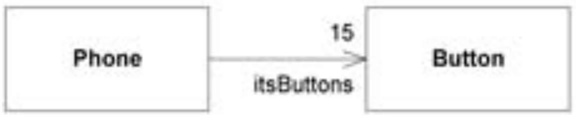
UML Class Diagrams - Introduction

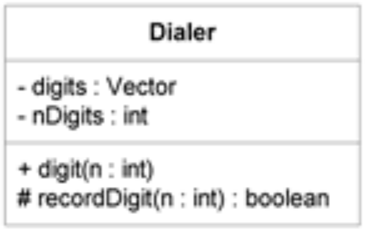
UML class diagrams allow us to denote the static contents of — and the relationships between — classes. This reading is adapted from Robert Martin’s book.

UML class diagrams allow us to denote the static contents of — and the relationships between — classes. In a class diagram we can show the member variables, and member functions of a class. We can also show whether one class inherits from another, or whether it holds a reference to another. In short, we can depict all the source code dependencies between classes.
This can be valuable. It can be much easier to evaluate the dependency structure of a system from a diagram than from source code. Diagrams make certain dependency structures visible. We can see dependency cycles, and determine how best to break them. We can see when abstract classes depend upon concrete classes, and determine a strategy for rerouting such dependencies.
The Basics
Classes



```
public class Phone
{
  private Button itsButtons[15];
}
```

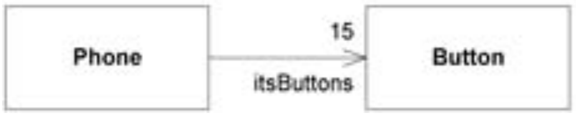
Above shows an elementary form of class diagram. The class named Dialer is represented as a rectangle. This diagram represents nothing more than the code shown to its right.



```
public class Dialer
{
  private Vector digits;
  int nDigits;
  public void digit(int n);
  protected boolean recordDigit(int n);
}
```

This is the most common way you will represent a class. The classes on most diagrams don't need any more than their name to make clear what is going on.

A class icon can be subdivided into compartments. The top compartment is for the name of the class, the second is for the variables of the class, and the third is for the methods of the class. This example shows these compartments and how they translate into code.



```
public class Phone
{
  private Button itsButtons[15];
}
```

Class icon compartments with corresponding code.

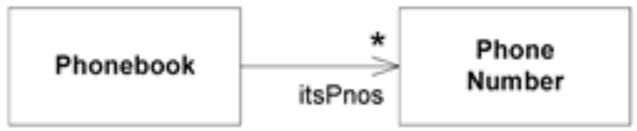
Notice the character in front of the variables and functions in the class icon. A dash (–) denotes private, a hash (#) denotes protected, and a plus (+) denotes public.

UML Class Diagrams - Introduction

The type of a variable, or a function argument is shown after the colon following the variable or argument name. Similarly, the return value of a function is shown after the colon following the function.

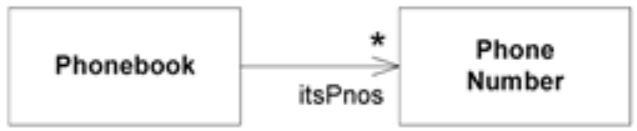
This kind of detail is sometimes useful, but should not be used very often. UML diagrams are not the place to declare variables and functions. Such declarations are better done in source code. Use these adornments only when they are essential to the purpose of the diagram.

Association
Associations between classes most often represent instance variables that hold references to other objects. For example, below 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.



```
public class Phonebook
{
  private Vector itsPnos;
}
```

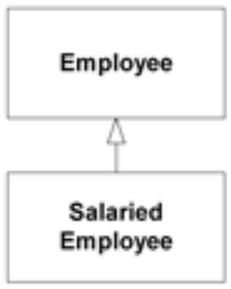
Above we saw that 15 Button objects were connected to the Phone object. In the next example below, we see what happens when there is no limit. A Phonebook is connected to many PhoneNumber objects. The star means many. In Java this is most commonly implemented with a Vector, a List, or some other container type.



```
public class Phonebook
{
  private Vector itsPnos;
}
```

Noticed we avoid using the word “has”. We could have said: “A Phonebook has many PhoneNumbers.” This was intentional. The common OO verbs HASA and ISA have special meanings. Some people engage in “hasa” and “isa” analyses that identify what properties an object has (“tom *has a* cat”) and what an object is (“the cat *is a* kind of animal). Consequently, we try to find more precise verbs to describe the action: terms that are descriptive of what actually happens in software, such as: “is connected to.”

Inheritance
There are many kinds of arrows used in UML - be aware!



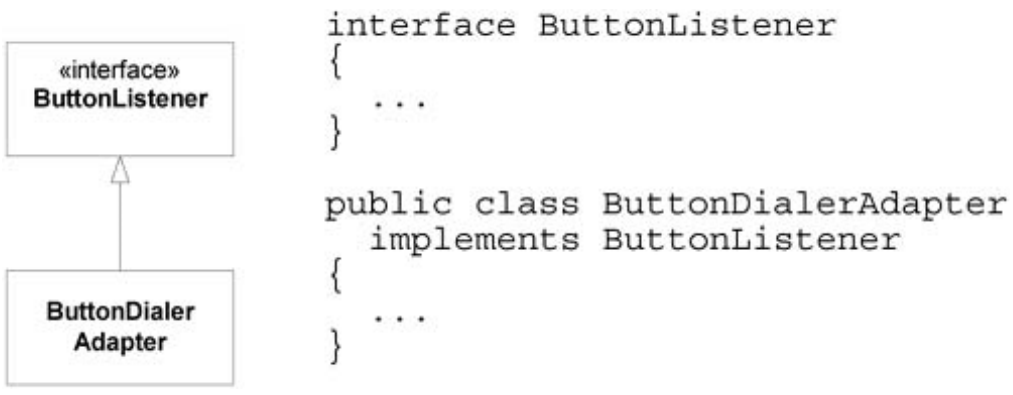
```
public class Employee
{
  ...
}

public class SalariedEmployee extends Employee
{
  ...
}
```

UML Class Diagrams - Introduction

The little arrowhead pointing at `Employee` denotes inheritance [or "generalization" in C++]. 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.

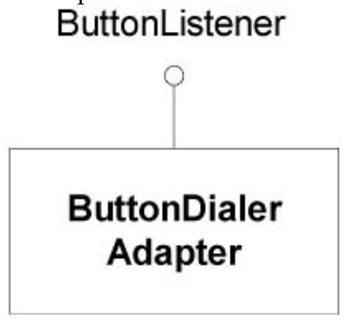
In UML all arrowheads point in the direction of source code dependency. Since it is the `SalariedEmployee` class that mentions the name of `Employee`, the arrowhead points at Employee. So, in UML, inheritance arrows point at the base class.

UML has a special notation for the kind of inheritance used between a Java class and a Java interface. It is shown blow as a dashed inheritance arrow. [Some people just skip using dashes that point to interfaces. Dashes are used often UML, tho, for other purposes.



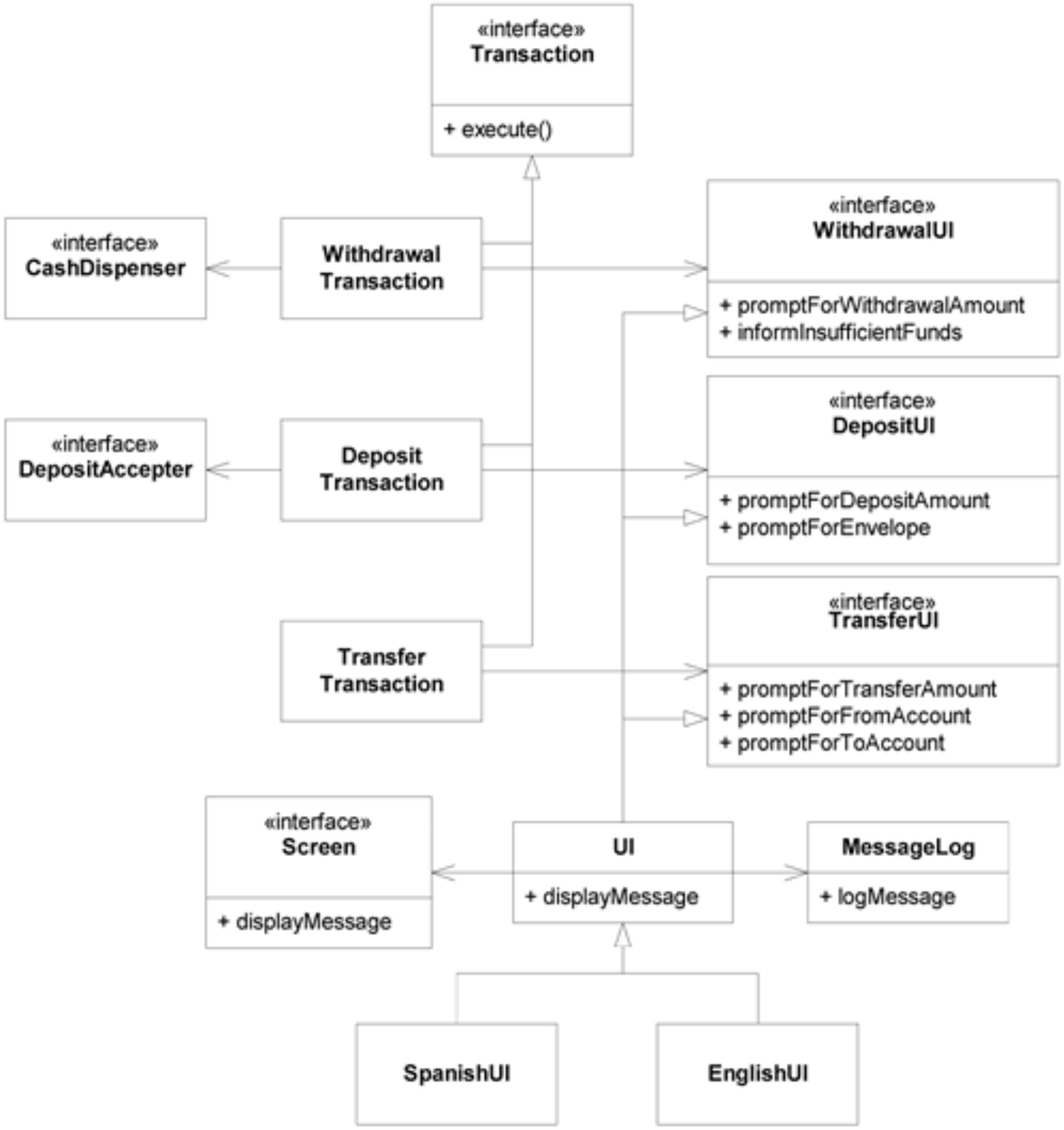
```
interface ButtonListener
{
  ...
}

public class ButtonDialerAdapter
  implements ButtonListener
{
  ...
}
```

Below is another way to convey the same idea. Interfaces can be drawn as little lollipops on the classes that implement them. We often see this kind of notation in COM (Communications) designs.



An Example Class Diagram
Here is a class diagram of part of a bank's ATM system. This diagram is interesting both for what it shows, and for what it does not show. Note that all the interfaces. [Some designers believe this ensures that programmers know what classes are intend to be interfaces and which are intended to be implemented. For example, the diagram immediately tells you that `WithdrawalTransaction` talks to a `CashDispenser` interface. Clearly some class in the system will have to implement the `CashDispenser`, but in this diagram we don't care which class it is.

UML Class Diagrams - Introduction



Note that this all is kinda sloppy ‘cause we have not been particularly thorough in documenting the methods of the various UI interfaces. Certainly WithdrawalUI will need more than just the two methods shown there. What about promptForAccount or informCashDispenserEmpty? Putting those methods in the diagram would just clutter it. By providing a representative batch of methods, I've given the reader the idea. That's all that's really necessary. [UML designers often drop what seems unnecessary in favor of simplicity of design (tho some designs can be awfully complicated!).

Again note the convention of horizontal association and vertical inheritance. This helps to differentiate these vastly different kinds of relationships. Without a convention like this it can be hard to tease the meaning out of the tangle.

Notice the three distinct zones. The *transactions* and their actions are on the left, the various *UI interfaces* are all on the right, and the UI *implementation* is on the bottom. Note also that the connections between the grouping are minimal and regular. In one case it is three associations, all pointing the same way. In the other case it is three inheritance relationships all merged into a single line. The grouping, and the way they are connected help the reader to see the diagram in coherent pieces.

You should be able to see the code as you look at the diagram. Is this code listing below close to what you expected for the implementation of UI?

```
public class UI implements
  WithdrawalUI, DepositUI, TransferUI
{
  private Screen itsScreen;
  private MessageLog itsMessageLog;

  public void displayMessage(String message)
  {
    itsMessageLog.logMessage(message);
    itsScreen.displayMessage(message);
  }
}
```

The Details

There are a *vast* number of details and adornments that can be added to UML class diagrams. Most of the time these details and adornments should not be added. But there are times when they can be helpful.

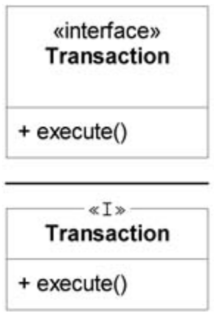
Class stereotypes

Class stereotypes appear between guillemet characters « », usually above the name of the class. You've seen them before above in the «interface» denotation as a class stereotype. «interface» is one of two standard stereotypes that can be used by Java programmers. The other is «utility».

«interface»

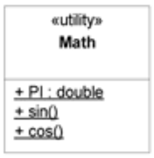
All the methods of classes marked with this stereotype are abstract. None of the methods can be implemented. Moreover, «interface» classes can have no instance variables. The only variables they can have are static variables. This corresponds exactly to Java interfaces. See below image.

UML Class Diagrams - Introduction



```
interface Transaction
{
  public void execute();
}
```

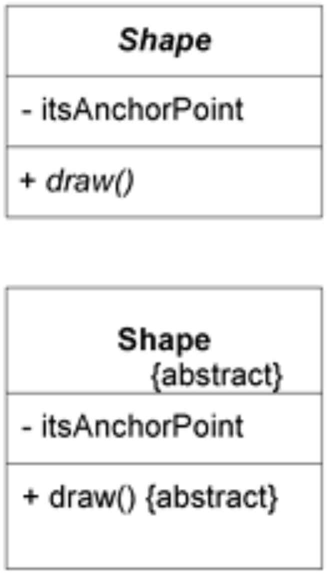
I draw interfaces so often that spelling the whole stereotype out at the white board can be pretty inconvenient. So I often use the shorthand in the lower part of Figure 3-9 to make the drawing easier. It's not standard UML, but it's much more convenient.
«utility»
All the methods and variables of a «utility» class are static. [Earlier authors call these class utilities.}



```
public class Math
{
  public static final double PI =
                         3.14159265358979323;

  public static double sin(double theta){...}
  public static double cos(double theta){...}
}
```

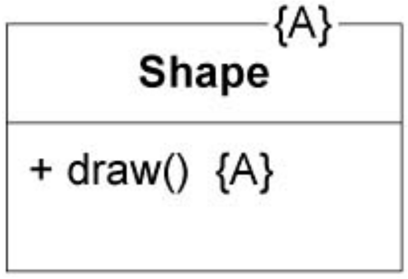
You can make your own stereotypes if you like. Some people often use stereotypes like «persistent», «C-API», «struct», or «function». You just have to make sure that the people who are reading your diagrams know what your stereotype means.

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 below:



```
public abstract class Shape
{
  private Point itsAnchorPoint;
  public abstract void draw();
}
```

Rather than write out all the details (such as during a meeting) some people use a shorthand version that is *not* standard UML. But there it is.

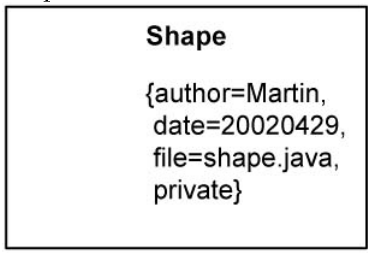


non-standard way of the same diagram above.

Properties
Properties, like {abstract} can be added to any class. They represent extra information that's not usually part of a class. You can create your own properties at any time.
Properties are written in a comma separated list of name–value pairs, like this:
`{author=Martin, date=20020429, file=shape.java, private}`

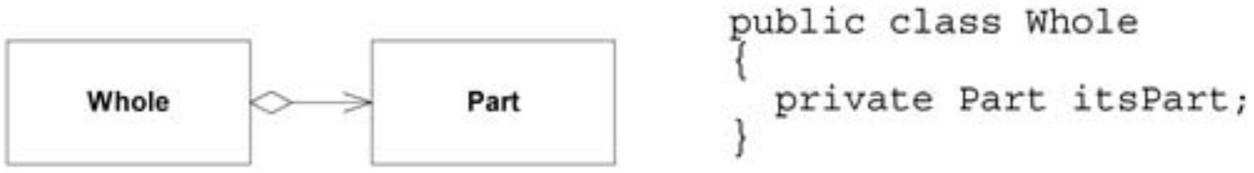
The properties in the preceding example are not part of UML. The {abstract} property is the only defined property of UML that Java programmers would find useful. If a property does not have a value, it is assumed to take the boolean value `true`. Thus, {abstract} and {abstract = true} are synonyms.

Properties are written below and to the right of the name of the class, as shown in Figure 3-13.



Other than the {abstract} property, , the class properties activity is pretty rare.

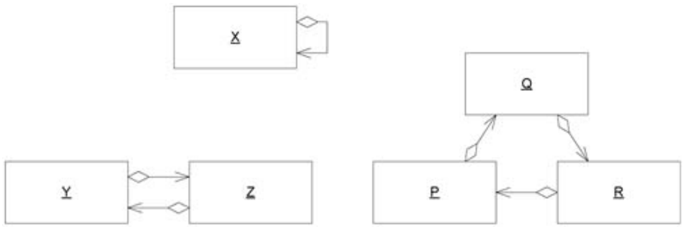
Aggregation
Aggregation is a special form of association that connotes a "whole/part" relationship. Below image shows how it is drawn and implemented. Notice that the implementation shown here is indistinguishable from association. That's a hint.



```
public class Whole
{
  private Part itsPart;
}
```

Unfortunately, UML does not provide a strong definition for this relationship. This leads to confusion because various programmers and analysts adopt their own pet definitions for the relationship. For that reason many people don't use the relationship at all. In fact, this relationship has been dropped from UML 2.0.

The one hard rule that UML gives us regarding aggregations is this: A whole cannot be its own part. Therefore instances cannot form cycles of aggregations. A single object cannot be an aggregate of itself, two objects cannot be aggregates of each other, three objects cannot form a ring of aggregation, and so on. See below for what *not* to do.

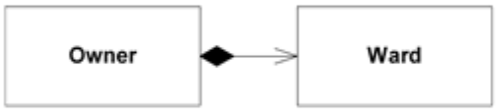


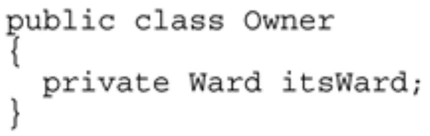
Illegal cycles of aggregation between instances.

I don't find this to be a particularly useful definition. How often am I concerned about making sure that instances form a directed acyclic graph? Not very often. Therefore I find this relationship useless in the kinds of diagrams I draw.

Composition

Composition is a special form of aggregation, as shown in below. 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.

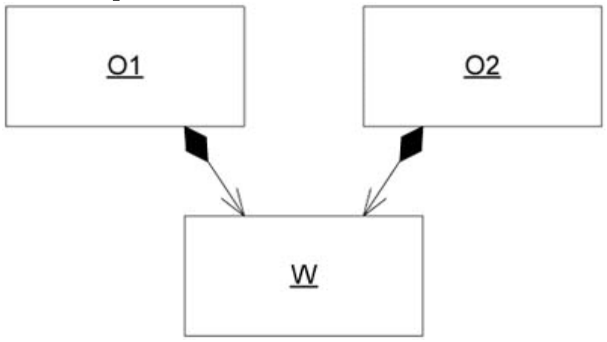




```
public class Owner
{
  private Ward itsWard;
}
```

The same rule applies to composition that applied to aggregation. There can be no cycles of instances. An owner cannot be its own ward. However, UML provides quite a bit more definition.

An instance of a ward cannot be owned simultaneously by two owners. The object diagram below is illegal. Note, however, that the corresponding class diagram is not illegal. An owner can transfer ownership of a ward to another owner.



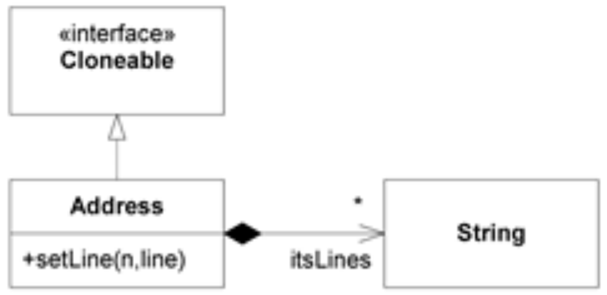
no. don't do this.

Illegal *design above!*

The owner is responsible for the lifetime of the ward. If the owner is destroyed, the ward must be destroyed with it. If the owner is copied, the ward must be copied with it.

In Java destruction happens behind the scenes by the garbage collector, so there is seldom a need to manage the lifetime of an object. Deep copies are not unheard of, but the need to show deep copy semantics on a diagram is rare. So, though I have used composition relationships to describe some Java programs, such use is infrequent.

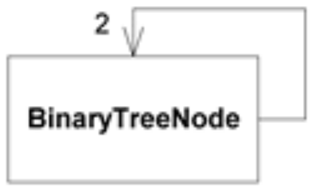
The next figure shows how composition is used to denote deep copy. We have a class named `Address` that holds many `Strings`. Each string holds one line of the address. Clearly, when you make a copy of the Address, you want the copy to change independently of the original. Thus, we need to make a deep copy. The composition relationship between the Address and the Strings indicates that copies need to be deep.



```
import java.util.Vector;

public class Address implements Cloneable{
  private Vector itsLines = new Vector();

  public void setLine(int n, String line) {
    if (n >= itsLines.size())
      itsLines.setSize(n+1);
    itsLines.setElementAt(line, n);
  }

  public Object clone() throws CloneNotSupportedException
  {
    Address clone = (Address)super.clone();
    clone.itsLines = (Vector)itsLines.clone();
    return clone;
  }
}
```

Multiplicity

Objects can hold arrays or vectors of other objects, or they can hold many of the same kind of objects in separate 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.

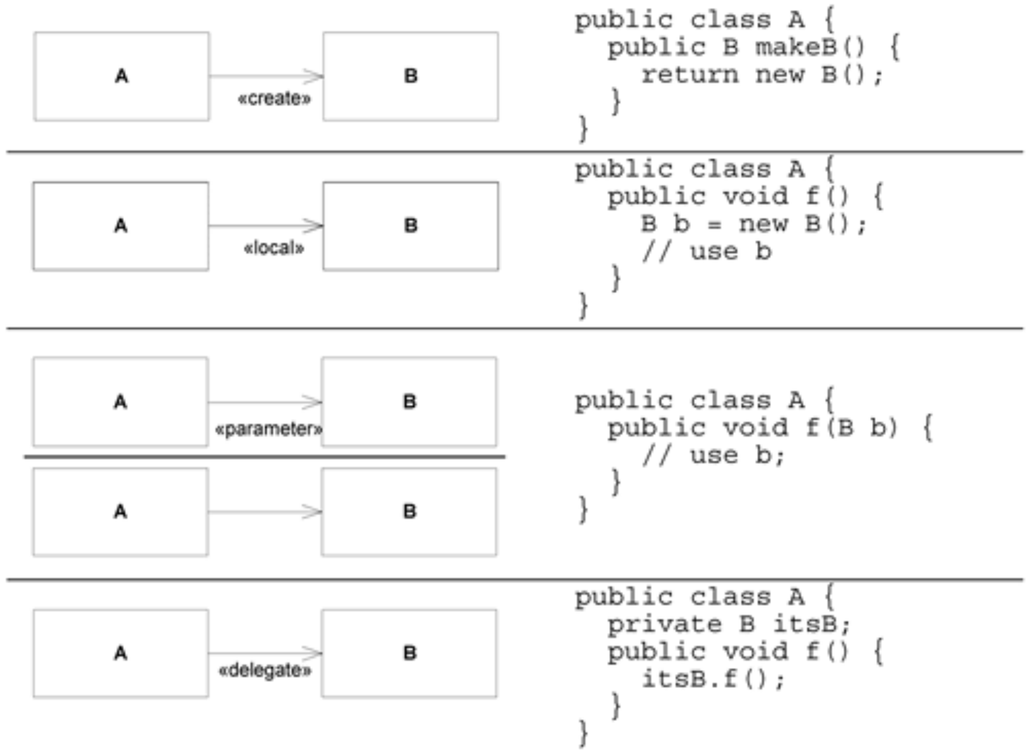


```
public class BinaryTreeNode
{
  private BinaryTreeNode leftNode;
  private BinaryTreeNode rightNode;
}
```

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.

Association stereotypes
Associations can be labeled with stereotypes that change their meaning. Figure 3-20 shows the ones that I use most often.



```
public class A {
  public B makeB() {
    return new B();
  }
}
```

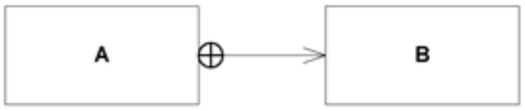
```
public class A {
  public void f() {
    B b = new B();
    // use b
  }
}
```

```
public class A {
  public void f(B b) {
    // use b;
  }
}
```

```
public class A {
  private B itsB;
  public void f() {
    itsB.f();
  }
}
```

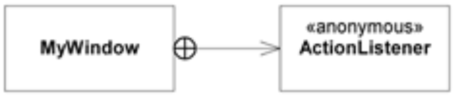
The **«create»** stereotype indicates that the target of the association is created by the source. The implication is that the source creates the target and then passes it around to other parts of the system. In the example I've shown a typical factory.
The **«local»** stereotype is used when the source class creates an instance of the target and holds it in a local variable. The implication is that the created instance does not survive the member function that creates it. Thus, it is not held by any instance variable nor passed around the system in any way.
The **«parameter»** stereotype shows that the source class gains access to the target instance though the parameter of one of its member functions. Again, the implication is that the source forgets all about this object once the member function returns. The target is not saved in an instance variable.
Using dashed dependency arrows, as the diagram shows, is a common and convenient idiom for denoting parameters. I usually prefer it to using the «parameter» stereotype.
The **«delegate»** stereotype is used when the source class forwards a member function invocation to the target. There are a number of design patterns where this technique is applied, such as PROXY, DECORATOR, and COMPOSITE . You might find this notation useful - or not - notice that your analyses will tend to follow the same paths of thought.

Inner classes
Inner (nested) classes are represented in UML with an association adorned with a crossed circle, as shown in Figure 3-21.



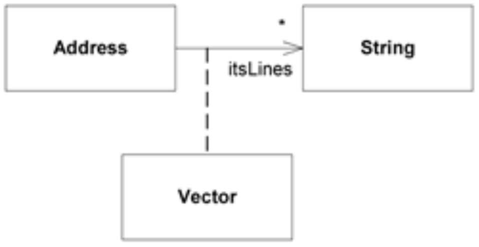
```
public class A {
  private class B {
    ...
  }
}
```

Anonymous inner classes
One of Java’s more interesting features is anonymous inner classes. This notation is concise and descriptive. The anonymous inner class is shown as a nested class that is given the «anonymous» stereotype, and is also given the name of the interface it implements.



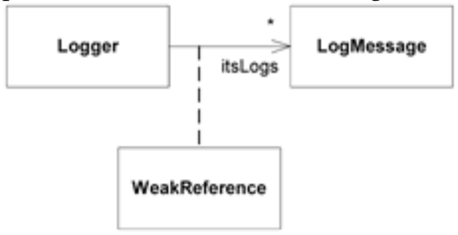
```
public class Window {
  public void f() {
    ActionListener l =
      new ActionListener() {
        // implementation
      };
  }
}
```

Association classes
Associations with multiplicity tell us that the source is connected to many instances of the target, but the diagram doesn’t tell us what kind of container class is used. This can be depicted by using an association class:

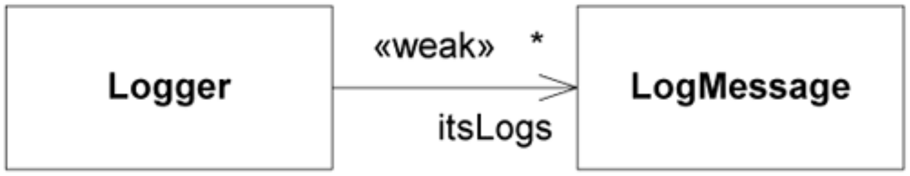


```
public class Address {
  private Vector itsLines;
};
```

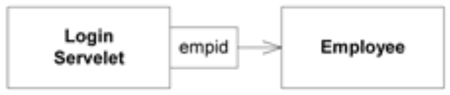
Association classes show how a particular association is implemented. On the diagram they appear as a normal class connected to the association with a dashed line. As Java programmers we interpret this to mean that the source class really contains a reference to the association class, which in turn contains references to the target.
Association classes can also be used to indicate special forms of references, such as weak, soft, or phantom references. The first figure below is the weak relationship.



```
public class Logger {
  private WeakReference itsLogs;
}
```



Association qualifiers
Association qualifiers are used when the association is implemented through some kind of key or token, instead of with a normal Java reference. The next figure shows a `LoginServlet` associated with an `Employee`. The association is mediated by a member variable named `empid`, which contains the database key for the `Employee`.



```
public class LoginServlet {
  private String empid;
  public String getName() {
    Employee e = DB.getEmp(empid);
    return e.getName();
  }
}
```

I find this notation useful in rare situations. Sometimes it's convenient to show that an object is associated to another through a database or dictionary key. It is important, however, that all the parties reading the diagram know how the qualifier is used to access the actual object. This is not something that's immediately evident from the notation.

Conclusion

UML Class Diagrams - Introduction

UML covers so much in detail, but skips a lot, too, that only an overview and practicing some areas that are comfortable to you and will likely be part of your future work can be addressed in class. Using too little of UML is almost always better than using too much.

Booch, G. (1994). *Object Oriented Analysis and Design with Applications.* Redwood City, CA.: Benjamin Cummings.
Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns*. Reading, Mass.: Addison-Wesley.