

# **The Non-Java Programmers Guide to Java**

**Copyright © 2008 by: Robert C. Ilardi**  
**Release: 9.18.2008**

**Contact Information:**

**Robert C. Ilardi**

**Email: [rilardi@gmail.com](mailto:rilardi@gmail.com)**

**Web Site: <http://www.roguelogic.com>**

# Syllabus

## Lesson 1 – My First Java Program

- The simplest Java Program. Outputs to the screen “Hello World”
- How to write a Java Program
- Compile a program from the command line
- Run a program from the command line
- What is the classpath?
- What is a Package?
- Environmental Settings for Java
  - Same on Windows or Unix/Linux: JAVA\_HOME, PATH, CLASSPATH
- What is a JAR File?

## Lesson 2 – Procedural Programming in Java?

- This shows you that you can actually just create a single Java class with “functions”
- You should never program in the real world like this.

## Lesson 3 – Classes and Objects

- Basic Object Oriented Programming (OOP) Guide
  - What is OOP?
  - What is a Class?
  - What is an Object?
  - What's the difference between a class and an object?
  - Java Class verses Java Interface

## Lesson 4 – Built In Java Data Types

- Two Types of Data Types in Java: Primitives and Objects
  - What's the difference?
- Primitive Types: int, float, double, char, long, byte, boolean, short
- Included Object Data Types: String, Date
- Primitive Wrapper Objects: Integer, Float, Double, Character, Long, Byte, Boolean, Short
  - In some cases you need to pass around a variable such as an int as an object. In this case you would use Wrapper Types.
  - For “MOST” Java programs you can usually ignore wrapper types, unless you want to store a primitive in a Collection (See the Lesson on Collections for details).
  - Notices there's one for each Primitive Type, only difference in most cases which for Character are the same name as the primitive, only with a Capital Letter as the first letter. In Java, everything is CASE-SENSITIVE.
  - Wrapper Types are used to “wrap” a primitive in an Object. Example If you want to wrap an would wrap an “int” variable in an Integer object.
- Type Casting
- Arrays

## Lesson 5 – Operators, Loops and Logic Statements (Control Statements)

## Lesson 6 – Collections

- Lists: ArrayList / Vector
- Maps: HashMap / HashTable

## **Lesson 7 – Exceptions**

## **Lesson 8 – JDBC (Java Database Connectivity)**

- How do we connect to a database in Java?
- Basic JDBC Programming from a Command Line Java Program.

## Reference Links for Java

The Java Programming Language Home Page: <http://java.sun.com>

+ Go here for ALL your Java needs. Including downloading the JDK.

Note: When downloading Java you want to download **Java SE (Java Standard Edition)**. For compiling you need the **JDK (Java Development Kit)**. There's also something called **JRE (Java Runtime Environment)**. This is included in the JDK so if you download and install the JDK you don't need to install a separate JRE. The JRE will allow you to run Java programs but does NOT include the compiler.

The **Standard Java Library** has a reference guide called **Java Docs**. There's a Java Docs web page for each every of Java. This guide is based on Java 1.5 but here are the links for the Java Docs for Java 1.5 and Java 1.6:

Java Docs for 1.5: <http://java.sun.com/j2se/1.5.0/docs/api/>

Java Docs for 1.6: <http://java.sun.com/javase/6/docs/api/>

The Java Docs is the reference guide for ALL classes included in the Standard Java Library.

Note on **J2EE**: This is the **Java Enterprise Edition**. This is basically just a Library and is used for enterprise programming like EJBs, JMS, Servlets and JSPs.

You will first need to install the Java Standard Edition before installing or using J2EE.

## **Intended Audience**

The intended audience for this guide on the Java Programming Language is Experienced Programmers that want a quick concise read to get them started with the Java Programming Language. This guide does NOT assume you have any knowledge of Java or Object Oriented Programming.

At the end of this guide you should be able start reading and modifying other peoples code, as well as start creating your own programs in Java from scratch. What this guide will NOT do is make you an expert with Java. You should read other documentation and books on Java and practice programming in Java, and perhaps even take a training course or two, if you want to be come a “senior” Java developer.

It is the author's opinion that, the only real way to become a highly productive programmer in any language including Java is to work with it on a frequent basis either at your job (as a professional programmer already, who wants to move to the Java language to participate in Java development projects within your company), or for programming projects at school and at home.

## **Additional Lesson Requests**

This Self Study Guide should be treated as a living document. It will be updated from time to time without notice. Please check <http://www.roguelogic.com> for updates. If you have a topic related to Java that you would like to see included, please email your request to: [rilardi@gmail.com](mailto:rilardi@gmail.com)

# Lesson 1 – My First Java Program

## The Simplest Java Program:

The simplest Java Program in these lessons is the HelloWorld.java class. It is just a single Java Class Source File which once compiled and ran will output to the screen the words "Hello World! - This is my first Java Program!!!"

Included in lesson one are two versions of the same program.

One is in the directory: "C:\java\_dev\lessons\lesson1" and is named "HelloWorld.java".

The second version which does the same exact thing, is in directory "C:\java\_dev\lessons\lesson1\examples\test1" and is named "HelloWorld.java" as well.

The only difference between the two files is the very FIRST line of the file "C:\java\_dev\lessons\lesson1\examples\test1\HelloWorld.java".

The first line of the second version of this file is: package examples.test1;

We will explore what this line means later in this lesson when we go over packages.

The HelloWorld programs in this example are good examples to show the make up of a Java program and explain some of the rules around File names and their relationship to Class names.

Let's take a look at the HelloWorld.java file in the lesson1 directory:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello World! - This is my first Java
Program!!!");
    }
}
```

The very first line of this file is the "CLASS DECLARATION." The keywords "public class" tells java that I am constructing a class and the word "HelloWorld" is the NAME of the class.

It is VERY IMPORTANT to note that the name of the class here "HelloWorld" is spelled EXACTLY the same way as the filename

"HelloWorld.java". Notice that even the Case of the letters are the same. The only difference is that we left off the ".java" part.

In java you MUST follow this rule. To create a Java Class you create a TEXT file (you can use Note Pad on windows for this) to store the source code in. This file MUST end with the file extension ".java" and the class name in the source code MUST be the same name of the file without the ".java" part.

Also, please note some additional rules for naming Classes: You can NOT until spaces. You can only use Letters and/or Numbers. And the first character of the name MUST be a letter NOT a number. It's just like the rules for naming Variables in most languages.

In Java like C or C++ "scope" is controlled by the curly brace characters; "{" is used to begin and "}" is used to end.

When we create a class ALL code MUST be within the "{" and "}" of the class. If you look at the code above, the line directly after the class declaration is simple "{". This tells the Java Compiler that the code of the class will start. And when we find a matching "}" this tells the java compiler that the code has ended for the class.

The "HelloWorld" class in this lesson is very simple and is barely a class. In fact it does NOT use object oriented programming almost at all, except for the fact that we are "storing" the code of this program inside a class called HelloWorld.

This example is more of an example of PROCEDURAL programming in java. In the java programming language, the MAIN entry point of a class for execution is a function (As Known in Java and other object oriented languages as a Method. We will use the words "function" and "method" interchangeably in our discussion in these lessons.), is a function called "public static void main(String[] args)". We will explain each part of what this actually means later on.

For now, just remember that the starting point of a java program is the MAIN method which is denoted by the words "public static void main(String[] args)". So whenever you need to find where a java program begins, look for this method and you can begin to trace through the code.

Notice again we START the code of the MAIN function with the open or begin curly brace, as highlighted in **bold** below:

```
public static void main(String[] args)
{
```

```
        System.out.println("Hello World! - This is my first Java
Program!!!");
    }
```

Remember just like in the case of the class as previous discussed, any code within this brace until we find a matching ending or closing curly brace as highlighted in **bold** below:

```
    public static void main(String[] args)
    {
        System.out.println("Hello World! - This is my first Java
Program!!!");
    }
```

It might be a good time to explain what we mean by "matching." You may find many sets of opening and closing curly braces within a block of code. As we have with the class itself in this example. There's two complete sets, one opening for the class and then one opening for the main method, then one closing for the main method, and then one closing for the class.

When we say matching it means a complete set. You can NOT have a starting or opening brace without a corresponding closing brace. And the closing brace will "close" the code scope for the immediate previously specified opening brace.

It is also a good time to let you know that in the Java programming language, WHITESPACES including NEWLINES are "IGNORED" by the compiler. So the following two sets of code are the same:

```
public static void main(String[] args)
{
```

and

```
public static void main(String[] args) {
```

Are exactly the same to the compiler.

Also, just like the NEWLINES do NOT mean anything to the compile, so do TABS or indentation.

We use indentation in Java just to make the code more readable. Indentation is used to show clearly the code scope.

Example, the functions that are part of the class HelloWorld. In this case on the MAIN function is indented with a single TAB after the opening curly brace of the class.

Then ALL code of the MAIN function itself is indented TWICE from the left margin, after the opening curly brace of the MAIN function. Once for the class and once for the function.

This is NOT necessary for the compiler, but is the normal coding convention of Java programmers to make sure the code is readable.

### **How to compile a Java program from the command line?**

The JDK (Java Development Kit) includes the Java Compiler, which is a program called javac (or javac.exe on windows).

To compile a Java source file (a file that ends with ".java") you just run from the command line javac and the java filename immediately after it.

Example:

```
javac HelloWorld.java
```

To compile multiple java files within the same directory you can just specify:

```
javac *.java
```

This will compile ALL java source files within the current directory you are in at the command line.

Once you run the compiler, if the code has no compile time errors in it, the compiler will create a "binary java executable" file which has the same filename as your source code file but it will replace the ".java" with ".class".

In this lesson if you run javac HelloWorld.java, the compiler will produce HelloWorld.class in the same directory as the HelloWorld.java file.

To be able to run javac you need the JDK's bin directory in your PATH environmental variable on Windows or Unix/Linux. The Java Dev CD that comes with these lessons has a JDK installed in

C:\java\_dev\jdk1.5.0\_06. The javac compiler program is located in C:\java\_dev\jdk1.5.0\_06\bin. You need to add this directory to your PATH. You could run the program with the full path but this is a waste of time, but here's how you would do it anyway using the JDK installed on the Java Dev CD. Run from the command line:

C:\java\_dev\jdk1.5.0\_06\bin\javac and then the filename of the java source file you want to compile. It is preferred that you ADD the JDK's BIN directory to your PATH.

Note: ALL lessons include a compile.cmd windows batch file which will compile everything in the LESSON you are in for you. It sets the PATH and other variables automatically for you using the set\_base\_env.cmd in the C:\java\_dev\support directory included on the Java Dev CD.

In the "real world" you won't have the java\_dev CD and the JDK will probably be installed in a directory like: "C:\Program Files\Java\jdk1.5.0\_09" You should read up on the internet (do a google search on: "setting up java on my computer").

### **How to run a Java program from the command line?**

The JDK (Java Development Kit) includes the JVM (Java Virtual Machine), which is a program called java (or java.exe on windows). Almost the same as the compiler, but without the "c" on the end of java. Obviously the "c" means compiler. You can think of this program as an Interpreter. But in actuality it is more complex than that. But it's easy to think of it as an Interpreter.

To run the HelloWorld java program which is compiled into the binary file "HelloWorld.class" in the lesson1 directory you simply run:

```
java HelloWorld
```

Notice that we did NOT write it as: java HelloWorld.class. This will cause an error and the program will NOT run.

What types of Java classes can I run? Well in large programs, there will be many binary class files. The only ones you can run are the ones that contain a MAIN method specified as previously discussed: "public static void main(String[] args)". Remember, this method name is the signature that tells Java, start running the program from here. Not all Java classes will have their own MAIN methods as we will see in later lessons, so it is important to note and remember that you can only run the java command with a Class Name of a class that has a MAIN method in it.

### **What is the classpath?**

The classpath in java is a list of directories, similar to the PATH environmental variable, which instead of the PATH which specifies directories for the Operating System to look for the programs to run, the CLASSPATH is an environmental variable that tells the JVM (the java interpreter) what directories to look for the Classes to be able to run. Since the JVM is a interpreter program it can read the CLASSPATH environmental variable, or you can specify the classpath

directly from the command line:

```
java -classpath C:\java_dev\lessons\lesson1 HelloWorld
```

We didn't have to do this, because if you DO NOT specify the classpath like we did with the `-classpath` option to the JVM, the java program will try to read the `CLASSPATH` environmental variable. If it doesn't find one, it will assume the classpath is the current directory.

It is important to understand the classpath when we speak about packages later in this lesson, because packages are specified and understood by the JVM as "relative" to the `CLASSPATH`.

Just remember this about the `CLASSPATH`:

The `CLASSPATH` is an environment variable that tells the `java.exe` where to find class files to interpret. It also tells `javac.exe` where to find already compiled classes to use when compiling other classes which may reference those other pre-compiled classes, such as Classes from a Library.

One last note on classpath: You can include MULTIPLE directories in the classpath. Each directory on windows would be separated by `;`. This can be used on both the environmental variable or the `-classpath` option on the `java.exe` program.

## **What are Packages?**

The creators of the Java programming language, tried to find a way for programmers to better be able to organize code. Their answer was Packages.

Packages are simple a directory path where the actual classes are stored. The only important thing to remember is that when you specify a package you do not specify the FULL PATH to the class you are try to reference, instead it is the relative path from where that package exists from one of the directories on the `CLASSPATH`.

In this lesson, the version of the HelloWorld program that's in directory: `C:\java_dev\lessons\lesson1\examples\test1\` is in package `examples.test1` when we set the classpath to `C:\java_dev\lessons\lesson1\`.

This can be confusing, but it's less confusing when you think that this is NOT a requirement of Java, but a requirement that the programmer of the HelloWorld.java file is setting on the program.

If the java programmer specified the package in the code to be

"lesson1.examples.test1" then the person running the program would have to set their CLASSPATH to "C:\java\_dev\lessons\" instead.

Let's go back to the first line of the file:

```
C:\java_dev\lessons\lesson1\examples\test1\HelloWorld.java:
```

```
package examples.test1;
```

This directive in the Java Programming language tells the compiler and the JVM (Java Virtual Machine, the "interpreter" of the Java Programming Language. AKA on windows java.exe) that the class file (the compiled version of the Java source file) MUST be in a directory examples.test1.

You can specify any valid directory name after the word "package" at the top of a Java source file. It can NOT include the drive letter like "C:\" and ALL "\" or "/" must be written as "." (Periods/Dots). It is also important to note that you DO NOT specify the FULL PATH or ABSOLUTE PATH of where the Java file is located. As mentioned above, you instead specify the RELATIVE PATH from where you want the user of your program to set their classpath to.

In this case, because we specified the package as "examples.test1" the user of the program once compiled MUST set their classpath to C:\java\_dev\lessons\lesson1\

Example to run the HelloWorld program that's in the examples\test1 directory:

```
java -classpath C:\java_dev\lessons\lesson1 examples.test1.HelloWorld
```

Let's take this command apart:

The first word in the command is "java" this is the JVM Interpreter program.

The second word is "-classpath" this is the option that is passed to the java program that tells it to read the next word on the command line as the CLASSPATH.

The third "word" is "C:\java\_dev\lessons\lesson1" this is the directory up until the point where the directory "examples" is found. This makes the package "examples.test1" relative to the CLASSPATH and is why we set the classpath to "C:\java\_dev\lessons\lesson1" in the first place. Because remember the programmer of the HelloWorld program specified the package to be "examples.test1". So that programmer forced us to set our classpath to the parent directory of "examples" so that it becomes relative to the classpath.

The fourth "word" is the class name specified in what we call the "Fully Qualified" name. This is because it contains the package "examples.test1" and then "." + the class name "Hello World".

If you ran in `java -classpath C:\java_dev\lessons\lesson1 HelloWorld` instead. You would be running the first version of class that's in directory `C:\java_dev\lessons\lesson1` instead of `C:\java_dev\lessons\lesson1\examples\test1`.

Go ahead of try it you will get different output:

```
java -classpath C:\java_dev\lessons\lesson1 HelloWorld
```

will give you the output:

```
Hello World! - This is my first Java Program!!!
```

-AND-

```
java -classpath C:\java_dev\lessons\lesson1 examples.test1.HelloWorld
```

will give you the output:

```
Hello World! - This is my first Java Program!!! - Package Version
```

They are TWO separate programs with the same class name but in different packages.

I know it's probably easier to forget about packages, because they seem like just an extract language feature, but it is important to understand packages because in the real world ALL java programs will be specified in a package.

### **What happens if my class is in one package and I need to reference a class in another package?**

There's a java directive statement called "**import**" which normally you will see at the top of a Class right below the package statement.

All you need to do to use a class from a different package is import that package.

This is how you would do it:

At the very top of the \*.java source file (right below the package statement if you have one) you write:

```
import [PACKAGE_NAME].[CLASS_NAME];
```

Example:

If I'm outside of the examples.test1 package and I want to use the HelloWorld class from this package all I need to do is write:

```
import examples.test1.HelloWorld;
```

There's also a short cut if you need to import many classes from the same package. You could write:

```
import examples.test1.*;
```

The above line will IMPORT ALL CLASSES from the package examples.test1.

One good thing about imports is:

Most people familiar with other languages that "link" libraries together in the executables would say if I **import package\_name.\***; the binary will become overly large because I'm linking or including other classes which I did NOT need.

This is NOT the case in Java. The compile will optimize the imports for you if you import "\*" from a package.

### **Importing Classes from the Standard Java Library:**

Imports are NOT just for USER CREATED Packages. If there's a class in the Standard Java Library you MUST import it as well if you want to use it.

The ONE AND ONLY exception to this is the "java.lang" package. Things like String are in here and you do NOT need to import this. Although you can if you really wanted to, but NO Java programmer would ever do so.

But other things like Date or other classes you might need to use are in other packages like java.util for example and if you want to use this you want to do:

```
import java.util.*;
```

### **Compiling and Running the Lesson Examples using the included Windows Batch Files:**

**How to Compile Lesson using the included Windows Batch Files:**

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson1`
3. `compile.cmd`

#### **How to Compile Lesson using javac.exe program directly:**

Note: This assumes you have the javac.exe program in your PATH.

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson1`
3. `javac HelloWorld.java`
4. `javac C:\java_dev\lessons\lesson1\examples\test1\HelloWorld.java`

#### **How to run the examples using the included Windows Batch Files:**

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson1`
3. To run the example without the package (Java Class: HelloWorld.java) run: `run_helloworld_nopackage.cmd`
4. To run the example with the package (Java Class: examples\test1\HelloWorld.java) run: `run_helloworld_withpackage.cmd`

#### **How to run the examples using the java.exe program directly:**

Note: This assumes you have the java.exe program in your PATH.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson1`
3. To run the example without the package (Java Class: HelloWorld.java) run:  
`java -classpath C:\java_dev\lessons\lesson1 HelloWorld`

4. To run the example with the package (Java Class: examples\test1\HelloWorld.java) run:  
java -classpath C:\java\_dev\lessons\lesson1 examples.test1.HelloWorld

## **Environmental Settings for Java**

Already in this lesson we spoke about the two main environmental variables for JAVA. The Path and the Classpath. There's one more that's usually a good idea to have set which is the JAVA\_HOME variable.

In this section we will go over 1. what these variables mean and actually do. 2. How to set them from a dos prompt. And 3. How to set them from Windows My Computer - System Properties.

### **What these variables mean:**

#### 1. PATH

In windows, the PATH is an environment variable that tells the command line/DOS prompt and Windows itself (example the "Run" from the Start Menu) the where to look for executable, like: \*.exe, \*.cmd and \*.bat files.

You can specify a list of directory in the PATH variable, each directory separated by a ";".

Related to Java, you would normally add the BIN directory of your JDK installation to your PATH variable on your computer. This is so java.exe and javac.exe and be easily found and worked with from anywhere in windows.

#### 2. CLASSPATH

We spoke about this in detail already in this chapter. But in summary, the classpath is an environmental variable that tells the java.exe interpreter program in which directories to look in for compiled Classes and as the parent directories for Packages.

You can specify a list of directory in the CLASSPATH variable, each directory separated by a ";" just like you do with the PATH.

#### 3. JAVA\_HOME

This is the only really new environmental variable that we haven't already mentioned in this lesson. The Java Home variable basically is the directory of the JDK installation. On the Java Dev CD included with these lessons the JAVA\_HOME should be set to: C:\java\_dev\jdk1.5.0\_06.

If you have the JDK installed in some other directory you should set this variable to that directory instead. This tells complex programs like Apache Tomcat where to find the entire Java Installation on the computer.

### **Here's a quick guide how to set these variables from a DOS Prompt.**

Note: When you set these variable from the DOS prompt, they will only exist and be set to the values you specify for that SINGLE DOS PROMPT Window you are running the "set" commands in.

To add the Java Dev CD's JDK BIN directory to the PATH from the command prompt type and run the following:

```
set PATH=%PATH%;C:\java_dev\jdk1.5.0_06\bin
```

To set JAVA\_HOME to the JDK directory included on the Java Dev CD:

```
set JAVA_HOME= C:\java_dev\jdk1.5.0_06
```

We will go over setting the classpath a little differently because it should be set to the classpath needed for each java program individual. You can specify a "global" classpath, but this is normally only used to include Libraries that are installed on your system.

In this case if you have a "global" classpath already set and you want to for example add lesson 1's directory to the classpath from a DOS prompt type and run:

```
set CLASSPATH=%CLASSPATH%;C:\java_dev\lessons\lesson1
```

Notice the %CLASSPATH% comes before the ";C:\java\_dev\lessons\lesson1" this is to make sure we concatenate the original classpath in front of the lesson1 directory on our new classpath.

If you wanted to overwrite the classpath completely or do NOT have a global classpath set on your system you would type and run:

```
set CLASSPATH=C:\java_dev\lessons\lesson1
```

instead.

### **How to set these variable using the Windows My Computer System**

## Properties.

Note: When you set the variable using this method, they will be set globally for your PC and will work on all newly opened DOS prompts and will be retained even after you reboot the PC. You can of course always repeat these steps to remove or change them.

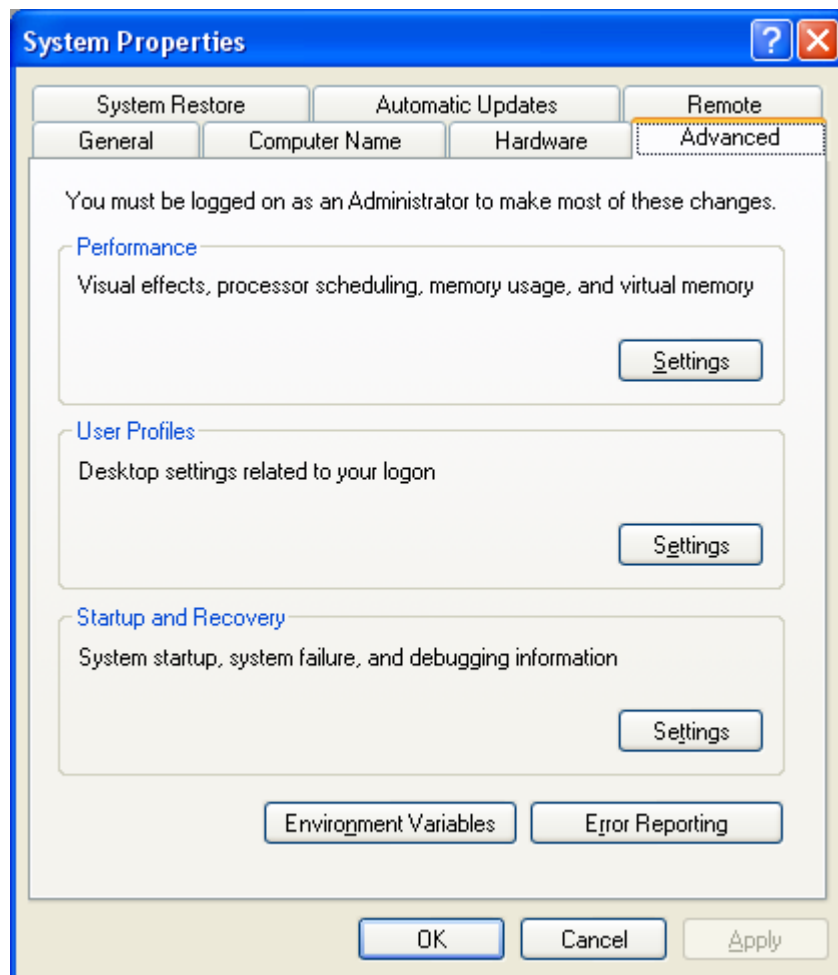
Steps:

1. Right Click on My Computer and click on "Properties". If you do NOT have the My Computer Icon on your desktop you can alternatively go to Control Panel and double click the "System" icon.

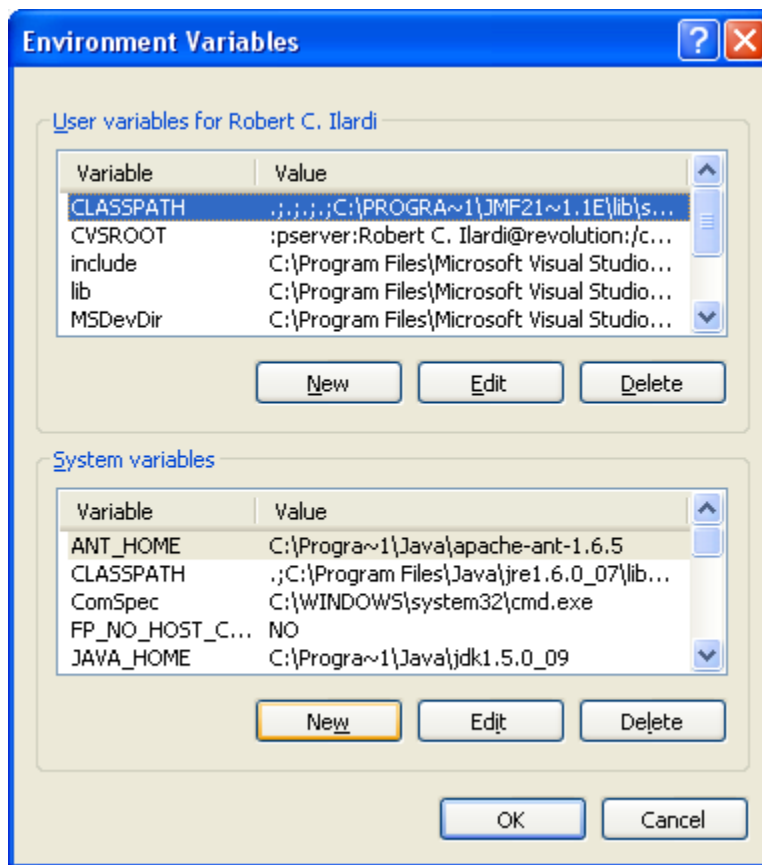
You will get this screen:



2. Click on the "Advanced" tab:



3. Click on the "Environmental Variables" button:



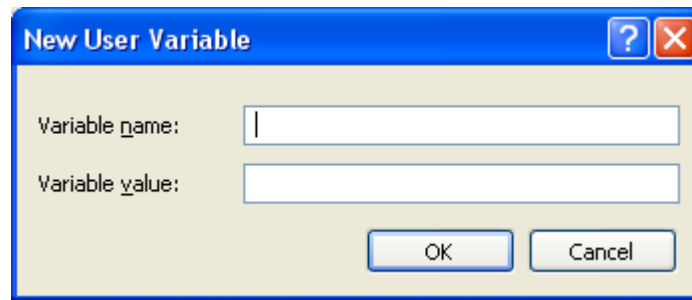
Note: As in the window above you will see two sections "User Variables" and "System Variables." If you are a computer administrator you can set the System Variables, which will set the variables for ALL users of the computer. If you are NOT an administrator or you only want to set the variables for yourself, you can just set the User Variables.

3. I recommend you just set the user variables for now until you are comfortable with editing these types of settings on your computer. So the remainder of the steps will focus on adding or editing the variables in the user variables section.

4. Browse the list of User Variables for the THREE Environmental Variables we spoke about: PATH, CLASSPATH, and JAVA\_HOME.

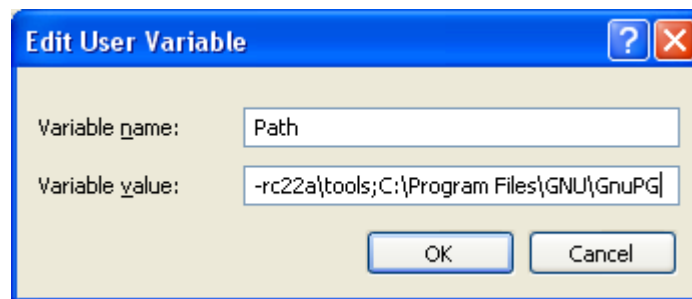
5. If you already have any of them, you can click the edit button to change the value, if you don't have any one of them you can use the New Button to add a new Variable.

6. Adding a new JAVA\_HOME variable. Click on the "New" button. You will get the screen below:



For the Variable Name if you did NOT see JAVA\_HOME already listed, type "JAVA\_HOME" and set it to the JDK's main directory. If you want you can set it to the JDK including on the Java Dev CD. To set the variable type or copy and paste the directory name in the Variable Value text box.

7. If You already have one of these variable you can edit it by clicking on the Edit button and you will get the screen below, which shows my Path variable opened for edit.



Note: If JAVA\_HOME is already set on your system you probably should leave it, unless you want to change it to a different JDK installation.

Note: For Path and Classpath, separate the different directories you want to include by ";".

### **What is a JAR File?**

A JAR file is a type of "ZIP" file which normally includes the compiled \*.CLASS files that make up a Java program. Usually libraries will be packaged inside a JAR file. This makes a simple and convenient way to distribute and store Java binaries in a single file. Most large Java programs will be made up of 100's if not 1000's of classes and therefore distributing them separately is very messy and the potential for mistakes in copying and deploying them is high.

JAR stands for "Java **AR**chive.". It really is just a ZIP file and can

be renamed \*.ZIP and opened with a program such as WinZIP. There's a command line utility for creating \*.JAR files included in the JDK. The program on windows is called: jar.exe. It is in the same directory as java.exe and javac.exe. So if you included the BIN directory in your PATH you can simply execute JAR right from the command line.

To create a JAR the command is: jar cvf [JAR\_FILE\_NAME] \*.class

JAR\_FILE\_NAME should include ".jar" in lowercase at the end of the name.

Do this from the ROOT directory of your Java program, where you compiled from. This is so all packages and sub packages in your program will be included.

Once you have a \*.JAR file you can directly include it in any CLASSPATH directly! You do NOT have to decompress the Archive! Java understands how to read from JAR files directly!

Example (Skipped ahead to lesson 8 on JDBC. The first lesson in this Guide to use a third party library):

```
java -classpath
C:\java_dev\lessons\lesson8;C:\java_dev\java_libs\mysql-connector-
java-5.1.5-bin.jar JDBCTest
```

Notice the part highlighted in bold above. After the lesson directory in the classpath, we put the Windows delimiter for classpaths which is ";" the semicolon, then we simply places the FULL PATH to the JAR file we want to include!

That's how simple it is to use JARs!

In a possible future lesson on the Java Build utility ANT, we will explore JARs in more detail. For now just know they exist and what they are used for and how to include them in the classpath, as show above, and used in Lesson 8!

## Lesson 2 – Procedural Programming in Java?

This lesson will be a short one. It is just to show that it is really easy to program in Java and you don't really have to worry too much about all the Object Oriented stuff just to get started in experiment and learning with the language.

We have already seen an example of a simple procedural program in Java with the HelloWorld class from Lesson 1.

With the HelloWorld example, there's only a single method or function which is the special MAIN method.

The method signature which was: `public static void main(String[] args)` has some special properties that allows it to be called by the JVM "outside" of the whole Object Oriented paradigm. We can follow this same pattern to create other method or functions which can be called without having to worry about OOP (Object Oriented Programming) as well.

Let's take a look at the main method a little more closely:

```
public static void main(String[] args)
```

"public" in bold above, is what is called the access modifier in Java. We will export the different types of access modifiers available in Java when we speak about OOP in the next Lesson.

For now, just assume that when you want to do procedural programming in Java you should make all your method **public**.

The next part of the method signature:

```
public static void main(String[] args)
```

is "static" this is a special keyword in Java that means the method is at the "Class Level". We will explore what this means in the next lesson on OOP. But for now, what this means is that you do NOT need to reference the method from an Object. Think of it as the keyword that marks the function as a "global" function.

The next part of the method signature is the return data type. Or what kind of variable can be returned by the function. In this case we specify the data type "void" as highlighted in bold below:

```
public static void main(String[] args)
```

void is a special data type which means the function returns NOTHING. You can use any data type in place of void if you need to return some

variable. It can even be Objects, but we will discuss this later when we talk about Data Types in Java.

The next part of the method signature is the NAME of the function. In this case it is the name "main" as highlighted in bold below:

```
public static void main(String[] args).
```

The final part of the method signature is the parameter list of arguments you will pass into the function.

In this case it is a single parameter which is a String array specified in the Java array notation with the square brackets immediately after the data type "String". This is highlighted in bold below:

```
public static void main(String[] args)
```

The "args" part is the variable name for the String array, so you can reference this string array from inside the function as the variable named "args".

### **So what does this all mean?**

Basically you just need to create a public static function in java if you want to do procedural programming. You really shouldn't do this in the real world unless you are creating a "helper" class, which is just a Class that has a collection of public static methods in it each of which might do a very simple but highly reusable function.

In the directory: C:\java\_dev\lessons\lesson2 there's an example of how to do procedural programming in java. Again we are taking a different twist on the HelloWorld example from Lesson 1.

This time we have split the Hello World into two classes. This is to illustrate how to split up methods into different classes and if they are static you can reference them directly from the Class itself. We will show in the next lesson there's one more step to do if you are creating "non-static" methods which means they are Object Level methods. For now, again, we are ignoring most principals of OOP and just focusing on the Procedural programming aspects of Java.

The first one Class is "HelloWorldService" (in file HelloWorldService.java) and contains the implementation of the functions used to create "Hello World" messages.

The second Class is "HelloWorldTester" (in file

HelloWorldTester.java) and contains the main method which will be used as the entry point into our program and will Call functions on HelloWorldService.

Think of HelloWorldService as a library module and you are referencing this from the HelloWorldTester program to reuse its functions.

Let's take a look at the HelloWorldService code:

```
public class HelloWorldService
{
    public static String sayHello(String name)
    {
        return "Hello " + name;
    }

    public static void printHello(String name)
    {
        String mesg = sayHello(name) ;

        System.out.println(mesg);
    }
}
```

In this class, we created a two methods. The first one is: **public static String sayHello(String name)**

The method is called "sayHello" and because we made it public static, is can be referenced in the procedural way from both within the HelloWorldService class itself as shown in method printHello. Or we can call it from the main program entry point class "HelloWorldTester" as shown in the main method of the code below:

Code from HelloWorldTester.java:

```
public class HelloWorldTester
{
    public static void main(String[] args)
    {
        //We are calling the sayHello method directly
        //which will return a message which we are
        //printing to the screen.
        String mesg;
        mesg = HelloWorldService.sayHello("Robert (returned)");
        System.out.println(mesg);

        //Here we are calling the printHello method which
```

```
        //does NOT return anything
        //and prints the Hello message for us on the screen.
        HelloWorldService.printHello("Robert (printed for me)");
    }
}
```

Notice there's one difference between the two lines in both above.

In the HelloWorldService class's method "printHello", we called sayHello without the "HelloWorldService." prefix. This is because in the case of printHello, we are in the same Class as the "sayHello" method so we don't need the prefix (although you could specify the prefix if you really wanted to).

In the HelloWorldTester class's method "main", we called "sayHello" by adding the prefix "HelloWorldService." which tells the compile to look for the function "sayHello" in the Class "HelloWorldService".

The second method "printHello" was added just to show that you can have as many "public static" procedural methods in a single class as you want and how to call them from both inside the same class and outside in a different class.

### **Compiling and Running the Lesson Examples:**

#### **How to Compile Lesson using the included Windows Batch Files:**

1. Start a DOS Prompt. And from the dos prompt run:
2. cd\java\_dev\lessons\lesson2
3. compile.cmd

#### **How to Compile Lesson using javac.exe program directly:**

Note: This assumes you have the javac.exe program in your PATH.

1. Start a DOS Prompt. And from the dos prompt run:
2. cd\java\_dev\lessons\lesson2
3. javac \*.java

#### **How to run the examples using the included Windows Batch Files:**

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson2`
3. To run the example from the program's main entry point class (Java Class: HelloWorldTester.java) run: `run_helloworldtester.cmd`

**How to run the examples using the java.exe program directly:**

Note: This assumes you have the java.exe program in your PATH.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson2`
3. To run the example from the program's main entry point class (Java Class: HelloWorldTester.java) run: `java -classpath C:\java_dev\lessons\lesson2 HelloWorldTester`

## Lesson 3 – Classes and Objects

### What is Object Oriented Programming (OOP)?

Basically, OOP is just a way of organizing code and maximizing reusability.

There are four basic concepts of OOP and ALL object oriented languages MUST follow these four concepts to be considered Object Oriented:

1. **Abstraction** - This basically means the ability of the language to allow programmers to create a "module" which will contain all the details of an implementation from other programmers reusing the code. This basically means the ability to create a Class or Object to represent logical groupings of functions and data in a reusable form.
2. **Encapsulation** (sometimes called Data Hiding) - This basically means you will hide all functions and data elements within a class. For example, if we have a Class to represent a Dog, there might be a function called "bark" within that class, which will act upon the Dog object. The Dog object may also contain a String data member (or member variable) called "name" to store the name of the dog. A data member (also known as member variable or field) can be thought of a "global variable" which exists only within the context of the Object.
3. **Polymorphism** - This is the ability of the language to allow child objects to act like their parents. At a lower level it also allows for function overloading. An example, if we have a Person class which we inherit from and create an Employee child class using this Person parent class. An Employee IS A Person and therefore has all the attributes and functions of the parent Person class as well. There's some caveats here, which we will explore with Inheritance and specific to Java with access modifiers.
4. **Inheritance** - This is the ability of the language to allow for one Class to extend or become a child class of another. As mentioned in Polymorphism a Person class can be extended to create an Employee class and the Employee class retains the functionality and attributes of the parent Person class.

### Object Oriented Programming in Java

In Java like many other OOP based languages, the root concept is the Class.

Think of Classes as Structures with functions inside of them as well.

Classes are also known as User Defined Data Types.

### **So what is the difference between a Class and an Object?**

The technical definition is that an Object is an in memory instance of a particular Class.

Basically you write code in a Class and when you create a variable having the Data Type of that Class, it becomes an Object instance of that class in memory.

That great advantage of this is that you create create N number instances of your class in memory. Therefore, let's take for example a class called Person. If a person contains a data member String name, and I create two Object instances of this class, then both object exist in memory at the same time with their OWN names, so that they do not overwrite each other.

If I relate this to a procedural language, then how would I do this? I an create a global variable called name, but then that name variable is shared throughout the code. There's many ways around this of course, I could create a Global Array of names, and somehow in code make sure that each in memory representation of a Person knows how to reference it's own array element in the name array.

But this is hard to do and take a lot of code, and with the idea of Objects, you don't have to worry about this. Each object has it's own "memory space."

For simplicity, just think of a Class is the original copy with your code and Objects are separate in memory copies of that Class.

### **Access Modifiers in Java -**

Let's quickly go over the Access Modifiers in Java before continuing. We already saw the first access modify, the keyword "public" when defining class and the "procedural methods" like "**public** static void sayHello(String name)" from the previous lessons.

First of all, what is an Access Modifier?

Basically all they are, are Java keywords which control the "visibility" of functions and member variables of classes. As mentioned in the concepts of OOP, the "level" Data Hiding or how hidden is something in a class is controlled by the Access Modifier.

There are THREE Access Modifiers in Java:

1. **public** - This means EVERYONE CAN SEE OR CALL the function or member variable of a class if the function or member variable is prefixed or marked as public.

2. **private** - This means that ONLY THE CLASS ITSELF CAN SEE OR CALL the function or member variable of a class. This means not even a Child Class which inherited from a parent can see these functions or member variables of the parent.

3. **protected** - This means that ONLY THE CLASS ITSELF AND IT'S CHILD CLASSES CAN SEE OR CALL the function or member variable of a class.

For now we will just focus on public and private and leave protected for a future lesson on Inheritance.

First, let's create a simple class and show how to create Objects of that class:

Note: See file: C:\java\_dev\lessons\lesson3\Person.java

```
public class Person
{
    private String name;

    public Person() {}

    public void setName(String aName)
    {
        name = aName;
    }

    public String getName()
    {
        return name;
    }
}
```

Let's go over the make up of this class.

The line "**private String name;**" is a member variable (aka data member or field) of the class. It has the access modifier private, which means only this class can directly see this variable.

Basically we are using this variable to store the name of the person as a String.

Remember what we said about member variables. They are sort of Global

Variables within the separate memory space of each Object instance of the class.

In our test of this class, we will create two Object Instance of the Person class, and show that they both have their own names in memory at the same time.

The next line is "**public Person() { }**". This is known as the **constructor** of the class. It is like a function. It actually is a function and is used to initialize a class. Notice here there are opening and closing curly braces without any code in between. This is because I didn't want to do anything special to initialize the Person Objects. If I had to do something special, I would put the code here. If not, just leave it empty as I did in this example.

We will see in the Tester example when the constructor is called. For now just know that every class should have a constructor, unless we are just putting procedural methods in the class, like we did for the Hello World examples in the previous lessons and in the tester classes which only have the main method.

As I mentioned constructors are functions, but there are a few differences. The first obvious difference is there is NO RETURN DATA TYPE. Not even void. This is just a rule for constructors.

The second is that the NAME of the constructor MUST be exactly the same name as the Class itself.

You can have as many constructors you want with different parameter lists, but the "default" constructor just follows this format: `public [CLASS_NAME] ()`.

The rest of the class contains code for two functions. Both are public, so they are visible to everyone and therefore everyone can call them, but inside and outside of this class.

The first method is "setName" and basically, this is used so that programmers making use of the Person class can set the name to the private String name member variable. You might be asking why can't I just make name public and do something like:

```
Person p = new Person();
```

```
Then p.name = "Some Name"
```

You can, but this is not GOOD OOP code and shouldn't be used.

In general the rule is make ALL member variables private or protected and have set and get methods to set and access them.

Also, in the current code, with "private String name."

If you did:

```
Person p = new Person();  
  
p.name = "Some Name";
```

Then you will get a compiler error, because it is private and you can NOT access things that a private.

The next method is the "getName" method which obviously just returns the name of the Person.

-----

Next Let's create a test program to use the Person Class:

```
public class PersonTester  
{  
    public static void main(String[] args)  
    {  
        Person p1, p2;  
  
        p1 = new Person();  
  
        p2 = new Person();  
  
        p1.setName("Robert");  
  
        p2.setName("Paula");  
  
        System.out.println(p1.getName());  
  
        System.out.println(p2.getName());  
    }  
}
```

So what are we doing in this class? Basically we are making use of the Person class and creating two Object Instances of the person to exist in memory at the same time.

Let's see how we are doing this:

The first line of the main method is: "**Person p1, p2;**" this is a variable declaration. The data type is class Person (which we just created.) and I'm creating two variables of type Person. One is p1 and the second is p2.

The next two lines are "**p2 = new Person();**" and "**p2 = new Person();**". These two lines actually create the Objects of type Person in memory. What does this mean? Well all it means is that the compiler will allocate memory for two objects big enough to store two SEPARATE copies of a Person in memory.

Remember I said we will see in the tester how the constructor is called. Well look at the line "**p1 = new Person();**" See the "Person()" part in bold. This is actually a call to the constructor!

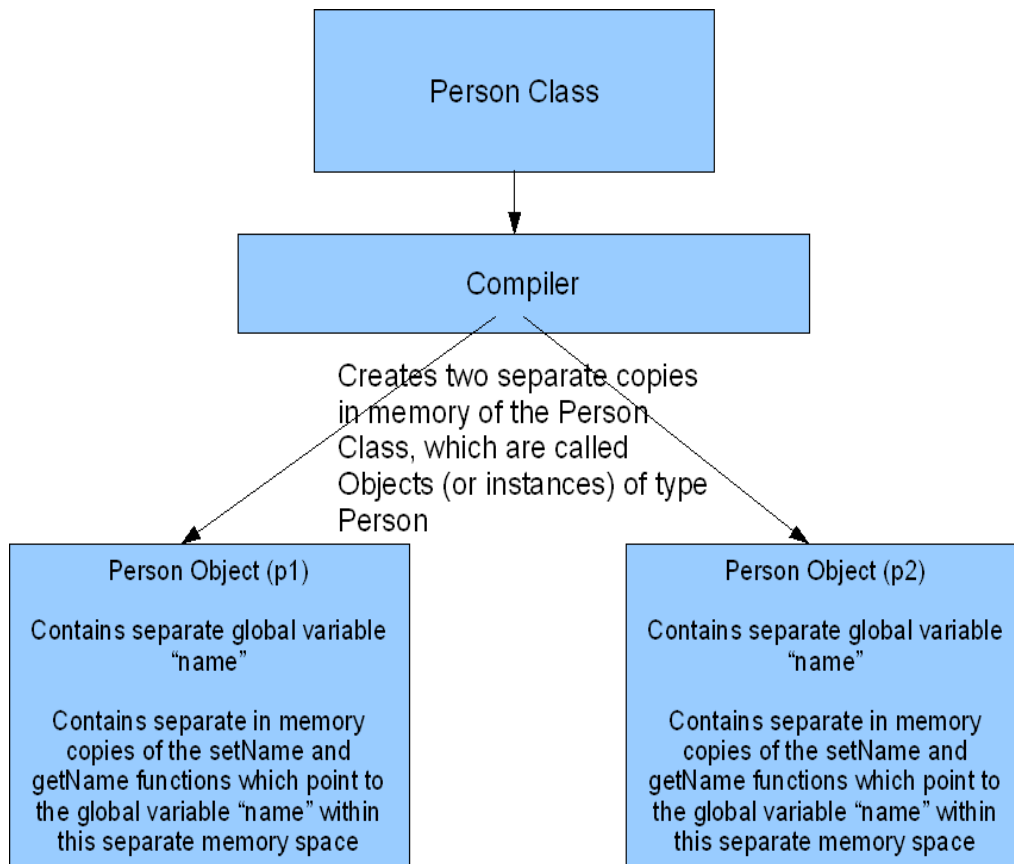
The keyword "new" is used for the compiler to create a "new" separate copy of the Person in memory.

The next two lines are "**p1.setName("Robert");**" and "**p2.setName("Paula");**". Obviously this is a call to the function "setName". But NOTICE that the function is prefixed by "p1." in the first call to setName and "p2." in the second call to setName.

As we saw when we had HelloWorldTester call "sayHello" or "printHello", which was prefixed, with "HelloWorldService", the prefix of the actual variable names tells the compiler to which in memory copy of the Person class to go to to invoke the code for function "setName."

When the Tester class calls setName on the variable p1, what it is actually doing is going to the separate memory space for the p1 instance of the person Object and setting the name "Robert" into the separate memory space for the name of the Person for p1. The same thing goes for p2, but obviously it is in p2's own memory space.

Let's see this on a diagram:



The next two lines `"System.out.println(p1.getName());"` and `"System.out.println(p2.getName());"` basically just output the name of the two separate Person objects to the screen.

The first one `System.out.println(p1.getName());` is just a call to the `getName` function within the `p1` Object's memory space, so that the name "Robert" is returned, and passed to the built in Java print function "System.out.println", because if you recall, we called `p1.setName("Robert")`.

And the second called, is the same thing but only on the Object `p2`, so it's a call to this function with `p2`'s separate memory space.

### The Java Class verses The Java Interface -

By now, you should already be familiar with a Class in Java. There's another programming structure in Java which is related to classes, but are entirely difference entities from a Class.

You should know already know that to create a class in Java you do the following: 1. Create a new text file named: `[CLASS_NAME].java`. 2. Within the text file type declare the class with the same name as the

file minus the ".java": public class CLASS\_NAME.

An Interface is declared very similarly:

Steps to creating an Interface:

1. Created a new text file named: [INTERFACE\_NAME].java  
+ The same naming rules apply to Interfaces as with classes.
2. In the text file you declare the Interface like the below:

```
public interface [INTERFACE_NAME]
{

}
```

**Note:** Again, just as with classes, the name of the interface itself MUST match the name of the .java file exactly including case.

### **So what is an interface used for and how are they different from classes?**

An Interface is like a "Header File" from other languages like C or C++ if you are familiar with those. It does NOT contain any implementations of functions. It only contains the signatures of functions.

Why use interfaces? Well, think of them as contracts between the creator of the interface and the user of a class which implements that interface.

Classes can implement multiple interfaces. If a class declares that it implements an interface, the Compiler will stop and report an error if the class that declared that it implements an interface did not implement ALL the functions declared in the interface. As mentioned above, it is a contract!

For a class to declare that it implements an interface it is quite simple:

Say we create an interface:

```
public interface Iexample
{
    public void runExample(int num, String name);
}
```

**Important:** Notice how we only "declared" the method "runExample"

instead of actually implementing it. You CAN NOT implement a method inside an interface. Remember it is just a contract or list of methods which implementing classes must implement.

Notice that the public simply ends with a semicolon ";" instead of the set of curly braces. This is how the compiler know it's a declaration and NOT an implementation.

Now, let's create a class which implements this interface called "Iexample":

```
public class Example implements Iexample
{
    public Example(){}

    public void runExample(int num, String name)
    {
        //Do something...
    }
}
```

Let's take a look more closely at the elements that are highlighted in bold above.

First, notice that in that class declaration line: "public class Example **implements Iexample**" there's a new keyword "implements" and then the interface name "Iexample". This is how you declare that this class implements some interface. In this case we say that Class Example implements Iexample. Just as it is written.

Next notice that we implemented the function that was declared in the interface "runExample".

If we did not implement this, since we declared that the Example class implements the Iexample interface, the compiler would tell us that some methods from Iexample are missing in Example.

If we wanted our class "Example" to implement another interface along with the "Iexample" interface, all we need to do is put a comma after the interface name and then add another Interface name to the list. In Java a class can implement N-number of interfaces.

Syntax Example for implementing multiple interfaces in a single class:

```
public class MyClass implements Interface1, Interface2, ...,
Interface(N)
```

Then you just have to implement ALL methods declared in ALL interfaces in your list.

It is important to know about Interfaces and how they are different from Classes, because as we will see when we speak about other topics in this guide, sometimes the Standard Java Library will give us an interface only and expect us to implement it. Or Java first gives us Interfaces and then the implementations so that we can easily work with these objects in groups based on their Interfaces. We will see this in particular when we talk about Collections in an upcoming lesson.

We will learn more about Interfaces in a probable future lesson on Inheritance, Abstract Classes, and Interfaces.

### **Compiling and Running the Lesson Examples:**

#### **How to Compile Lesson using the included Windows Batch Files:**

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson3`
3. `compile.cmd`

#### **How to Compile Lesson using javac.exe program directly:**

Note: This assumes you have the javac.exe program in your PATH.

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson3`
3. `javac *.java`

#### **How to run the examples using the included Windows Batch Files:**

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson3`
3. To run the example from the program's main entry point class (Java Class: PersonTester.java) run: `run_persontester.cmd`

**How to run the examples using the java.exe program directly:**

Note: This assumes you have the java.exe program in your PATH.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson3`
3. To run the example from the program's main entry point class (Java Class: PersonTester.java) run: `java -classpath C:\java_dev\lessons\lesson3 PersonTester`

## Lesson 4 – Built In Java Data Types

There are 8 built in Data Types in Java. They are also know as “**primitive**” data types.

They are: **byte, short, int, long, float, double, boolean, and char.**

The JDK and the Standard Java Library, comes with 100's of Classes. Recall from the previous chapter that a Class is simply a User Defined Structure Type with functions inside it. So these are Data Types as well, but are not considered primitives. Primitives are the only things in Java which are NOT classes, and that's where they get their name from.

In this lesson, besides the **8 primitives**, we will just talk about **String** and **Date** Classes of the Standard Java Library.

To go over the Built In Data Types, we will just proceed with an example as it is the easiest way to show what types of data can be stored in the 8 primitive types. Although, most of them are probably obvious as their names reflect the actual data they can store.

**Special Note on Chars and Strings:** In Java all Characters and therefore Strings are Unicode. Meaning they support international characters natively. Instead of the normal 8-bit (single byte) character with range from ASCII 0 to ASCII 255, Characters in Java are 16-bit (double byte) and go from Unicode 0 to Unicode 65535. This is why in Java they have the “byte” data type, so you can work with Binary Data without reading it two bytes at a time which gets very confusing.

The good thing is that Unicode supports the normal single byte characters for English Letters and Numbers and all other symbols on an English Keyboard. So when you program with Strings, you just do it like normal.

One this I want to show before we go over the example is a chart which shows the maximum values for each Built In Data Type:

### **Integer Types:**

For **byte**, from -128 to 127, inclusive

For **short**, from -32768 to 32767, inclusive

For **int**, from -2147483648 to 2147483647, inclusive

For **long**, from -9223372036854775808 to 9223372036854775807, inclusive

For **char**, from '\u0000' to '\uffff' inclusive, that is, from 0 to 65535

## Boolean Type:

For **boolean** the two possible values are TRUE or FALSE.

**Floats** and **Doubles** are a little more complex so please refer to the following link for details on ALL Java Built In Data Types:  
[http://java.sun.com/docs/books/jls/second\\_edition/html/typesValues.doc.html](http://java.sun.com/docs/books/jls/second_edition/html/typesValues.doc.html)

## Note on "Weird" Data Types and the way Java stores certain things in memory...

When you are starting on Java Programming, the whole Signed Byte thing and the Unicode (double/two byte characters) might make you want to look the other direction. But don't! Just ignore these. You probably will not start working with bytes or have to worry about Unicode for MOST projects you need to work on. Java and Unicode make things easy because ALL characters on the English Keyboard work like normal! So don't worry about these things. But I did want to explain them to the reader, so that you are aware for when you do see references to these things or if you need to work on International Text projects or projects where you need to read Binary Data.

One thing to remember is DO NOT STORE Binary Data within Java Strings or Characters. Java will think you are trying to use International Characters. If you need to work with Binary things, then use Bytes!

Now on to the Example finally! :)

We are taking a look at file:

C:\java\_dev\lessons\lesson4\BuiltInDataTypesTest.java

```
public class BuiltInDataTypesTest
{
    public static void main(String[] args)
    {
        byte bt;
        short sh;
        int i;
        long l;
        float f;
        double d;
        boolean bl;
        char ch;

        //BYTE----->
        bt=(byte)65; //From -128 to 127 (Java uses Signed Bytes,
```

```

        //which is while it's not 0 to 255 as you
        //might expect). You can convert this to
        //normal ASCII/Unsigned by doing a Binary And
        //with HEX value FF: int ascii=bt & 0xff;

System.out.println("Byte: " + bt);
//----->

//SHORT----->
sh=1234; //From -32768 to 32767

System.out.println("Short: " + sh);
//----->

//INT----->
i=5656246; //From -2147483648 to 2147483647

System.out.println("Integer: " + i);
//----->

//LONG----->
l=8000000000L; //From -9223372036854775808 to
               //9223372036854775807

System.out.println("Long: " + l);
//----->

//FLOAT----->
f=3.14F;

System.out.println("Float: " + f);
//----->

//DOUBLE----->
d=135.24542525D;

System.out.println("Double: " + d);
//----->

//BOOLEAN----->
bl=true; //TRUE OR FALSE
        //This might be over stated but for good measure
        //for FALSE obviously use: bl=false;

System.out.println("Boolean: " + bl);
//----->

//CHAR----->
ch='R'; //From Unicode '\u0000' to Unicode '\uffff'

```

```

        //inclusive, that is, from 0 to 65535

        System.out.println("Char: " + ch);
        //----->
    }
}

```

Because this is an example, I first declared ALL primitive or built in data types at the top of the main method:

```

byte bt;
short sh;
int i;
long l;
float f;
double d;
boolean bl;
char ch;

```

Then for each data type I did the exact same thing for each. Let's take a look at the **Int** data type.

The first line for Ints I set an example value:

```

i=5656246; //From -2147483648 to 2147483647

```

The second line for Ints I simply printed it to the screen:

```

System.out.println("Integer: " + i);

```

Each data type's two lines are enclosed within comments that have -----> on the end just to show you where each section starts and ends...

**Note:** In Java you can do this: **int num = 1234567;** You can declare and initialize a variable on the same line. It's probably over kill to mention this, but again, just for good measure! :)

## Strings and Dates -

Java of course includes a String data type as all modern languages do. As I mentioned at the beginning of this lesson, Java includes 100's if not 1000's of included Classes in the Standard Java Library which can be used as data types. You will have to explore the Java Docs to find out about others you might want to use in your programs,

so you don't have to write your own for a lot of common things. But I did want to include in this lesson Java's Date class which is included in the Standard Java Library along with String.

**Note on Imports for the next example:**

We have NOT imported anything as of yet for any other example code in our previous lessons.

Please recall from the first lesson on Packages and Imports that things in the java.lang package do NOT need to be imported but other things from the Standard Java Library do. String is in the java.lang package but Date is in the java.util package. So you will see we will import java.util.\* in the next example but NOT java.lang.

Let's take a look at file:

C:\java\_dev\lessons\lesson4\StandardJavaObjectTypes.java

```
import java.util.*;
```

```
public class StandardJavaObjectTypes
{
```

```
    public static void main(String[] args)
    {
```

```
        String s;
        Date d;
```

```
        //String----->
```

```
s="Hello World!"; //Note we do NOT need to do
                //s=new String("Hello World!");
                //Because String is part of the
                //Java Language and is why it is
                //in the java.lang package.
                //This this would actually work:
                //s=new String("Hello World!");
```

```
System.out.println("String: " + s);
//----->
```

```
        //Date----->
```

```
d=new Date(); //Create a new date object and since we are
              //not passing anything to the constructor it
              //will be initialized to the current time.
              //Note: there is also a Date class in the
              //java.sql package. This one is used for
              //Dates from a Database.
```

```
System.out.println("Date: " + d);
```

```
        //----->
    }
}
```

Notice that the first line of the file is the import statement for the Util Package:

```
import java.util.*;
```

This package contains 100's of useful classes. When we look at Collections we will see that they are all part of the java.util package.

Here we are import this package because the **Date** class is contained within this package.

As explained in lesson 1, we could have imported ONLY the Date class if we wrote the line like this:

```
import java.util.Date;
```

But it is more convenient to just import the entire package with the "\*" notation in place of the class name.

The rest of this example is just like the **BuiltInDataTypesTest** class. We declare the two data types we are working with at the top of the method and then we created a String and a Date and printed them on the screen.

## **Wrapper Objects**

In Java everything except for the 8 primitives are Objects. But it is often convenient to store primitives in thing that take Objects, such as Collections. The reason for this is that the "add" methods of these Collections as we will see in the lesson on Collections look like this: add(Object element). Since ALL classes in Java implicitly extend from the class called "Object" through Polymorphism you can add any object to a collection via these "add" methods. Since primitives are NOT objects they can NOT be stored in collections, so you either have to write a Class with a set and a get to store each separate primitive or use what is called the Wrapper Objects.

(There's a new language feature in JDK 1.5+ called Auto-Boxing and Unboxing which allows you to add primitives to Collection without first inserting the primitive into a Wrapper Object. But we will not focus on this new language feature. If you want to read more about this, please refer to this link:

<http://java.sun.com/j2se/1.5.0/docs/guide/language/autoboxing.html>).

## Wrapper Object to Primitives Mapping Table

Primitive	Wrapper Object	Constructor (For Setting)	Getter (For Accessing)	Comment
int	Integer	Integer(int x)	intValue()	
short	Short	Short(short x)	shortValue()	Notice the difference in the case of "S"
long	Long	Long(long x)	longValue()	Notice the difference in the case of "L"
byte	Byte	Byte(byte x)	byteValue()	Notice the difference in the case of "B"
char	Character	Character(char x)	charValue()	
float	Float	Float(float x)	floatValue()	Notice the difference in the case of "F"
double	Double	Double(double x)	doubleValue()	Notice the difference in the case of "D"
boolean	Boolean	Boolean(boolean x)	booleanValue()	Notice the difference in the case of "B"

**Reminder on Case:** Java is **CASE SENSITIVE**, so there's a difference between for example "boolean" and "Boolean".

### How to use wrapper Objects:

Say we want to store an int variable inside it's corresponding "Integer" wrapper object. It's pretty easy to do. All you need to do is pass the variable into the constructor of the Integer object.

Let's see an example:

```
//Declare and initialize an "int" variable.  
int x = 100;
```

```
//Store x in an Integer Object:
```

```
Integer num = new Integer(x);
```

```
-----
```

Now, the value of 100 is stored within the Integer object "num".

How to access the value?

```
int x2 = num.intValue();
```

The same goes for ALL other primitives and their corresponding wrapper objects. There's always a method that follows this format to access the actual value stored in the Wrapper Object:  
[WrapperObject].[primitiveName]Value().

Refer to the table above on Primitive and their Wrapper Objects for a reference on the constructors and accessing methods.

We will see in the lesson on Collections examples where we are using the Wrapper Objects so we can store primitives in collections!

### **Type Casting:**

In Java like in many other modern languages, you can "convert" from one data type to another using Type Casting. This usually only works if the two types have some type of relationship. In terms of Objects, they two Objects MUST have a Parent-Child Relationship in terms of inheritance. In terms of the primitives, the smaller types can be converted into a Larger type. For example an int can be converted into a long.

Let's see some examples:

```
int i=1024;  
long l=(long)i;
```

The part of the code above highlighted in bold is the type cast.

You basically just put the type you want as output inside a set of parenthesis on the right hand side of an equal sign right before the variable or value of the type you want to convert from.

It follows this syntax:

```
[FromDataType] inVariable = //some valid value;  
[ToDataType] outVariable = ([ToDataType])inVariable;
```

We will explore more Type Casting with Objects in the probable future lesson on Inheritance. We will also see some type casting with Objects in the lesson on Collections in this guide.

### **Arrays:**

Any data type including user defined classes can be used in Arrays.

Let's see how to define an array in Java:

The syntax is: `DATA_TYPE[] variableName;`

Notice the set of square brackets in bold after the words "DATA\_TYPE".

This is how you write an array declaration in Java.

Let's see some examples:

```
int[] numArr;  
String[] strArr;
```

The above two lines are variable declarations. The first line declares a "numArr" variable which is an Array of ints. The second line declares a variable named "strArr" which is an Array of Strings.

But as we already know in Java, you need to allocate the memory for everything except for the built in primitives. This rule applies to Arrays of primitives as well. You need to allocate the memory for the int array as well as for the String array.

Let's see how to do this:

```
int[] numArr;  
  
numArr = new int[5];
```

Once again we are using the keyword "new" which if you recalls, tell the compiler we want to allocate memory for some variable.

In this case we are telling the compiler we need to allocate memory for an Array of 5 ints.

The syntax for allocating an array is as follows:

```
//First declare the variable:
```

```
DATA_TYPE[] var;
```

```
//Second allocate the memory:
```

```
var = new DATA_TYPE[ARR_SIZE];
```

```
//Where ARR_SIZE is a whole number from 0 to MAX INT (2,147,483,647).
```

```
//Note: ARR_SIZE can be an int variable, if you want to dynamically
```

```
//allocate the size of the array at run time.
```

### **How to access array elements?**

To access an array Element you simple write the variable using this syntax:

```
//For assigning to the array
```

```
ARRAY_VARIABLE[INDEX] = VALUE
```

```
//For getting the value out of the array:
```

```
DATA_TYPE var = ARRAY_VARIABLE[INDEX];
```

Note: DATA\_TYPE must be the same data type of the ARRAY\_VARIABLE.

Example:

```
int[] nums = new int[21];
```

```
nums[5] = 1234;
```

```
int num = nums[5];
```

```
System.out.println(num); //Would print 1234
```

### **Important:**

When you allocate memory for an array of any data type that's an Object, like Strings or user defined classes. The default value is NULL. So if you traverse the array before you initialize all values, each element would be NULL. For build in data type arrays, like ints, etc, the value would be the default for that data type which is normally ZERO.

Let's first allocate an array to work with:

```
//Allocate a String Array of size 10 elements
String[] sArr = new String[10];

//The valid indexes of the array for size 10 is:
// 0 to 9

//More generically the valid indexes of ANY array is:
// 0 to ARRAY_SIZE - 1
```

To access the second element in the array "sArr" from above:

```
sArr[1] = "Hello"
```

Notice the part in bold in the line above has index "1" which since array elements start at ZERO, 1 is actually the second element in the array.

### **Determining the size of an array at runtime:**

Most modern languages provide some mechanism usually a function built in to the language's standard library. In Java each Array variable has a "property" called **length**. You can think of this property as a PUBLIC int data memory. The kind that they spoke about in the lesson on Object Oriented Programming and said never to do something like this. But since Array's are built into the language it is ok that Sun did this.

Example

```
//Declare and allocate a String array of 55 elements.
String[] sArr = new String[55];

int arrayLength = sArr.length;
//Notice the ".length" immediately after the sArr String array
//variable. This is how you access the "length" or size of the array
//at runtime. You can use this length for many useful things
//such as traversing an array.

System.out.println(arrayLength); //Will print 55.
```

### **Notice on jumping ahead to the lesson on Loops:**

To work with arrays in our example on Arrays, we are going to use for loops to traverse the arrays. Here's a quick look at the syntax of for loops, but if you feel more comfortable, skip the example on Arrays until you read the lesson on Loops and Logic. Then come back to this example.

### **For Loop Syntax:**

```
for ([loop variable initialization] ; [boolean logic statement for
ending the loop] ; [loop variable increment statement])
{
    //Code block within the for loop
}
```

Example:

```
for (int i = 0; i < 10; i++)
{
    //This will print Loop Iteration 1 to 10 each on its own line.
    System.out.println("Loop Iteration "+(i+1));
}
```

**NOTE:** As mentioned above, the lesson on Loops has a complete overview on For Loops. The above syntax and example are provided just so you can read the example on Arrays.

Let's take a look at an example on Arrays in Java. Please refer to file: c:\java\_dev\lessons\lesson4\ArrayTest.java

```
public class ArrayTest
{
    public static void main(String[] args)
    {
        //Declare an int array variable called "nums"
        int[] nums;

        //Allocate memory for an array of ints of size 100 elements
        nums = new int[100];

        //Initialize the int array with data
        //See the lesson on loops for details on For Loops for now
    }
}
```

```

//just know that this loop will go
//from 0 to nums.length - 1 (99)

//The i<nums.length will make this loop
//go from ZERO to the size of the array -1
//since we said i < (less than) the length...
//We are basically TRAVERSING the array here.
for (int i=0; i<nums.length; i++)
{
    //Set array element at index "i"
    //equal to some "random number" just for th test...
    //Note: Math.random is a built in library function.
    //You can look this up on the Java Docs.
    //Look for the class: Math.
    //Here's a direct link:
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html#random\(\)

    nums[i] = (int)(Math.random()*100);
}

//Traverse the array and print out the numbers
for (int i=0; i<nums.length; i++)
{
    System.out.println("Array Element " + i + " = " +
nums[i]);
}
}
}

```

-----

### **Compiling and Running the Lesson Examples:**

#### **How to Compile Lesson using the included Windows Batch Files:**

1. Start a DOS Prompt. And from the dos prompt run:
2. cd\java\_dev\lessons\lesson4
3. compile.cmd

#### **How to Compile Lesson using javac.exe program directly:**

Note: This assumes you have the javac.exe program in your PATH.

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson4`
3. `javac *.java`

**How to run the examples using the included Windows Batch Files:**

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson4`
3. To run the Built In Data Types example from the program's main entry point class (Java Class: `BuiltInDataTypesTest.java`) run: `run_builtintester.cmd`
4. To run the Standard Java Object Types example from the program's main entry point class (Java Class: `StandardJavaObjectTypes.java`) run: `run_stdobjectstester.cmd`
5. To run the Arrays example from the program's main entry point class (Java Class: `ArrayTest.java`) run: `run_arraytester.cmd`

**How to run the examples using the java.exe program directly:**

Note: This assumes you have the java.exe program in your PATH.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson4`
3. To run the Built In Data Types example from the program's main entry point class (Java Class: `BuiltInDataTypesTest.java`) run: `java -classpath C:\java_dev\lessons\lesson4 BuiltInDataTypesTest`
4. To run the Standard Java Object Types example from the program's main entry point class (Java Class: `StandardJavaObjectTypes.java`) run: `java -classpath C:\java_dev\lessons\lesson4 StandardJavaObjectTypes`
5. To run the Arrays example from the program's main entry point class (Java Class: `ArrayTest.java`) run: `java -classpath C:\java_dev\lessons\lesson4 ArrayTest`

## Lesson 5 – Operators, Loops and Logic Statements (Control Statements)

This is going to be a quick lesson to learn, as most Operators, Loops and Logic statements are common within all modern programming languages. This lesson is simply to provide a quick reference for experienced programmers to become familiar with the Java language syntax for these programming structures.

Besides this lesson you can read the official Sun documentation on Built In Operators at the following web page:  
<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html>

### Operators -

Note: The built in operators can only be applied to Primitives. For Objects, you need to call methods on those objects to perform an operation.

For example: You can do something like this with an "int":

```
int x = 10;
```

```
x = x + 1;
```

But you Can't use the "+" on an Object type:

If you wanted to do something like:

```
public class Foo
{
    private int x;

    public Foo() {}

    public void setX(int myX)
    {
        x = myX;
    }

    public int getX()
    {
        return x;
    }
}
```

You can't do this:

```
Foo f = new Foo();
```

```
f.setX(10);
```

```
f = f + 1; //This is an error that the compiler will complain  
about...
```

-----

Think about it... How would the compiler know what part of Object "f" to add 1 to?

You would have to modify the Foo class to have some method to add to x.

Like this:

```
public class Foo  
{  
    private int x;  
  
    public Foo() {}  
  
    public void setX(int myX)  
    {  
        x = myX;  
    }  
  
    public int getX()  
    {  
        return x;  
    }  
  
    public void addToX(int val)  
    {  
        x = x + val;  
    }  
}
```

Then you can do this:

```
Foo f = new Foo();
```

```
f.setX(10);
```

```
f.addToX(1);
```

-----

### Assignment Operator: =

The assignment operator "=" can be used on ALL primitives to set a value into it. It can also be used on Object variables, but instead of setting a value, it sets a reference or a "pointer" to the Object in memory.

### Arithmetic Operators:

These can be used on ALL INTEGER PRIMITIVES (int, short, long, char, byte) and ALL FLOATING POINT PRIMITIVES (float, double). But as we saw in the explanation above you CAN NOT use operators on Objects. The exception is "Strings" since it is part of the Language. The "+" operator does work with Strings for concatenation, but actually is NOT the "preferred" way to concatenate strings in Java. The preferred way is through a Class included in the Java Standard Library called "StringBuffer" but for now just use "+" for concatenation until you are comfortable with classes within the Java Standard Library and the StringBuffer class itself.

- + - Addition operator (also used for String concatenation)
- - Subtraction operator
- \* - Multiplication operator
- / - Division operator
- % - Remainder operator (Modulus)

Usage Table:

Operator	Description	Example	Output
+	Addition	int x = 10; int sum;  sum = x + 10;	System.out.println(sum); Would print: 20
-	Subtraction	int x = 20; int difference;  difference = x - 10;	System.out.println(difference); Would print: 10
*	Multiplication	int x = 10; int product;  product = x * 10;	System.out.println(product); Would print: 100
/	Division	int x = 10; int quotient;  quotient = x / 5;	System.out.println(quotient); Would print: 2

%	Modulus / Remainder	<pre>int x = 10; int remainder;  remainder = x % 7;</pre>	<pre>System.out.println(remainder); Would print: 3</pre>
---	---------------------	---	--

**The Increment Operators: ++ and +=**

The Increment Operators work for ALL **INTEGER AND FLOATING POINT** PRIMITIVE DATA TYPES.

The Increment Operators can be thought as of short cuts for this:

```
x = x + 1;
```

Can be written as:

```
x++; //This means increment the value of variable x by 1.
```

You can also write this as:

```
x+=1;
```

The nice thing about += you do NOT have to use "1" as the value to increment by. This is especially useful in loops if you want to increment the counter by a factor greater than 1.

So things like work fine:

```
x+=2; //Adds 2 to the value of variable x.
```

```
x+=100; //Adds 100 to the value of variable x.
```

**The Decrement Operators: -- and -=**

The Decrement Operators work for ALL **INTEGER AND FLOATING POINT** PRIMITIVE DATA TYPES.

The Decrement Operators can be thought as of short cuts for this:

```
x = x - 1;
```

Can be written as:

```
x--; //This means decrement the value of variable x by 1.
```

You can also write this as:

```
x-=1;
```

The nice thing about -= you do NOT have to use "1" as the value to decrement by. This is especially useful in loops if you want to decrement the counter by a factor greater than 1.

So things like work fine:

```
x-=2; //Subtracts 2 to the value of variable x.
```

```
x-=100; //Subtracts 100 to the value of variable x.
```

### **Note about x++ or ++x?**

When used in a for loop one of the most common places to see the ++ or - operators, there is no difference by the code below shows that there is a difference:

```
int x=10;  
int y;
```

```
y = x++;
```

```
System.out.println(y); //Prints 10  
System.out.println(x); //Prints 11
```

Why is this?

x++ actually means, return x first then increment.

If we wrote it like this:

```
int x=10;  
int y;
```

```
y = ++x;
```

```
System.out.println(y); //Prints 11  
System.out.println(x); //Prints 11
```

Which is probably what you expected the first example would have done.

The same rules apply to the -- operator as well.

Most times you are just incrementing a value without the assignment statement on the left, so you normally don't have to worry too much

about this rule, but I wanted you to see it for yourself in case you read someone else's code and they do some weird stuff like this...

In general you will see `x++` on it's own line like the below:

```
int x=10;
int y;

x++;

y=x;

System.out.println(y); //Prints 11
System.out.println(x); //Prints 11
```

But some programmers try to use "advanced" language short cuts (and as you know from other programming languages, these can be confusing and should be avoided in *most* of situation when you write code others need to read and work with. Don't try to show off that you know about the short cuts!) and combine the "`x++`" and "`y=x`" lines into one line.

In my opinion the more straight forward short cuts like `x++` on it's own line should be used instead of "`x=x+1`" but don't make it more complicated than that.

See the MathOperators example below to actually see the three examples above run...

For a runnable code example, see file  
C:\java\_dev\lessons\lesson5\MathOperators.java

Each Mathematical operator's test code section is separated by a comment line: "`//----->`"

```
public class MathOperators
{
    public static void main(String[] args)
    {
        int x;
        int sum;

        x = 10;
        sum = x + 10;

        System.out.println("The Sum of x + 10 (where x is 10) is "
+ sum);
```

```
//----->

x = 20;
int difference;

difference = x - 10;

System.out.println("The Difference of x - 10 (where x is
20) is " + difference);

//----->

x = 10;
int product;

product = x * 10;

System.out.println("The Product of x * 10 (where x is 10)
is " + product);

//----->

x = 10;
int quotient;

quotient = x / 5;

System.out.println("The quotient of x / 5 (where x is 10)
is " + quotient);

//----->

x = 10;
int remainder;

remainder = x % 7;

System.out.println("The remainder of x % 7 (where x is 10)
is " + remainder);

//----->

x = 10;
int y;

y = x++;

System.out.println("y after y = x++ : "+y); //Prints 10
because we used x++ instead of ++x with an assignment on the same
```

```

line
    System.out.println("x after y = x++ : "+x); //Prints 11
    //----->
    x = 10;
    y = ++x;
    System.out.println("y after y = ++x : "+y); //Prints 11
because we used ++x instead of x++ with an assignment on the same
line
    System.out.println("x after y = ++x : "+x); //Prints 11
    //----->
    x = 10;
    x++;
    y=x; //Because we separated the assignment from the
increment, it doesn't matter if the above line was x++ or ++x
    System.out.println("y after x++ and y=x on separate lines :
"+y); //Prints 11
    System.out.println("x after x++ and y=x on separate lines :
"+x); //Prints 11
    //----->
}
}

```

### **Equality and other Test Operators:**

You SHOULD NOT generally use these operators on Objects, they should only be used on PRIMITIVES. If you use these to test Objects, it will actually work on the memory location as a value for comparison. Which in general is NOT very useful.

```

== - Equal to
!= - Not equal to
> - Greater than
>= - Greater than or equal
< - Less than
<= - Less than or equal

```

The operators will return TRUE OR FALSE. You can actually set this value into a boolean or use it directly within logic statements or

the boolean control statement of a loop.

### **Note on "Equal to" (==) versus "Assignment" (=):**

There's a big difference between "==" (the equal to) and "=" (the assignment) operators. Do not use them interchangeably because they are NOT the same! The "double equal" as normally referred to by Java programmers is for testing equality and the single equal sign is for variable assignment (or in other words "setting" a value into a variable).

You will normally NOT use the single equal in a test statement like an if statement. Instead here, you would use the double equal.

**Code Example Note:** We will not take a look at the "Equality and other Test Operators" until we take a look at Logic statements first. As they are most useful in these cases.

### **Boolean Operators:**

These are used to create composite equality and test statements.

Example in Pseudo Code:

```
if a == b AND c == d then do_something
```

The **AND** is the operator we are talking about here.

There's also **OR** as well.

### **Boolean Operator List:**

```
&&   - AND  
||   - OR
```

Composite Boolean Statements can be used as the logic conditions for ALL Types of Loops and If statements in Java.

Please see the examples on If statements for code samples using Composite Boolean Statements. We will only use simple logic conditions in the examples on loops as the samples are there to illustrate the loop statements themselves.

**Note on Code Blocks within sets curly braces and single lines of code immediately after a If, For, or While:**

This applies to the next two sections of this lesson on Loops and If Statements:

In Java, it is preferred that you always use the sets of curly braces to show that a code is within a branch belonging to an If Statement or Loop. But if you only have a single line of code that needs to be executed you can forgo the set of curly braces and just write the single line of code immediately after the if statement or for or while loop.

If you see a line of code after an if, for, or while, without the curly braces, then ONLY that SINGLE line of code is within the scope of the if, for, or while.

Basically, to include multiple lines of code within a loop or if statement, you need to use the curly braces!

We will see examples of these in the next sections. I just wanted to mention this so that if you are reading other programmers code and you see code written as described above, you will not be surprised by it and know how to interpret it.

**Loops -**

**For Loops:**

Keyword: **for**

Syntax:

```
for ([loop variable initialization] ; [boolean logic statement for  
ending the loop] ; [loop variable increment statement])  
{  
    //Code block within the for loop  
}
```

Example (The loop below will execute 10 iterations, with "i" starting from 0 and ending with i = 9. Actually "i" ends being 10 but the loop does not execute so therefore the last value of "i" that can be seeing in the example below would have been 9):

```
for (int i = 0; i < 10; i++)  
{  
    //This will print Loop Iteration 1 to 10 each on its own line.
```

```
        System.out.println("Loop Iteration "+(i+1));
    }
```

Let's take a part the for loop line itself: "for (**int i = 0; i < 10; i++**)". In particular, we will be looking at the three parts within the parenthesis.

The first part "**int i = 0;**" is the loop variable initialization. The int variable "i" in this case will ONLY exist within the loop block of code itself. This is the scope of its lifetime in memory. Once code execution picks up after the closing curly brace of the for loop "i" no longer exists and can NOT be referenced.

If we did want it to be available both before and after the for loop executes, we could rewrite the for loop like this:

```
for (int i = 0; i < 10; i++)
```

would become:

```
int i;
```

```
//We can work with "i" out here.
```

```
//Once we hit the for loop statement for the
//first time, "i" will be set to 0 as we can
//see because of the loop variable initialization
//part of the for loop parameters within the parenthesis
//highlighted in bold:
```

```
for (i = 0; i < 10; i++)
{
    //i in here will become
    //values 0 to 9
}
```

```
//i out here will be 10 as soon as the loop is completed.
```

-----

The next part of the for loop statement is "**i < 10;**". This is the test which at execution when returns TRUE, will cause the for loop to exit.

In this case the test of  $i < 10$ , will cause the loop to execution from  $i = 0$  to 9 or 10 total iterations. The value of 10 will be reached but will NOT be seen from inside the for loop itself as

previously explained.

The final part of the for loop statement is `"i++"`. Notice there is NO `;"` at the end of this part. This is according to the syntax of for loops, so DO NOT add one...

This statement makes use of the shortcut for incrementing the int variable.

We could have rewrote this the long way: `"i = i + 1"` if we wanted to.

**Important:** This simply increments the loop variable by 1 **AT THE END** of each iteration of the loop.

### **Loop Control Statements:**

Keywords: break, continue

If you specify the keyword **"break"** with a semicolon at the end of it anywhere within the block of code within a Loop (for, while, etc), the loop execution will be immediately stopped and the next line of execution would be the first line immediately after the closing curly brace of the loop body.

If you specify the keyword **"continue"** with a semicolon at the end of it anywhere within the block of code within a Loop (for, while, etc), the loop execution will immediately jump back to the "top" of the loop body. If it's a for loop the loop variable increment would run and then the loop condition logic would be executed, then if the condition evaluates to true, the loop code would execute using the NEXT loop iteration. Otherwise the loop would exit. If it's a while loop, the same rules apply, but there is not loop variable to increment, so just the loop condition logic is executed.

Break is more common than Continue in most people's code you will find, but I wanted to mention it so in case you see it, you know what it's all about.

In the loop examples you will see an example block of code for the "break" statement only. I encourage you to write your own loop tests and experiment with the "continue" keyword.

Let's take a look at the example Java source file:  
C:\java\_dev\lessons\lesson5\ForLoops.java

```

public class ForLoops
{
    public static void main(String[] args)
    {
        //Basic for loop from 0 to 9
        for (int i=0; i < 10; i++)
        {
            System.out.println("i (inside loop) = "+i);
        }

        //If we try to reference i out here,
        //we will get a compiler error.
        //Try uncommenting the next line and recompiling if you
want to see the error for yourself:
        //System.out.println(" i (outside loop) = "+i);

        //----->
        System.out.println(); //Just printing a blank
                               //line to separate the output...

        //For loop with loop variable j
        //declared outside of the for loop statetment itself:

        int j = 99999; //I'm just setting j to an arbitrary
                       //value to show that once
                       //we enter the loop, the variable
                       //initialization part takes over.
                       //And sets the first value of
                       //j within the loop.

        System.out.println("j before for loop = "+j);

        for (j=0; j < 10 ; j++)
        {
            System.out.println("j (inside loop) = "+j);
        }

        //Just to show that j actually reaches 10, but the loop
        //does NOT execute on j=10 because it FAILS
        //the test "j < 10"
        System.out.println("j immediately after the for loop =
"+j);

        //We can continue to work with j if we want...
        //We can keep the value set to 10 if it's important to us
        //or reuse the variable for something else completely.

```

```

//----->
//Just printing a blank line to separate the output.
System.out.println();

//For loop with a break statement to
//"preempt" the loop execution.

for (int i=0; i < 10 ; i++)
{
    System.out.println("i (inside loop) = "+i);

    if (i==5)
    {
        System.out.println("Stopping the loop at i =
"+i);
        break; //This will cause the loop to stop and
//JUMP to the first line immediately
//after the loop...
    }
}

System.out.println("We are now out of the for loop...");
}
}

```

---

### **While Loops:**

Keyword: while

Syntax:

```

while ([boolean logic statement for ending the loop])
{
    //Code block within the while loop
}

```

### **Do While Loops:**

Keywords: do, while

Syntax:

**do**

```

{
    //Code block within the do while loop
} while ([boolean logic statement for ending the loop]);

```

The difference between a "while" loop and a "do while" loop is that: For the while loop to enter into the code block of the loop, the condition in the parenthesis must evaluate to true. Therefore the code may never run. Unlike the do while loop, which since the logic condition within the parenthesis of the while statement comes at the end, you will ALWAYS execute the do while loop code block at least once.

Let's take a look at the example in file:  
C:\java\_dev\lessons\lesson5\WhileLoops.java

```

public class WhileLoops
{
    public static void main(String[] args)
    {
        //While loop to count from 1 to 10
        int cnt=1;

        while (cnt<=10)
        {
            //Print out the loop iteration based
            //on the "cnt" int variable.
            System.out.println("cnt = "+cnt);

            cnt++; //Increment the counter variable by 1
        }

        //----->
        //Print blank line to separate the output...
        System.out.println();

        //Do While Loop to count down from 10 to 1
        cnt=10;

        do
        {
            System.out.println("cnt = "+cnt);
            cnt--; //Decrement the counter variable by 1
        } while (cnt>0);

        //----->

```

```

//----->
//Print blank line to separate the output...
System.out.println();

//While loop to count from 1 to 10, but will break at 5...

cnt=1;

while (cnt<=10)
{
    System.out.println("cnt = "+cnt);

    if (cnt==5)
    {
        System.out.println("Stopping the loop at cnt =
"+cnt);

        break; //This will cause the loop to stop and
                //JUMP to the first line immediately
                //after the loop...
    }

    cnt++; //Increment the counter variable by 1
}

System.out.println("We are now out of the for loop...");

//----->
}
}

```

## **If Statements:**

Keywords: if, else if, else

If statements are made up of various combinations of if's, else if's, and else's.

## **Variants of If Statements -**

### **Single If Block:**

Syntax:

**if** ([logic condition])

```
{
    //Block of code
    //Executes if logic condition of if statement
    //Evaluates to true.
}
```

#### **Single If Block with Else Block:**

```
if ([logic condition])
{
    //Block of code
    //Executes if logic condition of if statement
    //Evaluates to TRUE.
}
else
{
    //Block of code
    //Executes if the logic condition of the
    //if statement evaluates to FALSE.
}
```

#### **If Block with N-number of Else If's and Else at the end:**

```
if ([logic condition])
{
    //Block of code
    //Executes if logic condition of if statement
    //Evaluates to TRUE.
}
else if ([logic condition])
{
    //Block of code
    //Executes else if logic condition of if statement
    //Evaluates to TRUE.

    //NOTE: You can put multiple else if blocks
    //between the "if" and the "else"
    //Each with their own logic condition.

    //Important: ONLY one If or Else If will execute!
    //Even if multiple logic conditions are true,
    //Only the first true from the top will have
    //its block of code executed!
}
else
{
    //Block of code
```

```

    //Executes if the logic condition of the if
    //statement AND ALL Else If's logic conditions
    //ALL evaluated to FALSE.
}

```

**Note:** You actually can leave off the "else" block and just end with an else if block if you want. This basically is for "else" conditions that should not execute any code. Instead of writing the Else with an empty set of open and closed curly braces next to it { }, you can just leave off the else block.

Let's take a look at the example code in file:  
 C:\java\_dev\lessons\lesson5\IfStatements.java

```

public class IfStatements
{
    public static void main(String[] args)
    {
        int x, y;

        //Test code to show how a basic if statement works.
        //Since X is greater than 90 the logic condition of
        //the if statement evaulates to true and therefore
        //the code block within the curly braces of the
        //if statement executes.

        x=100;

        if (x>=90)
        {
            System.out.println("X is greater than or equal to
90!!!");
        }

        //----->
        System.out.println();

        //Test code to show how composite logic statements look and
work.

        x=100;
        y=10;

        if (x>=90 && y<=50)
        {

```

```

        System.out.println("X is greater than or equal to
90!!! AND Y is less than or equal to 50!!!");
    }

//----->
System.out.println();

//Test code to show how else blocks work...
//Since X is less than 40, the IF block will not execute
//because the logic condition of the if
//statement evaluates to FALSE
//therefore the ELSE block runs...

x=30;

if (x>40)
{
    System.out.println("X is greater than 40!!!");
}
else
{
    System.out.println("X is NOT greater than 40!!!");
}

//----->
System.out.println();

//Test code to show how else if's work...
//The second else if block will execute in this case
//as x is equal to 60 and the logic statement of the second
//else if says "Is X equal to 50 OR Is X equal to 60"

x=60;

if (x>=10 && x<=15)
{
    System.out.println("X is between 10 and 15");
}
else if (x>=20 && x<=30)
{
    System.out.println("X is between 20 and 30");
}
else if (x==50 || x==60)
{
    System.out.println("X is equal to 50 or 60...");
}
else

```

```

    {
        System.out.println("I'm in the ELSE Block...");
    }

//----->
System.out.println();

//Test code to show single statements
//within the scope of if statements
//without the corresponding set of curly braces.

x=10;

if (x==9)
    System.out.println("X equals 9"); //This will NOT
execute...

//At this point, we are NO longer within the scope
//of the above if statement as there is no curly brace
//and there is a single line of code
//the "System.out.println" after the if statement...

if (x==11)
    System.out.println("X equals 11");
else
    System.out.println("I guess X is not equal to 11
because I am the else and I'm executing...");

//Again, at this point, this line is no longer
//in the scope of the else block
//as the 'System.out.println("I guess X...'
//is immediately after the else without curly braces.
}
}

```

-----

### **Compiling and Running the Lesson Examples:**

#### **How to Compile Lesson using the included Windows Batch Files:**

1. Start a DOS Prompt. And from the dos prompt run:
2. cd\java\_dev\lessons\lesson5

3. compile.cmd

**How to Compile Lesson using javac.exe program directly:**

Note: This assumes you have the javac.exe program in your PATH.

1. Start a DOS Prompt. And from the dos prompt run:
2. cd\java\_dev\lessons\lesson5
3. javac \*.java

**How to run the examples using the included Windows Batch Files:**

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: cd\java\_dev\lessons\lesson5
3. To run the Map Operators examples from the program's main entry point class (Java Class: MathOperators.java) run: run\_mathstatements.cmd
4. To run the For Loops example from the program's main entry point class (Java Class: ForLoops.java) run: run\_forloops.cmd
5. To run the While Loops example from the program's main entry point class (Java Class: WhileLoops.java) run: run\_whileloops.cmd
6. To run the If Statements example from the program's main entry point class (Java Class: IfStatements.java) run: run\_ifstatements.cmd

**How to run the examples using the java.exe program directly:**

Note: This assumes you have the java.exe program in your PATH.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: cd\java\_dev\lessons\lesson5
3. To run the Math Operators examples from the program's main entry point class (Java Class: MathOperators.java) run: java -classpath C:\java\_dev\lessons\lesson5 MathOperators

4. To run the For Loops example from the program's main entry point class (Java Class: ForLoops.java) run: `java -classpath C:\java_dev\lessons\lesson5 ForLoops`

5. To run the While Loops example from the program's main entry point class (Java Class: WhileLoops.java) run: `java -classpath C:\java_dev\lessons\lesson5 WhileLoops`

6. To run the If Statements example from the program's main entry point class (Java Class: IfStatements.java) run: `java -classpath C:\java_dev\lessons\lesson5 IfStatements`

## Lesson 6 – Collections

### What are collections?

Collections are Data Structures like Vectors, HashTables, etc. Collections are classes that implement these classic data structures are included in the Standard Java Library for your convenience, so that you do not have to implement them if you need them and in most cases, using them makes your programs easier to implement, cleaner when other people read your code, smaller since you do not have to write the data structures, more stable and more bug free, and more efficient, since the programmers of Java made sure to use the most efficient implementations of these classic data structures.

They are called Collections, because they are data structures whose sole purpose is to store other things. So in essence they are literally collections of objects.

### Types of Collections:

There are three main groups of collections in Java. They are: **Lists**, **Maps**, and **Sets**. In this lesson we will explore some of the common implementations within the **Lists** and **Maps** groups. Actually each one, **List**, **Map**, **Set**, are actual names of Set Interfaces declared in Java.

**Note on Generics and changes to Collections in Java 1.5+:** This guide will focus on using Collections WITHOUT GENERICS. Please read on Generics here:

<http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html>

Generics are Java's version of Templates. Basically they allow you to Strong Type the type of Objects allowed to be stored in an Instance of a Collection.

Since this is an optional feature even in Java 1.5+, we will ignore it for now. In general Collections store type "Object", the universal object is which the parent of ALL objects in Java. Therefore a single collection can store many different objects of many different types.

When you use Generics to strong type a collection, you will only be able to store objects of the same type within that collection.

In this lesson we will refer to Java 1.4.2, since it is the last version of Java that is still supported by Sun that does NOT support generics and therefore makes it easier to learn about Collections.

## **Lists:**

Here we will examine two list implementations: **ArrayList** and **Vector**.

Actually we will really only focus on ArrayList. The reason I mention Vector is because programmers coming from other Languages such as C++ are probably more familiar with the name Vector than ArrayList.

In effect ArrayList is a newer form of the Vector data structure, at least in Java terms.

As you might already know from your experience with other languages a Vector is a "growable" array. That is you can add and remove elements to a Vector without manually resizing the array, which in most cases take some code to copy the old array to a new array with one element larger or smaller.

The difference between ArrayList and Vector is something to do with Threading in their Java implementations. Basically ArrayList is not synchronized and Vector is synchronized. We will ignore this for now until futures lessons on Multi-Threading and for now focus on the ArrayList version of this data structure.

Here's some quick links to the Java Doc documentation on Sun's web site for both **Vector** (<http://java.sun.com/j2se/1.4.2/docs/api/java/util/Vector.html>) and **ArrayList** (<http://java.sun.com/j2se/1.4.2/docs/api/java/util/ArrayList.html>).

## **More on the ArrayList Class -**

**Package:** java.util

**Class Name:** ArrayList

### **Description on Use:**

Basically you can use the ArrayList class as a growable or shrinkable array. It can also be used as a Queue.

### **Useful Basic Methods:**

You can **store** a value into the ArrayList using the "add" / "append" method: **public boolean add(Object o)**

You can **access** a previously stored value in the ArrayList using the "get" method: **public Object get(int index)**

You can **remove** a previously stored value in the ArrayList using the "remove" method: **public Object remove(int index)**

You can **check the size** of the ArrayList using the "size" method: **public int size()**

You can **clear all** value stored in the ArrayList using the "clear" method: **public void clear()**

### **Example:**

Let's look at a quick example that will create the ArrayList as a data member of the test Class, then one method will add elements to the array list and another method will traverse the array list to print them out.

We are now going to take a look at the file:  
C:\java\_dev\lessons\lesson6\ArrayListTest.java

```
//We are importing the ArrayList class from the util package!  
import java.util.*;
```

```
public class ArrayListTest  
{  
    //The data member, which is our ArrayList  
    private ArrayList arrLst;  
  
    //This is the first time we are using the Constructor  
    //to actually do something. In this case,  
    //We are using it to create the ArrayList object.  
    //It is very common to use the constructor to initialize  
    //our Object's Data Members.  
    public ArrayListTest()  
    {  
        arrLst=new ArrayList();  
    }  
  
    public void addWord(String s)  
    {  
        //Note: The class "Object" is the universal  
        //object. It is the parent of ALL other Objects  
        //in Java. Therefore due to Polymorphism,  
        //We can use it as a reference to ANY object.  
        //Normally it is only used in cases where you  
        //need to reference multiple types of objects  
        //that do not share any other common parent class  
        //or interface. The ArrayList class
```

```

        //uses Object for the Add and Get methods
        //so that you can store any type of object
        //in the collection.
        //Here's the add methods signature right from
        //the Java Doc (JDK 1.4.2):
        //public boolean add(Object o)
        //The add method "appends" to the end of the list...
        arrLst.add(s);
    }

public void printAll()
{
    String s;

    //We are going to use a for loop
    //to loop from element 0 to the SIZE of the arraylist.
    //The method "size()" returns an int representing the
    //number of elements added to and are current contained
    //within the arraylist.
    for (int i=0; i<arrLst.size(); i++)
    {
        //The "get" method will return
        //the Object to us. We need to type case
        //the Object back to the String object.
        s=(String)arrLst.get(i);

        //Print out the String on it's own line.
        System.out.println(s);
    }
}

//This method will remove
//an element from the ArrayList
//at the given index.
public void removeWordAt(int index)
{
    //We should only remove from the array list
    //if the index is a valid index within the Array List.
    //This is why we are checking if the value of index
    //is between 0 and the size of the array list.
    if (index>=0 && index<arrLst.size())
    {
        //If the index is valid
        //Do the actual removal.
        arrLst.remove(index);
    }
}
}

```

```

//Just a wrapper method around
//the "clear()" method of array list
//which will REMOVE all elements from the list.
public void clearAllWords()
{
    arrLst.clear();
}

public static void main(String[] args)
{
    //Create a new ArrayListTest object to use
    ArrayListTest alTest=new ArrayListTest();

    //Add Strings to my array list using the addWord
    //method on the ArrayListTest class we just created...
    alTest.addWord("Hello");
    alTest.addWord("World");
    alTest.addWord("This");
    alTest.addWord("Is");
    alTest.addWord("My");
    alTest.addWord("First");
    alTest.addWord("Test");
    alTest.addWord("With");
    alTest.addWord("ArrayLists!");

    //Use the print all method on the ArrayListTest
    //class we created to print each word we added
    //on it's own line on the screen.
    alTest.printAll();

    //Print a blank line to separate output!
    System.out.println();

    //Let's test the remove method and remove word "First"
    //which if we count is the 6th element
    //which is array index 5.
    //We need to use array index 5 instead of the
    //actual element number
    //index the first element in the array list is
    //called element 0 not 1!
    //Basically as we mentioned many times,
    //ArrayList is basically a
    //growable array!
    alTest.removeWordAt(5);
}

```

```
        //Let's print again, to show the changes to the list!
        alTest.printAll();

        //Since we are done with the list.
        //It's good practice to clear the list.
        //This helps Java clear the memory of unused
        //data FASTER.
        alTest.clearAllWords();
    }
}
```

---

### **Maps :**

Here we will examine two list implementations: **HashMap** and **Hashtable**.

Actually we will really only focus on HashMap. The reason I mention Hashtable is for the same reason I mentioned Vector when we spoke about lists. It is because programmers coming from other Languages such as C++ are probably more familiar with the name Hashtable than HashMap.

In effect HashMap is a newer form of the Hashtable data structure, at least in Java terms.

The same differences between ArrayList and Vector exists with HashMap and Hashtable. Hashtable is the synchronized version of the classic "hash table" data structure and HashMap is an unsynchronized version of the same data structure. Again, please ignore this difference for now until we talk about Threads. There's one other difference. HashMap can store a NULL as a key whereas Hashtable does NOT allow this. In most cases you will probably not use a NULL as a key so you don't have to worry about this all to much either, but you should know the difference in case you do need to do something like this in your programs.

Here's some quick links to the Java Doc documentation on Sun's web site for both **Hashtable** (<http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html>) and **HashMap** (<http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html>).

## More on the HashMap Class -

**Package:** java.util

**Class Name:** HashMap

### Description on Use:

HashMap is a standard Hash Table implementation that uses the bucket implementation for storage. Basically a Hash Code is calculated on the "key" used in the table and based on that Hash Code a bucket is located in the Hash Table data structure. A bucket is basically like a Linked List or some other growable list (ArrayList perhaps?), and then the bucket is traversed until the EXACT key is found which is associated with a value (Key-Value pair).

### Useful Basic Methods:

You can **store** a key-value pair into the HashMap using the "put" method: **public Object put(Object key, Object value)**

You can **access** a previously stored value in the HashMap using the "get" method: **public Object get(Object key)**

You can **remove** a previously stored value in the HashMap using the "remove" method: **public Object remove(Object key)**

You can **clear all** key-value pairs stored in the HashMap using the "clear" method: **public void clear()**

### Example:

Let's look at a quick example that will create the ArrayList as a data member of the test Class, then one method will add elements to the array list and another method will traverse the array list to print them out.

We are now going to take a look at the file:

C:\java\_dev\lessons\lesson6\HashMapTest.java

```
import java.util.*; //We are importing the HashMap
                    //class from the util package!
```

```
public class HashMapTest
{
    public static void main(String[] args)
    {
        String s;
        Integer num;
```

```

HashMap map=new HashMap(); //Create a new HashMap Object

//Store a key-value pair in the map
//The Key is a String "HelloMessage"
//The Value is also a String "Hello World!
//This is my HashMap Test!"
map.put("HelloMessage", "Hello World! This is my HashMap
Test!");

//Store a second key-value pair in the map
//The Key is a Integer Object wrapping the int: 2
//The Value is a String "TWO"
num=new Integer(2);
map.put(num, "TWO");

//Let's store the reverse as well.
//In this case the key is String "ONE"
//And the value is Integer(1)...
num=new Integer(1);
map.put("ONE", num);

//----->

//Retrieve the values from the map and print

//The Number Integer
num=(Integer)map.get("ONE");
System.out.println(num);

//The Number String
num=new Integer(2);
s=(String)map.get(num);
System.out.println(s);

//The Hello Message
s=(String)map.get("HelloMessage");
System.out.println(s);
}
}

```

-----

### **Compiling and Running the Lesson Examples:**

### **How to Compile Lesson using the included Windows Batch Files:**

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson6`
3. `compile.cmd`

#### **How to Compile Lesson using javac.exe program directly:**

Note: This assumes you have the javac.exe program in your PATH.

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson6`
3. `javac *.java`

#### **How to run the examples using the included Windows Batch Files:**

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson6`
3. To run the ArrayList example from the program's main entry point class (Java Class: ArrayListTest.java) run: `run_arraylist_test.cmd`
4. To run the HashMap example from the program's main entry point class (Java Class: HashMapTest.java) run: `run_hashmap_test.cmd`

#### **How to run the examples using the java.exe program directly:**

Note: This assumes you have the java.exe program in your PATH.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson6`
3. To run the ArrayList example from the program's main entry point class (Java Class: ArrayListTest.java) run: `java -classpath C:\java_dev\lessons\lesson6 ArrayListTest`
4. To run the HashMap example from the program's main entry point

```
class (Java Class: HashMapTest.java) run: java -classpath  
C:\java_dev\lessons\lesson6 HashMapTest
```

## Lesson 7 – Exceptions

### Terms :

**Exception** - An object that represents an error in Java.

**Throw/Thrown/Throws** - When an exception object gets created, you can explicitly "Throw" the exception UP the call stack. "throw" is a keyword in Java that will pass the exception object up the call stack.

**Catch** - When your code or the JVM receives an exception object that was thrown, this is called a catch. There is a keyword in Java called "catch".

### What are exceptions?

Exceptions are Java Objects which are used to represent errors. They include an error message, which is text, and more importantly, something called a Stack Trace.

The stack trace is a text dump of the execution call stack. You will be able to see the "path" of execution through each function call to the exact line number of where the exception got "thrown".

### How we use Exceptions in Java:

Keywords: throws, try, catch, throw.

Keyword: **throws**

This keyword is used in a method declaration to tell the compiler as well as users of the API that this method will throw the listed exceptions.

Syntax: [method signature] throws [EXCEPTION\_LIST]

Example with single exception:

```
public int readNumberFromFile(String filePath) throws IOException
```

Notice in the line above after the normal method signature we added the part in bold. The first thing is the keyword `throws`, and the second is the Exception Class Name that we will potentially throw within the method if there's an error.

Example with multiple exceptions:

```
public int dbLoginUsingPasswordFile(String filePath, String dbUrl)
throws IOException, SQLException
```

Notice in the line above the only difference in the part in bold is the exception exception class name separated by a comma. You can add N-number of exception class names to the list after the **throws** keyword, separated by commas if your code within the method may at some point possibly throw that type of exception.

Keyword: **throw**

This keyword (without the s on the end.) is used in a method's code block when you want to cause a new exception object to be "thrown" up the call stack. The type of exception MUST be of one of the types in the throws clause exception list. You can also throw any type of RuntimeException without having to list them in the throws clause exception list.

Syntax: `throw new [ExceptionClassName()];`

Example (assume readfile is a method that returns true or false):

```
//This says if NOT readfile()
//meaning if readfile returns FALSE
//Do something...
//In this case if FALSE throw a IOException.

if (!readfile())
{
    //All exceptions have a constructors
    //That exceptions a String to use as
    //an error message.
    throw new IOException("Error while Reading File!");
}
```

Next we will go over the **try**, **catch**, and **finally** keywords together as a group, because ALL catch and finally's MUST be preceded by a try.

Keyword: **try**

This keyword is used in a method's code block as a start of a nested block of code within the method that may throw some type of

exception. This is the first part of the code you need to explicitly write to satisfy the language and compiler requirements to explicitly CATCH Checked Exceptions.

The next two **catch** and **final** are optional, but at least one of the two MUST be present if there is a preceding try block. Also you can have all three: try, catch, then final

Keyword: **catch**

This keyword is used to start a catch block of code which is used to process a thrown exception within a previously defined try block of code. This block of code ONLY executes if there's an exception raised in the preceding try block.

You can have as many catch blocks, one for each type of exception that might be thrown in the preceding try block.

If you do have more than one catch block preceding a try block, you would start with the "LOWEST" exception class type in the class hierarchy first in the code and as the catch blocks continue, you should go up the chain.

Example:

If MyException2 extends from MyException.

Then IF I want to process exceptions that are thrown using the class MyException directly, and I also want to process exceptions that are thrown using the class MyException2 the first catch block should be MyException2, and the second catch block should be MyException.

Note: Just to reiterate, you do not have to have a catch block for every exception in the exception class tree, just the ones you are using...

Short Cut: You can just have one catch block to catch the top level exception "Exception" like this:

```
try
{
    //Code that may throw
    //Many different types of exceptions.
}
catch(Exception e)
{
    //This Catches ALL Exceptions!
}
```

Keyword: **finally**

This keyword is used to start a finally block of code which executed a block of code that will execute at the end of the try block if there is NO exception raised and will also run at the end of a catch block if an exception was raised. So unlike the catch block, this block of code will ALWAYS be executed. It is useful to use finally blocks to do clean up processes, such as closing files or database connections. Things that always need to run regardless whether or not there was an error or not.

#### **Syntax for Try-Catch-Finally blocks of code:**

```
try
{
    //Do something
    //We might throw an exception here on error
}
catch(ExceptionClassName variableName)
{
    //Do something
    //ONLY if an exception was thrown
    //OF TYPE ExceptionClassName.
    //You can access the
    //caught exception via
    //the variableName object.
}
finally
{
    //Do something
    //Whether or NOT
    //the catch block ran!
}
```

#### **Types or "Groups" of Exceptions**

There are two main types of exceptions:

1. Runtime Exceptions (aka Unchecked Exceptions). These exceptions are things like NullPointerExceptions, which do NOT need to be explicitly caught in your code and the compiler will NOT complain about these. They happen at runtime... This is why they are called "unchecked" exceptions, because the compiler does NOT check for them.
2. Exceptions (aka Checked Exceptions). These are exceptions which are checked by the compiler hence the name "Checked Exceptions." Examples of these are IOExceptions and SQLExceptions. You must explicitly write code to catch these types of exceptions. If the

compiler detects that there's a place in your code where an exception is thrown say from a function that opens a file, and the file APIs provided by the Standard Java Library throw an IOException on function calls to the File API, and you did not surround the calls to these functions in a try-catch block as described above, then the compiler will complain.

Let's quickly explore the Class Hierarchy. This gets into inheritance, which we essentially have ignored up until now. And still at this point, we will not get deep into inheritance. We will however mention it here as we explore how the Exceptions classes in Java are structured.

Here's the class hierarchy represented as a tree (all classes in the tree are in the package: java.lang and therefore are imported implicitly and do not need to be imported by your code!):

```
    Throwable
      |
    Exception
      |
  RuntimeException
```

At the very top we have the class "Throwable". This defines the base of what an exception is. And is why we call exceptions "throwables" because they inherit from the "Throwable" parent class.

At the second level in the tree, we have the class "Exception". All exceptions inherit from this class. Exceptions such as: IOException, SQLException, InterruptedException are all exceptions of Classes which extend from the Exception class. This makes them "Exceptions." And therefore fall into the Checked Exception Category, meaning they will be caught by the compiler as compile-time errors if your code does NOT explicitly catch them.

RuntimeExceptions which are at the third level in the tree, although extends from Exception, introduces one major difference. This difference allows the distinction from other exceptions because any exception class that extends from RuntimeException instead of Exception directly are "Unchecked Exceptions" and therefore NOT checked by the compiler. Some examples of RuntimeExceptions are: NullPointerException, ClassCastException, and IndexOutOfBoundsException.

## Creating your own new types of Exceptions:

Exceptions are just like any other class that you may create, with one significant difference. They MUST extend from the class: `Exception`, which as mentioned about is part of the `java.lang` package.

To create an exception you just create a new Java class and "EXTEND" from `Exception` as follows. (Notice, we will introduce a new keyword called "**extends**" but we will not get into it until a future lesson on inheritance. But for now just know that to inherit from a parent class you just need to add the `extends` clause to the class declaration as follows):

```
public class CLASS_NAME extends PARENT_CLASS_NAME
```

This is how you use `extends` to create a new `Exception`:

```
public class MyException extends Exception
{
    public MyException(String msg)
    {
        super(msg);
    }

    public MyException(String msg, Exception rootEx)
    {
        super(msg, rootEx);
    }

    public MyException(Exception rootEx)
    {
        super(rootEx);
    }
}
```

Until you are comfortable with inheritance, for now if you need to create a new type of `Exception` object. Just copy the code above and just change the class name from "`MyException`" to whatever you want to call your exception.

The above code is stored in file for your reference and usage:  
`C:\java_dev\lessons\lesson7\MyException.java`

**Note on Naming Convention:** By convention, class names of new types of `Exceptions` should end with the word "`Exception`".

## **Compiling and Running the Lesson Examples:**

### **How to Compile Lesson using the included Windows Batch Files:**

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson7`
3. `compile.cmd`

### **How to Compile Lesson using javac.exe program directly:**

Note: This assumes you have the javac.exe program in your PATH.

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson7`
3. `javac *.java`

### **How to run the examples using the included Windows Batch Files:**

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson7`
3. To run the ExceptionTests example from the program's main entry point class (Java Class: ExceptionTests.java) run:  
`run_exception_tests.cmd`

### **How to run the examples using the java.exe program directly:**

Note: This assumes you have the java.exe program in your PATH.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson7`
3. To run the ExceptionTests example from the program's main entry point class (Java Class: ExceptionTests.java) run: `java -classpath C:\java_dev\lessons\lesson7 ExceptionTests`

## Lesson 8 – JDBC (Java Database Connectivity)

JDBC is the Java standard for connecting to and interacting with databases.

It's an API framework that is made up of the following base Classes: DriverManager, Connection, Statement, PreparedStatement, CallableStatement, ResultSet.

There's a number of other classes besides the ones mentioned above, such as ResultSetMetaData, and DatabaseMetaData, among others. All Classes are part of the package: **java.sql**. For the full list of classes that made up the JDBC framework, please see the Java Docs. Here's a direct link to the Package on the Java Doc site:  
<http://java.sun.com/j2se/1.5.0/docs/api/java/sql/package-summary.html>

You will notice that all the classes I mentioned above except for DriverManager are actually Interfaces instead of Classes. This is because each Driver implements it's own versions of the interfaces above.

See the table below for a description of what each Class/Interface is used for in the JDBC Framework:

Class or Interface Name	Usage
DriverManager	Used to obtain a Connection implementation from the Driver. This is done via the getConnection method, which matches a JDBC URL prefix to a driver implementation. You do NOT use this class to "load" the driver. We will go over how to load drivers later on in this lesson.
Connection	This class represents a physical network connection to database. You can use this object to create Statements, PreparedStatements, and CallableStatements. You do NOT execute SQL directly on a connection, instead to execute the SQL through the created Statements that you create using the connection.
Statement	This is a basic "statement" implementation. It allows for

	<p>dynamic SQL execution. There's two methods: executeUpdate and executeQuery. You pass String representations of SQL statements to each of these methods to have them executed on the database. SELECTS are executed using the executeQuery method, and INSERTS, UPDATES, DELETES, DROPS, CREATES, etc are all executed via the executeUpdate method.</p>
PreparedStatement	<p>Implementations of the PreparedStatement will provide the same executeUpdate and executeQuery methods as the normal Statement object, but instead, PreparedStatements are used to "pre-compile" SQL statements on the DB and you can pass parameters to the prepared statement using the set[DataType] methods. These are used for fast execution of SQL statements that you need to call in a repetition fashion.</p>
CallableStatement	<p>This statement implementation is used to call Stored Procedures. There's a generic "execute" method provided by the CallableStatement object which can be used to execute a Stored Procedure that may do multiple statement executions, updates as well as queries.</p>
ResultSet	<p>A ResultSet is returned by the executeQuery method on the statement implementations. It is used to traverse a returned record set from the database.</p>

## How to create a connection to a Database:

Let's go over the steps of creating a connection using JDBC...

### 1. Load the driver:

Loading the driver is pretty easy. All you need to do is have the Java Class Loader load the class into memory. The driver class will do the rest.

How to have the class loader load the driver?

It's just one line of code:

```
Class.forName(FULLY_QUALIFIED_DRIVER_CLASS_NAME);
```

Example:

```
Class.forName("com.mysql.jdbc.Driver");
```

Notice: That the class name we passed into the static method `Class.forName`, include the package name "com.mysql.jdbc". This is what we mean by "FULLY QUALIFIED".

`Class.forName` load a class by name. We will talk more about this method in a possible future lesson on Java Reflection.

### 2. Obtain connection from the Driver Manager:

Call the "**getConnection**" method. This method is the "login" method and will return a `Connection` object.

There's three versions of the `getConnection` method. We will take a look at two of them:

The first version is:

```
public static Connection getConnection(String url,  
                                     String user,  
                                     String password)  
    throws SQLException;
```

This `getConnection` method above is the most simply to use. You simply provide it with three strings. The DB URL, Username, and Password.

The second version is:

```
public static Connection getConnection(String url,  
                                     Properties info)  
    throws SQLException;
```

Note: `Properties` is a Class included in the Standard Java Library in

the java.util package. It is basically a Hashtable, but has extra methods like void setProperty(String name, String value) and String getProperty(String name).

It is used to represent configurations in the form of name-value pairs of Strings.

Look it up in the Java Docs. Here's a direct link:

<http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html>

-----

This version of the getConnection method allows you to set many more configuration options besides a username and password. A lot of these configuration parameters are driver specific, so you will need to read the documentation from the Vendor such as Sybase, IBM, MySQL, etc for what properties are understood by the driver.

Normally at least **user** and **password** are properties that MUST be set if you use this method to get your connection.

Obviously the first version we looked at was much more simple, but in advanced programs you might want to consider using the second version as you can really control the database connection performance and behavior with the Properties version.

### 3. Creating Statements...

After the first two steps, we have already completed the steps needed to obtain a connection, but we will continue talking about how to use the JDBC framework to execute simple SQL statements.

Once you have a connection object you need to create a statement to execute the SQL on. This statement can be a normal Statement object, a PreparedStatement, or a CallableStatement. Use them as described previously in our table on basic JDBC classes.

To create a normal Statement you use this method on Connection:

**Statement createStatement() throws SQLException;**

To create a PreparedStatement you use this method on Connection:

**PreparedStatement prepareStatement(String sql) throws SQLException;**

To create a CallableStatement you use this method on Connection:

**CallableStatement prepareCall(String sql) throws SQLException**

See the Java Docs for more information on these APIs and how to use them. Here's a direct link to the Connection Java Doc Page:  
<http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Connection.html>

**One important note on statements is that:** Normally, a normal Statement can be reused for multiple SQL executions of different SQL statements. But PreparedStatement and CallableStatement can only be used to execute the same SQL or Stored Procedure multiple times.

#### **4. Closing your Statements and Connections...**

Finally once you are completed using your connection and the Statements you created, you **MUST** close all the statements and then the Connection in that order. Although if you forget to close the statements, closing the Connection will cause all previously opened statements on that Connection to be implicitly closed. But still it is a good idea to always **CLOSE** all your statements. This is important to note, because in large programs, we often keep the connection to the database opened for a long period of time, and we may want to open and close many statements. This usually happens if you create a statement in your programs initialization routine and set it say as a Data member in your class. Then you have multiple methods that execute SQL and each method will reference the connection you set as a data member in the class (Remember from our lesson on OOP a data member is like a global variable inside the separate memory space of an Object.) and will create their own statements to use. At the end of each method that creates its own Statements, you **MUST** always have a finally block (remember our lesson on Exceptions) and in the finally block you **MUST** close the statements. Each statement opens up remote resources on the database server, and too many opened statements will cause the database to stop allowing new statements or even possible connections to be opened.

On **ALL** statements there's a **public void close() throws SQLException** method. This same method also exists on the Connection object. Make sure to call them when you are done using them!

Let's now take a look at an example of a program that uses JDBC to connect to a local MySQL database (if you have the Java Dev CD that comes with this Java Guide), and it will first delete all records in a test table, then insert two new records, then selects them out of the table and prints them on the screen.

Please take a look at file: C:\java\_dev\lessons\lesson8\JDBCTest.java

```
import java.sql.*; //Import Connection, Statement, ResultSet, etc...
```

```
public class JDBCTest
```

```
{
```

```
    //Define DB Property Constants
```

```
    public static final String DB_DRIVER="com.mysql.jdbc.Driver";
```

```
    public static final String
```

```
DB_URL="jdbc:mysql://127.0.0.1:3306/test";
```

```
    public static final String DB_USERNAME="root";
```

```
    public static final String DB_PASSWORD="abcd1234";
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Connection conn=null;
```

```
        Statement st=null;
```

```
        ResultSet rs;
```

```
        int id, rowCnt;
```

```
        String name;
```

```
        try
```

```
        {
```

```
            //Initialize the MySQL JDBC Driver
```

```
            Class.forName(DB_DRIVER);
```

```
            //Create connection to the database
```

```
            conn=DriverManager.getConnection(DB_URL, DB_USERNAME,  
DB_PASSWORD);
```

```
            //Create a statement object to use
```

```
            //to execute sql statements on the DB.
```

```
            st=conn.createStatement();
```

```
            //Let's delete ALL records
```

```
            //from the test table.
```

```
            rowCnt=st.executeUpdate("DELETE FROM test.mytable");
```

```
            System.out.println("Rows Deleted: "+rowCnt);
```

```
            //Insert some test rows
```

```
            st.executeUpdate("insert into test.mytable(id, name)  
values(1, 'Robert')");
```

```
            st.executeUpdate("insert into test.mytable(id, name)  
values(2, 'Paula')");
```

```
            //Select out all rows from the table and print
```

```
            rs=st.executeQuery("select id, name from
```

```

test.mytable");

        rowCnt=1; //Set row count to 1 initially

        while (rs.next())
        {
            id=rs.getInt(1); //index 1 for column 1
                               //which is id
            name=rs.getString(2); //index 2 for column 2
                                   //which is name

            System.out.println("Row "+rowCnt+" : ID="+id+",
NAME="+name);

            rowCnt++; //Increment row count
        }
    } //End try block
    catch(Exception e)
    {
        //Log any errors...
        e.printStackTrace();
    }
    finally
    {
        //Make sure we ALWAYS close
        //the statement and connection
        //Error or no error!

        if (st!=null)
        {
            try
            {
                st.close();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }

        if (conn!=null)
        {
            try
            {
                conn.close();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}

```

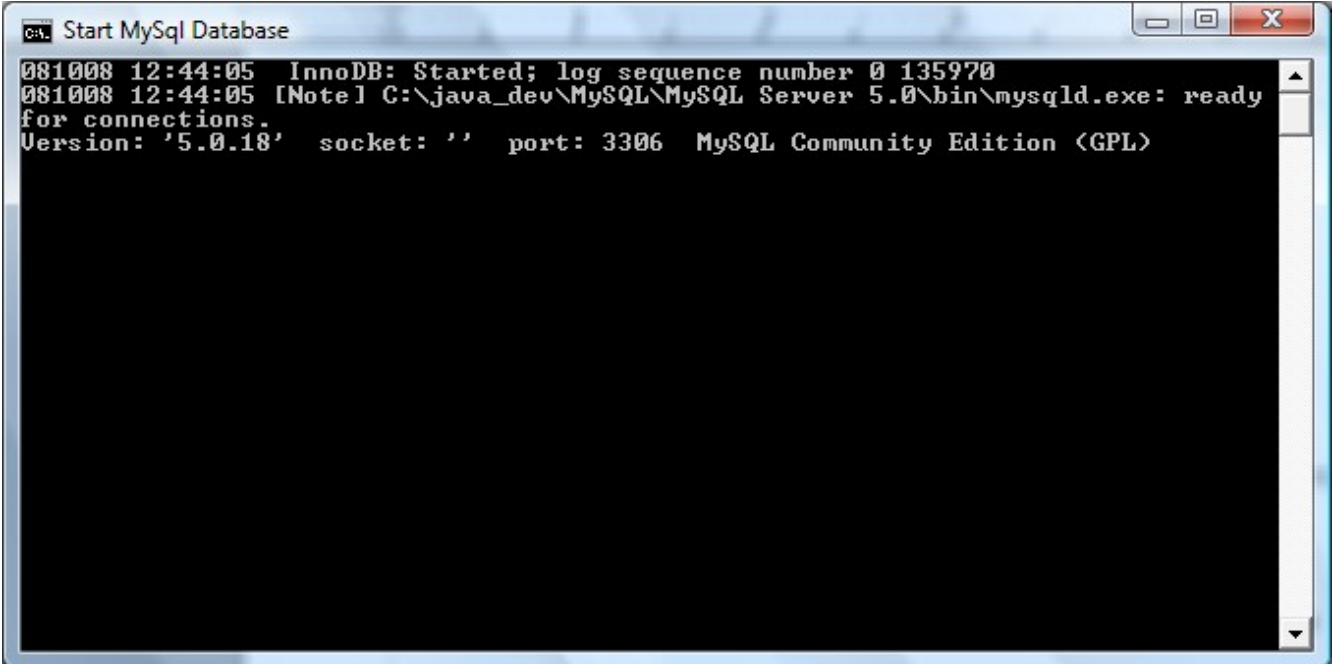
```
    }  
  }  
}
```

-----  
If you want to see the SQL used to create the test table see file:  
C:\java\_dev\lessons\lesson8\sql.txt

### **Starting the Local MySQL Database included on the Java Dev CD:**

Note: This MUST be done to successfully run the example above.

1. Open Windows Explorer and go to Directory:  
C:\java\_dev\lessons\lesson8
2. In this directory you will find a Windows Shortcut: "Start MySQL Database"
3. Double Click on this short cut and a DOS Prompt Window will appear, which will look similar to the below:



```
Start MySQL Database  
081008 12:44:05 InnoDB: Started; log sequence number 0 135970  
081008 12:44:05 [Note] C:\java_dev\MySQL\MySQL Server 5.0\bin\mysqld.exe: ready  
for connections.  
Version: '5.0.18'  socket: ''  port: 3306  MySQL Community Edition (GPL)
```

You may be prompted by Windows if you want to unblock the port 3306 on your local windows Firewall. You can if you like, if not it should still work since we are only connecting on localhost. However if you

have problems, please rerun this shortcut and unblock the port.

3. When you have finished the example you will need to shutdown the MySQL Database Server. To do this. Go to the MySQL Database Server DOS Prompt and press together Ctrl-C. You might have to do this twice. Give it a few seconds between each Ctrl-C to allow the DB server to shutdown.

### **Compiling and Running the Lesson Examples:**

#### **How to Compile Lesson using the included Windows Batch Files:**

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson8`
3. `compile.cmd`

#### **How to Compile Lesson using javac.exe program directly:**

Note: This assumes you have the javac.exe program in your PATH.

1. Start a DOS Prompt. And from the dos prompt run:
2. `cd\java_dev\lessons\lesson8`
3. `javac *.java`

#### **How to run the examples using the included Windows Batch Files:**

**IMPORTANT:** Make sure you FIRST have started the MySQL Database Server before running the example! Read "Starting the Local MySQL Database included on the Java Dev CD" above for instructions how to start the Database.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:
2. If not already in this directory run: `cd\java_dev\lessons\lesson8`
3. To run the JDBC Test example from the program's main entry point class (Java Class: JDBCTest.java) run: `run_jdbc_test.cmd`

#### **How to run the examples using the java.exe program directly:**

**IMPORTANT:** Make sure you FIRST have started the MySQL Database Server before running the example! Read "Starting the Local MySQL Database included on the Java Dev CD" above for instructions how to start the Database.

Note: This assumes you have the java.exe program in your PATH.

1. Start a DOS Prompt (It can be the same prompt that you compiled from if you still have that window opened). And from the dos prompt run:

2. If not already in this directory run: `cd\java_dev\lessons\lesson8`

3. To run the JDBC Test example from the program's main entry point class (Java Class: JDBCTest.java) run: `java -classpath C:\java_dev\lessons\lesson8;C:\java_dev\java_libs\mysql-connector-java-5.1.5-bin.jar JDBCTest`

**IMPORTANT Difference to Classpath in this example:**

Notice the Classpath parameter for the Java.exe (JVM) includes not only the lesson8 directory but also the JAR file: **mysql-connector-java-5.1.5-bin.jar**. This JAR includes the MySQL JDBC Driver and MUST be included in the classpath in order for the Class.forName method to find the JDBC driver and load it into memory.