第3章 类的设计

CHAPTER 3

预备知识

第1章 Java类
第2章 从已有类创建新类
附录D 文档与程序设计风格

目的

在学习过本章之后，读者应该能够：

- 描述封装、信息隐藏和数据抽象
- 为具有先决条件与后继条件的方法编写说明
- 编写一个类的Java接口
- 在程序设计过程中选择适宜的类与方法，包括那些可能已经编写好的类

面向对象的程序设计包含三个设计理念：封装、继承和多态性。前两章已经讨论了继承与多态性；本章将在之前关于类的讨论基础上介绍封装。封装是一种在类的设计过程中将实现的细节隐藏起来的方法。本章中将继续强调在具体实现一个方法之前先说明它的行为以及将这些说明在程序中用注释表达的重要性。

3.1 封装

如果想要学习驾驶汽车，关于汽车的最有用的说明是什么呢？显然不是关于它的引擎如何在一个周期中吸入空气与汽油、点燃空气与汽油的混和物并且排出废气的说明。这些细节对于学习驾驶而言是不必要的，事实上，这些细节反倒会成为学习的障碍。如果想要学习驾驶汽车，关于汽车最有用的说明如下：

- 若脚踩加速踏板，则汽车将行驶得更快。
- 若脚踩刹车踏板，则汽车将减速直至最终停止。
- 若将方向盘向右转动，则汽车将向右转。
- 若将方向盘向左转动，则汽车将向左转。

正如不需要告诉想驾驶汽车的人引擎的工作机制一样，也不需要告诉使用软件的人有关该软件的Java实现的所有精确细节。同样地，假如你已经为程序员创建了一个可供他在程序中使用的软件模块，则该模块的说明应该是告诉他如何使用这个模块，而不是告诉他有关该软件模块是如何编写的全部细节。

封装(Encapsulation)是面向对象程序设计的原则之一。“封装”听起来给人的第一印象是将东西放进一个包裹里，这个印象确实是正确的。封装隐藏了包裹中东西的详尽细节。由于这个原因，封装也常被称为**信息隐藏**。但是包裹中的东西的一部分是可见的。在汽车

中，有些东西是可见的，如踏板与方向盘，而另一些东西则藏在引擎盖下面。汽车被封装起来，隐藏其内部细节，只有驾驶汽车所需的控制部分是可见的，如图 3-1 所示。同样地，应该将 Java 代码封装起来，隐藏其细节，只让必需的控件是可见的。

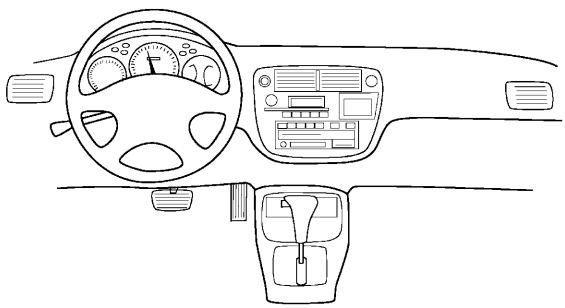

图 3-1　汽车的控制部分对于驾驶员是可见的，但其内部的机构则被隐藏

封装将数据与方法装入一个类中，将在使用这个类时不需要了解的实现细节隐藏起来。如果这个类设计得不错，使用它时就不需要先理解它是如何实现的。程序员在使用这个类的方法时可以对其如何编写的细节毫不知情，他只需要知道如何给一个方法提供适当的参数，然后就可以放心地让这个方法执行正确的操作。简而言之，程序员就不需要为类定义的内部细节操心。使用封装好的软件来编写更多的软件是一项较为简单的任务，从而，软件可以生产得更迅速，并且错误更少。

> 注　封装是面向对象编程的一条设计原则，它将数据与方法封装在类中，从而隐藏了类实现的细节。程序员获得的信息刚好足以能使用这个类。设计良好的类，即使在每一个方法的方法体都不可见的情况下也可以使用。

抽象(Abstraction)就是要求人们关注“做什么”而不是“怎样做”的过程。设计一个类实际上就是进行**数据抽象**(data abstraction)，其注意力放在想要利用或者针对这些数据去做什么，而不是操心如何实现这些任务以及如何表示这些数据。抽象要求人们关注哪些数据与操作是重要的。对某个事物进行抽象就是确定它的中心思想。例如，与整本书相比，一本书的抽象(摘要)是对这本书的一个简短描述。

程序员在设计类时，不应该考虑任何方法的实现细节。也就是说，不应该操心类的方法将如何达到它的目标。这种将说明与实现分离的做法，使得程序员可将注意力集中在更少的细节上，从而使设计更容易并更少出错。详细的、设计良好的说明，将使具体实现更能成功。

> 注　抽象的过程要求我们关注做什么而不是怎样做。

正确的封装应该是将类的定义分成两部分，这两部分分别称为**客户接口**(client interface)与**实现**(implementation)。客户接口描述了程序员在使用这个类时所需要知道的一切。它包括：类的公有方法的声明，程序员如何使用这些公有方法的注释，以及类的所有被定义为公有的常量。类定义的客户接口部分，必须包括在程序中使用该类时所需要知道的一切。

实现由全部的数据域和全部的方法定义构成，包括公有的、私有的和受保护的。尽管就

运行客户程序(一个使用该类的程序)而言,实现是必需的,但在编写这个客户程序时不需要知道实现的任何细节。图3-2所示为一个类的经封装的实现及其客户接口。虽然实现是对客户隐藏的,但接口是可见的,并且为客户提供了与实现之间定规有据的通信。

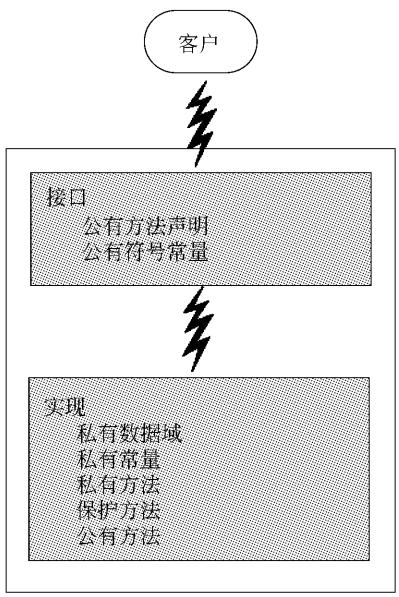

图3-2　接口为被隐藏的实现与客户之间提供了井井有条的通信

在Java类的定义中,客户接口与实现并不分开,而是相互混合。然而程序员可以创建一个单独的**Java接口**作为类的伴侣。Java接口包含类的公有方法的声明,并能定义公有的符号常量。本章的后几节将介绍如何编写和使用Java接口,在本书的后文中还将编写大量的Java接口。

自测题

1. 客户接口与类的实现有什么区别?

2. 试举一个不同于汽车的例子以说明封装的概念,例子中哪一部分对应于客户接口?哪一部分对应于实现?

3.2　方法的说明

将类及其方法的作用与它们的实现分离来,对于一个成功的软件项目而言是至关重要的。无须考虑其实现,就可以说明每个类和每个方法要干什么。此时所作的描述,可以帮助程序员捕获住最初的灵感并使之对以后的实现足够清晰。这些描述应该达到可以在程序中用作注释的程度,程序员视野应该开阔,不仅仅将注释看成是加在程序中使导师或老板满意的某种东西。

现在让我们将目光转向如何为类的方法编写注释。虽然各种组织都倾向于采用它们自己的注释风格,Java的开发人员已经为程序员制订了一种可遵循的注释风格。程序员如果

在程序中以这种风格来写注释，就可以运行一个称为 javadoc 的实用程序生成类的说明文档。这个文档将告诉其他程序员在使用这个类时所需知道的一切，同时省略所有实现细节，包括所有方法定义的方法体。

程序 javadoc 提取类的首部、全部公有方法的声明以及以特定的格式书写的注释。每一个注释都必须紧邻着一个公有类的定义或者一个公有方法的声明之前，并且必须以/**开头并以*/结束。由符号@开始的一些标记(tags)出现在注释中，用于标记方法的不同方面。例如，可以使用@param 标记一个参数，@return 标记一个返回值，使用@throws 标记由该方法抛出的一个异常。读者将会在本章写的注释中看到这样一些标记的例子。附录 D 提供了编写 javadoc 能接受的注释的详细说明。

在此不再进一步讨论有关 javadoc 的规则，而讨论方法说明中的一些重要事项。首先必须简明陈述方法的目的或者任务。以动词开头的陈述，将有助于避免实际上并不需要的冗余词汇。

在考虑一个方法的目的时，需要考虑它的输入参数。如果有的话，还要描述它们，还需要描述方法的结果。它是否返回一个值，是否导致某种动作，或者是否改变一个实参的状态？在书写这些描述时，始终应该记住如下几点。

先决条件(precondition)是对方法开始执行前必须为真的条件的陈述，它们是使用该方法必须满足的先决条件，否则就不应该使用这个方法，即使使用也不能够正确执行。先决条件可以与方法的形参相关。例如，一个计算 x 的平方根的方法可以将 $x \geqslant 0$ 作为先决条件。

后继条件(postcondition)是对方法执行之后将为真的事项的陈述。对于有值方法，后继条件将描述方法的返回值。对于无值方法，后继条件将描述方法执行的操作以及调用对象的变化情况。一般地，后继条件描述调用该方法产生的所有效果。

从后继条件的角度考虑问题将有助于阐明方法的目的。注意，从先决条件到后继条件都与“怎样做”无关——也就是说，方法的说明应与方法的实现相分离。

编程提示

如果一个方法不能满足其后继条件，即使满足了先决条件，也仍然可以抛出异常(参见附录 B 中关于异常的讨论)。

职责(responsibility) 先决条件定义了职责。如果客户能尽责保证在调用方法前满足某些条件，方法就不需要检查这些条件。反之，如果强制服从这些条件是方法的职责，客户就不需要去检查它们。对于给定条件由谁检查的清晰陈述，可以提高人们的办理效率，并避免了重复的劳动。

例如，上文提到的求平方根的方法，可以在其声明前添加如下的注释：

```
/** Task: 计算一个数的平方根
 *  @param x 一个非负的实数
 *  @return x 的平方根
 */
```

在此，方法假定客户将提供一个非负的实数作为实参。

另一方面，也可由方法承担检查实参的责任。此时的注释则应该如下所述：

```
/** Task: 计算一个数的平方根
 *  @param x 一个实数
 *  @return x的平方根,若 x>=0
 *  @throws ArithmeticException 若 x<0
 */
```

虽然如上述注释所示，先决条件与后继条件集成在一起，也可以将它们区分开来，正如可以将任务与实现区分开来一样。

编程提示

在每个公有方法的声明前的注释里，要对该方法作详细的说明。指出是由方法还是由方法的客户负责保证满足方法能成功执行所需的必要条件，由此可确保对先决条件的检查，而且不重复。然而，在调试阶段，方法始终应该检查其先决条件是否满足。

当使用继承和多态性来覆盖基类的方法时，派生类的方法可以与基类的方法不一致。先决条件和后继条件有助于避免这个问题。后继条件必须对一个方法在子类中的所有版本都有效。覆盖方法可以添加到后继条件，即它可以做得更多，但不可以做得更少。然而，覆盖的方法不可以对先决条件有异议。也就是说，它不能比该方法在基类中的版本要求的还多。

自测题

3. 假设Square类有一个数据域side和用于设置side值的方法setSide。应如何为该方法编写怎样的头部和注释？请注意先决条件和后继条件。

断言

断言(Assertions)是对程序逻辑中的某个事件为真的陈述。可将断言视作是一个布尔表达式，它在某处为真，或者至少应该为真。先决条件与后继条件都是断言的例子。如果一条断言为假，程序必定出错。

你可以在代码中将断言以注释的形式陈述出来。例如，假如在某个方法定义的某处，事先知道变量sum的值应该是正的，则可以编写如下的注释：

```
// Assertion: sum > 0
```

这样的注释指出了程序逻辑中可能不太容易看清楚的方面。并且，它们还提供了在调试时检查代码的准确位置。

自测题

4. 假设有一个正整数的数组，如下所列语句组的目的是查找数组中最大的整数。试问在如下循环语句中的if语句后面，应以注释的形式写一个什么样的断言？

```
int max = 0;
for (int index = 0; index <array.length; index++)
{
```

```
    if (array[index]>max)
        max = array[index];
    // Assertion:
}// end For
```

assert 语句　Java 不只允许你编写一个注释来做出一个断言。你可以如下使用 assert 语句来强制断言：

```
assert sum > 0;
```

如果保留字 assert 后面的布尔表达式为 true，该语句什么也不做。如果为 false，将发生一个**断言错误**(assertion error)，且程序运行终止，并显示如下的错误消息：

```
Exception in thread "main" java.lang.AssertionError
```

你可以给 assert 语句添加另一个表达式来明确化这种错误消息。另一个表达式必须表示一个值，因为其字符串所代表的值将会显示在错误消息中。例如，语句：

```
assert sum > 0 : sum;
```

在 s ＝＜ 0 时将把 sum 的值添加到错误消息中。例如，该错误消息可能是：

```
Exception in thread "main" java.lang.AssertionError: -5
```

默认情况下，在运行时 assert 语句被禁用了。因此，在完成编写程序后，你也可以把 assert 语句留在程序中，不会浪费运行时间。当运行程序时，如果你希望运行 assert 语句，就必须启用该语句。如何启用 assert 语句视你所使用的编程环境而定。

注　程序中的断言可以确保你的程序逻辑是正确的。在 Java 中，你可以使用 assert 语句来做出一个断言，其形式如下：

```
assert boolean_expression : valued_expression;
```

如果第一个表达式为 false，第二个可选表达式的值将出现在错误消息中。

编程提示

使用 assert 语句是在程序中查找错误的简单但有效的方法。除此之外，保留在程序文档中的断言可供需要修改或扩展程序性能的人使用。记住，Java 会忽略 assert 语句，除非程序用户专门设置不忽略它。

编程提示

可以用 assert 语句来强制满足方法的先决条件。

编程提示

assert 语句不可替代 if 语句。assert 语句只能是一种编程辅助方法，不是程序的逻辑代码。

3.3 接口

在本章的前几节中谈论了广义的客户接口，它告诉程序员在程序中使用一个特定类时所需知的一切。接口由类里的公有常量与公有方法的声明组成。虽然Java类将类的接口与类的实现混合在一起，程序员也可以编写单独的接口。

Java接口(Java interface)是程序的组件，它声明了一些公有方法。接口还应包含描述方法的注释，以便为程序员提供实现这些方法的必要信息。一些接口在一个类中描绘所有公有方法，而其他的则只描述其中一些。接口还可以定义公有的符号常量。

当你编写一个类，用于定义在接口中声明的方法时，我们就所该类**实现**(implement)了该接口。实现接口的类必须为接口中所描述的每个方法定义方法体。而接口则可能没有声明在类中定义的每个方法。

你可以编写自己的接口，也可以使用Java类库中的接口。当编写Java接口时，需要把它放在自己的文件中。也就是说，接口和实现它的类在两个不同的文件中。

3.3.1 编写接口

Java接口的开头类似于类的定义，只是需用单词interface替代class，即接口的开头是语句：

```
public interface 接口名称
```

而不是：

```
public class 类名称
```

接口可以含有任意数目的公有方法头声明，每一个后面都跟着一个分号。接口不会为类声明构造函数。注意，默认地，接口中的方法为公有的，因此你可以从它们的方法头中省去public。接口还可以定义任意数目公有符号常量。

举例 假设circle、square或plot对象都具有perimeter和area。我们希望这些对象的类具有能返回这些值的get方法。如果由不同的程序员来实现这些类，他们可能不会用同样的方法来为get方法命名或描述。为了确保这些类是以统一的方法来定义这些方法，我们可以编写如下接口：

```
public interface Mearsurable
{
    public int getPerimeter();
    public double getArea();
}
```

该接口为程序员提供了方法说明的简便归纳。程序员可以使用这些方法，而不用了解实现它们的类。

可以把接口定义保存在与该接口同名的文件中，其后缀为.java。例如，前面接口在Measurable.java文件中。

注　Java 接口是提供说明每个方法的目的、参数、先决条件和后继条件的注释的好地方。这样，程序员可以在一个文件里说明一个类，在另一个文件里实现这个类。

注　接口可以声明数据域，但它们必须是公有的。根据惯例，类的数据域是私有的，所以接口中的任何数据域都应该表示符号常量，因此它们是公有的、最终的和静态的。

注　在接口中声明的方法不能是最终的。但是，这些方法可以在实现接口的类中声明为最终的。

举例　回顾第 1 章中介绍的 Name 类。下列语句组为这个类定义了一个 Java 接口。为了节省篇幅，在此只为前两个方法写了注释：

```
public interface NameInterface
{
    /** Task: 设置姓与名
     *  @param firstName  待设置的名的字符串
     *  @param lastName   待设置的姓的字符串 */
    public void setName(String firstName, String lastName);
    /** Task: 获取全名
     *  @return 含有姓与名的字符串 */
    public String getName();
    public void setFirst(String firstName);
    public String getFirst();
    public void setLast(String lastName);
    public String getLast();
    public void giveLastNameTo(NameInterface child);
    public String toString();
} // NameInterface
```

此接口为客户提供了一个随手可用的方法说明的摘要，由此客户不需要去看类本身就能使用实现 NameInterface 的类。

注意方法 giveLastNameTo 中的参数以 NameInterface 为其数据类型，而不是像第 1 章那样以 Name 作为其数据类型。接口的编写应该与任何实现该接口的类独立开来。从第 3.3.3 节开始，我们将讨论作为数据类型的接口。

编程提示

为接口命名

接口的名称，尤其是 Java 中标准接口的名称，通常都由“able”结尾，例如 Comparable 与 Cloneable。这样的结尾并不总能提供一个好的名称，所以也常使用“er”或者“Interface”。正如 Java 的异常名称由“Exception”结尾一样，本书中将以“Interface”作为接口名称的结尾。

3.3.2 实现接口

任何实现接口的类定义都必须以 implements 子句开始陈述。例如，如果类 C 实现了 Measurable 接口，则类 C 的定义就应该如下所示开始：

```
public class C implements Measurable
```

随后，类必须为接口中声明的每个方法提供定义。在这个例子中，类 C 必须至少实现方法 getPerimeter 和 getArea。

如果要编写一个实现 NameInterface 接口的类 Name，那么该类应为如下所示：

```
public class Name implements NameInterface
```

该类的其余部分如第 1 章中定义的一样，只不过 giveLastNameTo 应具有一个数据类型为 NameInterface 而不是 Name 的参数，使其与接口中的声明匹配。

图 3-3 所示为包含 NameInterface、Name 及其客户的三个文件。

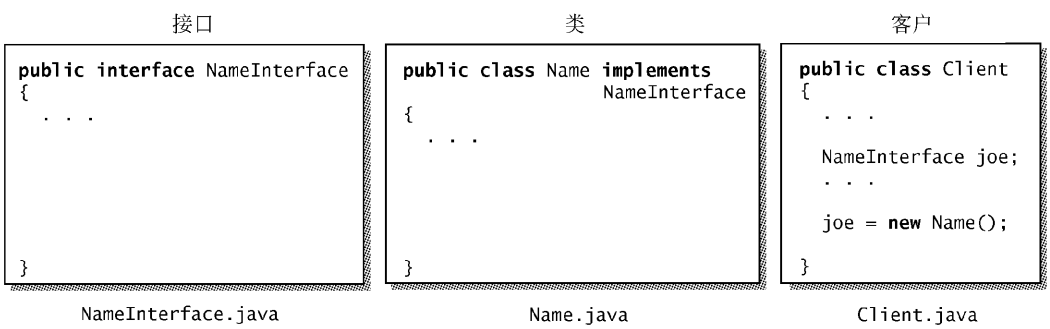

图 3-3 接口文件、实现该接口的类文件和客户文件

注 对类设计人员来说，编写接口是为其他程序人员描述方法的一种途径。对程序人员来说，实现接口是保证类定义了一些方法的途径。

注 可以有多个类以不同方式实现同一个接口。例如，可能有多个类都实现了 Measurable 接口，并且为 getPerimeter 和 getArea 方法提供了各自的版本。

多重接口 一个类还可以实现多重接口。在这种情况下，只要简单列出所有接口名称，彼此之间用逗号分隔即可。如果实现接口的类是由另一个类派生得到的，则 implements 子句应该总是跟在 extends 子句后面。因而，应该这样写

```
public class C extends B implements Measurable, AnotherInterface
```

实现多个接口的类必须在接口中定义每个方法。如果同一方法出现在某个类所实现的多个接口中，该类只需定义一个相应的方法即可。

第 2 章曾提及，一个类不能由一个以上的基类派生而来。这种限制避免了继承冲突实现发生的可能性。Java 接口包含的是方法说明，而不是实现。类可以实现这些方法说明，而不管它们是在一个还是多个接口中。通过允许某个类实现任意数目的接口，Java 就近似地获得了多个基类的功能，并且没有使问题复杂化。

自测题

5. 为第 2 章中给出的 Student 类写一个 Java 接口。

6. 必须对 Student 类做哪些改动，才能实现你在上一个题中编写的接口？

3.3.3　作为数据类型的接口

在声明变量、数据域或者方法的参数时，可以像使用数据类型那样使用 Java 接口。例如，方法 giveLastNameTo 可以有一个类型为 NameInterface 的参数：

```
public void giveLastNameTo (NameInterface aName)
```

传递给这个方法的任何实参都必须是一个实现 NameInterface 接口的类的对象。

为什么不像第 1 章那样把 aName 的类型声明为 Name 呢？我们希望接口与实现它的任何类无关，因为不止一个类可实现接口。通过用 NameInterface 作为该参数的类型，就可以确保方法的参数能够调用在 NameInterface 中声明的所有方法。通常，如果参数的数据类型是一个接口，那么就可以确保这个参数能够调用该接口中声明的方法。

如果类 C 没有以子句 implements NameInterface 开始，那么即使它实现了方法 compareTo，也不能将 D 的实例传递给 myMethod。

注　通过将接口作为一个对象的类型，就可确保这个对象具有一整套特定方法。

如下所示的变量声明：

```
NameInterface myName;
```

使 myName 成为一个引用变量，现在 myName 可以引用任何一个实现 NameInterface 的类的任何一个对象。因此，如果有：

```
myName = new Name("Coco", "Puffs");
```

则 myName. getFirst()将返回字符串“Coco”的引用。如果类 AnotherName 也实现了 NameInterface，并且随后写下：

```
myName = new AnotherName("April", "MacIntosh");
```

则 myName. getFirst()将返回字符串“April”的引用。

第 2 章讨论继承时介绍了多态变量，其中曾提及，如果类 B 由类 A 派生得到，就可以写

```
A item = new B();
```

变量 item 是多态的，因为它是动态类型，与静态类型是不同的。这里，变量 myName 也是多态的。因此，无论使用继承或接口都可能导致多态变量。

自测题

7. 如何使自测题 5 中编写的接口和实现该接口的 Student 类都可以使用 NameInterface？