第4章　UML和Java

- 表示结构
- 表示关系
- 小结

UML 和 Java 都是软件开发语言，但它们各自的使用方法并不相同。UML 是一种可视化的语言，而 Java 则是一种文字上的语言。

从某方面来说，UML 比 Java 更有意义，因为它提供了更抽象更强大的方式来表示具体的概念或关系。而通常只有一种方式来表示 Java 语言中的概念或关系。

例如，Java 变量声明在 UML 中可以用多种方法来表示。

本章概述了 UML 中与类相关的一些关键概念，以及如何实现这些概念。主要目的是了解在 UML 中新出现内容对我们有哪些帮助。第二个目的是，判断使用哪种 UML 标准化标记法可以有效增强特定 Java 代码块的作用，而无需改变等效的 Java 代码。

4.1　表示结构

结构性概念，例如类和接口，都是 Java 和 UML 中的基本概念。本节内容主要讲述这些概念是如何映射到 Java 和 UML 中的。

4.1.1　类

在 UML 中，是通过一个模块化的矩形表示 Java 类。其中有 3 个区域：

- 名称区域：显示 Java 的类名
- 属性区域：如果有的话，列出在类中定义的变量
- 操作区域：如果有的话，显示出定义在类中的方法。

图 4-1 所示的是一个没有任何变量和方法的简单 Java 类。

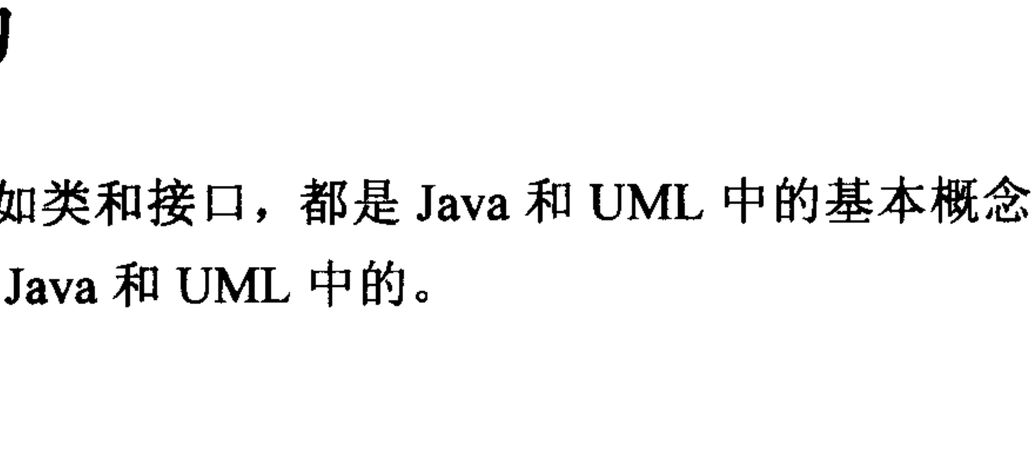


图 4-1　Java 和 UML 中的类

通过将类名变为斜体可以标识抽象类。

将模板并排显示在类名旁边，可以清楚地表示出 Java 类的具体类型，例如 applet(第 2 章中我们已讲过了模板这一概念)。你也可用模板来标识你自己的特定领域词汇表中类的具体类型(例如<<Business Entity>>)，这样可以使得这些类更有意义。

需要注意的是：如果用 UML 工具来生成 Java 代码，注意该工具会利用模板化机制来影响代码的生成。

图 4-2 表示的是一个模板化的类。

Java	UML

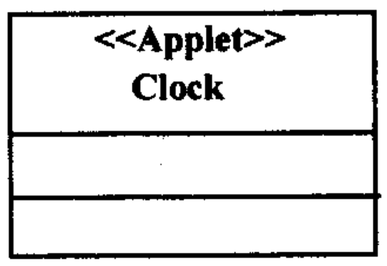
Public class Clock extends Applet{
…
}
	

图 4-2　模板化类

4.1.2　变量

在 UML 中，Java 变量的表示方式各不相同。图 4-2 就是一种实例，其中通过建模技术添加了源代码中未出现的变量。

将声明在类的属性区域中列出来，这种变量声明形式最简单。如果给属性加了下划线，表明的是变量的静态特性；在属性前加+，表明该属性可见性的作用域是 Public；加#表明是 Protected；加-表明是 Private。图 4-3 所示的是个带有属性的类。

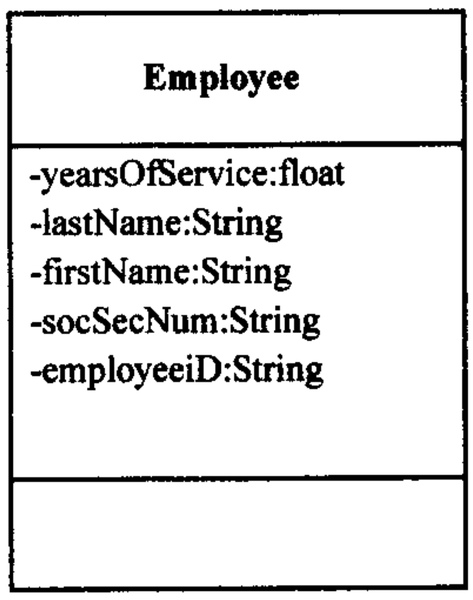
Java	UML
Public class Employee } private float yearsOfService; public String lastName; public String firstName; public String socScNum; public String employeeID; … }	

图 4-3　带有属性的类

如果某个类需要基本数据，那一般就采用这种形式的声明。从更为广义的建模角度来看，这种变量通常没有任何特定的意义。范例包含了要求能够存储表明对象的基本信息和处理内部逻辑等的变量。这种变量所基于的对象一般不能够被进一步分解。

变量也可以根据对象与其他对象(例如，某些种类的集合)间的关系来表述自己。本章后面的 4.2 节中，我们将讨论关系及其应用。

4.1.3　方法

在 UML 中，方法等效于对类进行的操作。方法显示在类的第 3 个区域中。与 4.1.2 这一节中定义类属性所使用的惯例一样，我们据此定义 UML 操作的可见性范围。

操作名前加下划线用于区分静态方法。在操作区块中以斜体列举操作，表明该方法是抽象方法。当然，您也可以根据细节的重要程度决定隐藏还是显示细节。例如，在图 4-4 中，并未显示操作签名。

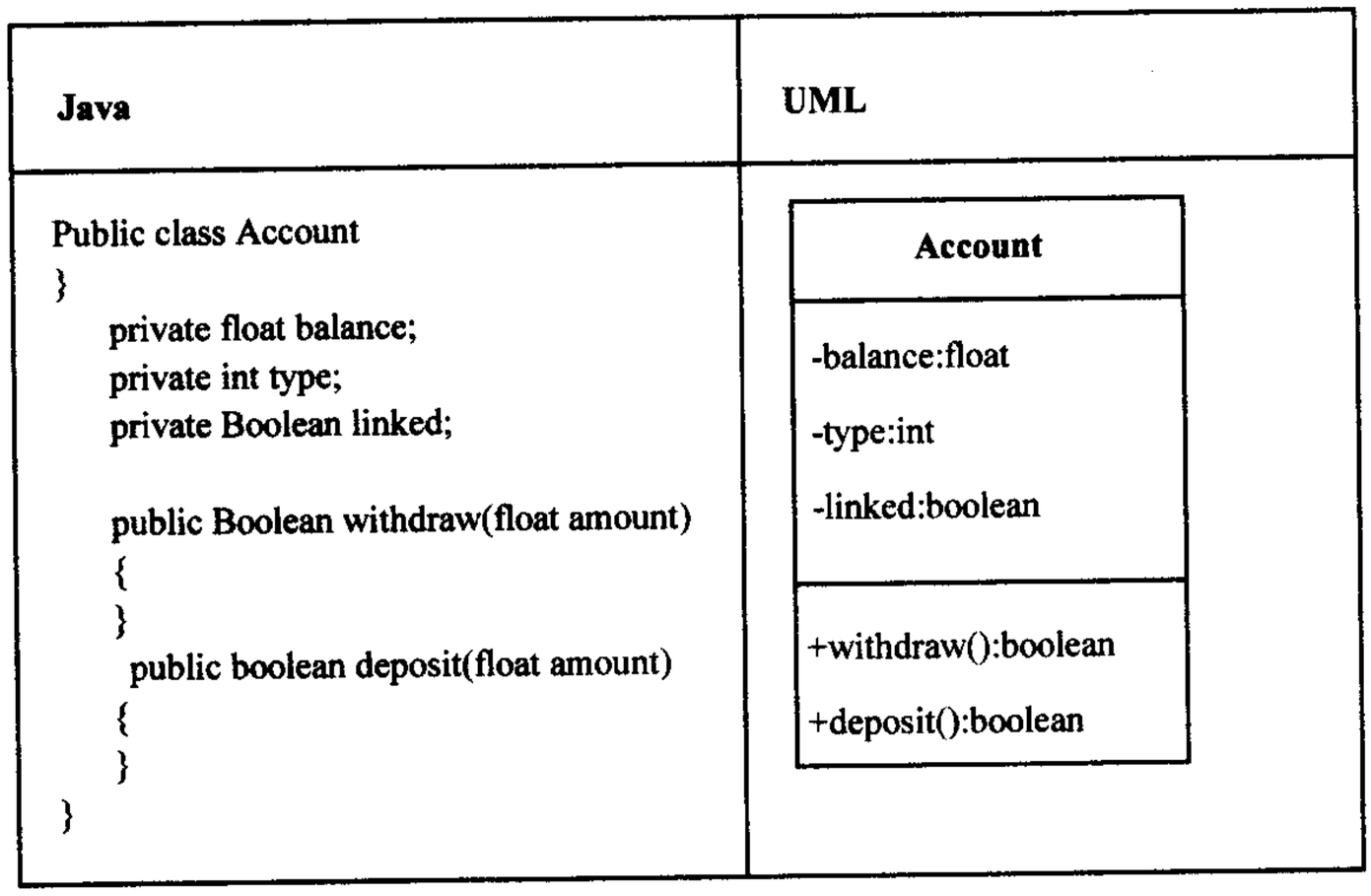


图 4-4　带有属性和操作的类

4.1.4　对象

虽然 Java 和 UML 中都有对象这一概念，但 UML 对象和 Java 代码间并没有直接的映射关系。之所以这样，是因为对象是个基于类定义的动态实体。Java 应用程序是根据 Java 类进行编写的。因此实际执行应用程序时，就会创建 Java 对象。

在 UML 中，通过交互作用图，用对象来模拟系统的动态性。带有对象名和(或)类名的矩形用来表示对象的标准化标记法。有时，我们希望在指定的位置显示对象的属性值，那么只要使用有两个区域的显示类属性的矩形就可以做到这一点，如图 4-5 所示。

Java	UML
No Java code equivalent	checking:Account Balance:float Type:int Linked:boolean

图 4-5　对象

4.1.5　接口

在 UML 中，Java 接口被描述为一个以<<interface>>为模板化的类。模板化的类可以随意选择与它们相关联的图标。就接口来说，UML 传统的表示方式是一个小圆圈。这种传统的表示方式通常在利用 UML 建模时表示 Java 接口。

图 4-6 所示的是个标准接口。

Java	UML

Public interface Control{
…
}
	<<interface>>
	Control

图 4-6　接口

图 4-7 所示的是一个备用的更为简洁的表示格式。

Java	UML

Public interface Control{
…
}
	Control

图 4-7　UML 中接口的另一种表示法

从建模的观点来看，每种方法都是可行的，随个人喜好而定。本书普遍使用以图表示的图标表示法。

4.1.6　包

Java 包映射为 UML 包。包多半是符合逻辑的，这就意味着只能将它们作为一种分组机制。包也可以是物理的，也就是说，在文件系统中，它们可以是物理目录。

UML 包以文件夹的形式表示出来，如图 4-8 所示。我们可以将包模板化，以区分包的各种类型，例如，利用<<subsystem>>表明包是个子系统。(子系统是指一组 UML 元素，

表示模型中的行为单元。它可以有接口和操作。从分析和设计角度来看，子系统通常比较有意义。在子系统和 Java 语言结构间没有直接的映射关系。)

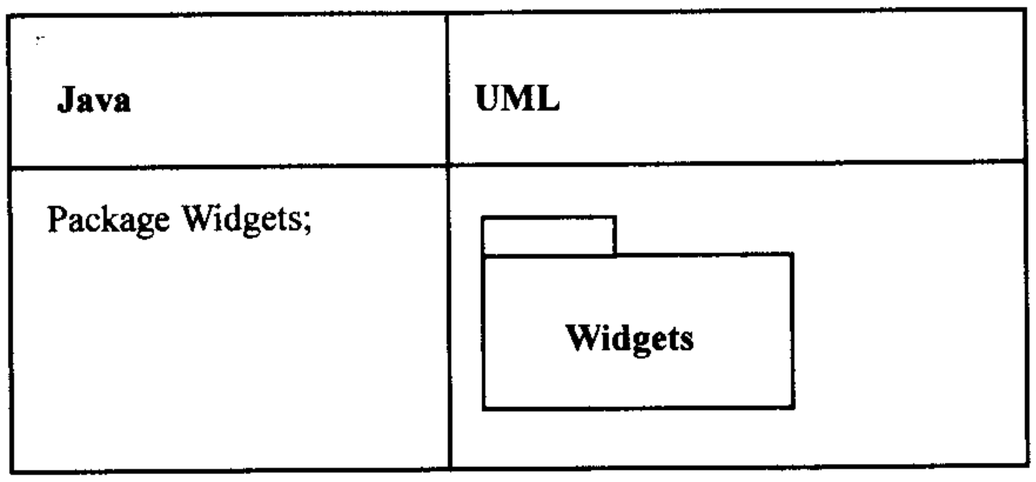


图 4-8　包

4.2　表示关系

关系在捕获和建模 Java 应用程序的重要结构方面发挥重要的作用。

某些关系，例如继承，可以通过预定义关键字在 Java 语言中显式标识。其他的关系在 Java 代码中不能这么容易地标识，但尽管如此，还是可以表示出来的。

4.2.1　继承

在 UML 中 Generalization 这个概念有点类似于 Java 中的继承概念。Generalization 可以直接映射为 extends 关键字，我们可以用一个带有三角的线条(三角的顶端紧靠着超类)来表示这种关系，如图 4-9 所示。

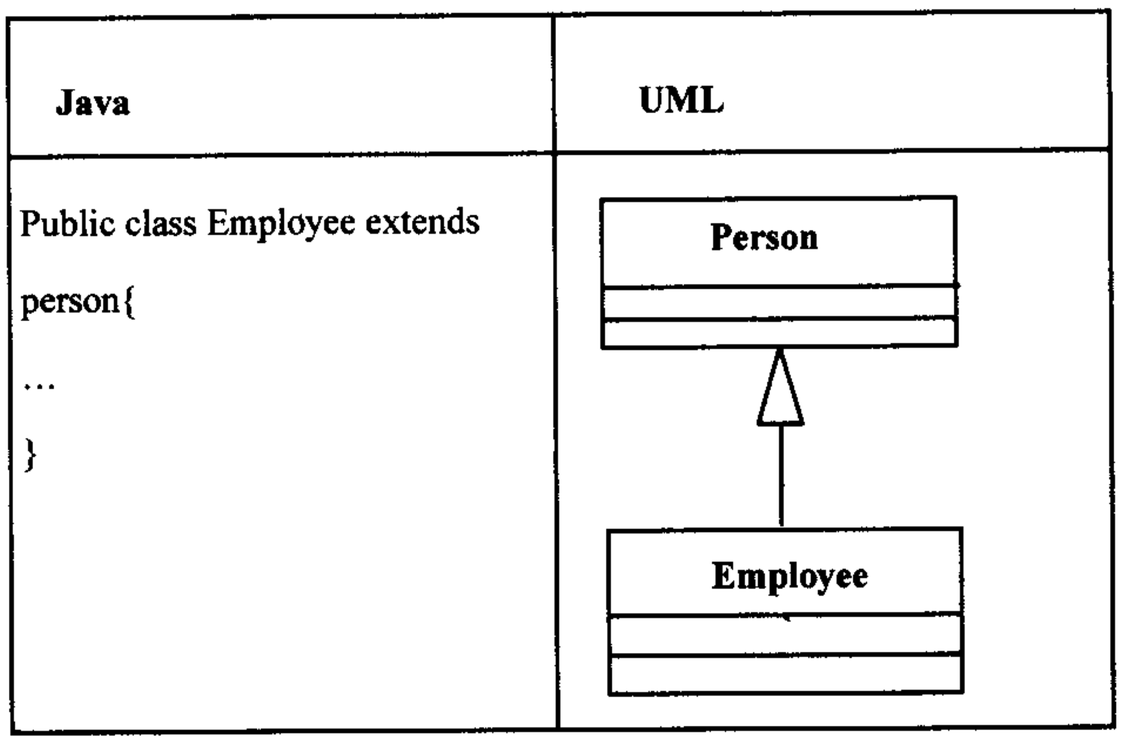


图 4-9　继承关系的表示

4.2.2　实现

在 Java 中，类可以实现一个或多个接口。Java 中的关键字 implements 映射 UML 中 realization 这个概念。

在 UML 中，实现可以用两种方法来表示。如果用模板化类的方法来表示接口，实现

由一条带有指向接口的三角的虚线表示。如果用圆圈来表示接口，那么就用一条实线来连接接口和实现类。

图 4-10 和图 4-11 所示的就是这两种方法。注意图 4-11 中所显示的方法是图 4-10 中方法的简写形式，最好不要将这两种方法混着使用。比如说，使用圆圈表示接口的同时，又使用带三角的虚线，这是不恰当的。

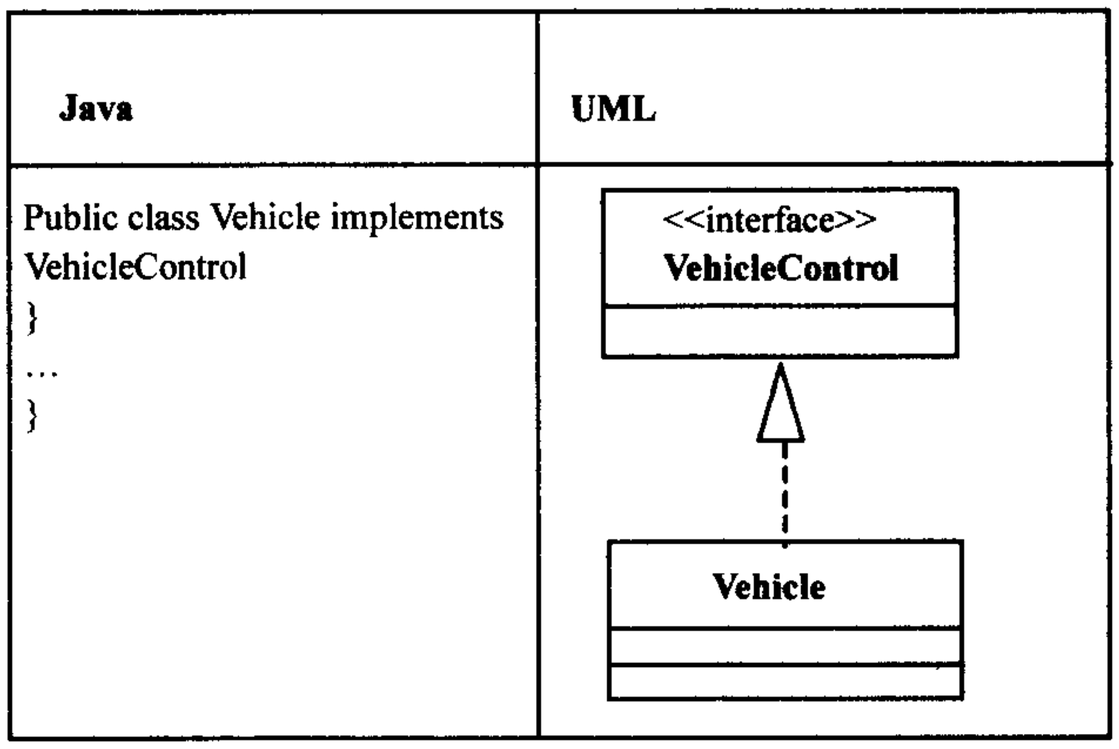


图 4-10 UML 实现

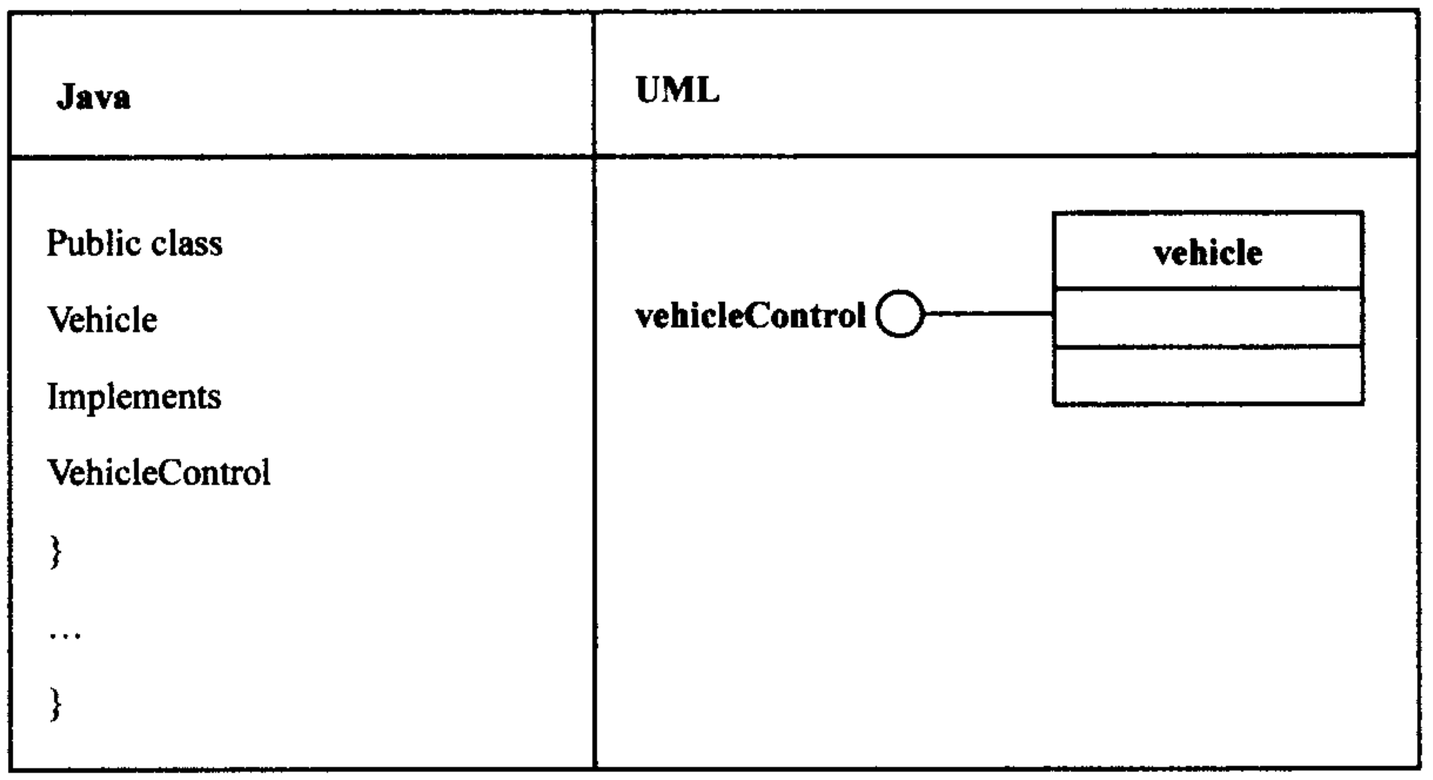


图 4-11 接口实现的另一种表示法

4.2.3 相关性

只要一个类以某种方式使用了另一个类，这两者间就有了相关性。其关系由用户所使用的类而定。在 UML 中，相关性由一条带有箭头的虚线表示，其中箭头指向引起相关性的类。

类具有相关性的条件是：

- 有一个基于其他类的局部变量
- 有一个直接对对象的引用
- 有一个对对象的引用，但是间接的。例如，通过一些操作参数

● 使用类的静态操作

在包含了相关联类的包之间，也会存在相关性关系。包间的相关性以带有箭头的虚线表示，如图 4-12 和图 4-13 所示。

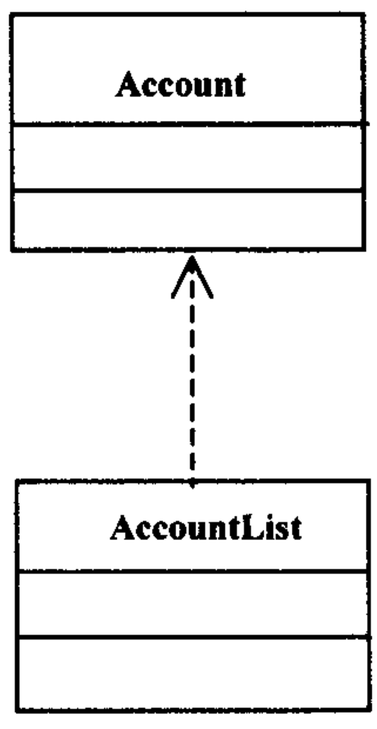
Java	UML
Dependencies between classes (AccountList depends on Account)	

图 4-12　类之间的相关性

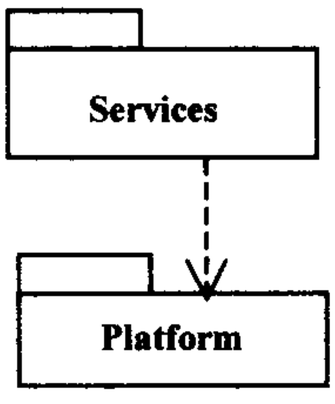
Java	UML
Dependencies between packages (Services depends on Platform)	

图 4-13　包之间的相关性

4.2.4　关联

从概念上来说，两个类间的关联表示这两个类间某种结构上的关系。

在 UML 中，关联的表示方法就是在有关系的类间画一条线。关联可能是单向，也可能是双向的。双向关联就以一条直线表示，而单向关联则以带有箭头的直线表示。

单向关联意思是箭头发出处类的对象(也就是，关联的没有箭头这一端的类)，可以调用箭头指向的类中的方法。在 Java 中，则该关联是可以调用方法的这个类中的实例变量。

图 4-14 所示的就是一个单向关联的例子。

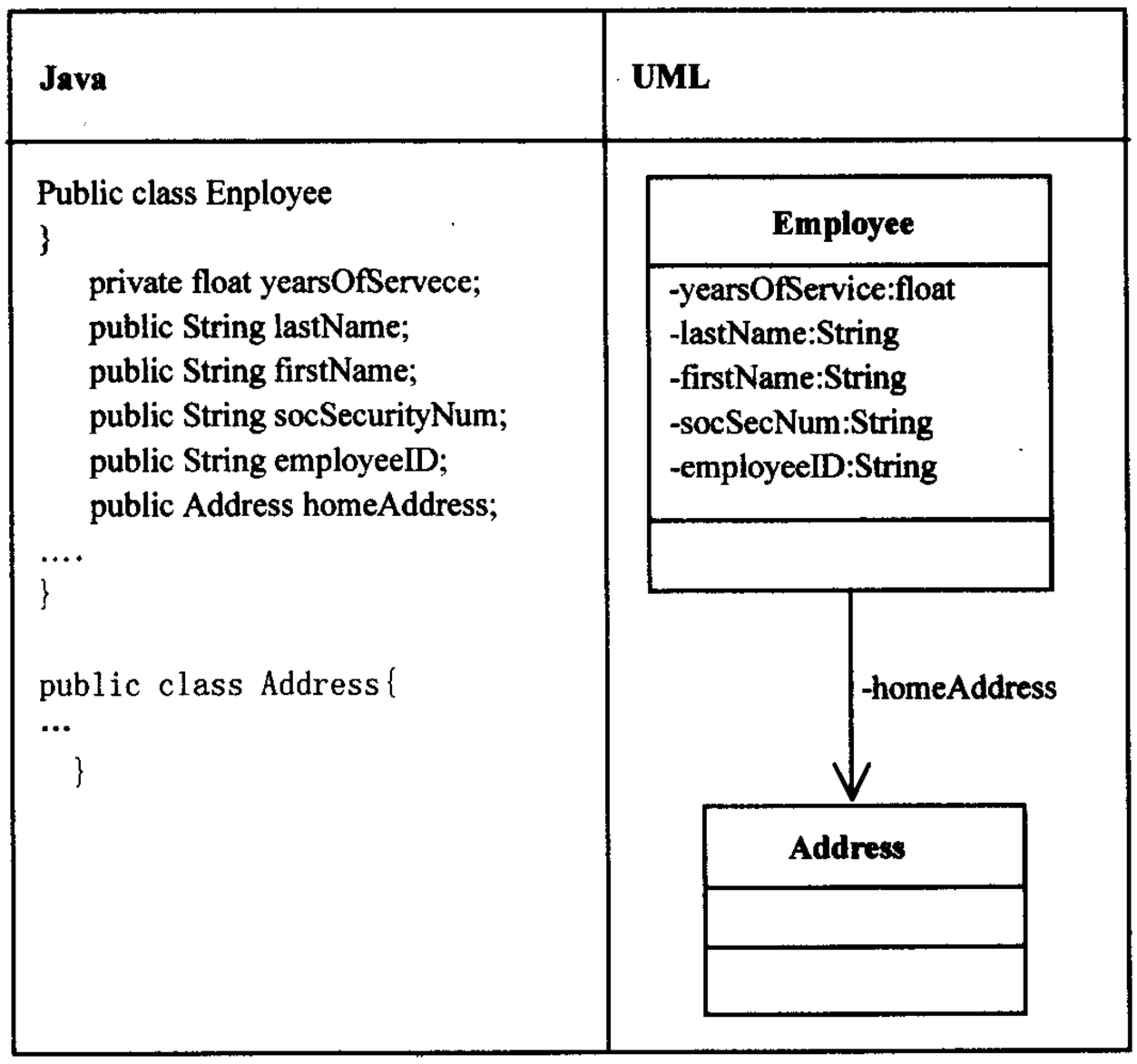


图 4-14　单向关联范例

绝大多数的关联都是单向关联，但有些关联也可以是双向关联的。双向关联只是表明关联中的每个对象都可以调用其他对象的方法。在 Java 中，双向关联是基于其他类的类中的实例变量。

双向关联的例子请参阅图 4-15。

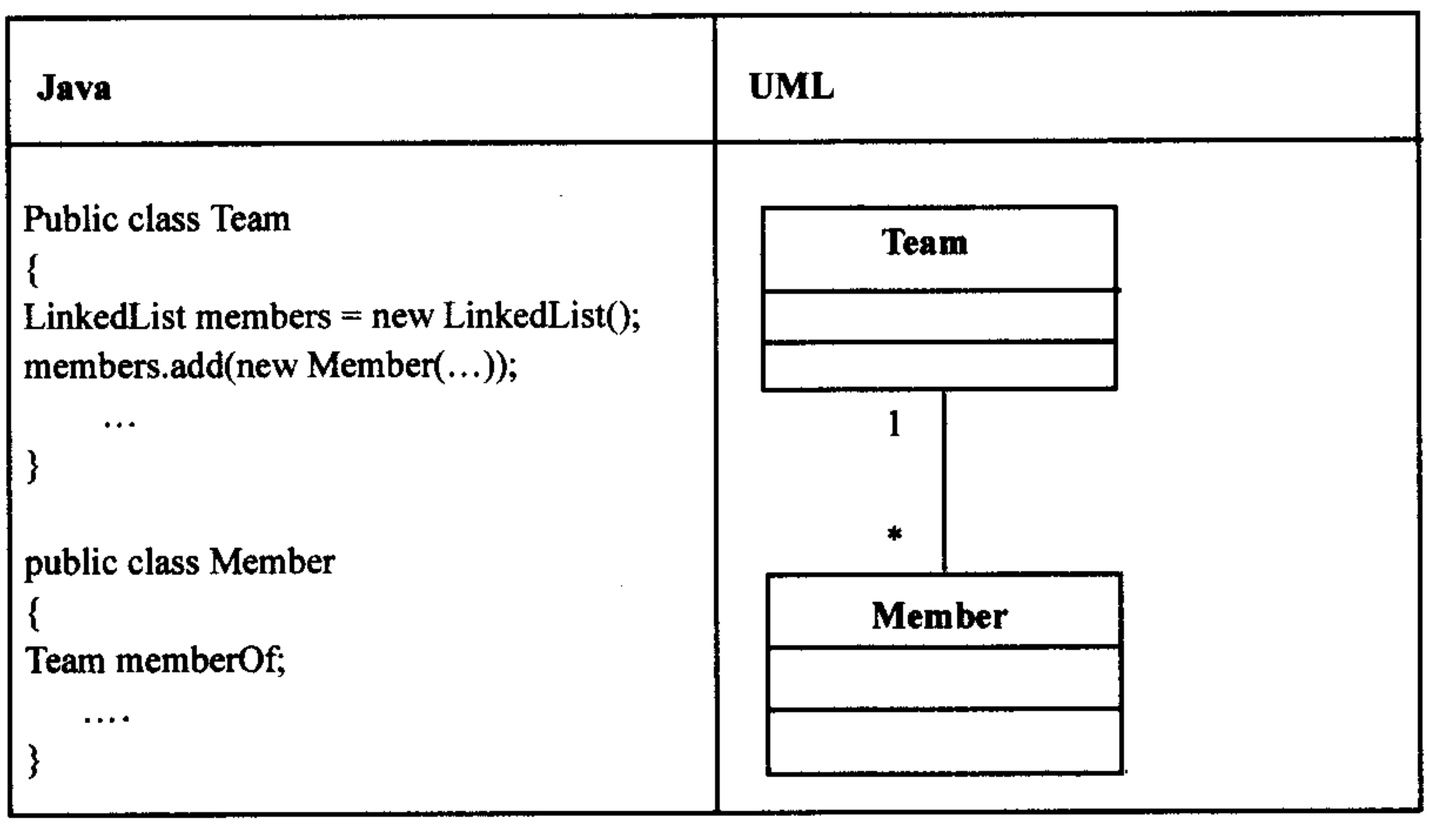


图 4-15　双向关联范例

那么该如何显示基本类型的关联呢？例如整型或逻辑型？当然，如果喜欢的话，也可以像上面这样做。事实上，在分析阶段，你就可以显示出大量实体间的关联，但在设计和实现时，识别每个关联的重要性，这样关联的数量就会大幅度减少。实际上，并没有必要添加更多的值来帮助理解设计，除了淆乱视觉，把关系显示出来并没有别的什么用处。用

关联来显示重要的关系应该更可取。

在 UML 中，关联的两端都是角色，而且都可被命名。举个例子来说吧，某个人应该与雇用他的公司之间有双向关联。在这种情况下，角色就该分别命名为雇主和雇员。如果用 Java 来实现，角色的名称应该符合各个类中实例变量的名称。如果给模型添加值以加深对它的理解，那么命名角色就比较有用，如果没有添加值，那不命名是很恰当的。在这种情况下，角色名只能是类的名称。

关于双向关联中角色的例子，请参阅图 4-16。

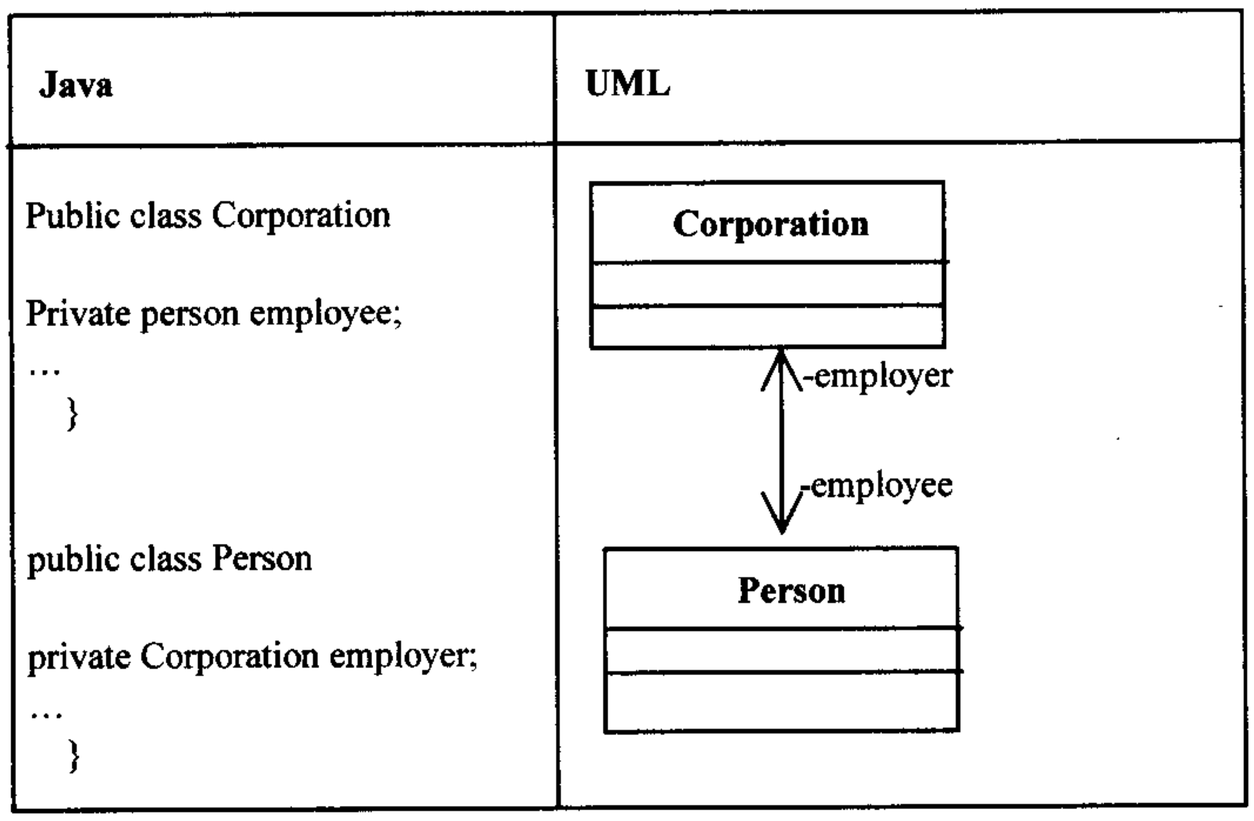


图 4-16　双向关联中角色范例

当然，一个类中的对象可能会与其他类中的对象有多个关联。例如，一个公司通常有多个雇员，而雇员可能会为多个公司工作。这可以通过给角色赋予多重任务进行模拟。多重任务可被描述为一个特定的值(例如，0、1、7)，或者一个范围(例如，0..1、1..5、1..*)。这里的星号表示一个没有限制的范围。例如，“*”可以是零或更大，“500..*”表明 500 或更多直到没有限制的数字。

根据 Java 的实现方式，多重性表明这是一个多值的实例变量。例如，假定某个公司雇用了一些人，而一个人最多在 3 个公司工作。对于变量的多个值，如果没有固定的上限，这就转换成了一个集合，表示只为一个公司工作的人。对于为 3 个公司工作的人，就会转换成一个带有 3 个元素的数组。

关于多重性的例子请参阅图 4-17。

与关联角色相关的信息不能一直驻留在关联所包含的类中。例如，将购物者与虚拟购物车之间的会话信息保存在各自的类中，这是不合适的。在这种情况下，可以用一个相关联的类来模拟这种情况，如图 4-18 所示。

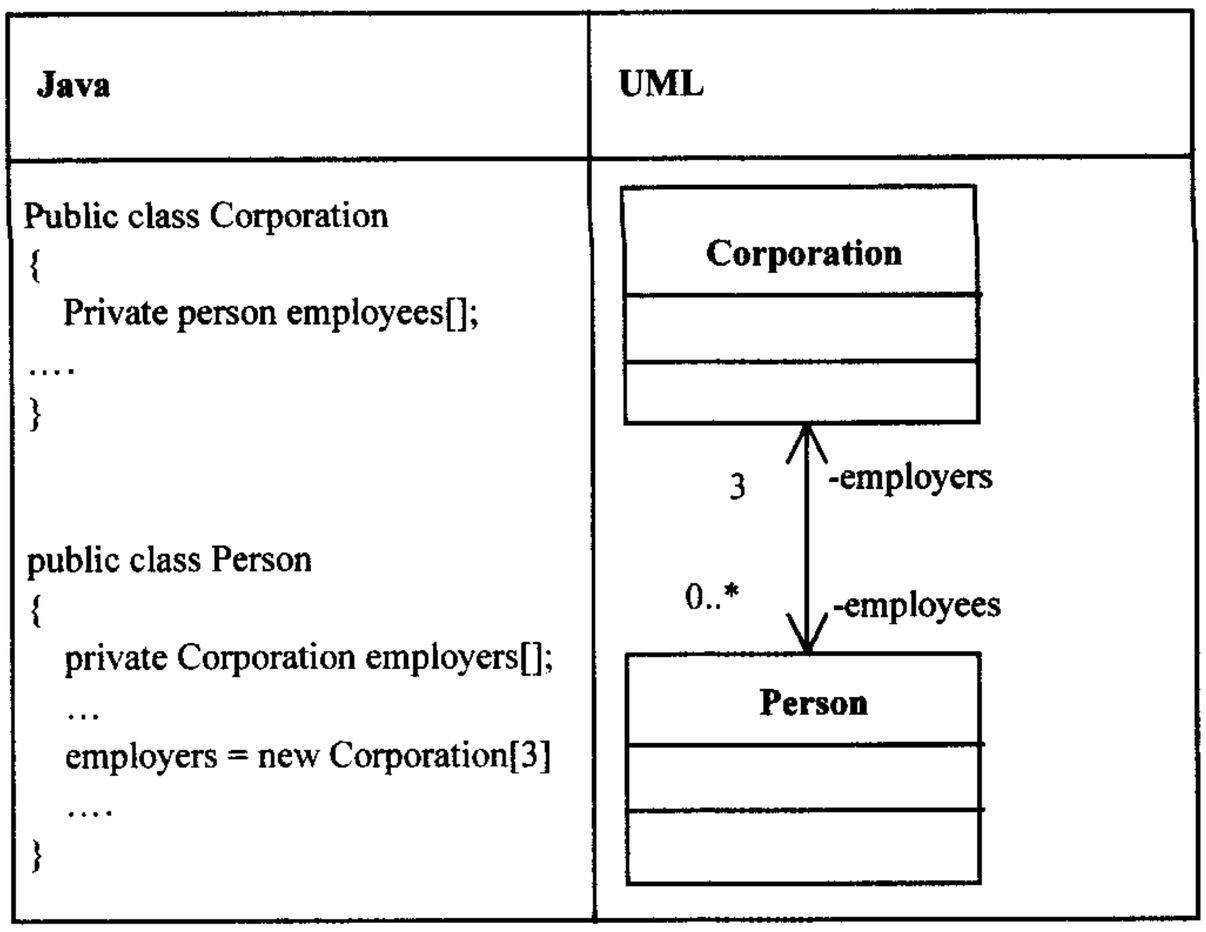


图 4-17　多重性范例

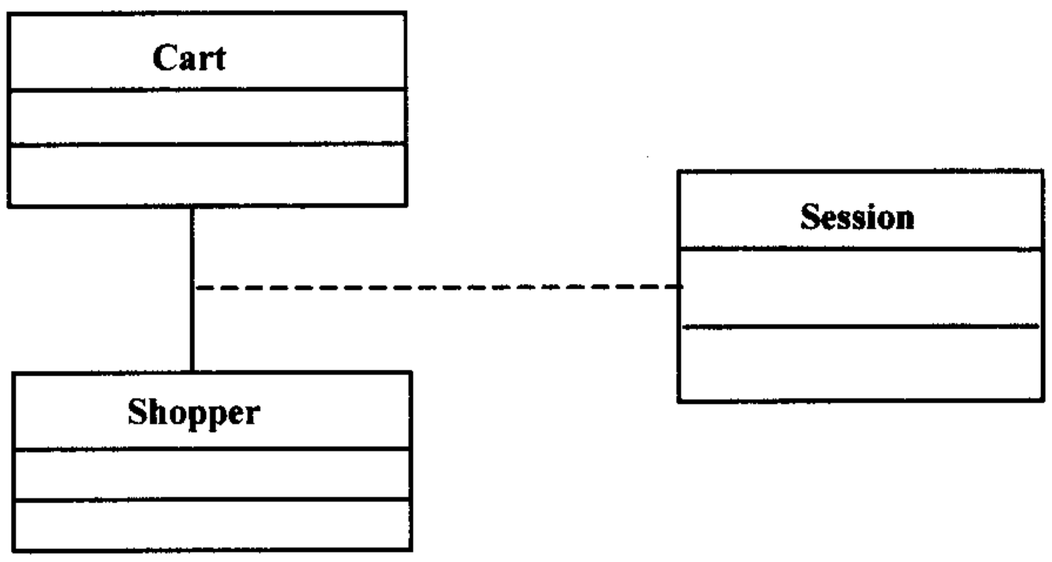


图 4-18　关联类

4.2.5　聚合

聚合是一种更有效的关联方式，可用来表示一种逻辑上的包容关系，也就是说，是一个由各部分组成的整体。虽然各部分与整体相对独立，但主要还是由它们形成一个整体。例如，以一台计算机为例，可以将它看成由主板、CPU、I/O 控制器等等组成。注意 I/O 控制器可以单独存在(例如，在计算机存储器中)，不过，它存在于整体环境中应该更为合适。

聚合可被建模为一个带有空心菱形的关联，该关联位于形成整体的类之间。因为它是一个关联，聚合就可以支持角色和多重性的概念。在 Java 中，聚合映射为类中的实例变量。

图 4-19 所示的是关于聚合的例子。

聚合的语义和约束与基本的关联并没有本质上的区别。尽管如此，许多人还是觉得聚合是必需的。

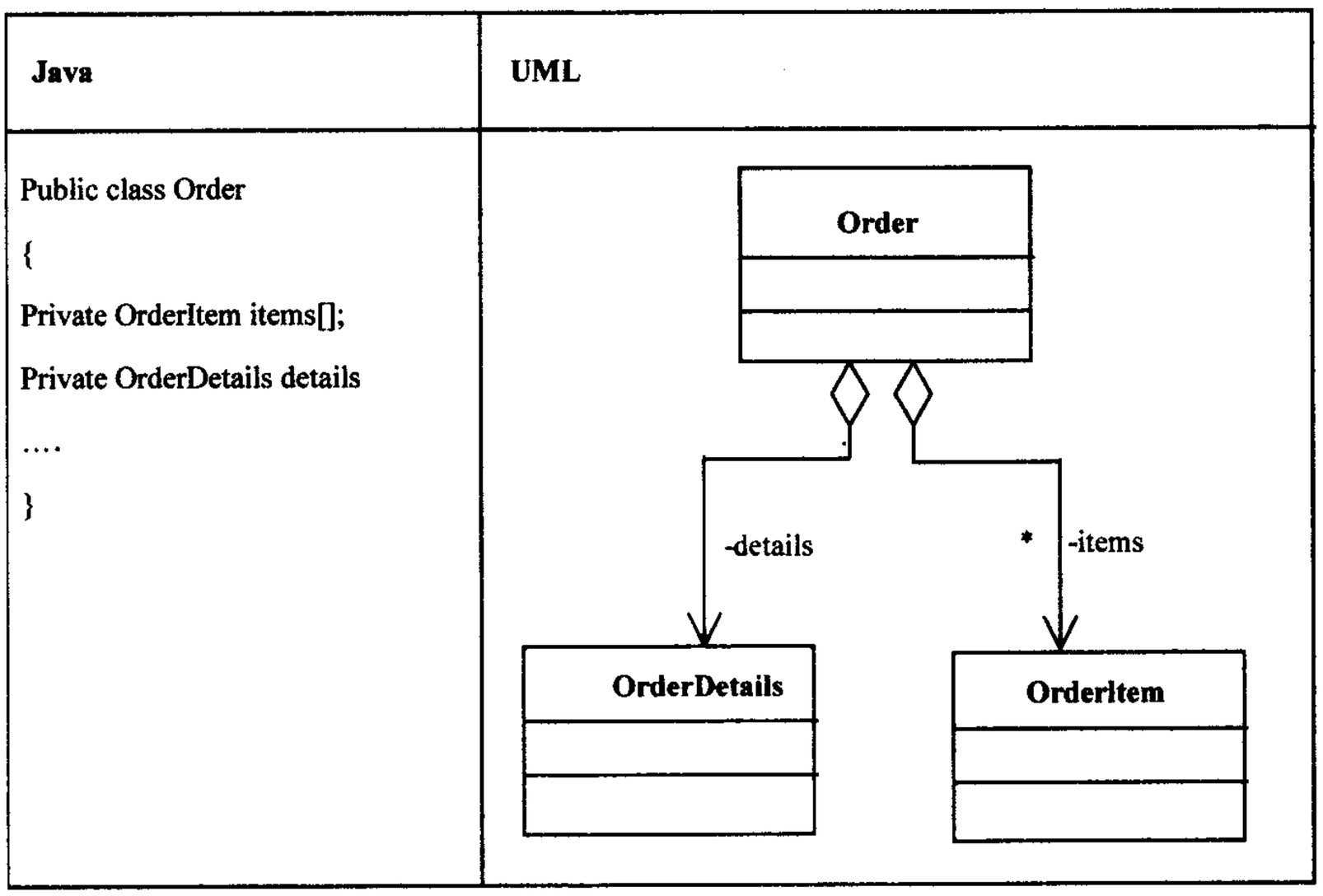


图 4-19　聚合范例

与关联的实例不同，进行聚合的实例之间只能是包含与被包含的关系。也就是说，对象不可能直接或间接地是它自己的组成部分。例如，如果一个 A 的实例聚合一个 B 的实例，那么这个 B 的实例本身就不能聚合这个 A 的实例。

大体上，除非想用聚合来添加值或说明什么问题，否则最好还是使用关联。(下面所介绍的合成是另一种类似的方法。)

4.2.6　合成

合成是关联的另一种形式，在某种程度上与聚合有些相似。不过，概念上比聚合更清晰。

需要建模物理容积的情况下，比较适合使用合成。这意味着在进行合成的各个部分之间，整体与部分间的耦合更为稳固，部分不能独立于整体而存在。也就是说，部分参与了整体的生命周期，它随着整体的出现而出现，随着整体的消亡而消亡。

在使用实现语言例如 C++时，与聚合和合成所对应的代码也不尽相同。例如，聚合中没有引用，而合成中不考虑值。但是，这种区别并不适用于 Java。因此，即使你用不同的方法模仿它们，以沟通设计意图和单独实现的重要元素，聚合和合成所对应的代码还是相同的。

除了在菱形已被填充的情况下以外，合成与聚合的用法都是一样的。

4.2.7　自反关系

类可以有一个自身的关联。例如，一个人雇用了另一个人，Person 类就会有一个与自身有关的关联，该关联中带有雇主和雇员的角色名称。这样的一种关系就称为自反关系。

这种标准化标记法可被看成是建模的一种简写形式。无需用两个类图标，只要用一个

就可以表示出这种关系。如图 4-20 所示，用它完全可以表示带有这种关系的两个相互独立的 Persion 类图标。不过，这样做会占用图中的一些空间。

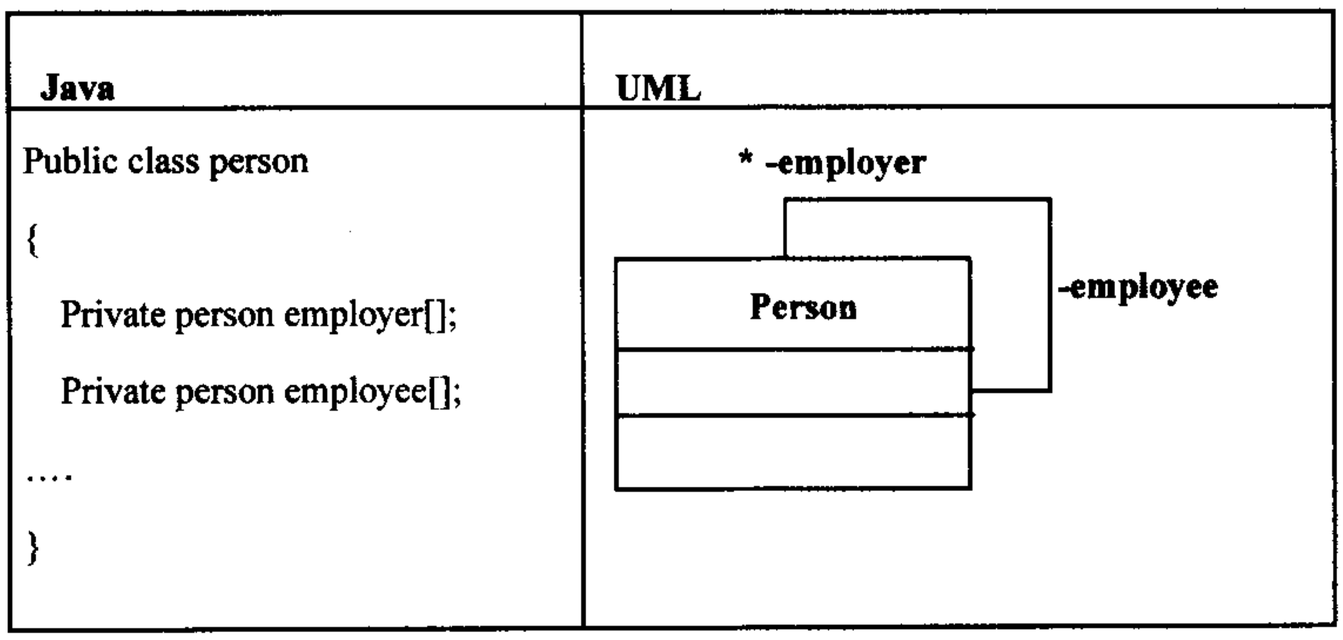


图 4-20　自反关系范例

4.3　小结

恰当地使用 UML 结构对整体设计有着极大的影响。不仅可以为设计提供相关内容，还可以使之更容易让人理解。

在本章中，我们主要关注与类图相关的主要概念，这些概念是：

- 类、属性和操作，以及在 Java 中它们之间的关系。
- 用作分组工具的包，以及其与 Java 的关系
- 类间各种不同的关系，以及在什么情况下使用下面这些关系：
  - 关联
  - 聚合
  - 合成
- 在 UML 中表示继承
- UML 中角色的实现，以及它是如何与 Java 实现语言中的 extends 相联系的。

建模并不是件容易的工作。与任何其他基于技术的工作相比，需要花费更大的精力才能掌握 UML 和建模技术。在下面几章中，我们将探讨一下在 J2EE 开发环境中包含了这些概念的应用程序。