Inheritance in Java

Inheritance is a compile-time mechanism in Java that allows you to extend a class (called the **base class** or s**uperclass**) with another class (called the **derived class** or **subclass**). In Java, inheritance is used for two purposes:

1. **class inheritance** - create a new class as an extension of another class, primarily for the purpose of **code reuse.** That is, the **derived class** inherits the public methods and public data of the **base class**. Java only allows a class to have one immediate base class, i.e., single class inheritance.
2. **interface inheritance** - create a new class to implement the methods defined as part of an **interface** for the purpose of **subtyping.** That is a class that implements an interface "conforms to" (or is constrained by the type of) the interface. Java supports multiple interface inheritance.

In Java, these two kinds of inheritance are made distinct by using different language syntax. For class inheritance, Java uses the keyword **extends** and for interface inheritance Java uses the keyword **implements**.

```
public class derived-class-name extends base-class-name {
    // derived class methods extend and possibly override
    // those of the base class
}

public class class-name implements interface-name {
    // class provides an implementation for the methods
    // as specified by the interface
}
```

Example of class inhertiance

```
package MyPackage;

class Base {
    private int x;
    public int f() { ... }
    protected int g() { ... }
}

class Derived extends Base {
    private int y;
    public int f() { /* new implementation for Base.f() */ }
    public void h() { y = g(); ... }
}
```

In Java, the **protected** access qualifier means that the protected item (field or method) is visible to a any derived class of the base class containing the protected item. It also means that the protected item is visible to methods of other classes in the same package. This is different from C++.

Q: What is the base class of class Object? I.e., what would you expect to get if you executed the following code?

```
Object x = new Object();
System.out.println(x.getClass().getSuperclass());
```

Order of Construction under Inheritance

Note that when you construct an object, the default base class constructor is called implicitly, before the body of the derived class constructor is executed. So, objects are constructed top-down under inheritance. Since every object inherits from the Object class, the Object() constructor is always called implicitly. However, you can call a superclass constructor explicitly using the builtin **super** keyword, as long as it is the *first* statement in a constructor.

For example, most Java exception objects inherit from the java.lang.Exception class. If you wrote your own exception class, say SomeException, you might write it as follows:

```
public class SomeException extends Exception {

    public SomeException() {
      super();  // calls Exception(), which ultimately calls Object()
    }

    public SomeException(String s) {
      super(s);  // calls Exception(String), to pass argument to base class
    }

    public SomeException (int error_code) {
      this("error”);  // class constructor above, which calls super(s)
      System.err.println(error_code);
    }
}
```

Abstract Base Classes

An **abstract class** is a class that leaves one or more method implementations unspecified by declaring one or more methods *abstract*. An abstract method has no body (i.e., no implementation). A subclass is required to override the abstract method and provide an implementation. Hence, an abstract class is incomplete and cannot be instantiated, but can be used as a base class.

```
abstract public class abstract-base-class-name  {
    // abstract class has at least one abstract method

    public abstract return-type abstract-method-name ( formal-params );

    ... // other abstract methods, object methods, class methods
}

public class derived-class-name extends abstract-base-class-name {

    public return-type abstract-method-name (formal-params) { stmt-list; }

    ... // other method implementations
}
```

It would be an error to try to instantiate an object of an abstract type:

```
abstract-class-name obj = new abstract-class-name();  // ERROR!
```

That is, operator new is invalid when applied to an abstract class.

Example abstract class usage

```
abstract class Point {
    private int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public void move(int dx, int dy)
    { x += dx; y += dy; plot(); }
    public abstract void plot();  // has no implementation
}

abstract class ColoredPoint extends Point {
    private int color;
    protected public ColoredPoint(int x, int y, int color)
    { super(x, y); this.color = color; }
}

class SimpleColoredPoint extends ColoredPoint {
    public SimpleColoredPoint(int x, int y, int color) { super(x,y,color); }
    public void plot() { ... } // code to plot a SimpleColoredPoint
}
```

Since ColoredPoint does not provide an implementation of the plot method, it must be declared abstract. The SimpleColoredPoint class does implement the inherited plot method. It would be an error to try to instantiate a Point object or a ColoredPoint object. However, you can declare a Point reference and initialize it with an instance of a subclass object that implements the plot method:

```
Point p = new SimpleColoredPoint(a, b, red); p.plot();
```

Interfaces

An abstract class mixes the idea of mutable data in the form of instance variables, non-abstract methods, and abstract methods. An abstract class with only static final instance variables and all abstract methods is called an **interface**. An interface is a *specification,* or contract, for a set of methods that a class that implements the interface must conform to in terms of the type signature of the methods. The class that implements the interface provides an implementation for each method, just as with an abstract method in an abstract class.

So, you can think of an interface as an abstract class with all abstract methods. The interface itself can have either public, package, private or protected access defined. All methods declared in an interface are implicitly abstract and implicitly public. It is not necessary, and in fact considered redundant to declare a method in an interface to be abstract.

You can define data in an interface, but it is less common to do so. If there are data fields defined in an interface, then they are implicitly defined to be:

- public.
- static, and
- final

In other words, any data defined in an interface are treated as public constants.

Note that a class and an interface in the same package cannot share the same name.

Methods declared in an interace cannot be declared final. **Why?**

Interface declaration

Interface names and class names in the same package must be distinct.

```
public interface interface-name {

    // if any data are defined, they must be constants
    public static final type-name var-name = constant-expr;

    // one or more implicitly abstract and public methods
    return-type method-name ( formal-params );
}
```

An interface declaraion is nothing more than a specification to which some class that purports to implement the interface must conform to in its implementation. That is, a class the implements the interface must have defined implementations for each of the interface methods.

The major reason for interfaces being distinct elements in Java is that you often want to define some operation to operate on objects that all conform to the same interface. So, you can define some code in a very general way, with a guarantee that by only using the methods defined in the interface, that all objects that implement the interface will have defined implementations for all the methods.

For example, you might define a Shape interface that defines an area() method, and so you would expect that any class that implements the Shape interface, would define an area method. So, if I have a list of references to objects that implement the Shape interface, I can legitimately invoke the area method, abstractly, on each of these objects and expect to obtain as a result a value that represents the area of some shape.

Separation of Interface from Implementations

Interfaces are specifications for many possible implementations. Interfaces are used to define a contract for how you interact with an object, independent of the underlying implementation. The objective of an object-oriented programmer is to separate the specification of the interface from the hidden details of the implementation.

Consider the specification of a common LIFO stack.

```
public interface StackInterface {
    boolean empty();
    void push(Object x);
    Object pop() throws EmptyStackException;
    Object peek() throws EmptyStackException;
}
```

Note that the methods in this interface are defined to operate on objects of type Object. Since a stack is a “container type”, it is very common to use the base class for all objects, namely Object, as the type of the arguments and results to a container type. Since all objects ultimately inherit from class Object, we can always generically refer to any object in the system using an Object reference. However, when we pop from a stack, we have to explicitly type case from the very general type Object to a concrete type, so that we can manipulate the object concretely.

Q: How many different ways are there to implement a stack? If we are using just using a Stack object (as opposed to implementing it ourselves) should we care?

Stack implementation of the StackInterface

```
public class Stack implements StackInterface {

    private Vector v = new Vector();  // use java.util.Vector class

    public boolean empty() { return v.size() == 0; }

    public void push(Object item) { v.addElement(item); }

    public Object pop() {
        Object obj = peek();
        v.removeElementAt(v.size() - 1);
        return obj;
    }

    public Object peek() throws EmptyStackException {
        if (v.size() == 0)
           throw new EmptyStackException();
        return v.elementAt(v.size() - 1);
    }
}
```

Should a stack use or inherit from a vector?

The java.util.Stack class is defined as a subclass of the java.util.Vector class, rather than using a Vector object as in the previous example. This sort of inheritance is not subtype inheritance, because the interface of a Stack object can be violated because a Vector has a “wider” interface than a Stack, i.e., a vector allows insertion into the front and the rear, so it is possible to violate the stack contract by treating a stack object as a vector, and violating the LIFO specification of a stack.

```
public class Stack extends Vector {
   public Object push(Object item) {addElement(item); return item;}
   public Object pop() {
        Object  obj;
        int len = size();
        obj = peek();
        removeElementAt(len - 1);
        return obj;
   }
   public Object peek() {
        int len = size();
        if (len == 0) throw new EmptyStackException();
        return elementAt(len - 1);
   }
   public boolean empty() { return size() == 0;}
}

Vector v = new Stack();  // legal - base class reference to subclass object
v.insertElementAt(x, 2);  // insert object x into Vector slot 2!!
```

When to use an Interface vs when to use an abstract class

Having reviewed their basic properties, there are two primary differences between interfaces and abstract classes:

- an abstract class can have a mix of abstract and non-abstract methods, so some default implementations can be defined in the abstract base class. An abstract class can also have static methods, static data, private and protected methods, etc. In other words, a class is a class, so it can contain features inherent to a class. The downside to an abstract base class, is that since their is only single inheritance in Java, you can only inherit from one class.
- an interface has a very restricted use, namely, to declare a set of public abstract method singatures that a subclass is required to implement. An interfaces defines a set of type constraints, in the form of type signatures, that impose a requirement on a subclass to implement the methods of the interface. Since you can inherit multiple interfaces, they are often a very useful mechanism to allow a class to have different behaviors in different situations of usage by implementing multiple interfaces.

It is usually a good idea to implement an interface when you need to define methods that are to be explicitly overridden by some subclass. If you then want some of the methods implemented with default implementations that will be inherited by a subclass, then create an implementation class for the interface, and have other class inherit (extend) that class, or just use an abstract base class instead of an interface.

Example of default interface implementations

```
interface X {
    void f();
    int g();
}

class XImpl implements X {
    void g() { return -1; } // default implementation for g()
}

class Y extends XImpl implements X {
    void f() { ... } // provide implementation for f()
}
```

Note that when you invoke an abtract method using a reference of the type of an abstract class or an interface, the method call is dynamically dispatched.

```
X x = new Y();
x.f();
```

The call x.f() causes a run-time determination of the actual type of object that 'x' refers to, then a method lookup to determine which implementation of f() to invoke. In this case, Y.f(x) is called, but the type of x is first converted to Y so that the 'this' reference is initialized to a reference of type Y inside of f(), since the implicit type signature of Y.f() is really Y.f(final Y this);