

## Inheritance in Java

Inheritance is a compile-time mechanism in Java that allows you to extend a class (called the **base class** or **superclass**) 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 methods of the base class.
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 the interface.

In Java, unlike C++, these two types 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**.

For example:

```
class Base {
    private int x;
    public int f() { ... }
    protected int g() { ... } // NOTE: protected access means visible to subclass only
}

class Derived extends Base {
    private int y;
    public void h() { y = g(); ... }
}
```

In Java, unlike C++, only single class inheritance is supported. I.e., for a given class, there is only one superclass. However, the superclass of a class can have a superclass, and so on, until the superclass is the class Object.

Q: What is the superclass 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());
```

## Final Classes

A class that is declared **final** cannot be subclassed, so the methods of a final class can never be overridden. The purpose of declaring a class final is to signify to a potential user of the class that the class is considered complete, from the perspective of the class designer, and that further specialization via class inheritance is unnecessary. Many of the classes found in the `java.lang` package are declared final, e.g., the `String` class.

Another way to prohibit a class from being used as a base class is to declare the default constructor to be private:

```
public class Single {
    static private Single single = new Single();
    private Single() {}
    public static Single makeSingle() { return single; }
}
```

The only way to obtain a reference to the `Single` object is to invoke its static class method `makeSingle`, which returns a handle to the one `Singleton` object that was statically created.

```
Singleton s = Singleton.makeSingleton();
```

This is an example of an *object-oriented design pattern*, called the `Singleton` pattern. It is commonly used to ensure that only one instance of an object is ever created during a program. Common examples include the `java.lang.Runtime` class, as there is only one runtime in a program. Another example is a windowing system that uses a “root” windows from which all other windows are sub-windows, such as the X-windows system.

A way to ensure that a class is always used as a base class for some other class is to declare its constructor “protected”. That way, an object of the class can only be constructed by constructing an object of one of its subclasses.

```
public class Base {
    protected Base() {} // can only be called by a subclass constructor
    ...
}
```

## Order of Construction under Inheritance

Note that when you construct an object, the default superclass constructor is called implicitly, before the body of the 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, and are written as a 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");  
        System.err.println(error_code);  
    }  
}
```

**Q:** What do you think happens in the case where you first call a local constructor using `this`? Does `super()` get called implicitly first?

Note that it is recommended in Java that you define your own exception classes as extension of the Java exception classes to cover the exceptions that might occur in your program.

## Abstract Base Classes

An **abstract class** is a class that leaves some implementation detail 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 for it. Hence, an abstract class is incomplete, but can be used as a base class. However, you cannot instantiate an instance of an abstract class, since it is not a complete class.

```
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(); // abstract method has no implementation
}

abstract class ColoredPoint extends Point {
    int color;
    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 SimplePoint
}
```

Since `ColoredPoint` does not provide an implementation of the `plot` method, it must be declared abstract too. The `SimpleColoredPoint` class does implement the `plot` method, and so only an object of type `SimpleColoredPoint` can be instantiated. 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

Abstract class mix the idea of mutable data in the form of instance variables, non-abstract methods, and abstract methods (i.e., methods with no implementation). An abstract class with all final data (if any) 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, 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 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
- 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.

In Java, static methods cannot be abstract, so methods in an interface cannot be static. **Why?**

Also, methods declared in an interace cannot be declared final. **Why?**

## 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;
}
```

**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?

```
class Stack implements StackInterface {
    private Vector v = new Vector(); // use a dynamic vector as the implementation

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

## Is a Stack a Subtype of 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();
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 there 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 signatures that a subclass is required to implement. An interface 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 have any methods that are to be 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.

```
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 abstract 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(); // dynamically determines which f() to invoke, and calls Y.f(this=(Y)x);
```