Design Patterns in Java

A brief introduction with useful examples

José Santos

November 2010

1 Fundamental Object Oriented Concepts in Java

1.1 Constructors

In Java, the class designer can guarantee initialization of every object by providing a special method called a **constructor**. If a class has a constructor, Java automatically calls that constructor when an object is created, before users can even get their hands on it. So initialization is guaranteed. The name of the constructor is the same as the name of the class. Note that the coding style of making the first letter of all methods lowercase does not apply to constructors, since the name of the constructor must match the name of the class exactly. A default constructor is one with no arguments that is used to create a basic object. If you create a class that has no constructors, the compiler will automatically create a default constructor for you.

1.2 Method Overloading

Method overloading allow the same method name to be used with different argument types. Java can determine which method you mean because each overloaded mehtod must have a unique list of argument types. Naturally, methods cannot be distinguished by their return types, since one can call a method and ignore its return value. This is often referred to *calling a method* for ist side effect.

1.3 The this keywork

When you invoke a method on an object, you don't explicitly pass a reference to the object as an argument to the method. However, this is implicitly done. So, as expected, inside the method code you may access the object fields as well as the other methods of that object. Now, suppose you're inside a method and you'd like to get a reference to the current object. Since that reference is passed secretly by the compiler, there is no identifier for it. However, for this purpose there is a keyword: **this**. The **this** keywork - which can be used only inside a method - produces the reference to the object the method has been called for. However, if you are calling another method of your class from within another method of your class you don't need to use the this keyword. You simply call the method. The current **this** reference is automatically used for the other method. Inside a constructor the **this** keywork takes on a different meaning when you give it an argument list. It makes an explicit call to the constructor that matches that argument list. However, you cannot call two constructors in such a way. Normally, the **this** keyword is used to distinguish between an argument.

1.4 Static Methods

With the **this** keyword in mind, you can more fully understand what it means to make a method **static**. It means that there is no **this** for that particular method. You cannot call non-static methods from inside static methods, and you can call a static method without any object. In fact, that's primarily what a static method is for. It's as if you're creating the equivalent of a global function from C.

1.5 Initialization

As we have already mentioned, the *initialization* of the object is performed in the constructor method - which we denote by *constructor initialization.* However, one direct way to do this is simply to assign the value at the point where it is defined in the class - this is called *automatic initialization.* When using both kinds of initialization one must keep in mind that automatic initialization happens before constructor initialization. To summarize:

1. The first time an object of the class is created or a static method or static field is accessed, the java interpreter must locate the class, which it does by searching through the classpath.
2. The class is loaded. All of its static initializers are run in the order in which they were specified. Thus, static initialization takes place only once, as the class object is loaded for the first time.
3. When the program tries to create a new object, the constructor process starts by allocating storage for the object on the heap.
4. This storage is wiped to zero, automatically setting all the primitive variables to zero and all the reference variables to null.
5. All automatic initializations are executed.
6. Constructors are now executed.

It is common to group static initializations inside a special *static clause.* Similarly, you can also group non-static initialization inside a special statement, which we illustrate in an example below:

```
public class X {
   static int x, y;
   int i,j;

   static {
     x = 47;
     y = 74;
   }

   {
     i = 2;
     j = 3;
   }
}
```

1.6 Arrays

There are three things which separate arrays from other types of containers: efficiency, type and the ability to hold primitives. The array is the most efficient way to store and randomly access a sequence of object references. The array is a simmple linear sequence which makes element access fast. The cost of this speed is that the size of the array is fixed and cannot change during the lifetime of the object. For example, with an **Arraylist** you won't get this problem, it automatically allocates more space, creating a new **Arraylist** and copying all the objects of the old Arraylist to the new one.
Before generics the container classes dealt with objects as if they did not have any specific type. They treated them as type **Object**. Arrays are superior to pre-generic containers because you create an array to hold a specific type. This means that you get *compile time* type checking to prevent you from inserting the wrong type or mistake the type that you are extracting.
Naturally, arrays are first class objects. The array identifier is actually a reference to a true object that is created on the heap. This is the object that holds the references to all the other objects which are stored in the array; it can be created implicitly in the array initialization or explicitly using the **new** keyword.
In java Arrays are declared in the following way:

```
int[] a1;
```

There are several ways you can initialize an array. The following examples illustrates all of them.

```
class Pessoa {
   String nome;
   int idade;

   public Pessoa(String nome) {
      this.nome = nome;
      this.idade = idade;
   }
}

public class ArrayTester{
   public static void main(String[] args) {
      Pessoa[] pessoas1;
      Pessoa[] pessoas2 = new Pessoas[2];

      Pessoa[] pessoas3 = {new Pessoa(``jose'', 24), new Pessoa(``pedro'', 21));
      pessoas2[0] = new Pessoa(``Nina'', 47);
      pessoas2[1] = new Pessoa(``Fernando'', 48);

      pessoas1 = pessoas2;
   }
}
```

The array pessoas2 was initialized by *aggregate initialization*. What if you want to create an anonymous array (for example: to pass it as an argument a function)? In this situation you can do as illustrated bellow:

```
methodXPTO(new Pessoa[]{ new Pessoa(``jose'', 24),
        new Pessoa(``Raquel'', 20), new Pessoa(``Ines'', 16)});
```

This kind of initialization is sometimes referred to *dynamic aggregate initialization*.
With respect to the lenght of the array, it is important to emphasize that **length** refers to size of the array. It does not refer to number of elements that the array contains in a given execution moment.

1.7 The JIT compiler

The JIT (*just in time*) compiler is one of the most important source of speedups in the JVM. It partially or fully converts a program into native machine code so that it doesn't need to be interpreted by the JVM and thus runs must faster. When a class must be loaded (typically, the first time you want to create an object of that class or when you want to invoke a static method), the **.class** file is located, and the byte codes for that class are brought into memory. At this point one approach is to simply JIT compile all the code, but this has two drawbacks:

- It takes a lot of time.
- It increases the size of the executable.

An alternative approach is lazy evaluation, which means that the code is not JIT compiled until necessary. Thus, code that never gets executed might never be JIT compiled.

2 Garbage Collection

Java provides a method called **finalize()** that you can define for your class. When the garbage collector is ready to release the storage used for your object, it will first call **finalize()**, and only on the next garbage-collection pass will it reclaim the object's memory. So if you to choose to **finalize()**, it gives you the ability to perform important cleanup at the *time of garbage collection*.
Although Java lets you implement a method which defines what to do before the garbage collector collects an object, one must have in mind two important things:

- Your objects might not get garbage collected.
- Garbage collection is not destruction.
- Garbage collection is only about memory.

What this means is that if there is some activity that must be performed before you no longer neeed an object, you must perform that activity yourself. The only reason for the existence of the garbage collector is to recover memory that your program is no longer using. So any activity that is associated with garbage collection, most notably the **finalize()** method, must also be only about memory and its deallocation. Does this mean that if your object contains other objects, **finalize()** should explicitly realse those objects? No!!! The garbage collector takes care of the release of all object memory regardless of how the object is created. So what's the use of finalize? Basically, the verification of the *termination condition* of an object. For example, if the object represents an open file, that file should be closed by the programmer before the object is garbage collected.

2.1 How a garbage collector works

First of all, in Java everything (except for primitive types) is allocated on the heap. When the garbage collector steps in, it releases the memory associated with object for which there are no more references and it compacts all objects on the heap, thus moving the "heap pointer" closer to the beginning of the page and farther away from page fault.
A simple but slow garbage collection technique is called *reference counting*. This means that each object contains a reference counter, and every time a reference is attached to an object, the reference count is increased. Every time a reference goes out of scope or is set to **null**, the reference count is decreased. Thus, managing reference counts is a smll but constant overhead that happens throughout the lifetime of your program. The garbage collector moves through the entire list of objects, and when it finds one with a reference count of zero it releases that storage. The one drawback is that if objects circularly refer to each other they can have nonzero reference counts while still being garbage. Locating such self-referential groups requires significant extra work for the garbage collector.

In faster schemes, garbage collection is not based on reference counting. Instead, it is based on the idea that any nondead object must ultimately be traceable back to a reference that lives either on the stack or in static storage. The chain might go through several layers of object. Thus, if you start in the stack and the static storage area and walk through all the references, you will find all the live objects. For each reference that you find, you must trace into the object that it points to and then follow all the references in that object, tracing into the objects they point to... until you've moved through the entire web that originated with the reference on the stack or in the static storage. Each object that you move through must still be alive. Note that there is no problem with detached self-referential object groups - these are simply not found, and are therefore automatically garbage.
One of these variants is *stop-and-copy.* This means that - for reasons that will become apparent - the program is first stopped (this is not a background collection scheme). Then, each live object that is found is copied from one heap to another, leaving behind all the garbage. In addition, as the objects are copied into the new heap, they are packed end-to-end, thus compacting the new heap.
Stop-and-copy may seem a reasonable approach, but once a program becomes stable, it might be generating little or no garbage. Despite that, a copy collector will still copy all the memory from one place to another, which is wasteful. To prevent thsi some JVMs detect that no new garbage is being generated and switch to a different scheme. This other scheme is called *mark-and-sweep.*
Mark-and-sweep follows the same logic of starting from the stack and static storage and tracing through all teh references to find live objects. However, each time it finds a live object, that object is marked by setting a flag in it, but the object isn't collected yet. Only when the marking process is finished does the sweep occur. During the sweep, the dead objects are released. However, no copying happens, so if the collector chooses to compact a fragmented heap, it does so by shuffling objects around.
In the JVM described here memory is allocated in big blocks. If you allocate a large object, it gets its own block. Strick *stop-and-copy* requires copying every live object from the source heap to a new heap before you could free the old one, which translates to lots of memory. With blocks, the garbage collection can typically copy objects to dead blocks as it collects. Each block has a *generation count* to keep track of whether it's alive.

3 Structuring your code: packages

The **package** keyword is used in Java to group classes together into a library. A package is what becomes available when using the **import** keyword.

```
import java.util.*;
```

If you do not want to use the whole package but only a single class, you can name the class in the **import** statement:

```
import java.util.ArrayList;
```

One of the main reasons to structure your code in packages is to manage name spaces.
When you create a source-code file for Java, it's commonly called a *compilation unit*. Each compilation unit must have a name ending in **.java**, and inside the compilation unit there can be a unique **public** class that must have the same name as the file (including capitalization, but excluding the **.java** filename extension). There can be only one public class in each compilation unit. If there are additional classes in that compilation unit, they are hidden from the world outside that package because they are not public, and they comprise support classes for the main public class.
When you compile a **.java** file, you get an output file for each class in the **.java** file. Each output file has the name of a class in the **.java** file, but with an extension of **.class**. Thus you can end up with quite a few **.class** files from a small number of **.java** files. A working program is a bunch of **.class** files, which can be packaged and compressed into a Java Archive (JAR) file. The Java interpreter is responsible for finding, loading and interpreting these files. A library is a group of these files. Each file has one class that is public, so there's one component for each file. If you want to say that all these components belong together, that's where the **package** keyword comes in. Thus, when you say:

```
package mypackage;
```

at the begining of a file, you are stating that this compilation unit is part of a library named **mypackage**. Or, put another way, you are saying that the public class name within this compilation unit is under the umbrella of the name **mypackage**, and anyone who wants to use this class must either fully specify its name or use the **import** keyword in combination with **mypackage**. Note that the convention for Java package names is to use all lowercase letters, even for intermediate words.
It is important to note that a package never really gets packaged into a single file. So, the **.class** files which comprise the package may be scattered through several different directories. However, a logical thing to do is to put all the **.class** files for a particular package into a single directory; that is, use the hierarchical file structure of the operating system to your advantage. Collecting the package files into a single subdirectory solves two other problems:

- Creating unique package names.
- Finding those classes that might be buried in the directory structure somewhere.

This is achieved by encoding the name of the location of the **.class** into the name of the package. By convention, the first part of the package name is the reversed Internet domain name of the creator of the class. Since Internet domains are guaranteed to be unique, by following this convention, your package name will be unique and you will never have name clash. But how can the JVM find the **.class** files? Resolving the **package** name into a directory on your machine.

The Java interpreter proceeds as follows. First, it finds the environment variable CLASSPATH, CLASSPATH contains one or more directories that are used as roots in a search for **.class** files. Starting at that root, the interpreter will take the package name and replace each dot with a slash to generate a path name form the CLASSPATH root (so **package foo.bar.baz** becomes **foo**
bar
baz or **foo/bar/baz**). This is then concatenated to the various entries in the CLASSPATH. There's a variation with JAR files, however. You must put the name of the JAR file in the CLASSPATH, not the just the path where it's located.

4 Java Access Specifiers

Access control is often referred to as "implementation hiding". Wrapping data and methods within calsses in combination with implementation hiding is often called *encapsulation.* Access control puts boundaries within a data type for two important reasons:

- To establish what the client programmers can and can't use.
- To separte the interface from the implementation. This is the most important reason: this allows you to change your code with the guarantee that all the that uses your code does not need to be changed.

When used, the Java access specifiers **public**, **protected** and **private** are placed in front of each definition for each member in your class, whether it's a field or a method. The default access has no keyword, but it is commonly referred to as *package access* (and sometimes "friendly"). It means that all the other classes in the current package have access to that member, but to all the classes outside of this package, the member appears to be **private**. Since a compilation unit can belong only to single package, all the classes within a single compilation unit are automatically available each other via package access.
The only way to grant access to a member is to:

- Make the member **public**. The everybody, everywhere, can access it.
- Give the member package access by leaving off any access specifier, and put the other classes in the same package. Then the other classes in that package can access the member.
- An inherited class can access protected members of its superclass as well as public members. However, it can only access *package-access* members if the two classes are in the same package, which might not be the case.
- You should provide *accessor/mutator* methods that read and change the fields of your class.

5 The Singleton design pattern

Note that a class cannot be **private** (that would make it acceccible to no one but the class) or **protected**. So, there are only two choices for class access: **package access** or **public**. If you don't want anyone else to have access to that class, you can make all the constructors **private**, thereby preventing anyone but you, inside a **static** member of the class, from creating an object of that class.
The idea of restricting the access to the constructor of a given class is the basic principle behind the singleton design pattern. The singleton desing pattern is used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalised to systems that operate efficiently when exactly one object exists, or that restrict the instantiation to a certain number of objects (say, five).
The singleton design pattern is obviously an elegant way to introduce global state in your program. However it is preferred to global variables because:

- They do not pollute the global namespace.
- They permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.

```
public calss Singleton {
   private static final Singleton instance = new Singleton();

   private Singleton() {}

   public static Singleton getInstance(){
      return instance;
   }
}
```

6 Composition and Inheritance

6.1 Composition versus Aggregation

Object composition is a way to combine simple objects into more complex ones. Composited objects are often referred to as having a *has a* relationship. In UML, composition is depicted as a filled diamond and a solid line. The more general form, *aggregation*, is depicted as an unfilled diamond and a solid line.
Aggregation differs from ordinary composition in that it does not imply ownership. In composition, when the owning object is destroyed, so are the contained objects. In aggregation, it is not necessarily true. For example, a university owns various departments and each department has a number of professors. If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist. Therefore a university can be seen as a composition of departments, whereas departments have an aggregation of

professors. In addition, a Professor can work in more than one department and even in more than one university, but a department cannot be part of more than one university.

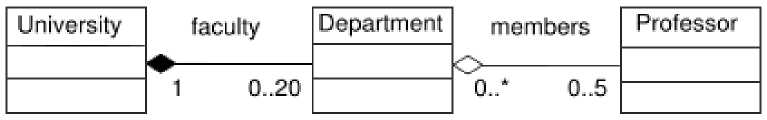

Figure 1: Aggregation versus Composition

Primitives that are fields in a class are automatically initialized to zero. But the object references are initialized to **null**. It makes sense that the compiler doesn't just create a default object for every reference, because that would incur unnecessary overhead in many cases. If you want the references initialized, you can do it:

- At the point the objects are defined. This means that they will always be initialized before the constructor is called.
- In the constructor for that class.
- Right before you actually need to use the object. This is often called: *lazy initialization.*

6.2 Inheritance

In OOP, *inheritance* is a way to compartmentalize and reuse code by creating collections of attributes and behaviors called *objects* which can be based on previously created objects. In *classical inheritance* where objects are defined by *classes*, classes can inherit other classes. The new classes, known as *subclasses*, inherit attributes and behavior of the pre-existing classes, which are referred to as *superclasses*. The inheritance relationship of *subclasses* and *superclasses* gives rise to a hirarchy.
In Java, it turns out that you are always doing inheritance when you create a class, because unless you explicitly inherit form some other class, you implicitly inherit form Java's standard root class **Object**. When you want to inherit from a specific class, you must put the keyword **extends** followed by the name of the base class.
Java has the keyword **super** that refers to the superclass that the current class has been inherited from.
When considering inheritance, the order of initialization is as follows:

- The storage allocated for the object is initialize to binary zero before anything else happens.
- The base-class constructor is called. This step is repeated recursively such that the root of the hierarchy is constructec first, followed by the next-derived class, etc., until the most-derived class is reached.

- Member initializers are called in the order of declaration.
- The body of the derived-class constructor is called.

When using composition and inheritance to create a new class, most of th etime you won't have to worry about cleaning up: subobjects can usually be left to the gargabe collector. If you do have cleanup issues, you must be diligent and create a **dispose()** method. Of course, with inheritance you must override **dispose()** in the derived class if you have any special cleanup that must happen as part of garbage collection. When you override **dispose()** in an inherited class, it is important to remember to call the base-class version of **dispose()**. The oreder of disposal should be the reverse of the order of initialization, in case one subobject is dependent on another. For fields, this means the reverse of the order of declaration (since fields are initialized in declaration order). For base classes, you should perform the derived-class cleanup first, then perform the base-class cleanup.

6.3 Inheritance versus Composition

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you embed an object so that you can use it to implement functionality in your new class, but the user of your new class sees the interface you have defined for the new class rather than the interface from the embedded object. For this effect, you embed **private** objects of existing classes inside your new class.
When you inherit, you take an existing class and make a special version of it. In general, this means that you're taking a general-purpose class and specializing it for a particular need. The *is-a* relation is expressed with inheritance, and the *has-a* relation is expressed with composition. One of the clearest ways to determine whether you should use composition or inheritance is to ask whether you will ever need to upcast from your new class to the base class. If you must upcast, then inheritance is necessary, but if you don't need to upcast, when you should look closely at whether you need inheritance.

6.4 The final keyword

Java's final keyword has slightly different meanings depending on the context, but in general it says: "This cannot be changed". You might want to prevent changes for two reasons: design and efficiency.
Many programming languages have a way to tell the compiler that a piece of data is constant. A constant is useful for two reasons:

- It can be a *compile-time constant* that won't ever change.
- It can be a value initialized at run time that you don't want changed.

It is important to note that if you make an object final, you are saying that you cannot change its reference. However, the object itself can be modified; Java does not provide a way to make any arbitrary object a constant.

Java allows the creation of *blank finals*, which are fields that are declared as **final** but are not given an initialization value. In all cases, the blank final must be initialized before it is used, and the compiler ensures this.
Java allows you to make arguments **final** by declaring them as such in the argument list. This means that inside the method you cannot change what the argument reference points to.
There are two reasons for **final** methods. The first is to put a lock on the method to prevent any inheriting class from changing its meaning. This is done for design reasons when you want to make sure that a method's behavior is retained during inheritance and cannot be overriden. The second reason for **final** methods is efficiency.
Naturally, any **private** methods in a class are implicitly **final**. Because you can't access a **private** method, you can't override it. This issue can cause confusion, because if you try to override a **private** method (which is implicitly **final**), it seems to work, and the compiler does not give an error message. However, it must be understood that "overriding" can only occur if something is part of the base-class interface. That is, you must be able to upcast an object to its base type and call the same method. If a method is **private**, it isn't part of the base-class interface. It is just some code that's hidden away inside the class, an it just happens to have that name.

```
public class PrivateOverride {
   private void f() {
      System.out.println(‘‘private f()’’);
   }

   public static void main(String[] args) {
      PrivateOverride po = new Derived();
      po.f();
   }
}

class Derived extends PrivateOverride {
   public void f() {
      System.out.println(‘‘derived f()’’);
   }
}
```

As you might expect considering the previous discussion, this program prints the following message: "private f()".
When you say that an entire class is **final** (by preceding its definition with the final keyword), you state that you don't wnat to inherit from thsi class or allow anyone else to do so.

6.5 Initialization - yet again

When the classloader tries to load a class, it will check if it has some base class. If the class being loaded has some base class, this base class will be loaded first. This process proceeds recursively untill all necessary classes have been loaded. After all necessary classes have been loaded, the object can be constructed. First, all the primitives in the object are set to zero and all references are set to null. Then the base-class constructor will be called. After the base-class constructor completes, the instance variables are initialized in textual oreder. Finally, the rest of the body of the constructor is executed.

6.6 Polymorphism

Taking an object reference and treating it as a reference to its base type is called *upcasting*. But, why should anyone intentionally forget the type of an object?

```
class Guitar extends Instrument {
  public void play(Note n) {
     System.out.println(‘‘Guitar.play() ’’ + n);
  }
}

class Flute extends Instrument {
  public void play(Note n) {
     System.out.println(‘‘Flute.play() ’’ + n);
  }
}

pulic class Music1 {
  public static void tune(Guitar g) {
    g.play(Note.MIDDLE_C);
  }

  pubic static void tune(Flute f) {
    f.play(Note.MIDDLE_C);
  }

  public static void main(String[] args) {
     Guitar g = new Guitar();
     Flute f = new Flute();

     tune(f);
     tune(g);
  }
}
```

This example illustrates why we need to upcast: we had to implement two "tune" methods whereas we could have only implemented one:

```
public static void tune(Instrument i) {
   i.play(Note.MIDDLE_C);
}
```

Note that in this example the method tune receives an Instrument reference. So how can the compiler possibly know that this Instrument reference points to a Flute and not a Guitar? In order to understand this, we must understand the iissue of dynamic binding.

6.7 Mehtod-call Binding

Connecting a method call to a method body is called *binding.* When binding is performed before the program is run (by the compiler and linker), it's called *early binding.* When the binding occurs at run time, it is deemed *late binding* (also called *dynamic binding* and *run-time binding*). When a language implements *late binding* there must be some mechanism to determine the type of the object at run time and to call the appropriate method. The late binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects.
All method binding in Java uses late binding unless the method is **static** or **final** (**private** methods are implicitly **final**). This clarifies one of the advantages of declaring a method final: it turns off dynamic binding, or rather it tells the compiler that dynamic binding is not necessary. This allows the compiler to generate slightly more efficient code to **final** method calls.
What happens if you are inside a constructor and call a dynamically bound method of the object being constructed? Conceptually, the constructor job is to bring the object into existence. Inside any constructor, the object being constructed may be only partially formed. You can know that the base-class objects have been initialized but you do not know which classes are inherited from you. A dynamically bound method, however, reaches "outward" into the inheritance hierarchy. It calls a method in the derived class. If you do this inside a constructor, you will be calling a method that might manipulate members that have not been initialized yet.

```
abstract class Glyph {
   abstract void draw();

   Glyph() {
      System.out.println(‘‘Glyph() before draw’’);
      draw();
      System.out.println(‘‘Glhph() after draw’’);
   }
}
```

```
class RoundGlyph extends Glyph {
   private int radius = 1;

   RoundGlyph(int r) {
      radius = r;
      System.out.println(``RoundGlyph(), radius = '' + radius);
   }

   void draw(){
      System.out.println(``RoundGlyph.draw(), radius = '' + radius);
   }
}

public class PolyConstructors {
   public static void main(String[] args) {
      new RoundGlyph(5);
   }
}
```

This program will print:

```
            Glyph() before draw
            RoundGlyph.draw(), radius = 0
            Glyph() after draw
            RoundGlyph(), radius =  5
```

The punchline is: "Inside a constructor do as little as possible to set the object into a good state, and if you can possibly avoid it, don't call any methods." The only safe methods to call inside a constructor are those that are **final** in the base class (this also includes **private** methods that are implicitly **final**).

6.8 Abstract Classes

An *abstract classes* is a class implemented in order to establish what's in common with all the derived classes. Generally, objects of an *abstract class* have no meaning, so is reasonable to establish that an abstract class cannot be instantiated.
In Java, it is possible to delare a method *abstract*:

```
abstract void f();
```

A class containing abstract methods is called an abstract class: if a class contains one or more abstract methods the class itself must be qualified as **abstract**. When you inherit form an abstract class and you want to make objects of the new type, you must provide mehtod definitions for all the abstract methods in the base class. If you don't, then the derived class is also abstract, and the compiler will force you to qualify that class with the abstract keyword.

It is possible to create a class as abstract without including any abstract methods. This is useful when you have got a class in which it does not make sense to have any abstract methods, and yet you want to prevent any instances of that class.

6.9 Pure Inheritance versus Extension

When studying inheritance, it would seem that the cleanest way to create an inheritance hierarchy is to take the "pure" aproach. That is, only methods that have been establish in the base class or **interface** are to be overridden in the derived class. This can be called a pure "is-a" relationship because the interface of a class establishes what it is. When you declare that a class **extends** another class, you are establishing a "is-like-a" relationship, because the derived class is like the base class - it has the same fundamental interface - but it has other features that require additional methods to implement. The extended part of the interface in the derived class is not available from the base class, so once you upcast, you can't call the new methods.

7 The State Design Pattern

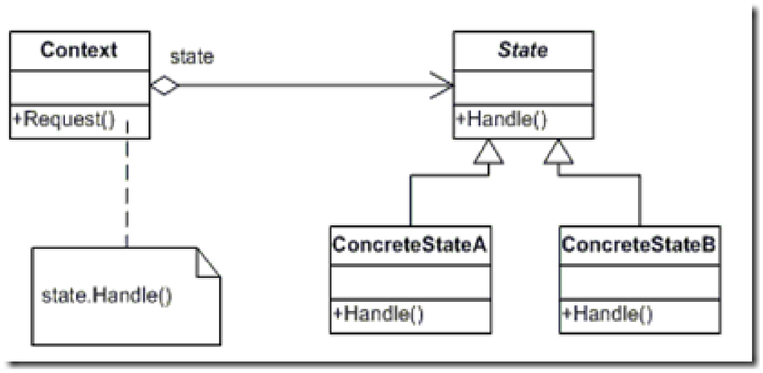

Figure 2: The State Design Patter

We introduce here the State design pattern as an example of designing with inheritance. The goal of this design pattern is to allow an object to alter its behavior when its internal state changes. The object will appear to change its class. One should use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enu-

merated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

```
abstract class Actor {
   public abstract void act();
}

class HappyActor extends Actor {
   public void act() {
      System.out.println(``Happy Actor'');
   }
}

class SadActor extends Actor {
   public void act() {
      System.out.println(``Sad Actor'');
   }
}

class State {
   private Actor actor = new HappyActor();

   public void change() { actor = new SadActor(); }
   public void performPlay() { actor.act(); }
}
```

8 Interfaces and multiple inheritance

The **interface** keyword produces a completely abstract class, one that provides no implementation at all. The **interface** keyword takes the **abstract** concept one step further. You could think of it as a "pure" **abstract** class. It allows the creator to establish the form for a class: method names, argument lists, and return types, but no method bodies. An **interface** can also contain fields, but these are implicitly **final** and **final**. An **interface** provides only a form, but no implementation. An interface says: "This is what all classes that implement this particular interface look like." So, the **interface** keyword is used to establish a protocol between classes.
To create an interface, use the **interface** keyword instead of the **class** keyword. To make a class that conforms to a particular **interface** (or group of interfaces), use the **implements** keyword, which says: "The **interface** is what it looks like, but now I'm going to say how it works." You can choose to explicitly declare the method declarations in an **interface** as public, but they are **public** even if you don't say it. So when you **implement** an **interface**, the methods from

the **interface** must be defined as public. Otherwise, they would default to package access, and you would be reducing the accessibility of a method during inheritance, which is not allowed by the Java compiler.
If you do inherit from a non-interface, you can inherit from only one. All the rest of the base elements must be **interfaces**. Note that when you combine a concrete class with interfaces this way, the concrete class must come first, then the interfaces. You place all the interface names after the **implements** keyword and separate them with commas. You can have as many interfaces as you want, each one becomes an independent type that you can upcast to.
Keep in mind that the core reason for interfaces is to be able to upcast to more than one base type. However, a second reason for using interfaces is the same as using an abstract base class: to prevent the client programmer from making an object of this class and establish that it is only an interface. How can you decide between an interface and an abstract class? If you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change to an abstract class, or if necessary to a concrete class.
When dealing with *multiple inheritance* one must be particularly careful with respect to *name collisions*. The difficulty occurs because overriding, implementation and overloading get unpleasantly mixed together, and overloaded methods cannot differ only be return type.

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }

class C {public int f() { return 1; } }

class C2 implements I1, I2 {
   public void f() {}
   public int f(int i) {return i; } //overloaded
}

class C3 extends C implements I2 {
   public int f(int i) { return i; } //overloaded
}

class C4 extends C implements I3 {}

//class C5 extends C implements I1 {}
//interface I4 extends I1, I3 {}
```

You can easily add new method declaration to an **interface** by using inheritance (using the keyword **extends**). You can also combine several **interfaces** into a new **interface** with inheritance.
Because any fields you put into an **interface** are automatically **static** and **final**, the **interface** is a convenient tool for creating groups of constant values.

```
package cronology;

public interface Months {
   int
     JANUARY = 1, FEBRUARY = 2, MARCH = 3,
     APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
     AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
     NOVEMBER = 11, DECEMBER = 12;
}
```

Notice the Java style of using all uppercase letters for **static finals** that have constant initializers. The fields in an interface are automatically public, so it's not necessary to specify that. In order to use the constants from outside the package, one only has to import **cronology.*** or **cronology.Months**, and referencing the values with expressions like **Months.JANUARY**. Additionally, the fields defined in interfaces are automatically **static** and **final**. These cannot be "blank finals", but they can be initialized with nonconstant expressions. Interfaces may be nested within classes and within other interfaces.

```
class A {
   interface B {
      void f();
   }
}

interface C {
   interface D {
      void f();
   }

   void g();
}

class CImp implements C {
   public void g() {}
}

class CDImp implements C.D {
   public void f() {}
}
```

Implementing a **private** interface is a way to force the definition of the methods in that interface without adding any type information (that is, without alllowing any upcasting).

```
class A {
   private interface D {
      void f();
   }

   public class DImp implements D {
      public void f() {};
   }

   public D getD() { return new DImp(); }
   private D dRef;

   public void receiveD(D d) {
      dRef = d;
      dRef.f();
   }
}

public class PrivateInterfaces {
   public static void main(String[] args) {
      A a = new A();

      //This does not work since D is private
      //A.D ad = a.getD();

      //This does not work since getD returns D a
      //and D is private
      //A.DImp di = a.getD();

      //Cannot access a member of the interface
      //a.getD().f();

      A a2 = new A();
      a.receiveD(a2.getD());
   }
}
```

This is a very intriguing example since **getD()** is a public method which returns a reference to a **private** interface. What can you do with the return value of this method? The only thing that works is if the return value is handled to an object that has permission to use it, in this case, another **A**, via the **receiveD()** method.

9 Inner Classes and Callbacks

Inner Class:

- A class defined within a method.

- A class defined within a scope inside a method.
- An anonymous class implementing an interface.
- An anonymous class extending a class that has a nondefault constructor.
- An anonymous class that performs field initialization.
- An anonymous class that performs construction using instance initialization.

```
public interface Destination {
   String readLabel();
}

public class Parcel {
   public Destination dest(String s) {
      class PDestination implements Destination {
         private String label;
         private PDestination(String whereTo) {
            label = whereTo;
         }
         public String readLabel() { return label; }
      }

      return new PDestination(s);
   }

   public static void main(String[] args) {
      Parcel p = new Parcel();
      Destination d = p.dest(‘‘Tanzania’’);
   }
}
```

10 Observer

The Observer pattern defines a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically. A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don’t want to achieve consistency by making the classes tightly coupled, because that reduces their usability. The Observer pattern describes how to establish these relationships. The key objects in this pattern are *subject* and *observer*. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject’s state.

This kind of intercation is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.
The Observer pattern should be used:

- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Implementation considerations:

- *Mapping subjects to their observers.* The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject.
- *Observing more than one subject.* It might make sense in some situations for an observer to depend on more than one subject. For example, a spreadsheet may depend on more than one data source.
- *Dangling references to deleted subjects.* Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference it.
- *Avoiding observer specific update protocol.* Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to **update**. The amount of information may vary widely. At one extreme, which we call the *push model*, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme is the *pull model*; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

We could implement the Observer pattern from scratch, but Java provides the Observable/Observer classes as built-in support for the Observer pattern.

```
public class EventSource extends Observable implements Runnable {
   public void run {
      try {
         final InputStreamReader isr = new InputStreamReader(System.in);
         final BufferedReader br = new BufferedReader(isr);
         while( true) {
            String response = br.readLine();
            setChanged();
            notifyObservers( response );
```

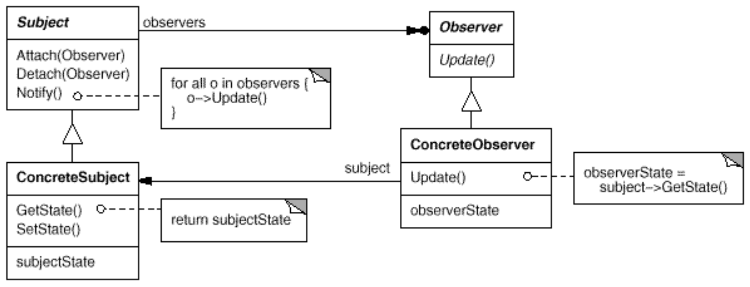

Figure 3: The observer design pattern

```
         }
      }
      catch{IOException e) {
         e.printStackTrace();
      }
   }
}

public class ResponseHandler implements Observer {
   private String resp;
   public void update(Observable obj, Object arg) {
      if (arg instanceof String) {
         resp = (String) arg;
         System.out.println("‘‘\nReceived Response: ’’+resp);
      }
   }
}

public class MyApp {
   public static void main(String[] args) {
      System.out.println(‘‘Enter Text > ’’);
      final EventSource evSrc = new EventSource();
      final ResponseHandler respHandler = new ResponseHandler();
      evSrc.addObserver(respHandler);

      Thread thread = new Thread(evSrc);
      thread.start();
   }
}
```

11 Command Design Pattern

11.1 Callbacks - review

A *callback* is a function that is made known to the system to be called at later time when certain events occurs. In C and C++ we can implement *callbacks* using pointers to functions, but this is not possible in Java. In Java we must use an object that serves the role of a pointer to a function. We will use the term *functor* to denote a class with only one method whose instances serve the role of a pointer to a function. *Functor* objects can be created, passed as parameters and manipulated wherever fuction pointers are needed.
Obviously, a **Comparator** object is a funtor, since it acts like a pointer to the **compare()** method.
Consider a Java **Observable** object and its **Observer** objects. Each **Observer** implements the **Observer** interface and provides an implementation of the **update()** method. As far as the **Observable** is concerned, it essentially has a pointer to an **update()** method to callback when **Observers** should be notified. So, in this case, each **Observer** is acting as a *functor*.

11.2 The command pattern

The main idea behind the command pattern is to encapsulate a request as an object, thereby letting you manipulate the requests in various ways. This pattern should be used when:

- You want to implement a callback function capability.
- You want to specify, queue and execute requests at different times.
- You need to support undo and change log operations.

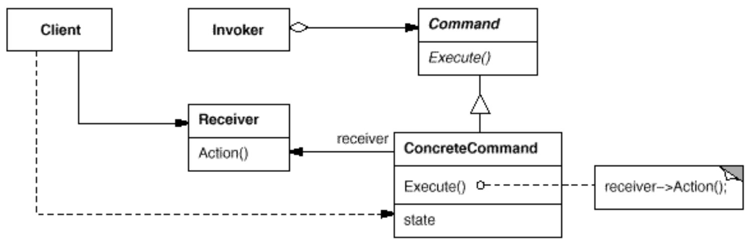

Figure 4: The command design pattern

The participants in the command design pattern are:

- **Command:** Declares an interface for executing an operation.

- **ConcreteCommand:** Defines a binding between a Receiver object and an action. And implements the **execute()** method by invoking the corresponding operation on Receiver.
- **Client:** Creates a concrete command and sets its receiver.
- **Invoker:** Asks the command to carry out the request.
- **Receiver:** Knows how to carry out the operations associated with carrying out a request.

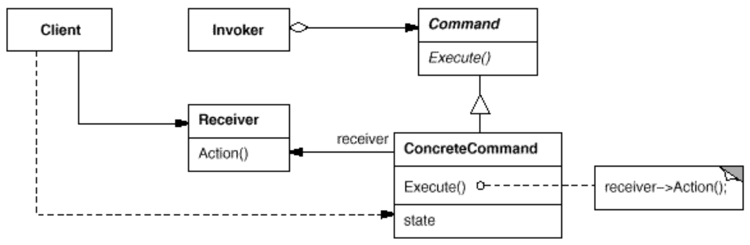

Figure 5: Interactions between the participants in the command design pattern

Naturally, the command pattern allows you to decouple the object that invokes the operation from the one that knows how to perform it. Additionally, commands are first-class objects. They can be manipulated and extended like any other object (for instance, you can use the composite design pattern).

12 Swing

Java 1.1 introduced a new GUI model based on the Observer pattern. GUI components which can generate events are called *event sources*. Objects that want to be notified of GUI events are called *event listeners*. Naturally, a event source corresponds to a concrete subject and a event listener corresponds to a concrete observer. An event listener must implement an interface which provides the method to be called by the event source when the event occurs. Unlike the Observer pattern, there can be different kinds of listener interfaces, each tailored to a different type of GUI event:

- Listeners for Semantic Events:
  - ActionListener
  - AdjustmentListener
  - ItemListener
  - TextListener

- Listener for Low-Level Events:
  - ComponentListener
  - ContainerListener
  - FocusListener
  - KeyListener
  - MouseListener
  - MouseMotionListener
  - WindowListener

Some of these listener interfaces have several methods which must be implemented by an event listener. For example, the WindowListener interface has seven such methods. In many cases, an event listener is really only interested in one specific event, such as the Window Closing event. Java provides "adapter" classes as a convenience in this situation. For example, the WindowAdapter class implements the WindowListener interface, providing "do nothing" implementation of all seven required methods. An event listener class can extend WindowAdapter and override only those methods of interest.

13 Exceptions

14 Strategy Pattern

According to the *Strategy* pattern, behaviors of a class should not be inherited. Instead, they should be encapsulated using interfaces. The *Strategy* pattern uses composition instead of inheritance. In this pattern behaviors are defined as separate interfaces and specific classes that implement these interfaces. This allows better decoupling between a behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes. Behaviors can be changed at run-time as well as design-time. This gives greater flexibility in design and is in harmony with the **Open/Close Principle** (OCP) which states that classes should be open for extension but closed to modification.

14.1 Examples

14.1.1 Listing Files and directories Version 1 - Java Example

Suppose you want to list all the directories and files in a given directory. The **File** object can be listed in two ways. If you call **list()**, you will get the full list that the **File** object contains. However, if you want a restricted list - for example, if you want all of the files with an extension **.java** - then you should use a "directory filter", which is a class that tells how to select the **File** objects for display.

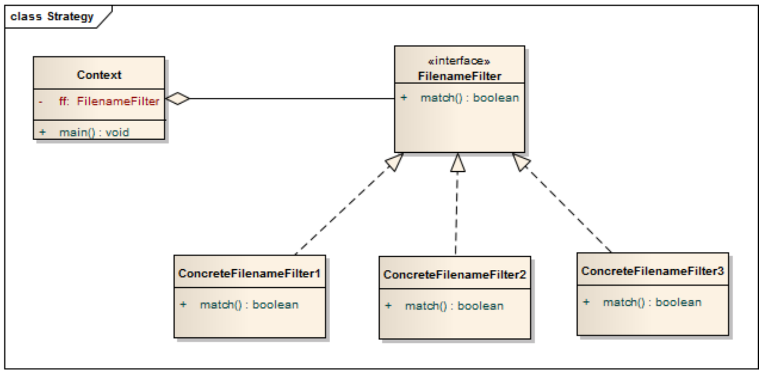

Figure 6: Class diagram that corresponds to the listing files example

```
public class DirList{
   public static void main(String[] args) {
      File path = new File(``.'');
      String[] list;
      if(args == 0)
        list = path.list();
      else
        list = path.list(new DirFilter(args[0]));

      Arrays.sort(list, new AlphabeticComparator());
      for(int i=0; i<list.length; i++)
         System.out.println(list[i]);
   }
}


class DirFilter implements FilenameFilter {
   private Pattern pattern;
   public DirFilter(String regex) {
      pattern = Pattern.compile(regex);
   }

   public boolean accept(File dir, String name) {
      return pattern.matcher(
         new File(name).getName()).matches();
   }
}
```

The **DirFilter** class implements the interface **FilenameFilter**:

```
public interface FilenameFilter {
   boolean accept(Filter dir, String name);
}
```

The whole reason behind the creation of this class is to provide the **accept** method the list method so that the method **list()** can determine which files should be listed. The **accpet** method must accept a **File** object representing the directory that a particular file is found in, and a **String** containing the name of that file. To make sure the element you are working with is only the file name and contains no path information, all you have to do is take the **String** object and create a **File** object out of it, then call **getName()** which strips away all the path information. Then **accept()** uses a regular expression **matcher** object to see if the regular expression **regex** matches the name of the file.
A brief note on regular expressions in Java. A **Pattern** object represents a compiled version a regular expression. The static **compile()** method compiles a regular expression **String** into a **Pattern** object. Then, one can use the **matcher** method and a **String** which we want to compare against the pattern to produce a **Matcher** object. The **Matcher** object can be used in different ways, for instances:

- `boolean matches()`

  - Returns true if the pattern matches the entire input.

- `boolean lookingAt()`

- `boolean find()`

  - Can be used to discover multiple pattern matches.

14.1.2 Listing Files and directories Version 2 - Java Example

The example presented above is ideal for rewriting using an **anonymous inner class**. Generally, a class is bound to a name or identifier upon definition. However, Java also allows classes to be defined without names. Such a class is called an **aonnymous inner class**.

```
public class DirList{

   public static FilenameFilter filter (fineal String regex) {
      return new FilenameFilter() {
         private Pattern pattern = Pattern.compile(regex);
         public boolean accept(File dir, String name) {
            return pattern.matcher(new File(name).getName()).matches();
         }
      };
   }

   public static void main(String[] args) {
      File path = new File(``.'');
      String[] list;
      if(args == 0)
        list = path.list();
      else
        list = path.list(filter(args[0]));

      Arrays.sort(list, new AlphabeticComparator());
      for(int i=0; i<list.length; i++)
         System.out.println(list[i]);
   }
}
```

15 Decorator Pattern

The decorator pattern can be used to make it possible to extend (decorate) the functionality of a class at runtime, independently of other instances of the same class, provided some groundwork is done at design time. This is achieved by designed a decorator class that wraps the original class. The decorator pattern was conceived so that multiple decorators can be stacked on top of each other, each time adding new functionality to the overriden methods. The decorator method is an alternative to subclassing. Subclassing adds behaviour at compile time, decorating can provide new behaviour at runtime for individual objects.
Decorators are often used when simple subclassing results in a large number of classes in order to satisfy every possible combination that is needed - so many classes that it becomes impractical. The main disadvantage of the decorator pattern is that althought it gives the programmer more flexibility when writing a program, it can make the use of the basic functionality more complex.
We provide a very simple example to motivate the use of this pattern.

```
interface Window {
   public void draw();
   public String getDescription();
}

public class SimpleWindow implements Window {
   public void draw(){
      //Draw a Simple Window
   }

   public String getDescription(){
      return ‘‘Simple Window’’;
   }
}

abstract class WindowDecorator implements Window {

   protected Window decoratedWindow;

   public WindowDecorator(Window decoratedWindow){
      this.decoratedWindow = decoratedWindow;
   }

   public void draw(){
     decoratedWindow.draw();
   }

}

public VerticalScrollBarDecorator extends WindowDecorator {
   public VerticalScrollBarDecorator (Window decoratedWindow) {
      super(decoratedWindow);
   }

   public void draw() {
      drawVerticalScrollBar();
      super.draw();
   }

   private void drawVerticalScrollBar() {
      //draw the vertical scroll bar
   }

   public String getDescription() {
      return decoratedWindow.getDescription() + ‘‘, including vertical scroll bar’’;
   }
}

class HorizontalScrollBarDecorator extends WindowDecorator {
   public HorizontalScrollBarDecorator (Window decoratedWindow) {
      super(decoratedWindow);
   }

   public void draw() {
      drawHorizontalScrollBar();
      super.draw();
   }
   protected void drawHorizontalScrollBar() {
      //Draw Horizontal Scrollbar
   }

   public String getDescription() {
      return decoratedWindow.getDescription() + ‘‘, including horizontal scrollbar’’;
   }
}

public class DecoratedWindowTest {
   public static void main(String[] args) {
      Window decoratedWindow = new HorizontalScrollBarDecorator(
                new VerticalScrollBarDecorator(new SimpleWindow()));
      System.out.println(decoratedWindow.getDescription());
   }
}
```

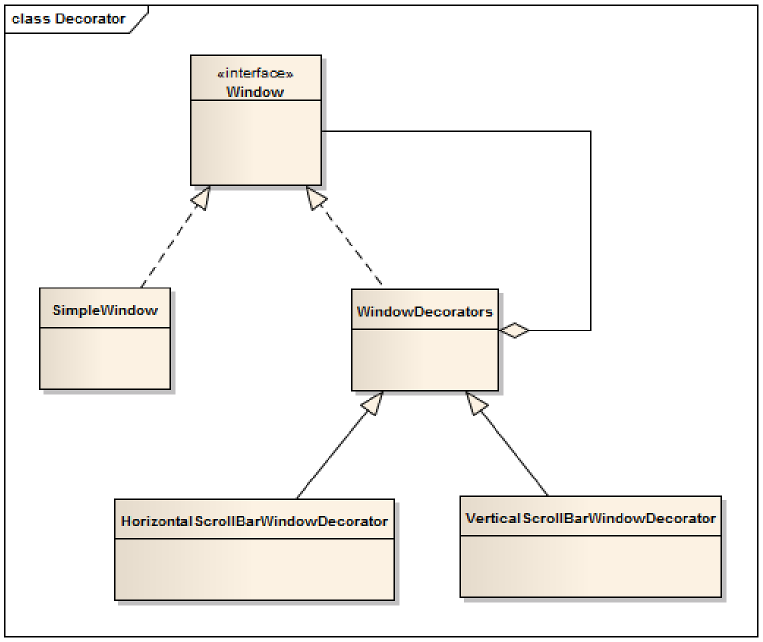
class Decorator
«interface»
Window
SimpleWindow
WindowDecorators
HorizontalScrollBarWindowDecorator
VerticalScrollBarWindowDecorator

Figure 7: Class diagram that corresponds to the window’s example

16 Java Input Output

In section 14 we presented an illustration of the *Strategy* pattern which used Java Input/Output Libraries. In this section we will discuss the design principles that rule Java I/O.
I/O libraries often use the abstraction of a stream, which represents any data source or sink (e.g. an array of bytes, a **String** object, a file, a ”pipe´´, etc) as an object capable of producing or receiving pieces of data. The stream hides the details of what actually happens to the data inside the I/O device.
The Java library classes for I/O are divided by input and ouput and by character-oriented and byte oriented. Table 1 presents the classes which constitute the fundamental core of Java I/O.

	Character-Oriented	Byte-Oriented
Input	**InputStreamReader**	**InputStream**
Output	**OutputStreamWriter**	**OutputStream**

Table 1: Java I/O

These are the core classes. However, they are not commonly used. They exist so that other classes can use them - these other classes provide a more useful interface. The goal of the Java I/O designers was to let the programmer combine different classes in order to get the functionality he or she wants (hence, the use of the *Decorator* pattern). Thus, you'll rarely create your stream object by using a single class, but instead you will layer multiple objects together to provide your desired functionality.