
51          }
52      }
53      //静态方法,用于获取当前所有充电站的总充电次数
54      public static void getTotalCharges() {
55          //静态变量存储总充电次数
56           System.out.println("充电站总充电次数: " + totalCharges);
57      }
58      //静态方法,用于记录充电次数
59      private static void incrementTotalCharges() {
60          totalCharges++;
61      }
62  }
63  public class Example20 {
64      public static void main(String[] args) {
65          CyberDog myDog1 = new CyberDog();
66          myDog1.run();
67          myDog1.rollOver();
68          //myDog1 奔跑、打滚后电量下降
69          myDog1.checkBatteryStatus();
70          System.out.println("-------------------------------------");
71          CyberDog myDog2 = new CyberDog("富贵", 20, 1.0, new double[] { 0.0, 0.0 },
            "poweredOff");
72          //myDog2 出厂后即电量不足
73          System.out.println("-------------------------------------");
74          ChargingStation station = new ChargingStation();
75          //将两只 CyberDog 送入充电站
76          station.charge(myDog1);
77          station.charge(myDog2, 50);
78          System.out.println("-------------------------------------");
79          //充电等待中...
80          station.stop(myDog1);
81          station.stop(myDog2);
82          System.out.println("-------------------------------------");
83          //检查电量
84          myDog1.checkBatteryStatus();
85          myDog2.checkBatteryStatus();
86      }
87  }
```

运行例 5-20,运行结果如下所示。

```
旺财初始化完成,设置电量为 100
旺财正在奔跑,消耗 15 % 电量.
旺财正在打滚,消耗 8 % 电量...
当前电量为 77 %,请放心使用!
富贵初始化完成,设置电量为 20
旺财开始充电...
设置电量成功
富贵开始充电...
设置电量成功
旺财停止充电.
富贵停止充电.
当前电量为 100 %,请放心使用!
当前电量为 70 %,请放心使用!
充电站总充电次数: 2
```

例 5-20 定义了一个名为 ChargingStation 的类,用于模拟充电站。对 ChargingStation 类进行解析如下所示。

(1) 成员变量。

portCount:表示充电站的充电端口数量。

chargingDogs:存储当前正在充电的 CyberDog 对象的列表。

totalCharges:一个静态变量,用于记录所有充电站的总充电次数。

(2) 构造方法。

无参构造方法,将 portCount 默认设置为 5,表示可以接入 5 个 CyberDog,并初始化 chargingDogs 列表。

有参构造方法,允许设置充电端口的数量 portCount,并初始化 chargingDogs 列表。

(3) 方法。

charge:通过重载 charge 方法,实现充满自停和充入任意电量的功能。charge(CyberDog dog)为 CyberDog 对象充电至满电量。charge(CyberDog dog, int chargeAmount)为 CyberDog 对象充入指定的电量。

stop(CyberDog dog):对指定的 CyberDog 对象停止充电,并从 chargingDogs 列表中移除。

getTotalCharges():返回所有充电站的总充电次数。

incrementTotalCharges():私有静态方法,表示增加总充电次数,类内调用即可,无须对外暴露,因此设置为私有方法。

(4) main()方法。

main()方法用于演示如何使用 ChargingStation 类,创建两个 CyberDog 对象,分别模拟不同的场景(消耗电量后)。创建一个 ChargingStation 对象,并使用它为两个 CyberDog 充电。

5.10 本章小结

本章主要讲解了 Java 面向对象中类与对象的内容。介绍了类的声明、对象的创建,以及如何通过对象来调用方法和访问属性。同时,也介绍了访问修饰符的重要性,它确保了数据的安全性和类的封装性。

面向对象编程的核心在于将现实世界的事物抽象成代码中的类，并通过对象来模拟现实世界中的行为。这种编程范式提高了代码的可读性、可维护性和可扩展性。

习　题　5

一、填空题

1. 类的封装性可以通过使用_____访问修饰符来实现。

2. Java 中如果类中的成员变量有_____变量，那所有的对象为此类变量分配相同的一处内存空间。

3. 方法重载是指在同一个类中可以有多个同名方法，它们具有相同的方法名但_____不同。

4. this 关键字用于引用当前对象的_____。

5. Java 的包声明是通过使用_____关键字来完成的。

二、选择题

1. 下列选项中不是 Java 访问修饰符的是(　　)。

A. public　　B. private　　C. protected　　D. this

2. 关于以下程序代码的说明正确的是(　　)。

```
1   public class HasStatic {
2       private static int x = 100;
3       public static void main(String args[]) {
4           HasStatic hsl = new HasStatic();
5           hsl.x++;
6           HasStatic hs2 = new HasStatic();
7           hs2.x++;
8           hsl = new HasStatic();
9           hsl.x++;
10          HasStatic.x--;
11          System.out.println(" x=" + x);
12      }
13  }
```

A. 程序通过编译，输出结果为：x=102

B. 程序通过编译，输出结果为：x=103

C. 10 行不能通过编译，因为 x 是私有静态变量

D. 5 行不能通过编译，因为引用了私有静态变量

3. 以下说法正确的是(　　)。

A. public 关键字只能修饰类名

B. public 关键字只能修饰方法

C. public 关键字只能修饰成员变量

D. 以上说法都不对

4. 在 Java 中，假设 A 有构造方法 A(int a)，则在类 A 的其他构造方法中调用该构造方法和语句格式应该为(　　)。

A. this.A(x)　　B. this(x)　　C. super(x)　　D. A(x)

三、简单题

1. 解释类的封装性，并给出一个封装类的示例。

2. 描述构造方法的作用，并说明如何在 Java 中调用它们。

3. 解释方法重载的概念，并给出一个包含方法重载的类的示例。

4. 阐述 this 关键字的作用，并给出使用 this 的代码示例。

5. 描述 static 关键字的用途，并给出一个使用 static 的类的例子。

6. 解释包在 Java 中的作用，并说明如何声明和使用它们。

四、编程题

1. 定义一个 Circle(圆)类，提供显示圆周长、面积的方法。

2. 创建一个 Calculator(计算器)类，在其中定义两个私有变量作为操作数，再定义四个方法分别实现和、差、商、积。

3. 按照以下要求设计类：

(1) 编写一个名为 Car 的类，包含品牌(brand)、型号(model)和颜色(color)三个属性，以及一个构造方法和相应的 getters 和 setters。

(2) 在 Car 类中添加两个重载的 display 方法，一个打印车辆的基本信息，另一个还额外打印车辆的生产年份(year)。

(3) 在 Car 类中，使用 this 关键字来区分局部变量和实例变量。

(4) 在 Car 类中添加一个 static 变量 count，用于跟踪创建的 Car 对象的数量。同时，添加一个 static 方法来获取当前创建的 Car 对象总数。

(5) 将 Car 类放入名为 com.example.vehicles 的包中，并编写一个测试类 CarTest，位于同一个包中，用于创建 Car 对象并调用其方法。

(6) 修改 Car 类的属性访问权限，使它们只能通过类的方法被访问和修改，体现封装的概念。