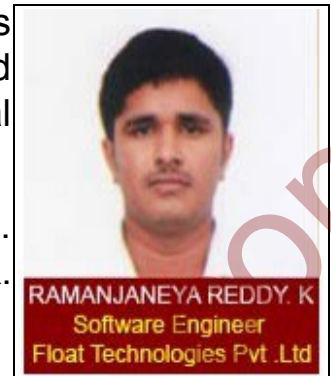


Import Statement

After learning the concept of package, the next part is import statement to the program and run in the command prompt. We can import the statement at some logical statements only.

For example if we are executing a program on circle. According to the mathematical statement we know $c=2\pi R$. Here $\pi=3.1415$



Let's execute a program using the statement in java

The java.lang.Math class defines the PI constant and many static methods, including methods for calculating sines, cosines, tangents, square roots, maxima, minima, exponents, and many more.

Example statement using PI value

```
public static final double PI = 3.141592653589793;
public static double cos(double a)
{
    */////////////////
    ///////////////////*
}
```

If we observe the statement particularly, it imported the PI value into java. But the PI value is taken as double in the program which is the statement written by Sun Micro Systems and it is imported as "Final". Many import statements are taken as Static where the value should not be changed and made it is a public. So we can say that, to import a statement on PI we need statement:

```
Public static final double PI=3.141592653589793;
```

These statements will be under package of java.lang.math. All the arithmetic operations are imported from this package. The package can be described as briefly in this way: The operating view of the package is taken as java and is the package folder where lang is the sub-folder. Here we have one more package called as math, which is called as "Sub-sub package". In this package we have import statement.

We can do sines, cosines, tangents, square roots, maxima, minima, exponents which is under the method. By importing this statement a direct call to this method can be made in the program and execution process is very easy.

Example statement in the java.lang.math:

For importing BigInteger:

```
package java.math;
import java.util.Random;
import java.io.*;
public class BigInteger extends Number implements Comparable<BigInteger>
{
    *////////////////
    ///////////*
}
```

We have imported the statement under the package of java.math, which is fully classified package name and the imported two statements for program it is java.util.Random and java.io.*. For execution we need to have the Random and io.* statements mainly.

For importing Integer:

```
package java.lang;
import java.util.Properties
    public final class Integer extends Number implements
Comparable<Integer>
{
    *////////////////
    ///////////*
}
```

The above example is bit different from the previous example, where we imported only statement of util.Properties. If we observe clearly we had made this imported and class as PUBLIC.

Diagram view of package:

Import Statements

Syntax for import statements:

There are two types of import statements for a package. There are listed as below:

```
Import    pack1.[pack2[.....[packn]]].*;
```

This syntax makes us to understand how to refer all classes/ interfaces of separate package in current java program. This approach is not recommended to use in industry even if we are using two classes of one package, because uninterested classes and interfaces are also loaded in main memory and becomes waste of memory use.

If use statement 1 in our current java program then we can refer all the classes and interfaces of p1 package, but we can't refer all the classes and interfaces of p2 package.

If we use statement 2 in our java program then we can refer all the classes /interfaces of package p2 but we can't refer classes/interfaces of its upper package and its lower package.

Here, we have

P1, p2, p3 are packages

C1, c2, c3 are classes

I1, i2, i3 are interfaces

Syntax 2:

```
Import pack1[.pack2.....[packn]].class/interface name;
```

This syntax makes us understand a specific class/interface of a particular package in our current java program.

Example:

```
Import p1,c1;
```

```
Import p1.p2.c2;
```

```
Import p1.p2.p3.c3;
```

If we observe the above statements in our java program we can refer statement 1 for only class c1 of package p1 but we can't refer other classes and interfaces of same package p1. This type of syntax 2 is highly recommended in industry for real-time experience. Because a programmer can refer only interested class/interfaces by eliminating to import uninterested class and interface.

Static import statement:

Static import statement is new feature in JDK 1.5 version onwards. The purpose of the new facility available is to eliminate repeated referring of class name or interface name before the static data members and class names before the static methods. This jdk 1.5 is also called as "tiger" software. In other words static imports make us to refer static features directly without referring their class name and interface names.

Syntax 3:

Import static pack[.pack2[.....[packn]]].class/interface name.

This method of writing the import statement refers the "static data member level".

Syntax 4:

Import static pack1[.pack2[.....[packn]]]. Classname.methodname;

This method of writing syntax in import statements refers to "Static method name".

Example program;

```
class Languages
{
    public static void main(String[] args)
    {
        display();
    }

    static void display()
    {
        System.out.println("Hello welcome to my article");
    }
}
```

```
}  
}
```

Excetuion:

Hello welcome to my article

Synatx 5:

Import static pack1[.pack2[.....[packn]]].class name/interface name.*;

Example

```
class Util  
{  
    public static void method()  
{  
        // write your method  
    }  
}
```

This method of writing syntax refers to the “static features level”

Importing a Package Member

Import a package member of Rhombus which is user-defined package

We can import the package not only for arithmetic operations in the program; the import statement is used for graphical representation also. In the same way we can create a user-defined package by following the rules. To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one. Here's how you would import the Rectangle class from the graphics package. Here is one of the created package for Rhombus.

Example

```
import graphics.Rhombus;  
  
{  
  
Rhombus my Rhombus = new Rhombus ();  
  
    *////////////////////
```

```
//////////////////////////*
```

```
}
```

Above is the dynamic creation.

Import a package member of Square which is user-defined package:

We can import the package with square, which is already present in the user defined one. As we know from the mathematical order, insert the statement for finding the perimeter and area of the given square.

Where,

Perimeter of the square= $4 \times \text{square}$

Area of the Square= S^2

Example

```
import graphics.Square;

{
Sqaure mySquare = new Square ();

    *//////////////////////////
    ////////////////////////////*
}
```

Importing an Entire Package

To import a package is different from import a package member. In importing package we import fully classified name. To import all the types contained in a particular package, use the import statement with the asterisk (*) wildcard character.

```
import graphics.*;
```

Now you can refer to any class or interface in the graphics package by its simple name.

```
{
Circle myCircle = new Circle();
Rhombus myRhombus = new Rhombus();
}
```

The asterisk in the import statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the graphics package that begin with A.

// does not work

```
import graphics.A*;
```

Instead, it generates a compiler error. With the import statement, you generally import only a single package member or an entire package.

Importing user defined package:

As we discussed earlier we can create and define user packages in the program. These packages are made public to a limited and used in companys.

```
Import.java.io.*;
```

```
Import.java.io.DataInputStream;
```

Here in this package we have both the implicit import of the package and explicit import of the package.

Example for creating a demo package;

Create a new package demo_pack by creating a new directory by the name demo_pack. Assume we are in G:drive. Now move the directory demo_pack and create a new file in it by the name **demo2.java** as

```
G:\demo_paack> edit demo2.java
```

Example program for creating a demo package:

```
Package demo_pack;
```

```
Public class demo2
```

```
{
```

```
    Public void show()
```

```
    {
```

```
        System.Out.Println("Welcome to this package")
```

```
}  
Public int mul(int x , int y)  
{  
    Return x*y;  
}  
}
```

As if you observe there are two public functions is **show** and **mul**.

```
Import demo_pack.demo2;
```

```
Class main
```

```
{  
    public static void main(string[]args)  
    {  
        demo2 d=new demo2();  
        d.show();  
        int m=d.mul(40,50);  
        system.out.println("mul"*m);  
    }  
}
```

Before compiling the program, set the classpath, if not set earlier

```
Set=classpath=.;G;\;
```

It is a dynamic selection of setting class path.

Execution:

Compile and run the program you will have the output as

Welcome to my package

Mul=2000

Q) Write a java program which list of all static import for referring static features:

//program

```
/*Import static java.lang.math.Sqrt;
```

```
    Import static java.lang.math.Pow.*;
```

(OR)

```
    Import static java.lang.math.*;
```

```
    Import static java.lang.system.out;
```

```
Class arithmeticdemo2
```

```
{
```

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

```
{
```

```
        System.out.println("val of pi="+pi);
```

```
        System.out.println("val of ri="+E);
```

```
        System.out.println("Sqrt of 64"="+Sqrt(64));
```

```
        System.out.println("Sqrt of 121"="+Sqrt(121));
```

```
        System.out.println("8 to the power of 2="+pow(8,2));
```

```
    }
```

```
}
```

Understanding CLASSPATH:

In java, setting path is the most important concept. To run or execute the program setting class path should be understand. There are few rules and few concept oriented process to set the class path in java environment. There are two ways of setting class path in java programming language. One approach can be done directly (permanent) and another approach by directory path(temporary).

For example write a program in java.

```
package com.Tommorow;
```

```
public class Test1
```

```
{
public static void main(String[] args)
    {
        System.out.println ("Run Test1.main()");
    }
}

package com.Tomorrow;
public class Test2
{
public static void main(String[] args)
    {
        System.out.println ("Run Test2.main()");
        CPTest1 cpt1 = new Test1();
    }
}

} //program end
```

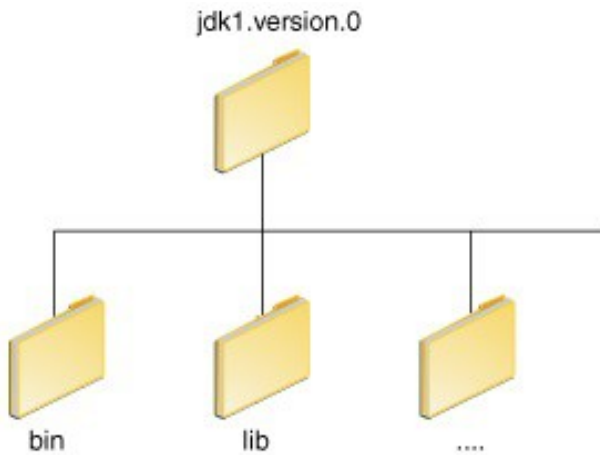
Execution of Program:

Rather than specifying class search path on the javac command line, we can make use of a 'system' class path. This is the class path that will be used by both the Java compiler and the JVM in the absence of specific instructions to the contrary. In both Unix and Windows systems, this is done by setting an environment variable. For example, in Linux with the bash shell:

and in Windows:

set CLASSPATH=c:\folder\subfolder

Setting permanent class path:



For setting class path in windows environment variables follow the steps:

You can run Java applications just fine without setting the PATH environment variable. Or, you can optionally set it as a convenience.

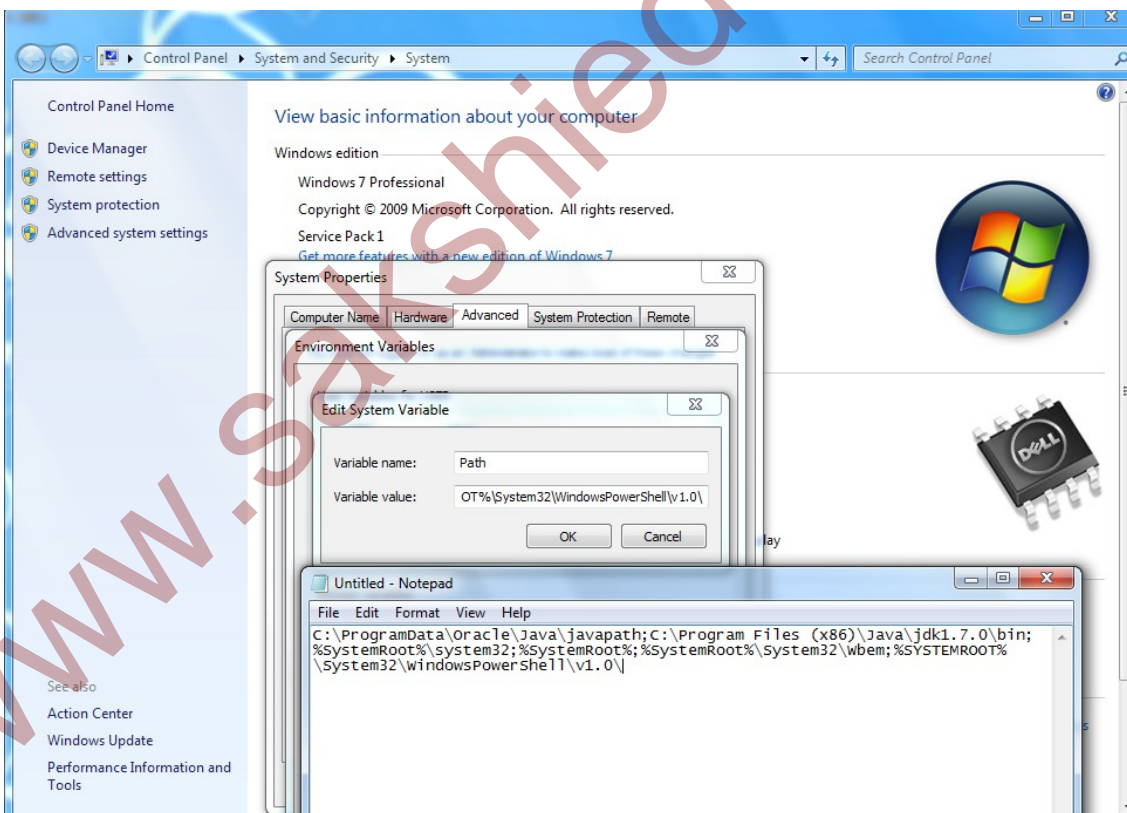
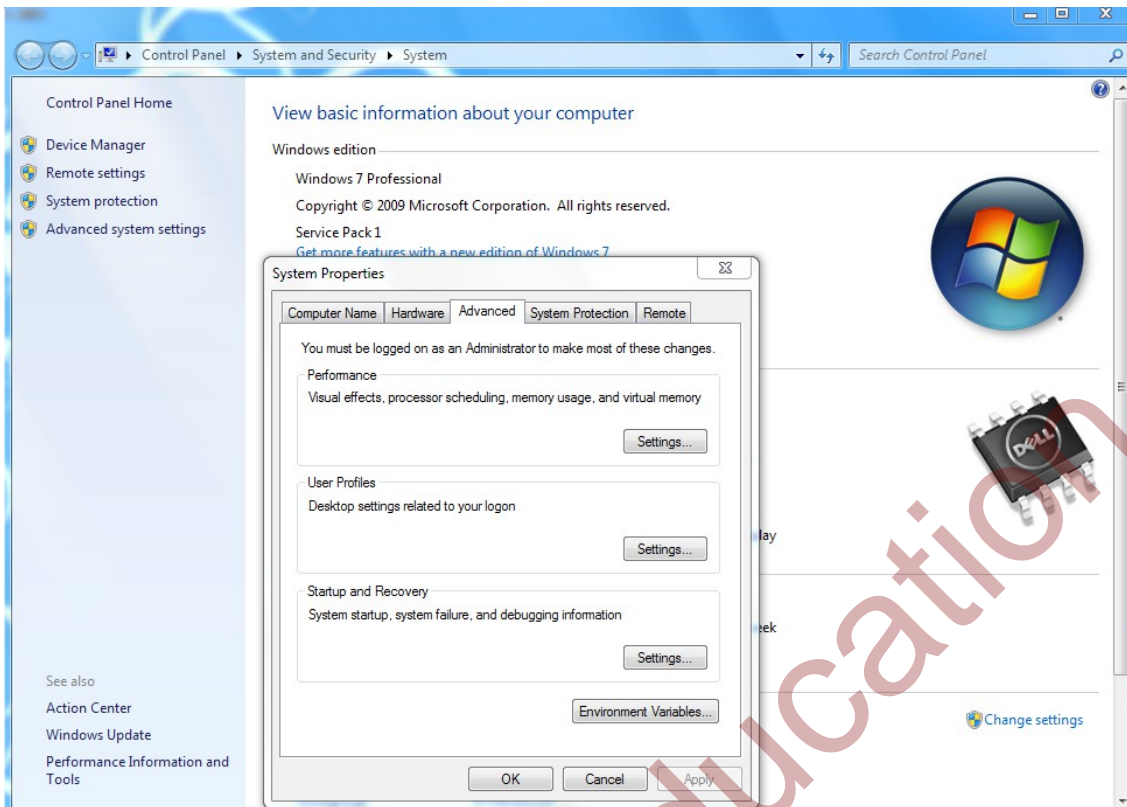
Set the PATH environment variable if you want to be able to conveniently run the executable (javac.exe, java.exe, javadoc.exe, and so on) from any directory without having to type the full path of the command. If you do not set the PATH variable, you need to specify the full path to the executable every time you run it, such as:

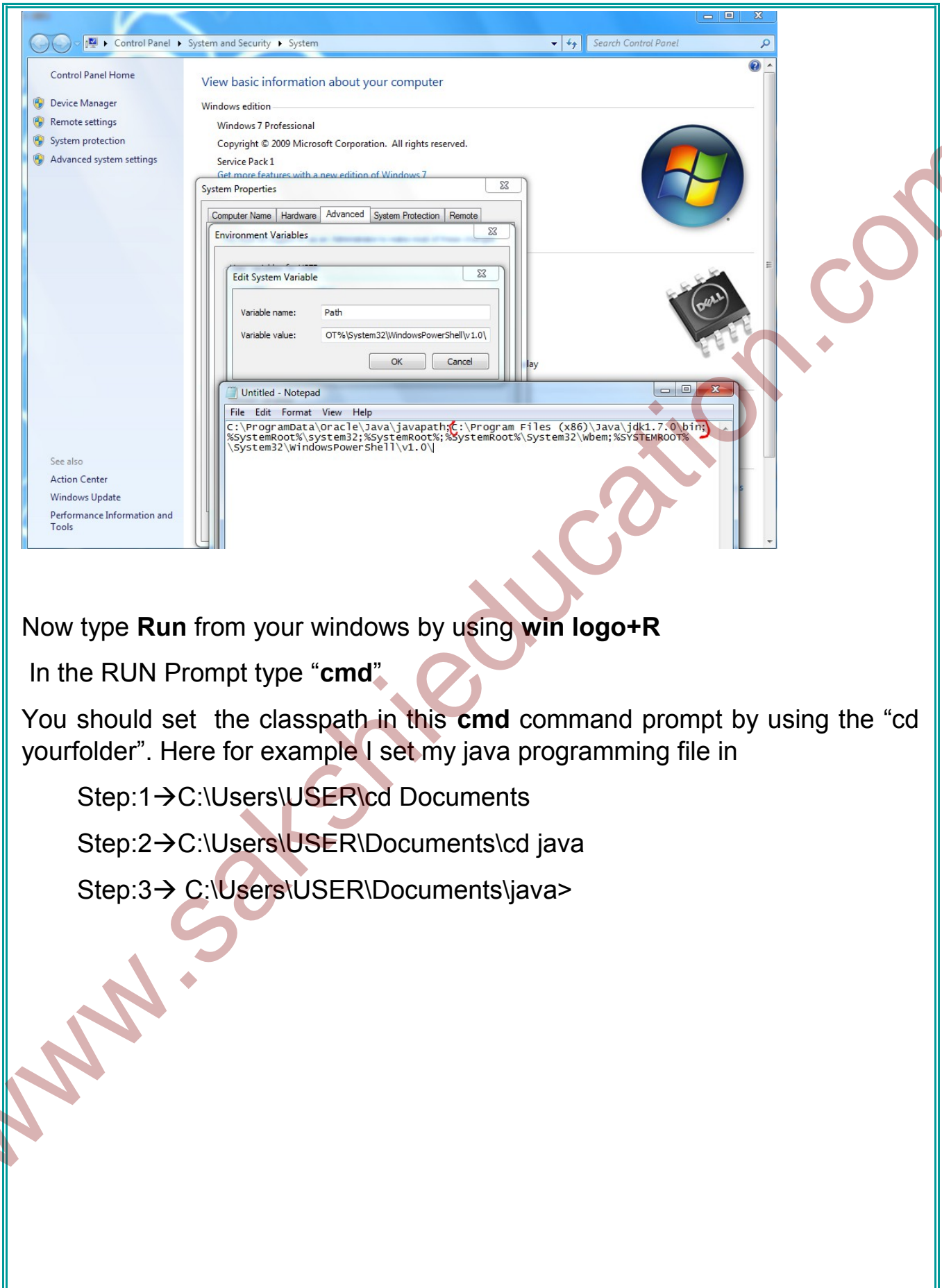
C:\Java\jdk1.7.0\bin\javac Hello.java

For setting environment variables in windows:

Steps to follow:

1. From the desktop, right click the **Computer** icon.
2. Choose **Properties** from the context menu.
3. Click the **Advanced system settings** link.
4. Click **Environment Variables**. In the section **System Variables**, find the PATH environment variable and select it. Click **Edit**. If the PATH environment variable does not exist, click New.
5. In the **Edit System Variable** (or **New System Variable**) window, specify the value of the PATH environment variable. Click **OK**. Close all remaining windows by clicking **OK**.





Now type **Run** from your windows by using **win logo+R**

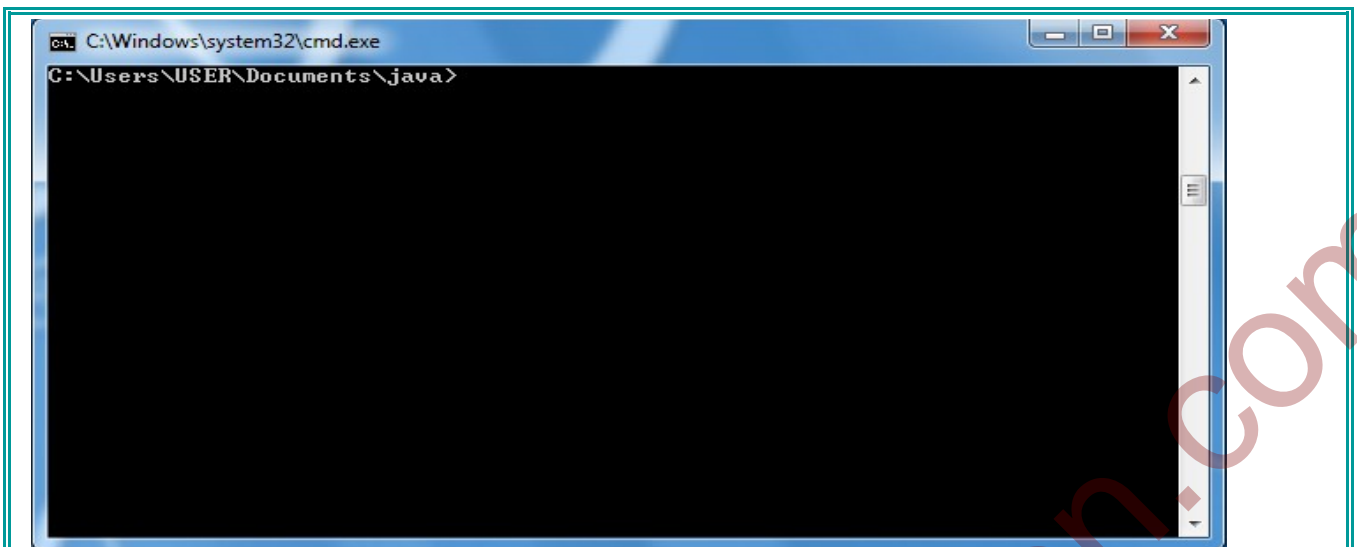
In the RUN Prompt type “**cmd**”

You should set the classpath in this **cmd** command prompt by using the “cd yourfolder”. Here for example I set my java programming file in

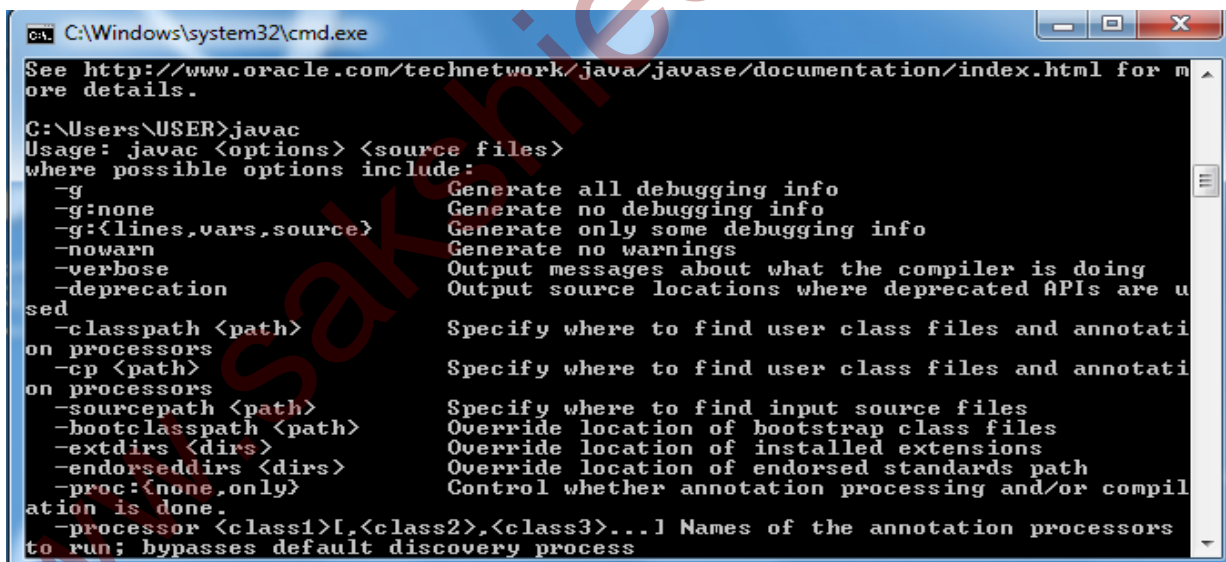
Step:1→C:\Users\USER\cd Documents

Step:2→C:\Users\USER\Documents\cd java

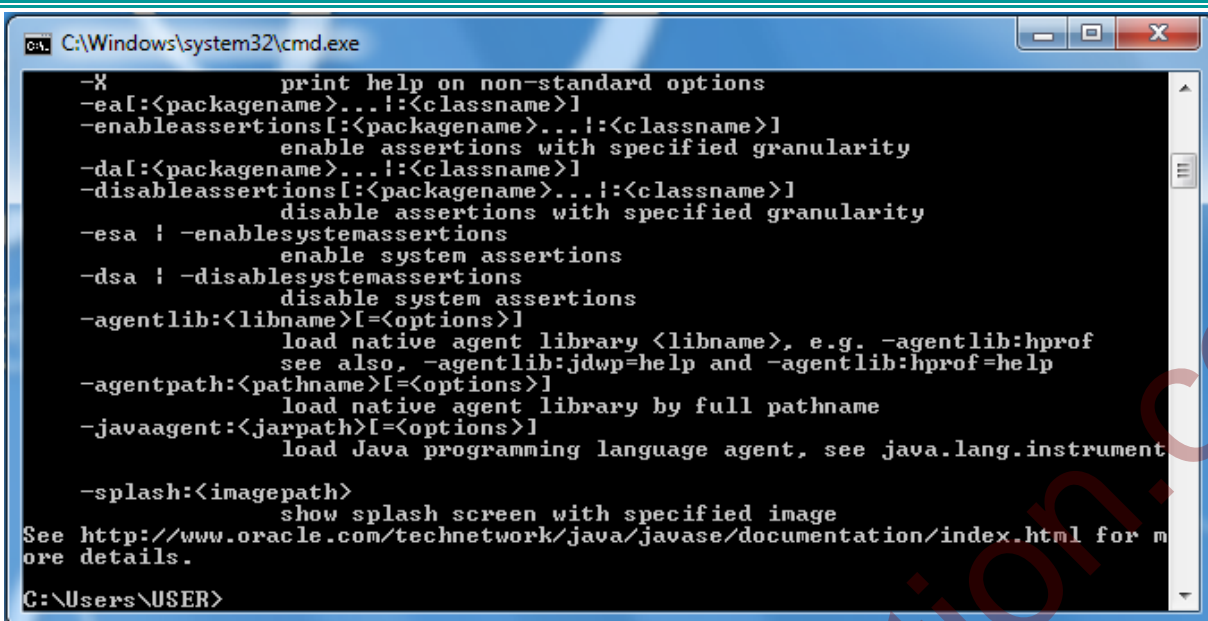
Step:3→ C:\Users\USER\Documents\java>



Note: For checking whether your java programming is running or not, type java in the open command prompt. You can see that after setting the path file the term “**java**” will show the list of source files which are given by Oracle (Previously developed by Sun Micro Systems).



Check whether the your java programming was supported for compiler. Type “javac” which is a java cmd for compiling the program which already set in environment variables in my computer.



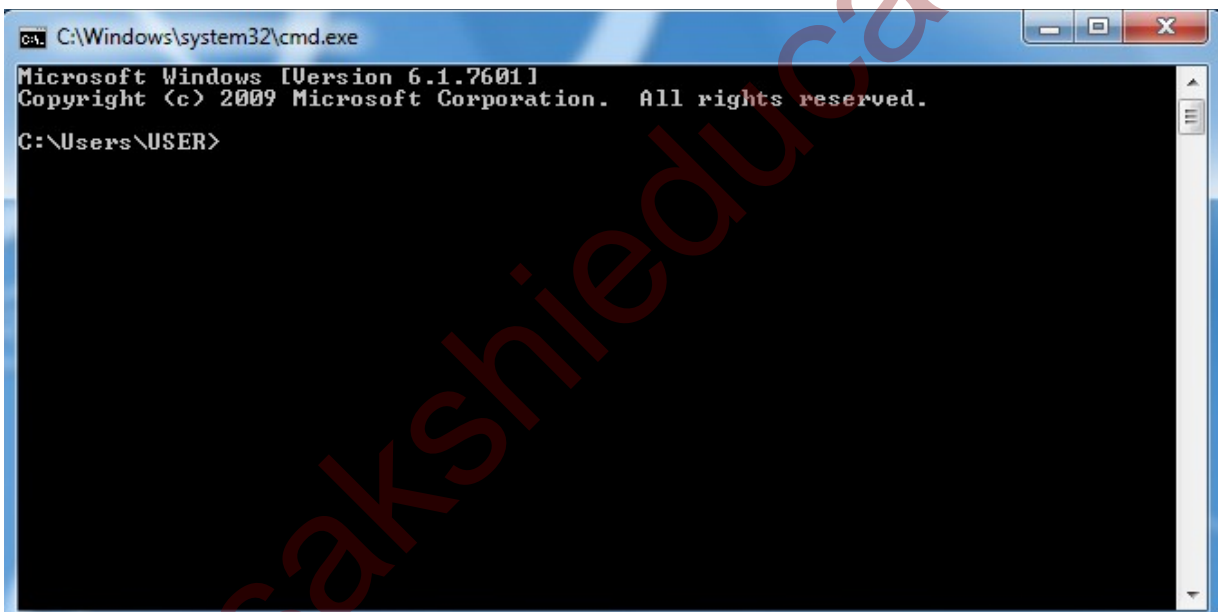
```
C:\Windows\system32\cmd.exe

-X                print help on non-standard options
-ea[:<packagename>...![:<classname>]
-enableassertions[:<packagename>...![:<classname>]
                  enable assertions with specified granularity
-da[:<packagename>...![:<classname>]
-disableassertions[:<packagename>...![:<classname>]
                  disable assertions with specified granularity
-esa : -enablesystemassertions
                  enable system assertions
-dsa : -disablesystemassertions
                  disable system assertions
-agentlib:<libname>[=<options>]
                  load native agent library <libname>, e.g. -agentlib:hprof
                  see also, -agentlib:jdwp=help and -agentlib:hprof=help
-agentpath:<pathname>[=<options>]
                  load native agent library by full pathname
-javaagent:<jarpath>[=<options>]
                  load Java programming language agent, see java.lang.instrument

-splash:<imagepath>
                  show splash screen with specified image

See http://www.oracle.com/technetwork/java/javase/documentation/index.html for more details.

C:\Users\USER>
```



```
C:\Windows\system32\cmd.exe

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

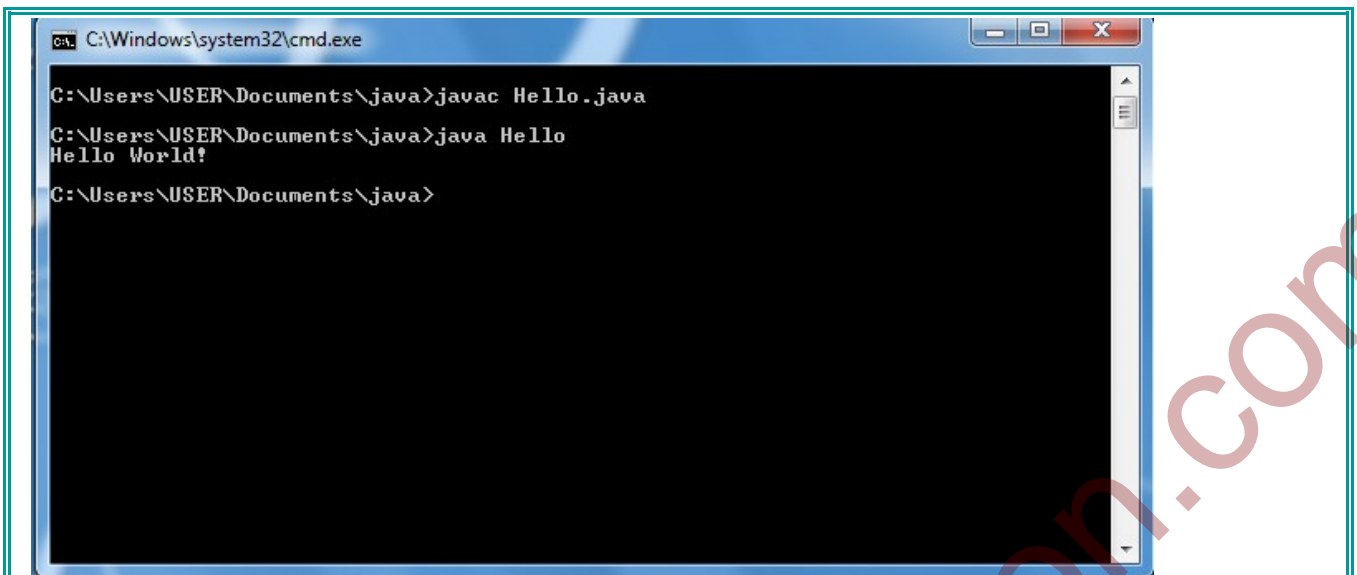
C:\Users\USER>
```

You have set the command prompt temporarily to call the compiler for execution of program and by using “cd folder name” which is a dynamic entry in prompt and now execute the program.

Javac Hello.Java

Java Hello

Ans: **Hello World!**



```
C:\Windows\system32\cmd.exe

C:\Users\USER\Documents\java>javac Hello.java
C:\Users\USER\Documents\java>java Hello
Hello World!
C:\Users\USER\Documents\java>
```

Setting permanent CLASSPATH:

To set class path in Windows:

```
CLASSPATH=c:\myclasses\myclasses.jar
```

This procedure is fine for short-term changes to the system CLASSPATH, but if you want these changes to be persistent you will need to arrange this by yourself as details vary from system to system.

Setting the system CLASSPATH is a useful procedure if you have JAR's full of classes that you use all the time. For example, if I am developing Enterprise Java Bean (EJB) applications using Sun's J2EE 'Reference Implementation', all the EJB related classes are in a JAR called 'j2ee.jar' that comes with the distribution. I want this JAR on the class search path all the time. In addition, most people want to ensure that the current directory is on the search path, whatever the current directory happens to be.

```
CLASSPATH=/usr/j2ee/j2ee.jar:.;export CLASSPATH
```

Here the "." indicates current directory'.

Suppose that a program has been enclosed in a [Jar file](#) called *helloWorld.jar*, put directly in the *D:\myprogram* directory. We have the following file structure:

```
D:\myprogram\
| ---> helloWorld.jar
|---> lib\
```


|---> supportLib.jar

Here we can see there is a .jar file supported in the lib folder.

Main-Class: org.mypackage.HelloWorld

Class-Path: lib/supportLib.jar

It's important that the manifest that the file ends with either a new line or carriage return.

To launch the program, we can use the following command:

```
java -jar D:\myprogram\helloWorld.jar [app arguments]
```

This will automatically start the org.mypackage.HelloWorld specified in the Main-Class with the arguments and user cannot replace this class name using java -jar options. The Class-Path meantime describes the location of the supportLib.jar file relative to the location of the helloWorld.jar. Neither absolute file path (which is permitted in -classpath parameter on the command line) nor jar-internal paths are supported. This particularly means that if main class file is contained in a jar, org/mypackage/HelloWorld.class must be a valid path on the root within the jar.

Multiple class path entries are separated with spaces:

Class-Path: lib/supportLib.jar lib/supportLib2.jar

Example programs:

Each package name has a corresponding directory name. In the following examples, the source files are located at /home/ws/src/java/awt/*.java.

Compiling One or More Packages

To compile a package, the source files (*.java) for that package must be located in a directory having the same name as package. If the package name is made up of several identifiers (separated by dots), each identifier represents a different directory. Thus, all java.awt classes must reside in a directory named java/awt/.

First, change to the parent directory of the top-most package. Then run javac, supplying one or more package names.

```
% cd /home/ws/src/
```

```
% javac java.awt java.awt.event
```

Compiles the public classes in packages java.awt and java.awt.event.

Compiling One or More Classes

Change to the directory holding the class. Then run javac, supplying one or more class names.

```
% cd /home/ws/src/java/awt/
```

```
% javac Button.java Canvas.java
```

Compiles the two classes.