Lecture 1 – Part I

Java Programming Fundamentals

Topics in Quantitative Finance: Numerical Solutions of Partial Differential Equations

Instructor: Iraj Kani

Introduction to Java

We start by making a few introductory remarks about the Java programming language, which we will be using for programming purposes throughout this course:

- Java is a state-of-the-art *object-oriented* programming language, created by Sun Microsystems.

- *Syntactically* Java is similar to C programming language, and although considerably different, it accommodates many of the object oriented concepts that exist in the C++ and other object oriented programming languages.

- Java is a *pure* object-oriented language based on a pure object hierarchy, and is viewed to be considerable simpler and more robust than most other object oriented programming languages, like C++.

- Java is *platform-independent,* with Java programs potentially capable of running on any platform without modification.

- This is possible because Java is an *interpreted* programming language. Java programs are *portable* to any hardware platform to which Java interpreter has been ported. Java is now successfully ported to all major hardware platforms and operating systems.

However, to better understand Java we must distinguish between *Java Programming Language*, *Java Virtual Machine*, and *Java Platform*. Below we will briefly describe the latter two concepts.

Java Virtual Machine

Java is an interpreted programming language and depends on Java Virtual Machine for its basic functioning:

- Java Virtual Machine (JVM) or Java *Interprete*r is the engine of Java programming environment.

- It is a *software system / CPU architecture* in which Java program are interpreted and executed.

- *Java Byte Code*, resulting from compilation of a Java program, is the machine language of JVM.

- Java byte code can be ported and run without modification to any other platform for which a Java Virtual Machine exists.

- Unlike most interpreters, Java Virtual machine is a remarkably *high performance* software system.

- *Just-in-time* (JIT) compilers were a significant VM technology that contributed to Java's high performance, e.g. Hotspot technology from Sun Microsystems.

- Today there is a JVM on every major desktop platform, various hand-held devices and other hardware devices.

Java Platform

Java Platform is a collection of constructs that provide the basic functionality and programming blueprints for the Java environment:

- All programs written in Java programming language rely, explicitly or implicitly, upon Java Platform constructs.

- It consists of a set of pre-defined *classes* that are organized into groups called *packages*.

- Java Platform contains packages for numerical calculations, input/output (I/O), networking, graphics, database access, security etc.

- Although not an Operating System, Java Platform provides all necessary *Application Programming Interfaces* (APIs) for accessing functionality of native operating system.

- Java offers comparable API to those available when you program in the native operating system, while remaining independent of (transparent to) the native machine and its operating systems.

- Java has built-in features like *garbage collection* and *internationalization* that make writing Java programs easier, shorter, more universal and with fewer bugs**.**

A Basic Java Program

Here is an example of a simple Java Program:

```
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello world!");
    }
}
```

- The program begins with declaration of a Java class, named *HelloWorld*.

- *Class* is the most fundamental programming construct in Java. All Java programs are declaration of class. A Java class represents a template upon which the behavior and functionality of a Java program is built.

- Here we have declared the HelloWorld as a *public* class, which the use of the *public modifier*.

- The HelloWorld class consists of one class *member*, the main() method, with one argument named *args*, and of type String[] (i.e. String array).

- A *method* is a parameterized collection of statements or procedures that perform an action. Methods may have parameters and return values.

- Here the *main* method does not return a value (i.e. is declared to have *void* as return type), but it performs an action by printing "Hello World!" into the standard (system) output stream.

- The main method has two keyword modifiers, *public* and *static*, indicating that the method can be accessed without restriction and that its action is specific to the class and not to any particular instance/object of the class.

- The System class used within the main method is a standard part of Java Platform. It is a class in *java.lang* package that is automatically available to any Java program.

- Specifically, here the System.out.println() is invoked. This method takes a string argument and prints it to a system's standard output stream.

Another Example

Here is another example of a Java program:

```
/**
* This program computes factorial of a number.
*/

public class Factorial {
        public static void main(String[] args) {
                int x = 5;  // test the program with input value of 5.
                double fact = factorial(x);
                System.out.println("result: " + fact);
        }
        public static double factorial (int x) {
          if (x < 0)
                return 0;
          double fact = 1.0;
          while (x > 1) {
              fact = fact * x;
              x = x – 1;
          }
          return fact;
        }
  }
```

- Java programming language supports three types of comments, all of which are ignored by the Java compiler:

  (i) comments of the form /** … */ : multi-line comments used by Javadoc
  (ii) comments of the form /* … */ : multi-line comments for programmers
  (iii) comments of the form // … : single line comments

- Factorial class contains two methods, *main* and *factorial*. Both methods are declared as *static*. The *factorial* method has *double* as return value.

- The first line of main declares an *int* (integer) variable x, and assigns it the value 5. The second line declares a *double* variable fact and assigns it the return value of the *factorial* method. Both *int* and *double* are examples of Java *primitive types*.

- When we run a Java program the main method is invoked by default. In this case it prints "result: 120" in the system's standard output stream.

Running a Java Program

To compile and run a Java program you require a Java software development kit (SDK), such as the freely available JDK (Java Development Kit) from Sun Microsystems.

Basically these are the steps:

1. Type the *Factorial* program in your favorite text editor, and then save it in a file specifically named *Factorial.java* .

2. *Compile* the program using java compiler. For Sun's JDK this compiler is known as *javac*, which is a command-line tool invoked at the command line as:

```
C: \> javac Factorial.java
```

Any compilation errors will appear at this time. If successful, the compilation process creates an output file (known as a *class file*) with the name *Factorial.class*.

3. Java class files require Java Interpreter for their running. In Sun's JDK the interpreter is a command-line program named *java*:

```
C: \> java Factorial
```

Java Class Members

Java classes provide the basic functional blue print for their objects. A Java *object* is an *instance* of its Java class. Though the functionality of an object is *derived* from its class, an object possesses its own private run time *state*.

To provide this functionality to its objects each class is made up of class *members*. Some important class members are:

Constructors – Special constructs used for instantiating objects from a class.

Fields – Store variable information within objects of a class. The values of the fields determine the *runtime state* of an object. A field may be declared as *static* (*class field*), in which case it's value at any point is *shared* by all objects of the class. Otherwise, it is an *instance field*, associated with (and a having value depending on) each specific object.

Methods - These are the subroutines associated with the class. Methods are invoked using a reference to the class (class methods / static methods) or an object of the class (instance methods).

Inner Classes – Java classes can be nested, contained within other classes as members. Inner classes may have their own fields, methods, or their own nested inner classes.

Here is a mock-up of class used for computing the Black-Scholes price of a standard stock option:

```
public class BlackScholes {
      public static final int CALL = 0;
      public static final int PUT = 1;

      private int type;
      private double stockPrice;
      private double strikePrice;
      private double timeToExpiration;
      private double volatility, discountRate, dividendYield;     // variables of same type

     public BlackScholes(double stockPrice, double strikePrice, double timeToExpiration,
            double volatility, double discountRate, double dividendYield, int type) {
          this.stockPrice = stockPrice;
          this.strikePrice = strikePrice;
          . . .
     }

     public BlackScholes() {  // default constructor setting default values of parameters
          this(100, 100, 1.0, 0.10, 0.0, 0.0, CALL);
     }

     public void setStockPrice(double s) {
          stockPrice = s;
     }

     public double getStockPrice() {
          return stockPrice;
     }
     . . .

     public double getPrice() {
           . . .
     }
}
```

A typical invocation of this class may look something like this:

```
. . .
BlackScholes bs = new BlackScholes(100, 100, 1.0, 0.20, 0.10, 0.05, BlackScholes.CALL);
double callPrice = bs.getPrice();
. . .
```

Here is another invocation using the *default constructor* and *setter/getter* methods for the fields:

```
BlackScholes bs = new BlackScholes();
bs.setStockPrice(90);
. . .
bs.setType(BlackScholes.PUT);
double putPrice = bs.getPrice();
```

Note the declaration of the class fields CALL and PUT as *public, static* and *final*. Also note the *private* modifier for the parameter fields, making the following statement *illegal*:

```
bs.stockPrice = 110;    // invalid Java statement: this field is private.
```

Packages and Java Namespace

A *package* is a named collection of classes. It serves to group classes and define a *namespace* for the classes within the package.

Java packages are defined using the keyword *package*, which is expected to be the first token in the definition of the class, as illustrated by the following example:

```
package edu.columbia.ieor.course.util;

public class Gaussian {

      public static double pdf(double x) { // gaussian probability density function
         return Math.exp(-x*x/2) / Math.sqrt(2*Math.PI);
      }

      public static double cdf(double x) { // gaussian cumulative density functions
         . . .
      }
}
```

This class has the simple name Gaussian and the fully qualified name *edu.columbia.ieor.course.util.Gaussian*.

The Math class is contained in *java.lang* package. This special package is automatically imported in all Java classes. Packages such as *java.io*, *java.util*, and *java.net* are other examples of pre-defined packages within the Java Platform.

Java Syntax

Identifiers and Reserved Words

An *identifier* is a symbolic name associated with some construct in a Java program. For example, class, method, field, parameters and variable names are all identifiers. Examples of valid identifiers are: a, b0, var_1, my_field_name;

An identifier can not include punctuation symbols other than underscore and currency symbols. It can contain numbers, but not as their first character. Also they can not contain most punctuation and escape characters (like '.' and '\n').

A *reserved* word is a keyword or literal that is a part of Java programming language, e.g. public, abstract, double, new, null, if, throws, 2. You can not use a reserved word as an identifier in a Java program.

Primitive Data Types

There are eight *primitive* data types in Java, consisting of *boolean* type, *character* type, four *integer* types and two *floating-point* types. They are summarized below:

boolean	true or false
char	16-bit (Unicode) character set
byte	8-bit integer (signed)
short	16-bit integer (signed)
int	32-bit integer (signed)
long	64-bit integer (signed)
float	32-bit floating point
double	64-bit floating point

Here is some examples of defining primitive types in Java:

```
int n = 10;
long l = 1023L;
double d = -12.345;
byte b = 15;
double d = 1.02e+10;
short s = -125;
float f = -3.22e-7f;
char c = “a”;
```

Strings and Characters

Strings represent textual array of characters. In Java strings are not primitive types, but are represented by *java.lang.String* class, with many useful methods for manipulating strings:

```
String s = “hello”;
String s1 = “hello” + “ world”;
String s2 = new String(“hello world”);
String s3 = s2.toUpperCase();
int n = s3.indexOf(“W”);
```

Note that Java strings are *immutable* objects. However, a closely related class *java.lang.StringBuffer* provides mutability for strings:

```
StringBuffer sb = new StringBuffer(s):
sb.append(“ world”);
```

For each Java numeric type (byte, short, int, long, float, double) there is a corresponding class in *java.lang* package: *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, each a subclass of *java.lang.Number* class.

These define useful methods and constants, including MIN_VALUE and MAX_VALUE, which specify the range of that numeric type. There are also methods for converting strings to numbers:

```
Double db = new Double(1.5e4);
double d = db.doubleValue();
String s = "1.5e4";
double d1 = Double.parseDouble(s);
double d2 = Double.valueOf(s).doubleValue();
```

Here is an example of the use of these methods inside the *main* routine of the *Factorial* class, where the arguments can be passed at the command line to the string array parameter *args*:

```
public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    double fact = factorial(n):
    System.out.println("result: " + fact);
}

C: \> java Factorial 5
```

Arithmetic operations in Java can produce special floating-point values: *zero*, *Infinity*, *-Infinity* and *NaN*. These values can also be directly accessed as public fields of *java.lang.Double* class:

```
double inf = Double.POSITIVE_INFINITY;          // result of 1/0
double negInf = Double.NEGATIVE_INFINITY;   // result of -1/0
double nan = Double.NaN;                                 // result of 0/0
```

The *java.lang.Math* class provides a number of useful methods and field for performing algebraic, exponential, logarithmic, trigonometric operations on numbers:

```
double d = Math.PI;
double d1 = Math.sqrt(d);
double d1 = Math.exp(-2.5);
double d2 = Math.log(d1);
double d3 = Math.cos(Math.toRadians(30));   // converting degrees to radians and taking cosine
```

Another useful class for generating pseudo-random numbers is *java.util.Random* class:

```
java.util.Random rand = new java.util.Random(1);    // creates a random object with seed value 1

for (int i = 0; i < 10; i++) {
  System.out.println("rand: " + i + "]: " + rand.nextGuassian();
}
```

Reference Types

Java *classes* and *arrays* constitute Java's *reference types*. They represent composite types with varying size in memory depending on what they contain. Here are some examples of defining arrays in Java:

```
double[] d = new double[10];   // array of ten doubles all initialized to value 0
double[] d1 = {1.0, -2.3e-4, Math.E}; // initialized array of doubles
double[][] d2 = {{1,2},{3,4},{5,6}};    // regular 2-dim array of doubles: matrix
double[][] d3 = {{1}, {2,3}, {4,5,6}};  //irregular 2-dim array of doubles

String[] s = {"hello", "world"};
Double[] db = {new Double(1.1), new Double(2.2)};

StandardStockOption[] so = new StandardStockOption[2];
so[0] = new StandardStockOption(100, . . .);
so[1] = new StandardStockOption(90, . . .);

for ( int i = 0; i < so.length; i++) {
    System.out.println("price[" + i + "]: " + so[i].getPrice());
}
```

Java handles objects and arrays *by reference*. When you assign an object or array to a variable, you are assigning to the variable a reference to that object or array.The same happens when you pass an object or array as parameter to a method.

Operators

There are several types of operators in Java, including *arithmetic operators*, *increment operators*, *comparison operators*, *shift operators*, *assignment operators*, and *conditional operators*. Here are a few examples using these operators:

```
int k = 8 % 5;                  // modulo operator, returning 3
int k1 = k++;                   // unary post-increment operator, k1 equals 3, k equals 4
int k2 = --k1;                  // unary pre-decrement operator, k2 equals k1 equals 2
boolean b = k1 == k2;           // comparison operator, b equals true
boolean b1 = (k != k1);         // comparison operator, b1 equals true
boolean b2 = b && b1;           // boolean and operator, b2 equals true;
k *= 2;                         // assignment operator, k equals 8
k %= 4;                         // assignment operator, k equals 0
int k3 = (k >= 2) ? k : k1;     // conditional operator, k3 equals k1
boolean b3 = (new BarrierOption() instanceof Option);  // instanceof operator, b3 equals true
```

Statements

Java statements are basic building block of Java programs. There are several types of statements in Java, including *block statements*, *expressions*, *variable declarations*, *flow control statements*. Here are some common examples of flow control statements:

if-else statement:

```
double h = 10;

if (h < 4) {
  System.out.println(“h < 4”);
}
else if (h < 8) {
  System.out.println(“4 <= h < 8”);
}
else {
  System.out.println(“h >= 8”);
}
```

while statement:

```
import java.util.Random;
int count = 0;
Random r = new Random();

while(r != null && count < 100) {
  double g = r.nextGaussian();
  count++;
  System.out.println(“rand: “ + g);
}
```

for statement:

```
import java.util.Random;

int count = 0;
double g = 0;

for (Random r = new Random(); r != null && count < 100; g = r.nextGaussian(), count++ ) {
     System.out.prinln(“rand: “ + g);
}
```

try/catch/finally statement:

```
double[] d = {1, 2, 3};

try {
   for ( int m = 0; m < 10; m++) {
     System.out.println(d[m]);
}
catch( IndexOutOfBoundsException e)   {
   System.out.println(“index out of bounds: “ + m);
}
catch(Throwable f) {
  System.out.println(“error: “ + f.getMessage());
}
finally {;}
```