

Composition vs Inheritance

A fundamental design choice that we have to make when considering whether or not to use inheritance is to examine the relationship that exists between two classes. The simple way to do this is the 'Is-a' and 'has-a' rule.

We say an object *X* is-a *Y*, if everywhere you can use an object of type *Y*, you can use instead of object of type *X*. In other words, *X* is a **proper subtype** of *Y*. Usually, this means that *X* implements the same interface as *Y*. So, you know that *X* and *Y* conform to the same set of method type signatures, however, their implementation may be different.

We say an object *X* has-a *Y*, if *Y* is a part-of *X*. So, you typically think of *X* containing an instance of *Y*, not *X* inheriting from *Y*.

For example:

1. Would you say that an Email Message is-a Header or an Email Message has-a Header?
2. Would you say that a Stack is-a Vector or a Stack has-Vector?
3. Would you say that Circle is-a Shape or a Circle has-a Shape?

In each case, you need to consider whether or not the relationship between two objects is simple one of "using the object" or "being the object". If you just need to use an object, then that implies a composition relationship. If an object behaves like another object, then that implies a subtype relationship.

Subtyping, and subtype polymorphism, are one of the most important contributions of object-oriented programming to programming language theory in general.

Java implements a very general form of subtype polymorphism, because every object is-a *Object* since every class inherits implicitly from the class *Object*. So, you can always treat any object in Java, not matter what its class type is, as a subtype of *Object*. So, everywhere you expect to have an *Object*, you can use any Java object. However, this is a very generic way to treat ALL objects in a program, which may in fact be too general, as it forces the programmer to do a lot of **upcasting** and **downcasting**. You cast up from a concrete type to a more general type and you cast down from a more general type of a more concrete type. *Downcasting however can result in run-time type errors.*

Polymorphism

As we have already seen, the *Object* class is the base class for every object. This idea is borrowed from Smalltalk, and is different than C++. The *Object* class permits a form of **subtype polymorphism** in Java that is not found in C++. Since every object "is-a" *Object*, then it is possible to define heterogeneous collections of objects.

For example, the `java.util.Vector` class, is implemented as a resizable array of type *Object*. That means that you can insert any type of java object into a single *Vector*. However, when you do this, you "lose" the type of the object. For example:

```
Vector v = new Vector();
v.addElement(new String("hello, world"));
v.addElement(new Integer(5));
...
Object s = v.firstElement();
Object i = v.lastElement();
```

In this simple example, you know that *s* is really a reference to an object of type *String*, and *i* is a reference to an element of type *Integer* (note: *Integer* is a "wrapper" class for the builtin type `int` and builtin types do not inherit from *Object*. The are also `Long`, `Float`, `Double`, `Boolean`, `Character` wrapper classes defined).

What you really want to write is an explicit downcast:

```
String s = (String) v.firstElement();
Integer i = (Integer) v.lastElement();
```

Polymorphism in Java permits this type of explicit **downcasting** of an *Object* reference to a reference of the object's real type. If you get it wrong, the java run-time will raise an exception called the `ClassCastException`. This is a case where you should use the **instanceof** language feature to check the real type of a subtype of the *Object* class at runtime before performing the type cast if you do not know for sure that you are casting to the correct type.

Base Class Finalization

The *Object* class defines a special "finalize" method that is called by the garbage collector to allow an object to finalize any cleanup that needs to occur before the memory resources for the object are reclaimed. So, the `finalize` method, which can be overridden by an class, provides a type of destructor. The `finalize` method should be implemented for any class that uses system resources and needs to release those resources as part of implicit destruction by the garbage collector. For example:

```
public class DerivedFile extends BaseFile {
    private RandomAccessFile file;

    public DerivedFile(String path) {
        ...// do initialization
    }

    public void close() {
        if (file != null) {
            try { file.close(); file = null; } // allow gc to collect Stream object
            catch (IOException e) { ... }
        }
    }

    public void finalize() throws Throwable {
        close(); // ensure that file was closed before we die
        super.finalize(); // explicitly call BaseFile.finalize() since it is overridden
    }
}
```

The `finalize` method should always call the superclass `finalize` method as the last thing it does, so that the superclass has a chance to finalize itself. This is very much like the virtual destructor chain in C++, except that in Java, you have to explicitly initiate the call to the superclass `finalize` method. The garbage collector will call the first `finalize` method for you, and then you have to program the rest of the finalization calls.

Finding the Class of an Object at Runtime

Because every instance of a class is also an instance of the *Object* class, you can use an *Object* reference to determine at run-time the class of an object. In fact, Java defines a class called "java.lang.Class" that allows you to do some interesting things that are not easily achievable in C++. For example, you can query an object to determine its type:

```
Object x = vector.removeLast();
Class type = x.getClass(); // get the class of the thing that was in the vector
System.out.println(type.getName());
```

You can then ask the *Class* object for certain kinds of information about the class for an object, such as its superclass (there's only one superclass in java since there is only single inheritance) and the names of all the interfaces implemented by the class.

```
Class superclass = type.getSuperclass();
System.out.println(superclass.getName());
```

```
Class[] interfaces = type.getInterfaces();
for (int i = 0; i < interfaces.length; i++)
    System.out.println(interfaces[i].getName());
```

This type of flexibility is useful for obtaining type information at run-time, but it is not as useful as it could be, since you (the programmer) still have to ultimately know what type you are dealing with since you will need to eventually **downcast** an *Object* reference to a reference to the real type in order to use the type. The *Object* class mostly provides a convenient handle that allows you to treat all objects generically, for example, in a container type such as a *Vector*, *Stack*, *Queue*, etc.

The ability to compute type information about objects in a program at run-time is an object-oriented concept called **reflection**. The idea of being able to perform reflective computation in a program originated in Artificial Intelligence, with the language 3-Lisp. The deep idea underlying reflection is that programs should be able to evolve and adapt their behavior overtime as the result of feedback from execution. Java does not provide as powerful a model of reflection as does 3-Lisp, but it is a beginning. However, you can expect to see more of this type of programming.

More on the class `java.lang.Class`

An interesting feature of the `Class` class is a static public method call `Class.forName` that allows you to construct a `Class` object given a `String`, then ask the class object to construct an object that is an instance of the class name given in the string. This is useful for constructing objects from user input. For example:

```
public static void main(String[] args)
{
    Object[] objects = new Object[args.length];
    try {
        for(int i = 0; i < args.length; i++) {
            Class type = Class.forName(args[i]); // construct Class for each class name
            objects[i] = type.newInstance();
        }
    } catch (Exception e) { System.err.print(e); }
}
```

Of course, you have to eventually cast each constructed object down to the type of the `Class`, which you need to know in advance when you write the program. So, Java provides some dynamic type programming features, but the strong typing isn't gone.

The types of exceptions that can arise are:

1. `ClassNotFoundException` thrown by `Class.forName` because the `String` given does not match a known class
2. `NoSuchMethodError` thrown by `Class.newInstance` because the `Class` does not define a constructor that takes no arguments. The `Class.newInstance` method constructs an object using the default constructor.
3. `IllegalAccessException` thrown by `Class.newInstance` because the default constructor is not accessible.
4. `InstantiationException` is thrown by `Class.newInstance` because the class is abstract or is an interface.