Java Tutorial for C++ Programmers

Java is a strongly typed object-oriented programming language that was first developed in the early 1990's at Sun Microsystems, and inspired by the C++ programming language. The improvements of Java over C++ include a more streamlined syntax, easy-to-use general purpose libraries that address all modern computing needs, garbage collection, platform independence (e.g. a Java program developed on Linux does not need modification to work on Windows), and the ability to be executed over the internet. Java's main disadvantage is in terms of performance, since executing a Java program requires a Java Virtual Machine which adds an extra layer of inefficiency to the computing process.

Writing a Java Program. Since Java is an OOP language, a Java program consists of a set of objects, with each object having private and public data and methods (i.e. "functions"). Java objects are instantiated from Java classes that can either be listed as `public` or `private`. Private classes must be nested within some public class. For example,

```
public class A
{
.
.
.
    private class B
    {
    .
    .
    .
    }
}
```

In C++ a class is divided into both .h and .cpp files. In Java each source file (excluding packages) is a .java file which consists of exactly one public class definition, with private classes possibly nested inside. Thus, Java source code will usually consist of a set of .java files. When a source file is compiled into Java byte code (a code that is executed by the JVM), the byte code is stored in a .class file.

In C++ there must be a .cpp file with a `main()` function that gets executed first. In Java there are two possibilities for how a program starts execution.

1. There is one public class that has a public static method called `main()`. Like C++, it is this method that begins the execution of the program.

2. There is exactly one public class that is of type Applet or JApplet. The corresponding .class file can then be executed using a web browser or some other applet viewing program.

Programming style. Java programming style is very similar to C++, however the identifiers for classes, attributes, references, methods, and constants follow more strict guidelines. Class identifiers normally begin with a capital letter. If the class name consists of more than one word, then the words are concatenated together with capital letters used to begin each new word. (e.g. "BinarySearchTree"). Attributes, methods, and references follow a similar rule, with the exception that the first letter of the identifier is lowercase (e.g. BinarySearchTree myTree). Finally, constants are identified using words that are written in all capitals and separated with underscores (e.g. MAX_HEIGHT).

References. In C++, a distinction can be made between a reference (pointer) to an object and an actual object. For example,

```
A* a1; //a pointer variable that can be assigned to reference a type A object
A a2;  //represents an object of type A
```

In Java *every* declaration of the form `A a` indicates that `a` is a *reference* of type `A`. There are two ways to attach a reference to an actual object.

1. **new operator.** For example, `a = new A()`.
2. **assignment.** `a = EXPR`, where EXPR is some expression of type A. For example, EXPR might be a function that returns a type-A object, or it might be another type-A reference that is already attached to some type-A object.

Note that references are automatically assigned `null` until they attach to some object.

Primitive data types. There is one exception to the notion that `A a` always implies that `a` is a reference. This is not the case when `A` represents a *primitive data type*. Similar to C++, Java primitive data types include `byte, short, int, long, float, double, char, and boolean`. Thus, given `int a = 1`, `a` represents an integer that holds the value of 1.

Attribute and Method Modifiers.

1. **public, private, protected.** These serve the same purpose as in C++, with protected allowing for access to derived classes.

2. **final.** This modifies an attribute that is to serve as a constant. The attribute must be assigned a predetermined value that does not change during the running of the program. For example,

   ```
   final double CAPACITY=12.0;
   ```

   It can also be used to modify a method that is not to be overridden by any future descendants.

3. **abstract.** This can be applied to a method (and hence the encompassing class) to indicate that the method will be overridden and implemented in some derived class For example,

   ```
   public abstract double getArea(double radius);
   ```

4. **static.** This is applied to an attribute or a method to indicate that one does not need to create an actual object to use the method or attribute. For example,

   ```
   class A
   {
       public static getArea(double radius);
   }
   ```

   allows one to use `getArea()` without creating an `A` object. For example,

   ```
   double a = A.getArea(radius);
   ```

   would be appropriate.

Java libraries and classes. You can obtain documentation regarding any class of the Java platform by visiting

`http://java.sun.com/j2se/1.3/docs/api/`

Fundamental Java Classes.

- **String.** Analogous to the C++ string class.
- **Math.** Contains several standard (statically defined) mathematical functions
- **Random.** Used for generating pseudorandom numbers.
- **Keyboard.** This not part of the Java library, but was developed for the book "Java Software Solutions", by John Lewis and William Loftus. Contains very convenient functions for reading various input types.
- **NumberFormat.** Provides generic formatting capabilities for numbers.
- **DecimalFormat.** Provides generic formatting capabilities for numbers.
- **StringTokenizer.** Allows for the conversion of a string into its respective tokens based on a set of user-specified delimiters.

Garbage Collection. Garbage collection refers to the process of automatically recovering dynamically allocated memory without user programming or intervention. Java has this feature, and Java uses neither the `delete` function nor the class destructor from C++.

Wrapper Classes. Unlike C++, in Java one cannot define an array of elements of a primitive data type, such as the `int`. Arrays must be arrays of objects, and primitive data types do not have the same status as an object. Thus, Java has the wrapper classes which are in one-to-one correspondence with primitive data types. The wrapper classes are Byte, Short, Integer, Long, Float, Double, Character, and Boolean. So instead of creating an array of elements of type int, one creates an array of elements of type Integer.

Nested Classes. A *nested class* is a private class that is nested within a public class. For example the public class `List` might have the private class `Link` nested within it. Note that the nested class has access to all private methods and attibutes of the outer class, while the outer class only has access to those methods and attributes of the inner class that have been declared as public.

Abstract Interfaces. An *abstract interface* is a collection of constants and abstract methods. An abstract method is a method that does not have an implementation. Abstract methods are valuable to Java, since Java does not have multiple inheritance.

As an example, consider the `Comparable` abstract interface, which consists of the one abstract method `int compareTo(Object)`. If one wants to compare employees, then one can define the `Employee` class as

```
public class Employee implements Comparable
{
.
.
.
}
```

while, implementing the `compareTo()` function within the class definition. The usefulness of this can be seen if one wanted to place employee objects within a `PriorityQueue` object. Such an object would have an `insert` method for inserting objects into the queue. The method can now take as input objects of type `Comparable`, which in turn allows for `Employee` objects to be inserted (or any other `Comparable` object for that matter). So in this sense, implementing an abstract interface can be viewed as a form of inheritance in Java.

Arrays. Let `C` be any class type. Then the lines of code

```
//three steps to creating an array of objects
C[] c;    //reference to a C array

c = new C[20];   //c now references 20 C references

for(int i=0; i < 20; i++)
    c[i] = new C();      //attach the ith C reference
```

are typical for how to create an array of `C` objects.

2-Dimensional Arrays. Two-dimensional arrays are handled similarly.

```
C[][] c;    //reference to a C 2-D array

c = new C[20][30];   //c now references 20x30 C references
```

Inheritance. As mentioned earlier, Java only allows for single inheritance, and uses abstract interfaces to allow for a class to have more than one super type associated with it. The notation for deriving class `B` from class `A` is

```
public class B extends A
{
.
.
.
}
```

`B` then has all the attributes and methods of `A` that were either declared `public` or `protected`.

Derived Class Constructors. Naturally, a derived class's constructor will need to initialize attributes that were inherited from its parent. This can be accomplished by invoking the `super()` method, and placing in it a list of parameters that are needed for the parent's constructor.

Method Overriding. Any method that was inherited from an ancestor may be overridden (i.e. re-defined) in the derived class. If a method is not to be overridden by any future descendants, then the `final` modifier can be used on the method.

The Object Class. The `Object` class is the default parent of every other Java class. Thus three methods that are automatically inherited by any class include:

```
boolean equals(Object obj)
//returns true if this object is an alias of the specified object

String toString()
//returns a String representation of this object

Object clone()
//creates and returns a (shallow) copy of this object
```

Reading From a File.

```
FileReader fr = new FileReader(filename);
BufferedReader inFile = new BufferedReader(fr);

line = inFile.readline();
while (line != null)
{
    tokenizer = new StringTokenizer(line);
    name = tokenizer.nextToken();
}
```

Writing to a File.

```
FileWriter fw = new FileWriter(filename);
BufferedWriter bw = new BufferedWriter(filename)
PrintWriter outFile = new PrintWriter(bw);
//outFile now has capabilities such as print(), and println()
.
.
.
outFile.close()
```