

Structure of Java Program

Java is an object-oriented programming, **platform-independent**, and **secure** programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications. So, before diving in depth, it is necessary to understand the **basic structure of Java program** in detail. In this section, we have discussed the basic **structure of a Java program**. At the end of this section, you will be able to develop the Hello world Java program, easily.

A typical structure of a Java program contains the following elements:

- Documentation Section
- Package Declaration
- Import Statements
- Interface Section
- Class Definition
- Class Variables and Variables
- Main Method Class
- Methods and Behaviours

Documentation Section

The documentation section is an important section but **optional** for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name, and description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. To write the statements in the documentation section, we use **comments**. The comments may be **single-line, multi-line, and documentation** comments.

Single-line Comment: It starts with a pair of forwarding slash (//). For example:

```
//First Java Program
```

Multi-line Comment: It starts with a /* and ends with */. We write between these two symbols. For example:

```
/*It is an example of  
multiline comment*/
```

Documentation Comment: It starts with the delimiter (/**) and ends with */. For example:

```
/**It is an example of documentation comment*/
```

Package Declaration

The package declaration **is optional**. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. Note that there can be **only one package** statement in a Java program. It must be defined before any class and interface declaration. It is necessary because a Java class can be placed in different packages and directories based on the module they are used. For all these classes package belongs to a single parent directory. We use the keyword **package** to declare the package name. For example:

```
package mypack; //where mypack is the package name
```

```
package com.mypack; //where com is the root directory and mypack is the subdirectory
```

Import Statements

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the **import** keyword to import the class. It is written before the class declaration and after the package statement. We use the import statement in two ways, either import a specific class or import all classes of a particular package. In a Java program, we can use multiple import statements. For example:

```
import java.util.Scanner; //it imports the Scanner class only
```

```
import java.util.*; //it imports all the class of the java.util package
```

Interface Section

It is an **optional** section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An **interface** is a slightly different from the class. It contains only **constants** and **method** declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the **implements** keyword. An interface can also be used with other interfaces by using the **extends** keyword. For example:

```

interface car
{
void start();
void stop();
}

```

Class Definition

In this section, we define the class. It is **vital** part of a Java program. Without the **class**, we cannot create any Java program. A Java program may contain more than one class definition. We use the **class** keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the `main()` method. For example:

```

class Student //class definition
{
}

```

Class Variables and Constants

In this section, we define **variables** and **constants** that are to be used later in the program. In a Java program, the variables and constants are defined just after the class definition. The variables and constants store values of the parameters. It is used during the execution of the program. We can also decide and define the scope of variables by using the modifiers. It defines the life of the variables. For example:

```

class Student //class definition
{
    String sname; //variable
    int id;
    double percentage;
}

```

Main Method Class

In this section, we define the **main() method**. It is essential for all Java programs. Because the execution of all Java programs starts from the `main()` method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods. We use the following statement to define the `main()` method:

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

For example:

```
public class Student //class definition
{
    public static void main(String args[])
    {
        //statements
    }
}
```

Methods and behaviour

In this section, we define the functionality of the program by using the **methods**. The methods are the set of instructions that we want to perform. These instructions execute at runtime and perform the specified task. For example:

```
public class Demo //class definition
{
    public static void main(String args[])
    {
        void display()
        {
            System.out.println("Hello World.....!");
        }
        //statements
    }
}
```

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity** – Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.

- **Class Names** – For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

Example: *class MyFirstJavaClass*

- **Method Names** – All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

Example: *public void myMethodName()*

- **Program File Name** – Name of the program file should exactly match the class name.

When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

But please make a note that in case you do not have a public class present in the file then file name can be different than class name. It is also not mandatory to have a public class in the file.

Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as '*MyFirstJavaProgram.java*'

- **public static void main(String args[])** – Java program processing starts from the main() method which is a mandatory part of every Java program.

First Java Program | Hello World Example

We can write a simple hello Java program easily after installing the JDK.

To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.

The requirement for Java Hello World Example

For executing any Java program, the following software or application must be properly installed.

- Install the JDK if you don't have installed it, and install it.
- Set path of the jdk/bin directory
- Create the Java program
- Compile and run the Java program

Creating Hello World Example

Let's create the hello java program:

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

Save the above file as Simple.java.

To compile:

javac Simple.java

To execute:

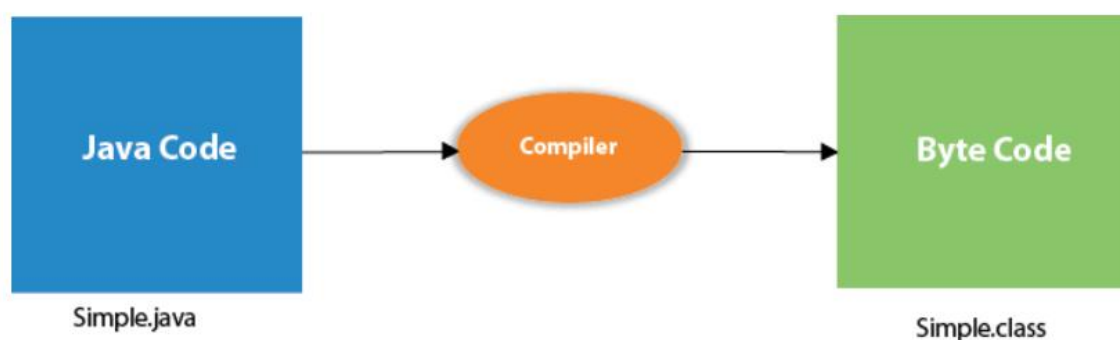
java Simple

Output:

Hello Java

Compilation Flow:

When we compile Java program using javac tool, the Java compiler converts the source code into byte code.



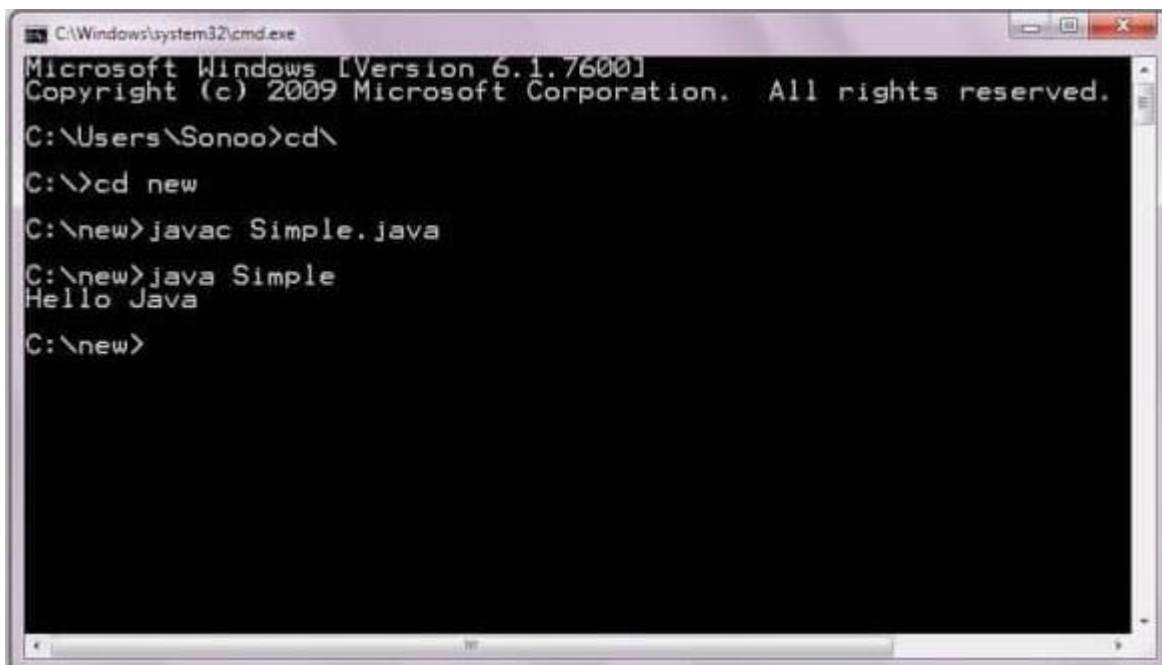
Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.

- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class.

As displayed in the above diagram, write the simple program of Java in notepad and saved it as Simple.java. In order to compile and run the above program, you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt**. When we have done with all the steps properly, it shows the following output:



```

C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>cd\
C:\>cd new
C:\new>javac Simple.java
C:\new>java Simple
Hello Java
C:\new>
  
```

To compile and run the above program, go to your current directory first; my current directory is c:\new. Write here:

To compile:

javac Simple.java

To execute:

java Simple

In how many ways we can write a Java program?

There are many ways to write a Java program. The modifications that can be done in a Java program are given below:

1) By changing the sequence of the modifiers, method prototype is not changed in Java.

Let's see the simple code of the main method.

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

2) The subscript notation in the Java array can be used after type, before the variable or after the variable.

Let's see the different codes to write the main method.

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

3) Having a semicolon at the end of class is optional in Java.

Let's see the simple code.

```
class A{
    static public void main(String... args){
        System.out.println("hello java4");
    };
}
```

Valid Java main() method signature

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


Invalid Java main() method signature

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

Resolving an error "javac is not recognized as an internal or external command"?

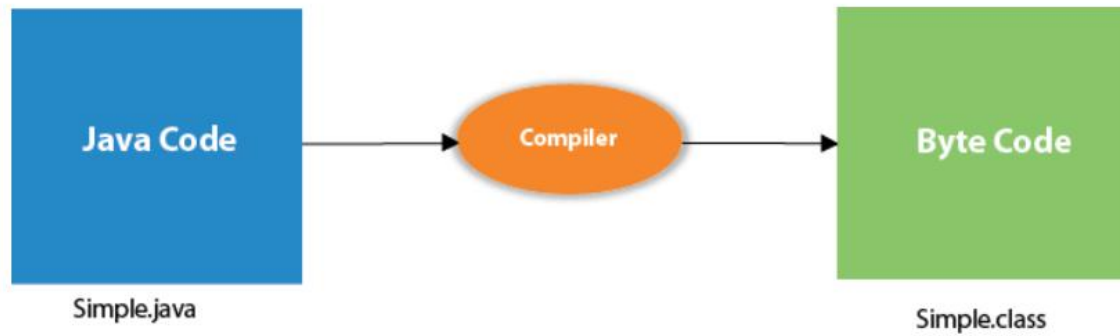
If there occurs a problem like displayed in the below figure, you need to set a path. Since DOS doesn't recognize javac and java as internal or external command. To overcome this problem, we need to set a path. The path is not required in a case where you save your program inside the JDK/bin directory. However, it is an excellent approach to set the path.



Internal Details of Hello Java Program

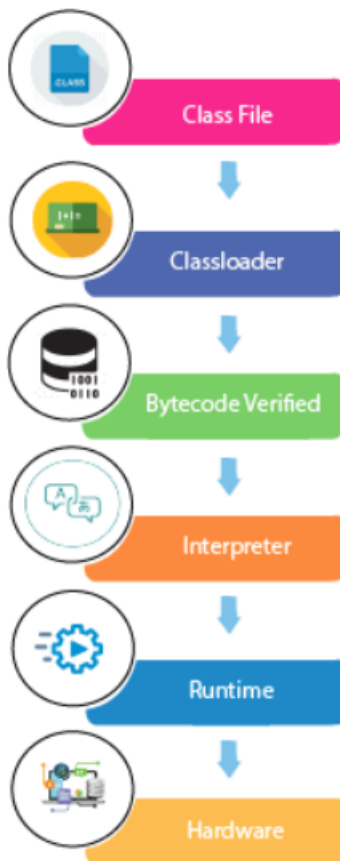
What happens at compile time?

At compile time, the Java file is compiled by Java Compiler (It does not interact with OS) and converts the Java code into bytecode.



What happens at runtime?

At runtime, the following steps are performed:



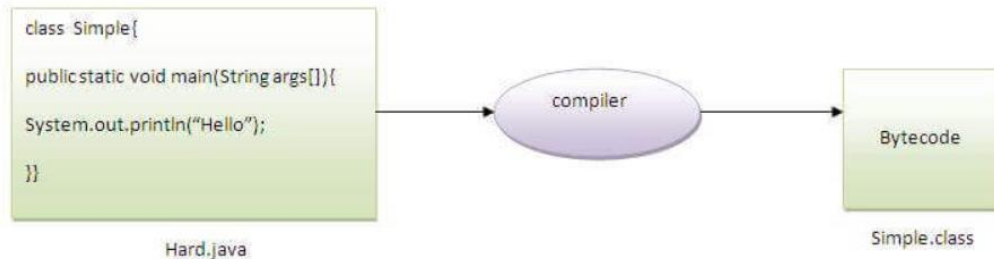
Classloader: It is the subsystem of JVM that is used to load class files.

Bytecode Verifier: Checks the code fragments for illegal code that can violate access rights to objects.

Interpreter: Read bytecode stream then execute the instructions.

Q) Can you save a Java source file by another name than the class name?

Yes, if the class is not public. It is explained in the figure given below:



To compile:

javac Hard.java

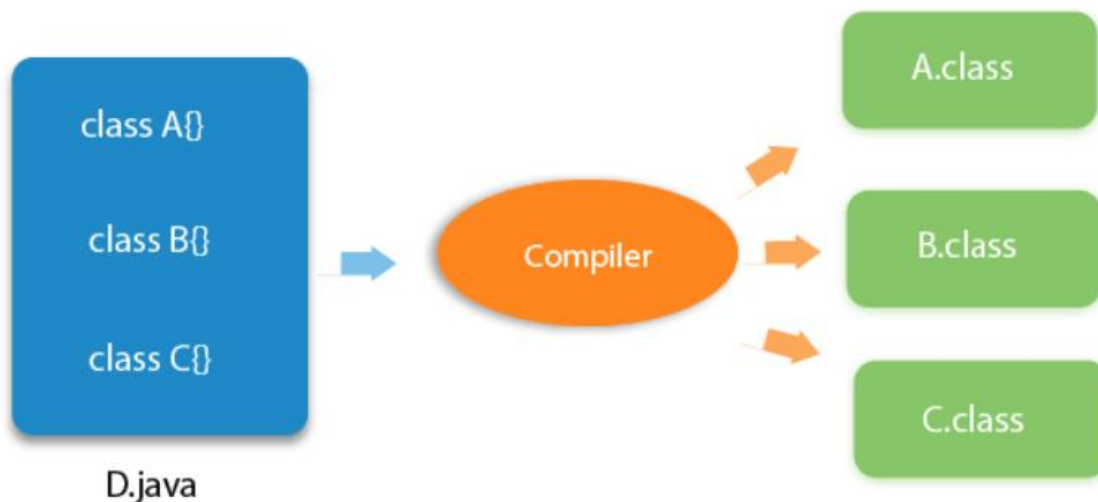
To execute:

java Simple

Observe that, we have compiled the code with file name but running the program with class name. Therefore, we can save a Java program other than class name.

Q) Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:

**How to set path in Java**

The path is required to be set for using tools such as javac, java, etc.

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

1) How to set the Temporary Path of JDK in Windows

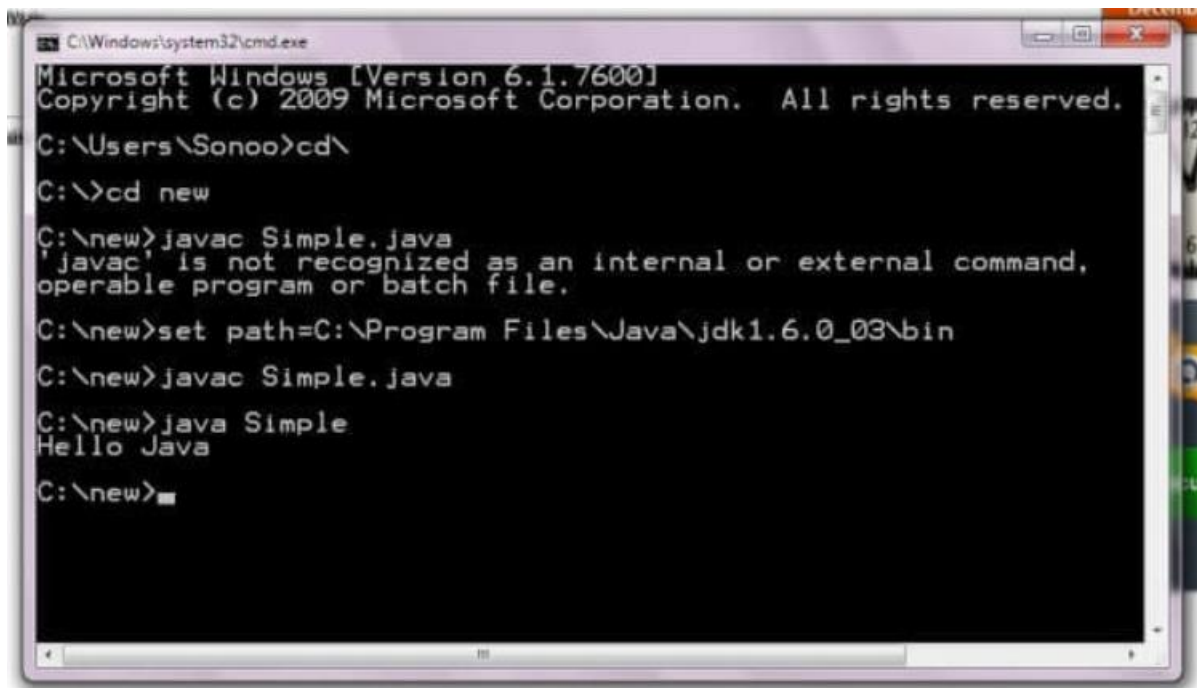
To set the temporary path of JDK, you need to follow the following steps:

- Open the command prompt
- Copy the path of the JDK/bin directory
- Write in command prompt: set path=copied_path

For Example:

```
set path=C:\Program Files\Java\jdk1.6.0_23\bin
```

Let's see it in the figure given below:



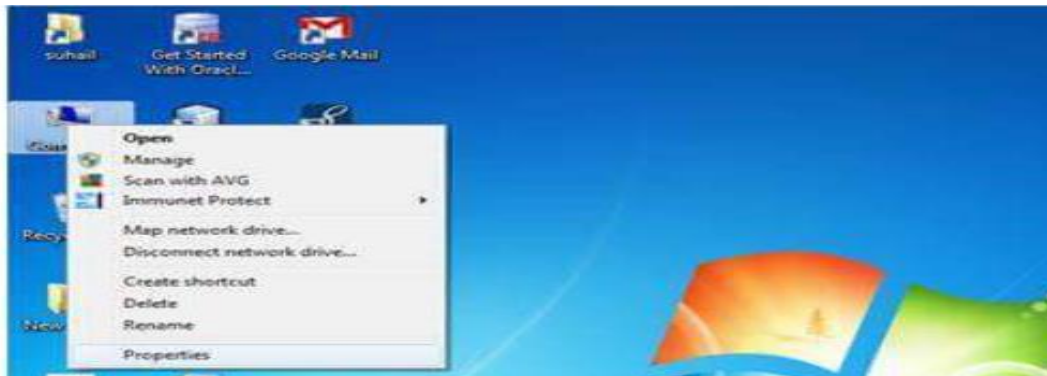
2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

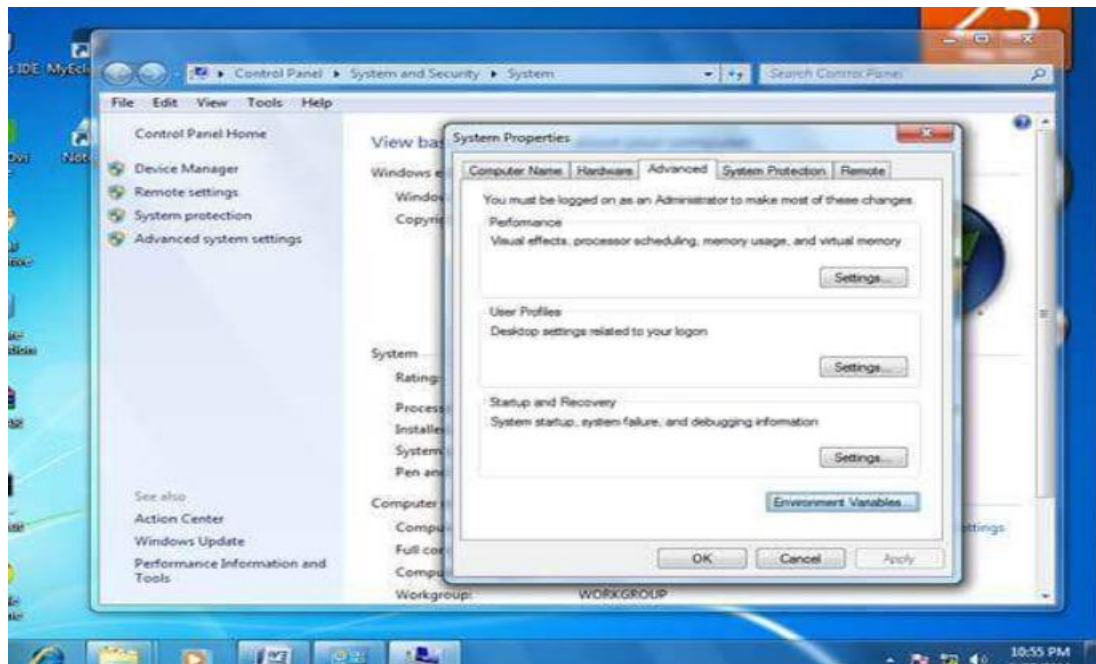
1) Go to MyComputer properties



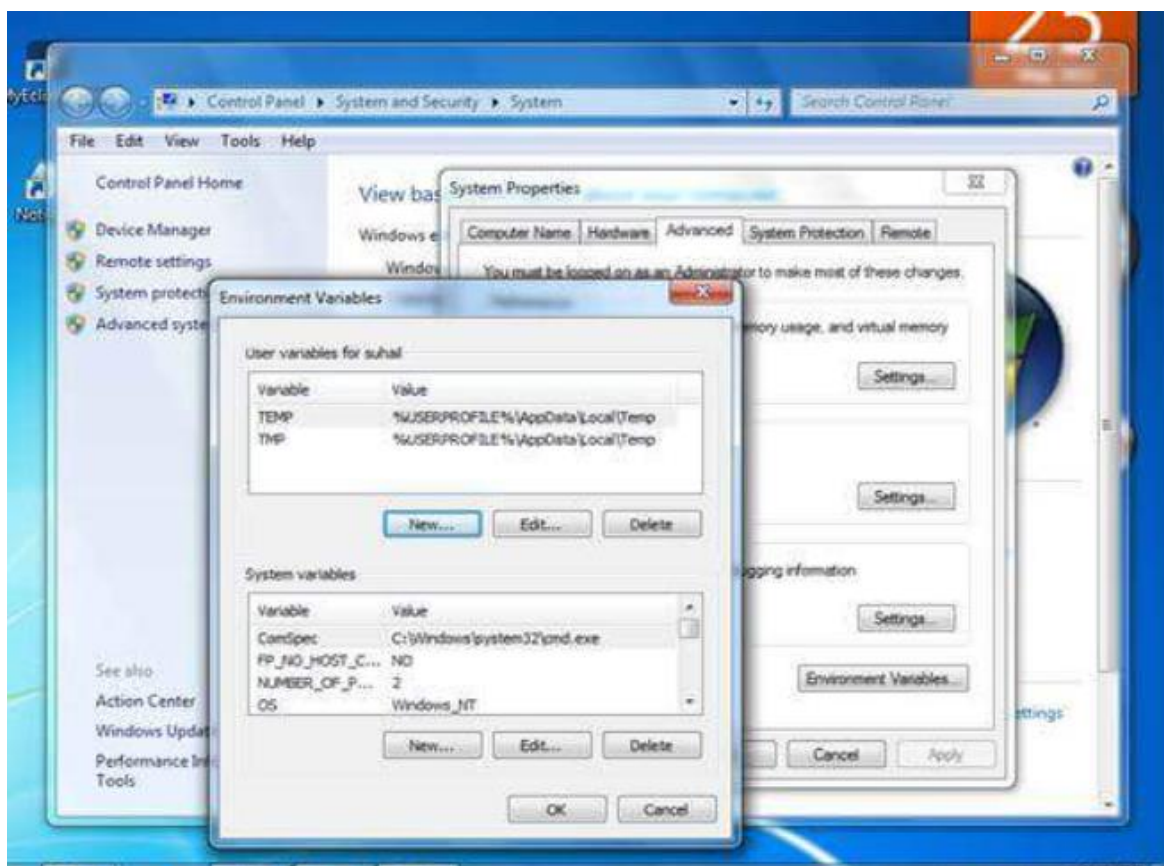
2) Click on the advanced tab

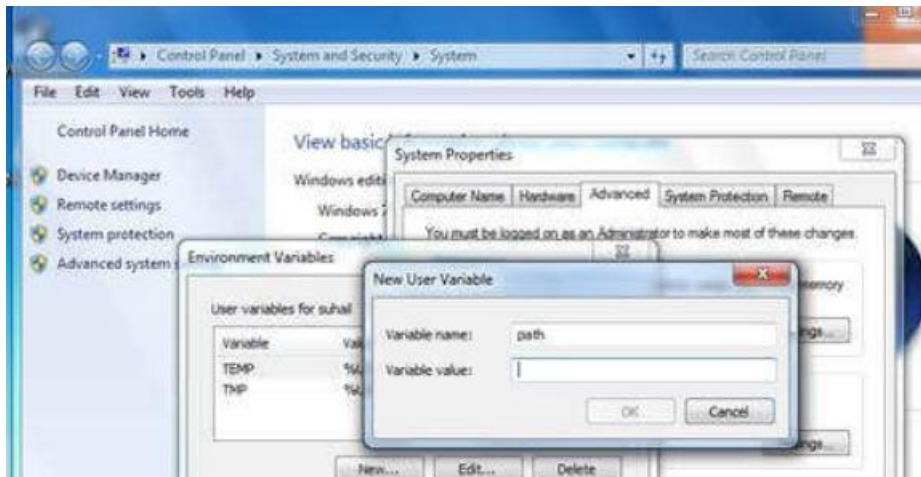
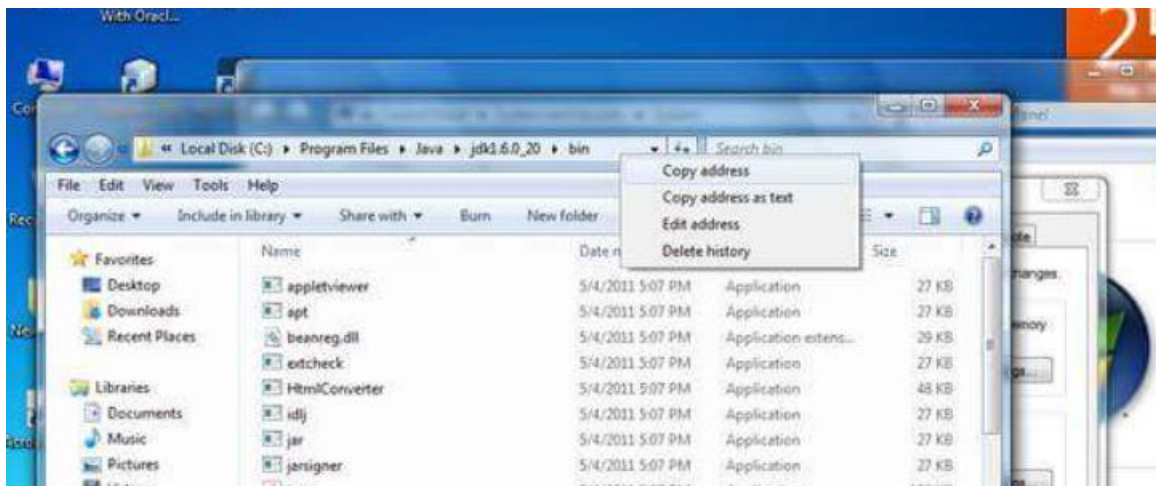
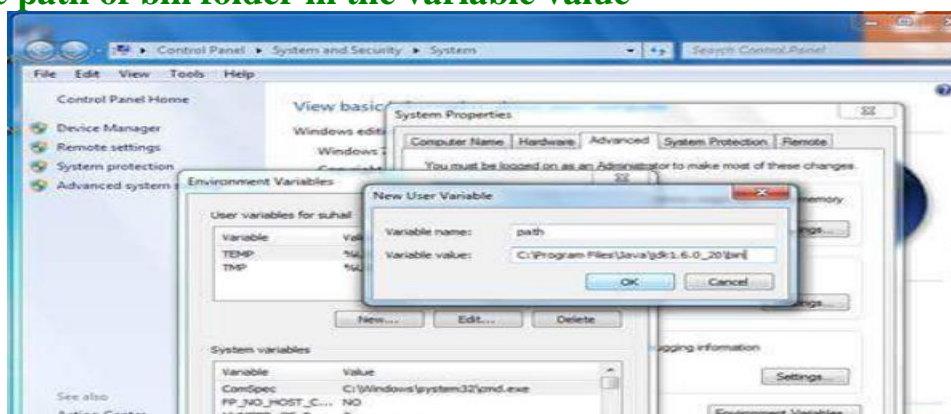


3) Click on environment variables



4) Click on the new tab of user variables



5) Write the path in the variable name**6) Copy the path of bin folder****7) Paste path of bin folder in the variable value****8) Click on ok button****9) Click on ok button**

Now your permanent path is set. You can now execute any program of java from any drive.

How to Set CLASSPATH in Java

CLASSPATH: CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files. The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH.

You need to set the CLASSPATH if:

- You need to load a class that is not present in the current directory or any sub-directories.
- You need to load a class that is not in a location specified by the extensions mechanism.

The CLASSPATH depends on what you are setting the CLASSPATH. The CLASSPATH has a directory name or file name at the end. The following points describe what should be the end of the CLASSPATH.

- If a JAR or zip, the file contains class files, the CLASSPATH end with the name of the zip or JAR file.
- If class files placed in an unnamed package, the CLASSPATH ends with the directory that contains the class files.
- If class files placed in a named package, the CLASSPATH ends with the directory that contains the root package in the full package name, that is the first package in the full package name.

If CLASSPATH finds a class file which is present in the current directory, then it will load the class and use it, irrespective of the same name class presents in another directory which is also included in the CLASSPATH.

If you want to set multiple classpaths, then you need to separate each CLASSPATH by a semicolon (;).

Step 1: Click on the Windows button and choose Control Panel. Select System.



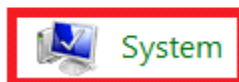
Recovery



Region and Language

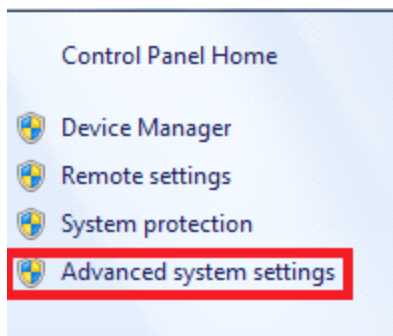


Sync Center

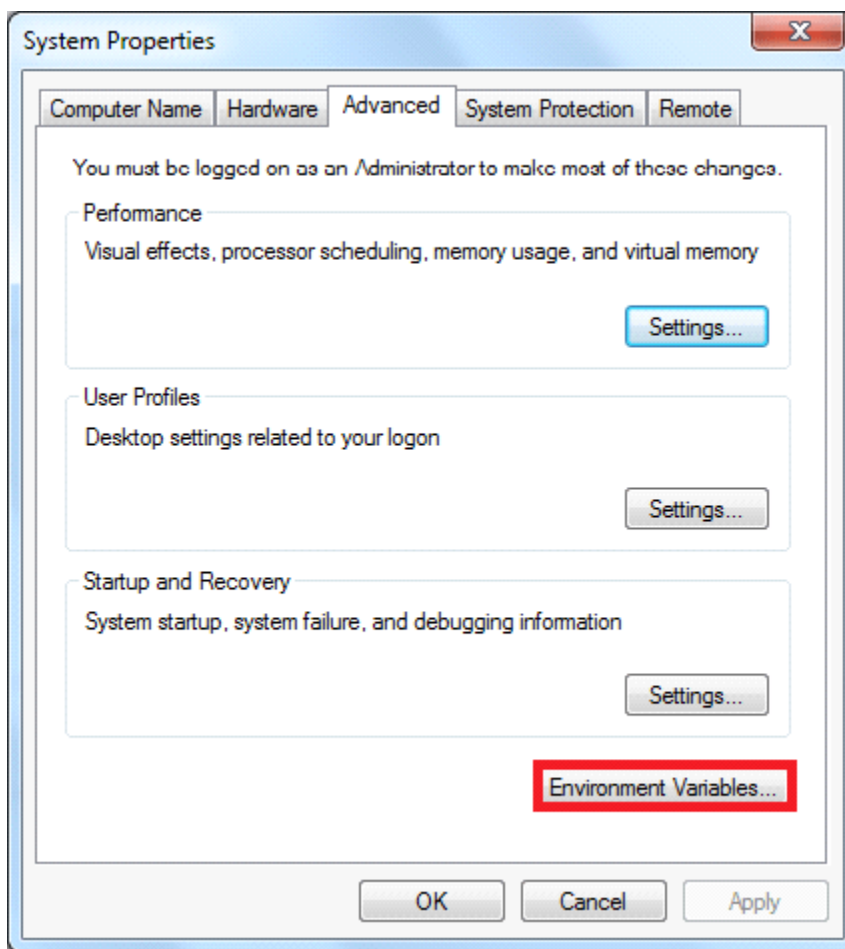


System

Step 2: Click on **Advanced System Settings**.



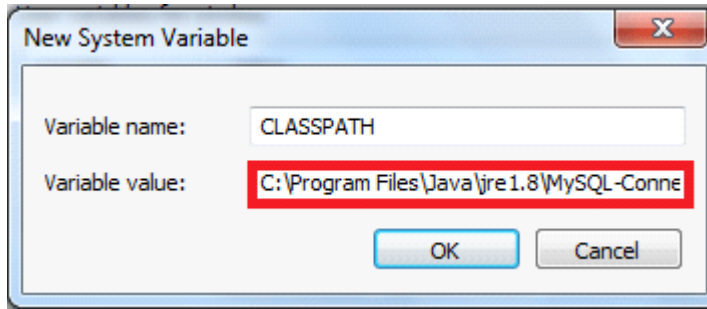
Step 3: A dialog box will open. Click on Environment Variables.



Step 4: If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

If the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as *C:\Program Files\Java\jre1.8\MySQL-Connector Java.jar;;*

Remember: Put ;, at the end of the CLASSPATH.



Difference between PATH and CLASSPATH

PATH	CLASSPATH
PATH is an environment variable.	CLASSPATH is also an environment variable.
It is used by the operating system to find the executable files (.exe).	It is used by Application ClassLoader to locate the .class file.
You are required to include the directory which contains .exe files.	You are required to include all the directories which contain .class and JAR files.
PATH environment variable once set, cannot be overridden.	The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command.

Difference between JDK, JRE, and JVM

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

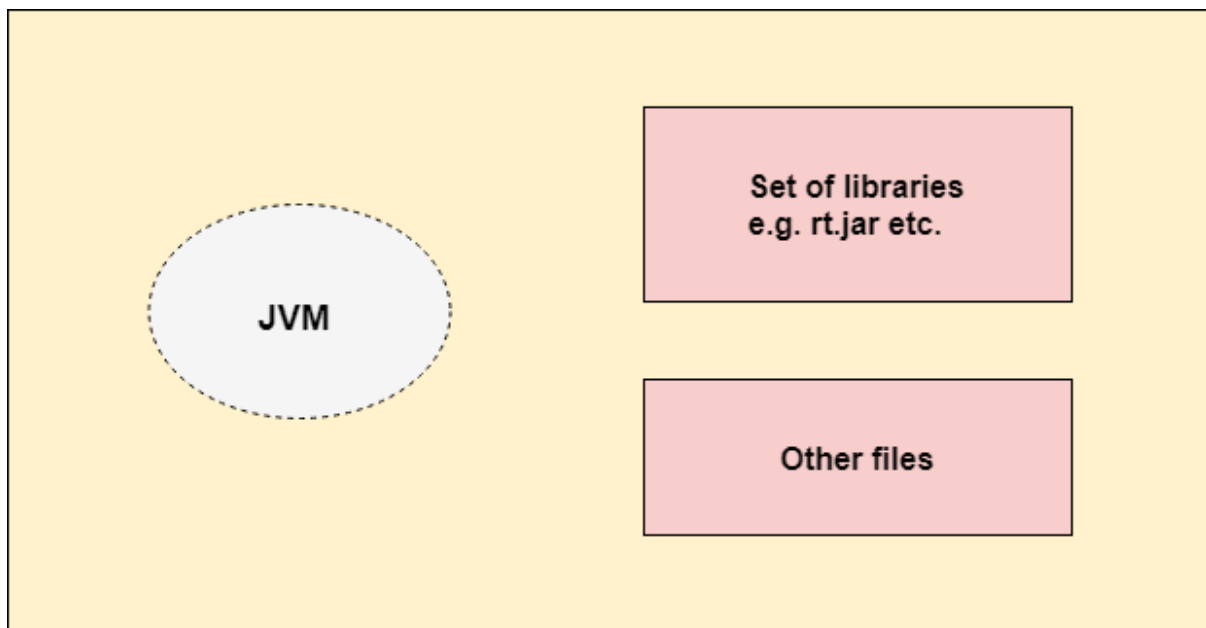
The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JRE

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets.

It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

