

## **Lecture 1 – Part II**

# **Java Programming Language**

### **Additional Features and Constructs**

**Topics in Quantitative Finance: Numerical Solutions of  
Partial Differential Equations**

**Instructor: Iraj Kani**

# Subclasses and Inheritance

Central to the object-oriented programming paradigm is the ability to add to the functionality of a class by *subclassing* or *extension*.

To extend a class the key word *extends* is used, as in the following example:

```
public class StandardStockOption {
    public double stockPrice;
    public double strikePrice;
    ... // other fields

    public StandardStockOption() {...} // default constructor

    ... // other constructors including all the parameters
    ... // getters and setters etc.

    public double getPrice() {
        BlackScholes bs = new BlackScholes(stockPrice, strikePrice, ...);
        return bs.getPrice();
    }
}

public class ForwardStartStockOption extends StandardStockOption {
    public double forwardTime;

    public ForwardStartStockOption(double forwardTime) {
        super(); // invokes the super class constructor
        this.forwardTime = forwardTime;
    }
    ... //other constructors, e.g. including all the parameters of StandardStockOption

    public double getPrice() { // overriding the getPrice method of the superclass
        if (forwardTime >= timeToExpiration)
            return 0;
        ... // remaining implementation
    }
}
```

We refer to ForwardStartStockOption as a *subclass* of StandardStockOption class, or to StandardStockOption class as the *superclass* of ForwardStartStockOption class.

ForwardStartStockOption class *inherits* the fields and methods of the StandardStockOption class, as the following code segment shows:

```
ForwardStartStockOption fso = new ForwardStartStockOption(1);
fso.stockPrice = 100;
fso.strikePrice = 95;
fso.timeToExpiration = 2;
...
double fsoPrice = fso.getPrice();
```

Every subclass is a perfectly valid *instance* of its superclass, therefore any ForwardStartStockOption object can be assigned *without a cast* to a StandardStockOption variable:

```
ForwardStartStockOption fso = new ForwardStartStockOption(1);
StandardStockOption so = fso; // assigned to an StandardStockOption variable
```

However, the reverse assignment will require *casting*:

```
ForwardStartStockOption fso = new ForwardStartStockOption(1);
StandardStockOption so = fso;
ForwardStartStockOption fso1 = (ForwardStartStockOption)so; // casting needed here
```

The ability to *override* the methods of the superclass is another essential feature of the object-oriented paradigm.

Every Java class has a superclass. If a class is not a direct extension of another class through the *extends* keyword, then its superclass is *java.lang.Object* class.

The *java.lang.Object* class is the only class that does not have a superclass. It is the *root* of the Object Hierarchy in Java.

Java ensures that the constructor method of a class is invoked when an object of a class or any subclass of the class is initialized. If necessary, Java will insert a call to `super()` as the first statement in the constructor methods. This continues all the way up the class hierarchy, including the Object class.

All Java classes inherit the methods of the Object class, some of which are shown below:

```
public String toString();
public boolean equals(Object obj);
public int hashCode();
```

It is sometimes convenient to override one or more methods inherited from the Object class, as shown in the following modification to the StandardStockOption and the ForwardStartOption classes:

```
public class StandardStockOption {
    ...
    public String toString() {
        StringBuffer sb = new StringBuffer();
        sb.append("stockPrice: " + stockPrice);
        sb.append("strikePrice: " + strikePrice);
        ...
        return sb.toString();
    }
}
```

```

public class ForwardStartStockOption extends StandardStockOption {
    ...
    public String toString() { // overrides toString method of superclass
        StringBuffer sb = new StringBuffer();
        sb.append(super.toString()); // invokes the toString() method of the superclass
        sb.append("forwardTime: " + forwardTime);
        return sb.toString();
    }
}

```

The special syntax *super.toString()* is used to specifically invoke the superclass's *toString* method.

## Abstract Classes and Methods

Suppose we wanted to implement a number of options on the stock, such as standard options, binary options, barrier options etc., each with its own special implementation of *getPrice()* method. To work with this array of claims it would make sense to define a common superclass, *StockOption*:

```

public abstract class StockOption {
    public double stockPrice;
    public double dividendYield;
    public double volatility;
    public double discountRate;
    public double strikePrice;
    public double timeToExpiration;
    ...
    public abstract double getPrice(); // an abstract method of abstract StockOption class
}

```

In Java you can define a method without implementing it by using the *abstract* keyword. Any class containing an abstract method must also be declared an *abstract class*. An abstract class can not be instantiated. Any extension of an abstract class that does not implement all the abstract methods it inherits *must* also be declared abstract.

Methods that are *static*, *private* and *final* methods can not be declared abstract since they can not be overridden in a subclass.

The following is a possible mock-up of the standard option, binary option and barrier option on a stock:

```

public class StandardStockOption extends StockOption {
    public static final int CALL = 0;
    public static final int PUT = 1;
    ...
    public double getPrice() {
        ...
    }
}

```

```

public class BinaryStockOption extends StockOption { // one-touch digital option
    public double barrier;
    public double payoff;
    ...
    public double getPrice() {
        ...
    }
}

public class BarrierStockOption extends StockOption { // standard single barrier option
    public static final int UP_OUT_CALL = 0;
    public static final int UP_IN_CALL = 1;
    ...
    public double barrier;
    public double rebate;
    ...
    public double getPrice() {
        ...
    }
}

```

## Interfaces

Suppose for all one-touch-type options we wanted to set/get barrier level and obtain expected first passage time to the barrier. We can define an abstract `OneTouchOption` class and extend it, but since Java does not allow multiple inheritance we can not extend the `StockOption` class as well. Defining an *interface*, `OneTouch`, is Java's solution to this:

```

public interface OneTouch {
    public void setBarrier(double barrier);
    public double getBarrier();
    public double getExpectedFirstPassageTime();
}

public class BinaryStockOption extends StockOption implements OneTouch {
    ...
    public void setBarrier(double barrier) {
        this.barrier = barrier;
    }
    public double getBarrier() {
        return barrier;
    }
    public double getExpectedFirstPassageTime() {
        ...
    }
}

public class BarrierStockOption extends StockOption implements OneTouch {
    ... // similar implementation for barrier options
}

```

An interface is similar to an abstract class, and is defined using the keyword *interface*. It

contains no implementation for its methods, which are implicitly *abstract*, with the keyword `abstract` omitted.

All interface methods must be *instance methods* and are implicitly *public*. In a sense an interface provides a pure public API.

An interface can contain fields but they must be declared as *static* and *final* (i.e. should be a *Constant*). Like an abstract class an interface can not be instantiated.

A class declares an interface in its *implements* clause. If a class implementing an interface does not provide an explicit implementation for all the interface methods, it must be declared *abstract*.

Java classes can implement multiple interfaces. For example, we wanted to access payoff along a given path for all path-dependent options. We can define:

```
public interface PathDependent {
    public double getPayoff(double[] pathTimes, double[] pathPrices);
}
```

Now we can make binary and barrier options implement both interfaces:

```
public class BinaryOption extends StockOption implements OneTouch, PathDependent {
    ... // implementations of OneTouch interface
    public double getPayoff(double[] pathTimes, double[] pathPrices) {
        ...
    }
}
```

Another possibility is to allow `OneTouch` interface to *extend* `PathDependent` interface, and let these options implement all the methods of `OneTouch` interface, as shown below:

```
public interface OneTouch extends PathDependent {
    ...
}
```

Here is an example usage of these constructs involving a portfolio consisting of different types of options and making use of Java's *instanceof* operator:

```
Option[] opts = new StockOption[10];
opts[0] = new StandardStockOption(...);
opts[1] = new BarrierStockOption(...);
...
for (int i = 0; i < 10; i++) {
    System.out.println("price[" + i + "]: " + opts[i].getPrice());
    if (opts[i] instanceof OneTouch) {
        OneTouch ot = (OneTouch)opts[i];
        System.out.println("first passage time[" + i + "]: " +
            ot.getExpectedFirstPassageTime());
    }
}
```

# Encapsulation, Inheritance and Polymorphism

*Encapsulation or Data Hiding* refers to hiding of the data within the class and making the data only accessed through trusted methods. It helps the programmer hide the inner workings of the class from users, and to ensure proper working of the class against unintentional disruption by the user.

For Java class members this implemented by using the access control modifiers: *public*, *protected*, *private*, and package access.

**public:** member is accessible anywhere the class is accessible.

**private:** member is not accessible anywhere, except within the class.

**protected:** member is accessible to all classes within the package and all subclasses regardless of their package.

**package access:** member is accessible only to all classes in the same package.

```
public class Demo {  
    public int field1; // accessible by everyone  
    protected double field2; // accessible to classes in the same package and all subclasses  
    private long field3; // only accessible in Demo class  
    String field4; // accessible to classes in the same package  
    ...  
}
```

Java classes can be extended through the process of *Inheritance*. A subclass inherits all non-*private instance* fields and methods of its superclass that are accessible to it. In particular, for subclasses in a different package only *public* and *protected* members are inherited. Note that private fields and methods, class fields and methods, and constructors are never inherited

	member visibility			
Accessibility:	public	protected	package	private
Defining Class	Yes	Yes	Yes	Yes
Class / same package	Yes	Yes	Yes	No
Subclass / different package	Yes	Yes	No	No
Non-subclass / different package	Yes	No	No	No

*Polymorphism* refers to the availability of multiple methods with the same name but different signature within the same Java class:

```
public class Curve {
    ...
    public double interpolate(double time) {
        ...
    }
    public double[] interpolate(double[] times) {
        ...
    }
}
```

## Java Exceptions

A Java *exception* is a signal that an error or an exceptional condition has occurred in the Java program. To *throw* an exception is to indicate occurrence of an exceptional conditional within the program. To *catch* an exception is to detect it and do whatever handling necessary to recover from it.

Every exception in Java is an object. The type of this object is *java.lang.Throwable* or some subclass of *Throwable* that more specifically handles a given type of exception that has occurred.

You can actively throw an exception from a method using the *throw* statement, as in the following example:

```
public double getPrice() {
    if ( stockPrice < 0 ) {
        throw new IllegalArgumentException("stock price must be positive!");
    }
    ...
}
```

There are two basic subclasses of *Throwable*: *java.lang.Error* and *java.lang.Exception*. The subclasses of *java.lang.Error* generally indicate unrecoverable problems during execution. The subclasses of *java.lang.Exception* are generally recoverable – these can be generally caught and handled.

Java also distinguishes between *checked* and *unchecked* exceptions. Any exception object that is an *Error* is always unchecked. Any exception object of type *Exception* is checked, except for subclasses of *java.lang.RuntimeException*, the occurrence of which may not always be predictable.

Examples of runtime exceptions in Java are *java.lang.NullPointerException*, and *java.lang.IllegalArgumentException*.

Any method that throws a *checked* exception, or invokes a method that throws such a checked exception, must explicitly indicate so by use of the *throws* clause in its signature:

```
public static void read(String filename) throws java.io.IOException { // must throw
    FileReader reader = new FileReader(filename);
    ...
}
```

The `Throwable` class has a `String` field containing the error message, which can be read by invoking the `getMessage` method once the exception is caught using the *try/catch/finally* statement:

```
try {
    read(filename);
}
catch(IOException e) {
    System.out.println("error: " + e.getMessage());
}
finally {
    ... // do any necessary clean-up
}
```

# Homework Assignment 1:

1. If you have a laptop and do not already have a Java IDE installed, begin by installing a JBuilder Foundation from [www.borland.com](http://www.borland.com):

Downloads -> JBuilder -> Select Foundation (Windows, 02/12/05, 51.2-96.5 Mb)

2. Create a new project in JBuilder:

Name: course, Directory: z:\

3. Create the Gaussian class in this project, as discussed in the lecture:

Name: Gaussian, Package: edu.columbia.ieor.course.util

Write the implementation for pdf() and cdf() methods of the Gaussian class based on the lecture and using the handouts, and

**Exercise 1:** Calculate pdf( $x$ ) for values of  $x$  in the range  $[-5,5]$  in intervals of size 0.2.

**Exercise 2:** Calculate cdf( $x$ ) for the same range of  $x$  values as above.

4. Create and fully implement StandardStockOption class as discussed in lecture (in edu.columbia.ieor.course.analytic package).

**Exercise 3:** Assume  $S = 100$ ,  $K = 100$ ,  $T = 3$ ,  $r = 0.05$ ,  $\delta = 0.03$ ,  $\sigma = 0.2$ . Compute the price, delta, gamma, theta, vega, and rho (use *numerical* calculation for computing all sensitivities) for both standard European *call* and *put* options with the following ranges of parameters (in each case keeping other parameters unchanged as given above):

- Strike price  $K$  in the range  $[80, 120]$  in steps of 2
- Maturity  $T$  in the range  $[0.2, 5]$  in steps of 0.2
- Interest rate  $r$  in the range  $[0, 0.20]$  in steps of 0.01
- Volatility  $\sigma$  in the range  $[0.02, 0.5]$  in steps of 0.02

**Extra Credit:** Create and implement a BarrierStockOption class in the same package and provide implementation for pricing UP\_OUT\_CALL and DOWN\_IN\_CALL options, using the pricing formulas provided in the handouts, and using *numerical* calculations for all sensitivities (delta, gamma, theta, vega, and rho).

Assume  $S = 100$ ,  $r = 4\%$ ,  $\delta = 2\%$ ,  $\sigma = 15\%$ . Compute the price and sensitivities for:

Type	Strike	Barrier	Payoff	Expiration
Up-Out-Call	100	110	5	2
Down-In-Call	95	90	0	3

