Inheritance

Inheritance is the act of deriving a new class from an existing one. Inheritance allows us to extend the functionality of the object. The new class automatically contains some or all methods and variables of the original class. The new derived class is called a child class, or subclass. The original class is called parent class, or superclass, or base class. Since the derived class is always more specific, the relation between these classes is called is-a relation. Class extension can be used for a number of purposes. The most common use is a specialization - where the extended class defines new behaviors and thus becomes a specialized version of its superclass. Consider an example in which we extend the functionalities of the ArrayList to add in a sorting ability. The new class (let us call it SortedArrayList) is a specialized version of the ArrayList, since obviously not each ArrayList is a sorted list. The language syntax is pretty much straightforward:

```java
public class SortedArrayList extends ArrayList
{
    public void sort()
    {
        Object[] tmp = toArray();
        Arrays.sort(tmp);
        clear();
        for(int i = 0; i < tmp.length; i++)
            add(i, tmp[i]);
    }
}
```

All public members declared in ArrayList will be available in SortedArrayList. That is it; we call toArray, clear and add directly, without using the class (or object) name. We say that SortedArrayList inherits public members of ArrayList, but the latter does not inherit anything from the former – the sort method won’t be available in ArrayList. Inheritance is a directional process.

The above implementation is not particular elegant, by two reasons. First, you cannot sort objects of different types, and second, a user has an ability to add a new element, making a disorder in the list. If the first problem can be easily handled using an exceptions mechanism, the second presents a real threat. You do not have a control over this, and cannot removing the add method from the base class. Inheritance does not allow you to do this. If you are not happy with a particular method implementation in the base class, the only choice you have is to provide a different implementation in the subclass. This process is called overriding. When you override the base class method you must keep its signature and returning type.

Inheritance allows the following two changes in derived class:

1. add new members;
2. override existing (in base class) methods.
There are two forms of inheritance:

- *inheritance of type*: whereby the subclass acquires the type of the base class;
- *inheritance of implementation*: whereby the subclass acquires the implementation of the base class;

These two forms always occur together.

**protected**

The derived class inherits all variables and methods, which are public. It does not inherit those variables and methods, which are private. However, we do not want to declare all variables and methods to be public, since they will be visible for everybody. Therefore, Java offers another type called **protected**. Protected variables are not accessible for other classes, but only for derived class. In the above example, we could declare the field `pages` as protected, and thus in Dictionary we would simply call `pages` instead of the method `getPages()`.

**super:**

Constructors are not inherited in a derived class. Moreover, a child's constructor is responsible for calling the parent's constructor. This is Dictionary's constructor:

```java
public Dictionary(int d, int p, String s)
{
    super(p, s);
    definition = d;
}
```

By keyword **super** Java provides a mechanism to call parent's constructor. The above `super(p, s)` is actually a call for Book's constructor `Book(p, s)`. How would call Book's constructor from Dictionary? There is no other way as to use **super**. The super method can be only used as a first line (literally) of a constructor.

The super reference can also be used to invoke other methods in parent's class. For example, how would you call Book's `toString` method from inside Dictionary class? `super.toString()`. We say that Dictionary's `toString` overrides Book's `toString()`.

If the child class constructor does not call **super**, the parent's constructor with no arguments will be implicitly called.

**super(arg list)** as the first line in a constructor explicitly calls the parent's constructor with the given signature.
If parent class implements a constructor with arguments and has no a constructor with no arguments, then the child constructors must explicitly call a parents constructor.

This code fragment demonstrates implicit call to super:

```java
public class Demo
{
    public static void main(String[] args)
    {
        System.out.println("Object Parent");
        Parent sleepy = new Parent();
        System.out.println("Object Child");
        Child care = new Child();
        System.out.println("Object Child Two");
        care = new Child("");
    }
}

class Parent
{
    public Parent()
    {
        System.out.println("    In Parent");
    }
}

class Child extends Parent
{
    public Child()
    {
        System.out.println("    In Child");
    }
    public Child(String args)
    {
        System.out.println("    In Child with argument");
    }
}

Here is the output:

Object Parent
    In Parent
Object Child
    In Parent  (implicit call to super)
    In Child
Object Child Two
    In Parent  (implicit call to super)
    In Child with argument

Single inheritance:
Java does not support multiple inheritance, but only single. This means that a derived class can have only one parent. But a parent can have a few children!

**Overriding:**

A child class can override the parent's definition. However, a method can be defined with the `final` modifier. A child class cannot override a `final` method. Also, a subclass cannot override methods that are declared `static` in the superclass.

**Hierarchies:**

Multiple classes can be derived from a single parent. There is no limit to the number of children a class can have (but a child can have only one parent). Two children of the same parent are called *siblings*. Siblings are NOT related to each other by inheritance. Inheritance is transitive. All methods and fields are passed from a parent to the children and then from children to their children and further. There is no single best hierarchy, the decision is made when you design your classes. This is a picture of the class hierarchy, going bottom-up from subclass to superclass:

```
  Object
    ^
   |  
  Book
    ^
   |  
Dictionary
```

**Class Object** *(see API)*

Every class is an extended class, whether or not it is declared as such. In Java all classes are derived from the `Object` class. So, there are two equivalent definitions:

```java
class Bag
{
  . .
}

class Bag extends Object
{
  . .
}
```

Object class has several methods, here are a few of them, all of them can be overridden

**String toString()**

Object's `toString` method returns a String representation of the object. You use `toString` along with `System.out.println` to display a text representation of an object.

**Object clone()**
You use the clone method to create an object from an existing object. To create a clone, you write: `yourObject.clone();` Object's implementation of this method checks if the object on which clone was invoked implements the `Cloneable` interface, and throws an exception if it does not. The simplest way to make your class cloneable is to add `implements Cloneable` to your class's declaration. Object's implementation of the clone method creates an object of the same type as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables. If a class contains a member variable, which is a reference, then the original class and its clone are aliases (they refer to the same object). If you want to have a deep copy, you have to provide your implementation of `clone()`.

```java
boolean equals(Object obj)
```

The `equals` method compares two objects for equality and returns true if they are equal. The `equals` method provided in the `Object` class uses the identity function to determine if objects are equal. In most cases two objects are equal if they are aliases to each other. However, two distinct objects might be considered equal if they contain the same information. So you have to distinguish the method `equals` and the sign `==`. Consider the following example:

```java
Integer a = new Integer(1);
Integer b = new Integer(1);
Integer c = b;
```

If we compare `a == b`, this is false, since `a` and `b` refer to different objects, however `c == b` is true, since they are aliases. Now if we compare `a.equals(b)`, this is true, since they have identical values stored.

```java
void finalize()
```

The `Object` class provides a method, finalize, that cleans up an object before it is garbage collected. The `finalize` method is called automatically by the system and most classes you write do not need to override it. So you can generally ignore this method. If you override `finalize`, your implementation should call `super.finalize()` as the last thing it does. Method `finalize` should always be defined as protected so subclasses have access to the method.

Garbage Collector:

Managing memory explicitly is tedious and error prone work. Some languages require that you keep track of all the objects - you create them and then you destroy them. In Java you don't have to worry about destroying objects. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called garbage collection. An object is eligible for garbage collection when there are no more references to that object. This happens in two cases: 1) the variable goes out of scope; 2) the variable is set to null. The garbage collector does its job automatically, although, you may want to run the garbage collection explicitly by calling `System.gc()`. Before an
object gets garbage-collected, the garbage collector gives the object an opportunity to clean up after itself through a call to `finalize`. Java does not guarantee the order in which objects will be garbage collected; therefore, it cannot guarantee which object's finalizer will execute first.

**Polymorphism**

The term *polymorphism* can be defined as having many forms. A polymorphic reference is one that refer to different types of the objects. One meaning of it is to use a switch statement to take an appropriate action on each object based on that object’s type. Polymorphism forces you to think in types.

Consider a collection of greeting cards: the super class is *Card* which has a subclass *Holiday* which has a subclass *AprilFool*. The single variable *card* is used with different types. It is called a *polymorphic reference*:

```java
public class CardTester {
    public static void main (String[] args) {
        Card card = new Holiday("Charlie");
        card = new AprilFool("Rosanne");

        // Holiday hol = new Card("fill the blank"); you cannot do it!
    }
}

class Card {
    String toWhom;

    public Card(String name) {
        toWhom = name;
        System.out.println("Dear " + toWhom + "!");
    }
}

class Holiday extends Card {
    public Holiday(String name) {
        super(name);
        System.out.println("Season's Greetings!");
    }
}

class AprilFool extends Holiday {
    public AprilFool(String name) {
        super(name);
        System.out.println("Just Kidding!");
    }
}
```
Observe that the declaration `Holiday hol = new Card("fill the blank");` is incorrect - a variable can hold a reference to an object who's class is a descendant of the class of the variable.

**Classification:**

Luca Cardelli and Peter Wegner, authors of "On Understanding Types, Data Abstraction, and Polymorphism", 1985 (see their paper at http://research.microsoft.com/Users/luca/Papers/OnUnderstanding.pdf) divided polymorphism into two major categories:

- **Ad Hoc Polymorphism** (it primary means using one name to denote values of different types)
  - type conversion (casting)
  - overriding (including "shadowing")
  - overloading
- **Pure (or Subtype) Polymorphism**
  It was introduced in Simular in 1967.
  ML in 1976 was the first language with polymorphic typing.
  - polymorphic attributes (variables)
  - polymorphic methods (or virtual methods)
  - parametric classes (not implemented in Java)

Review the code example `Polymorphism.java`, which has the following hierarchy of classes:

Point <- Circle <- Cylinder

An object of the type Cylinder can be declared either of the type Cylinder or Circle or Point; an object of a type Circle can be declared either Circle or Point. Using the effect of polymorphism we declare an array of different shapes and then compute their total surface area:

```java
public static double totalArea (Point[ ] figures) {
    double total=0;
    for(int = 0; i < figures.length; i++)
        total += figures[i].area();
    return total;
}
```