Java Basics

---

1. INHERITANCE

Code sharing is a good thing. First, you do not want to re-write the same stuff more than once. Also, code is likely to be more reliable as it only has to be debugged once. A mechanism for sharing code is *inheritance*.

```
class A {
   //variables
   //methods
}
class B extends A {
  //class B inherits ALL variables and methods of class A
}
```

We say that class A is the parent of class B, or the *super-class* of class B; class B is the *sub-class*. A class can inherit from a single other class (in Java).

The sub-class inherits all the variables and methods in the super-class. It is as though the variables and methods of the super-class would be textually present in the sub-class.

Example:

```
class OneDPoint

class TwoDPoint extends OneDPoint

class ThreeDPoint extends TwoDPoint
```

Example:

```
class Person {
   String name;  //the name

   void Person(String s) {
       name = new String(s);
   }

   String getName() {
      return name;
   }
};
class Student extends Person {

  int class; //the class of graduation
```

```
  void Student(String n, int year) {
    super(n);
    class = year;
  }
}
```

Now we can say:

```
class Student s = new Student("Jon Myers);
System.out.println("name is " + s.getName());
```

In the constructor of the sub-class, the first thing to do is invoke the constructor of the super-class.

In our toy example above this makes no sense because class Person is empty and it does not even have a constructor. But in general it takes some work to "construct" a parent class, so in a sub-class we first need to invoke the appropriate constructor and pass it the arguments that we want.

Object-oriented programming allows to define class hierarchies. That is, it allows for classes to inherit from other classes, which inherit from other classes, and so on.

1.1 Overriding methods

We said earlier that a sub-cass inherits all the methods of the parent classes. Sometimes it is the case that a sub-class is not happy with the definition of a method inherited from the parent; it might want to do something different. In this case it may chose to redefine the method. This is called "overriding a method".

1.2 The Java class hierarchy

In Java the root of all classes is class `Object`. That is, every Java class inherits from it.

Class `Object` has no variables, but has a bunch of methods. Thus any class will inherit these methods, and their default definition in class `Object`. Here are a few that we'll be using later:

```
//Indicates whether some other object is "equal to" this one.
boolean equals(Object obj);

//Returns a string representation of the object.
String toString()
```

2. JAVA INTERFACES

Java has the concept of an `interface`. An interfaces is like a class, just that it does not give the implementation, just the *signatures* of the methods. All methods in an interface must be public.

```
public interface xxx {

// list the signatures of the methods
}
```

For example, here is an interface that you'll use all the time:

```
public interface MouseListener {
   public void  mouseClicked(MouseEvent e);
   public void mouseEntered(MouseEvent e);
   public void mouseExited(MouseEvent e);
   public void mousePressed(MouseEvent e);
   public void mouseReleased(MouseEvent e);
}
```

Interfaces are used to describe a class *without* an implementation.

A class may chose to *implement* an interface, or several. Every class that implements an interface must define all methods in the interface.

When compiling a class that implements an interface, the compiler checks to see that *all* methods specified in the interface are implemented; the parameter types and return type must match; if not, it gives an error. So you can think of an interface is a way of specifying a set of methods that an object must support. An interface is a contract between a class and the outside world.

It is possible that a class implements several interfaces—then it needs to implement *all* methods in *all* interfaces.

You probably won't be defining interfaces, but you'll write all the time classes that implement interfaces.

Interfaces are useful because sometimes you don't know exactly what class you'll be using, you just know it has to have some functionality. You can think of interfaces as *generic* classes. For example, here is the set of methods that we'd expect from a `List`:

```
public interface List {

  //return the size of the list
  public int size();

  //return true if list is empty
  public boolean isEmpty();

  //return true if the list contains the given element
  public boolean contains(Object value);

  //add one element to the list
```

```
  public void add(Object value);

  //remove the element from the list
  public void remove(Object value);
}
```

Suppose we have two classes that implement `List`

```
public class ArrayList implements List {

  //... implements all methods in the interface, and possibly more
}

public class Vector implements List {

  //... implements all methods in the interface, and possibly more
}
```

Now we can write a piece of code that works with a generic List:

```
void collectData(List l) {
    ....
}
```

and we can call this passing an argument either an `ArrayList` or a `Vector`.