OOP UNI

Overview

Class Visibility

Visibility	Symbol (prefix)	Description
public	+	available to all other classes
protected	#	available to subclasses (see chapter 5)
private	-	available only to the class itself
package	%	available to all classes in the same package

Basics ...

Classes

Figure 3-1 shows the simplest form of class diagram. The class named `Dialler` is represented as a simple rectangle. This diagram represents nothing more than the code shown to its right.

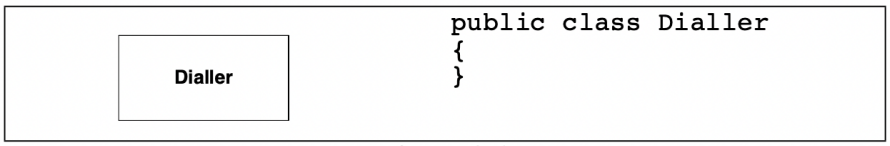

Figure 3-1
Class Icon

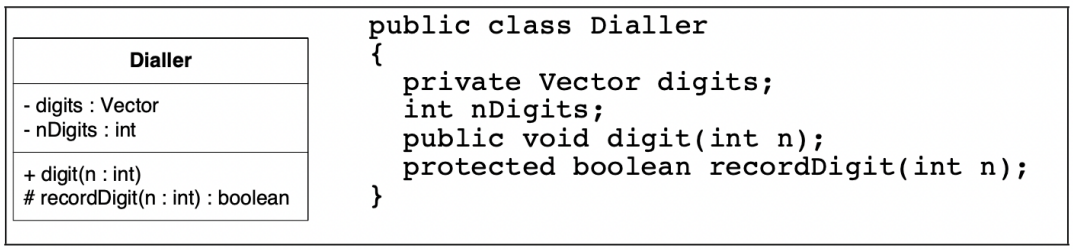

Figure 3-2

Association

Associations between classes most often represent instance variables that hold references to other objects. For example, in Figure 3-3 we see an association between `Phone` and `Button`. The direction of the arrow tells us that `Phone` holds a reference to `Button`. The name near the arrowhead is the name of the instance variable. The number near the arrowhead tells us how many references are held.

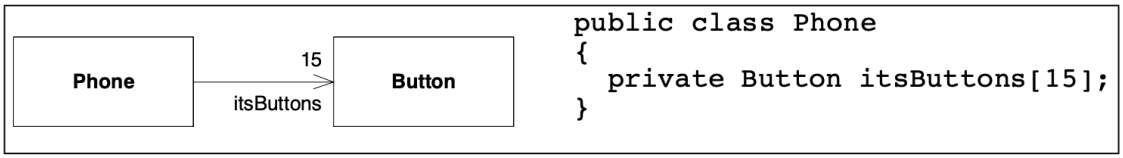

Figure 3-3

Multiplicity

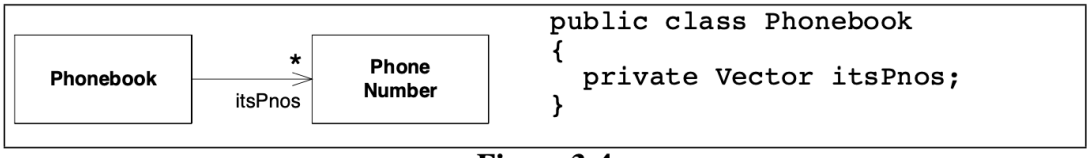

Figure 3-4

Inheritance

You have to be very careful with your arrowheads in UML. Figure 3-5 shows why. The little arrowhead pointing at `Employee` denotes *inheritance*[1]. If you draw your arrowheads carelessly, it may be hard to tell whether you mean inheritance or association. To make it clearer, I often make inheritance relationships vertical and associations horizontal.

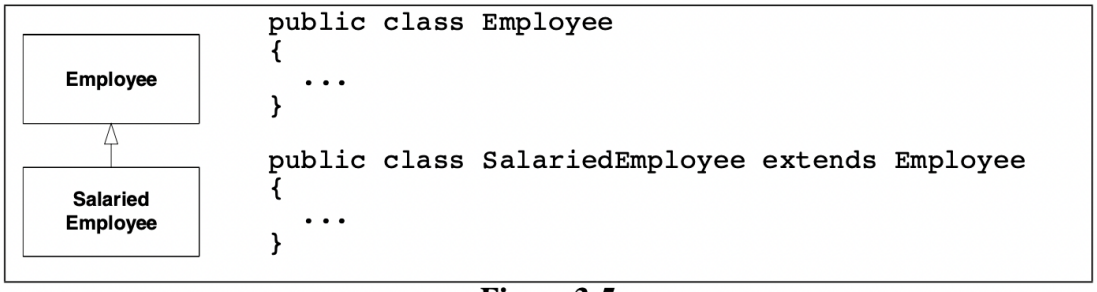

Figure 3-5

suggest you forget to dash the arrows that you draw on whiteboards too. Life's too short to be dashing arrows.

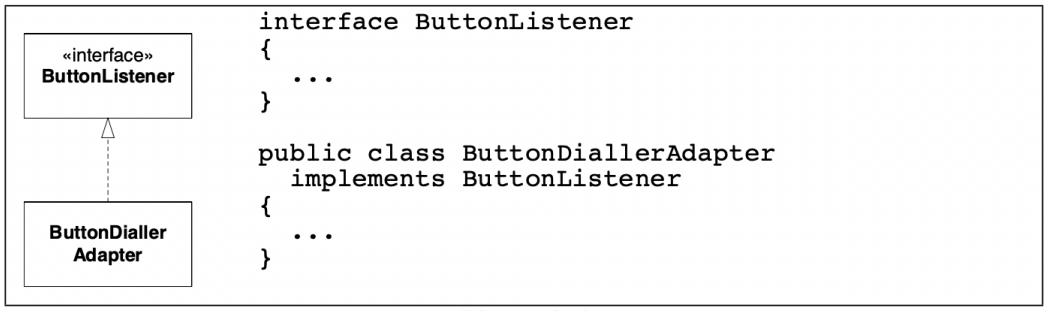

Figure 3-

«interface». All the methods of classes marked with this stereotype are abstract. None of the methods can be implemented. Morevover, «interface» classes can have no instance variables. The only variables they can have are static variables. This corresponds exactly to java interfaces. See Figure 3-9.

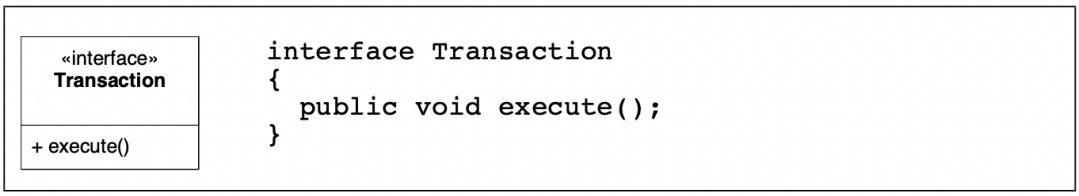

Figure 3-9

Abstract classes

In UML there are two ways to denote that a class, or a method, is abstract. You can write the name in italics, or you can use the {abstract} property. Both options are shown in Figure 3-11.

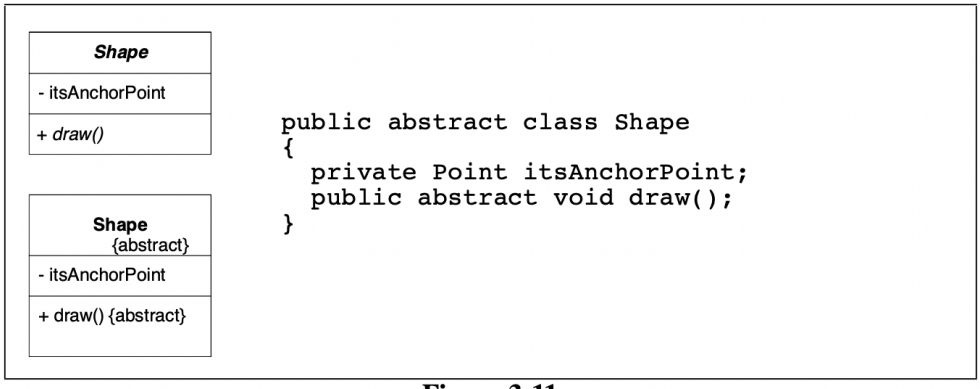

Figure 3-11

Aggregation

Aggregation is a special form of association that connotes a "whole/part" relationship. Figure 3-14 shows how it is drawn and implemented. Notice that the implementation shown in Figure 3-14 is indistinguishable from association. That's a hint.

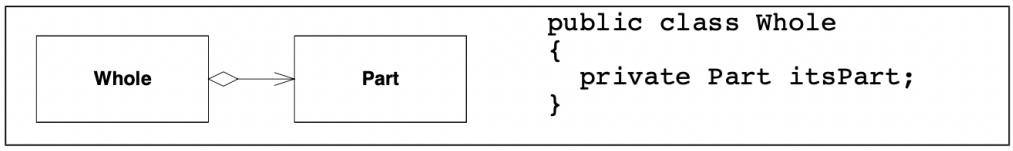

Figure 3-14

Composition

Composition is a special form of aggregation shown in Figure 3-16. Again, notice that the implementation is indistinguishable from association. However, this time the reason is not due to a lack of definition; this time it's because the relationship does not have a lot of use in a Java program. C++ programmers, on the other hand, find a *lot* of use for it.

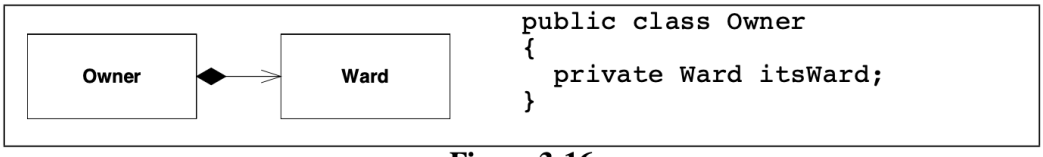

Figure 3-16

Multiplicity

Objects can hold arrays or vectors of other objects; or they can hold many of the same kind of object in seperate instance variables. In UML this can be shown by placing a *multiplicity* expression on the far end of the association. Multiplicity expressions can be simple numbers, ranges, or a combination of both. For example Figure 3-19 shows a `BinaryTreeNode`, using a multiplicity of 2.

Here are the allowable forms:

- Digit. The exact number of elements
- * or 0..* zero to many.
- 0..1 Zero or one. In Java this is often implemented with a reference that can be `null`.
- 1..* One to many.
- 3..5 Three to five.
- 0, 2..5, 9..* Silly, but legal.
