Inheritance

Inheritance defines a relationship among classes. Key words often associated with inheritance are extend and implements. Implements is used by classes that inherit from interfaces. Interfaces can never be extended by a class.

Java allows only single inheritance for class extension but allows multiple extensions for interfaces.

Example:

```java
public class NewClass extends BaseClass implements Interface1, Interface2,... InterfaceN
{
    ....
}
```

An interface can extend multiple interfaces:

Example:

```java
public interface NewInterface extends Interface1, Interface2, ..., InterfaceN
{
    ....
}
```

Extend

When a class, called C2 extends C1, then C2 “IS-A” subclass of C1, and C1 is a superclass of C2. All public and proteted members of the superclass are accessible in the extended classes. Let’s look at an example:

```java
public class Shape{
}
```

```java
public class Rectangle extends Shape{
}
```

```java
public class Circle extends Shape{
}
```

```java
public class Square extends Rectangle{
}
```

Shape is the superclass of Rectangle and Circle

Rectangle and Circle are subclasses of Shape

Square is a subclass of both Rectangle and Shape

```java
public static void main(String args[]){
    Shape shape = new Shape();
    Rectangle rect = new Rectangle();
}
```
Square square = new ();
System.out.println(rect instanceof Shape);
System.out.println(square instanceof Rectangle);
System.out.println(square instanceof Shape);
}
}
This would produce the following:
true
true
true

Constructors of Extended Classes
To construct an extended class you need to initialize the fields inherited from the superclass and the fields declared in the extended class. We use the keyword super to call a super class constructor and method. Let’s look at an example.

```java
public class ColoredPoint extends Point {
    public Color color; //the Color class is part of java.awt.Color class
    public ColoredPoint(final double x, final double y, final Color color) {
        super(x,y);//this calls the construct for Point the Superclass
        // that matches the Point(double, double) constructor
        this.color = color;// provides the value for the color field of this class.
    }
}
```
The super keyword must be the first statement of the subclass’s constructor. This is the only way to invoke a superclass constructor. If no constructor is defined in the extended class, the no-arg constructor is provided by default. The default no-arg constructor simply invokes the no-arg constructor of the superclass. For example, the following no-arg constructor will be provided implicitly for the Extended class when no constructor is provided explicitly.

```java
public class ExtendedClass extends SuperClass {
    public ExtendedClass() {
        super();
    }
    //methods and fields
}
```
For the above example, the SuperClass must provide a no-arg constructor otherwise a compile error will result. To call a method of a superclass you do the following: super.method();

Subtypes and Polymorphism
So what is a subtype? Our book defines subtype as: Type T1 is a subtype of T2 if every legitimate value of T1 is also a legitimate value of T2. In this case, T2 is a supertype of T1. Well, that is
clear as mud, so I looked this up from the Java Tutorials. It is possible to assign an object of one
type to an object of another type as long as they are compatible. Let’s look at an example – Keep
in mind that an Object is a supertype of Integer.

Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger; //this is acceptable

Let’s talk a little about polymorphism. In “C”, the rule of assignments is that the left-hand side and
the right-hand side of an assignment must be of compatible types. In OOP we have polymorphic
assignments, which means the type of expression at the right-hand side of an assignment must
be a subtype of the type of the variable at the left-hand side of the assignment. Or as [http://www.tutorialspoint.com/java/java_polymorphism.htm](http://www.tutorialspoint.com/java/java_polymorphism.htm) tells us, polymorphism is the
ability of an object to take on many forms. The most common of polymorphism in OOP occurs
when a parent class reference is used to refer to a child class object. We can use the “IS-A” test to
determine if polymorphism.
Let’s look at another example.

```java
//this example comes from the tutorialspoint.com site in footnote 1.
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}

/*The Deer class is polymorphic since it has multiple inheritance
  (extends and implements)
  * A Deer IS-A Animal
  * A Deer IS-A Vegetarian
  * A Deer IS-A Deer
  * A Deer IS-A Object
  * All of the above make the following legal
 */

Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;

// As you would expect all the reference variable “d, a, v, o” refer
to the same Deer object in the heap.

This next example shows how casting works.

class Student {....}
class Undergraduate extends Student {....}
class Graduate extends Student {....}

//two instances

Student student1, student2;
student1 = new Undergraduate(); //polymorphic assignment
student2 = new Graduate(); // polymorphic assignment

Graduate student3;
student 3 = student2; //this will cause a compile error

/*the right had side is a Student and the left hand side is a Graduate.
  Although student2 holds a reference to an instance of Graduate it
  is not a subtype of the left hand (student3). Casting is
  necessary because type checking is carried out at compile time and
  is based on the declared types of variables. */

[https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html](https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html)
Overriding Methods

First let's discuss the difference between overriding a method and overloading a method. From our book, “Overriding is concerned with methods of different classes that have an inheritance relationship. In overriding methods share the same name, signature, and return type. In contrast, overloading methods are part of the same class, but have different signatures.

The book gives us an example the difference between overriding and overloading methods. However, the following webpage has the following example [http://www.tutorialspoint.com/java/java_polymorphism.htm](http://www.tutorialspoint.com/java/java_polymorphism.htm)

/* File name : Employee.java */
public class Employee {
    private String name;
    private String address;
    private int number;
    public Employee(String name, String address, int number) {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }
    public void mailCheck() {
        System.out.println("Mailing a check to " + this.name + " " + this.address);
    }
    public String toString() {
        return name + " " + address + " " + number;
    }
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String newAddress) {
        address = newAddress;
    }
    public int getNumber() {
        return number;
    }
}

//Now suppose we extend Employee class as follows:

/* File name : Salary.java */
public class Salary extends Employee {
    private double salary; //Annual salary
    public Salary(String name, String address, int number, double
}
salary)
{
    super(name, address, number);
    setSalary(salary);
}
public void mailCheck()
{
    System.out.println("Within mailCheck of Salary class ");
    System.out.println("Mailing check to " + getName() + " with salary " + salary);
}
public double getSalary()
{
    return salary;
}
public void setSalary(double newSalary)
{
    if(newSalary >= 0.0)
    {
        salary = newSalary;
    }
}
public double computePay()
{
    System.out.println("Computing salary pay for " + getName());
    return salary/52;
}

/* File name : VirtualDemo.java */
public class VirtualDemo
{
    public static void main(String [] args)
    {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3,
        3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();
        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}

The output is as follows:
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0

Here is an explanation of the above output:
Here, we instantiate two Salary objects. one using a Salary reference \( s \), and the other using an Employee reference \( e \).

While invoking \( s\.mailCheck() \) the compiler sees \( mailCheck() \) in the Salary class at compile time, and the JVM invokes \( mailCheck() \) in the Salary class at run time.

Invoking \( mailCheck() \) on \( e \) is quite different because \( e \) is an Employee reference. When the compiler sees \( e\.mailCheck() \), the compiler sees the \( mailCheck() \) method in the Employee class.

Here, at compile time, the compiler used \( mailCheck() \) in Employee to validate this statement. At run time, however, the JVM invokes \( mailCheck() \) in the Salary class.

This behavior is referred to as virtual method invocation, and the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

Just a few more thoughts before I rap this section up.

Final: a method declared final cannot be overridden in a subclass. Final methods are useful in preventing accidental overriding of subclass methods. Also, final methods also allow the java compiler and JVM to optimize byte-code.

Invoking Overridden Methods: The keyword “super” should be used to invoke an overridden method. The super keyword represents the superclass.