Table of Contents

[**1. JAVA Basics – Starting with Java** 4](#_Toc12814680)

[**1A. Java Basics – Interview Questions** 23](#_Toc12814681)

[**2. What is Class and Object in Java?** 24](#_Toc12814682)

[**2A. Java Basics – Class and Object Interview Questions** 26](#_Toc12814683)

[**3. Java Variables** 28](#_Toc12814684)

[**3A. Java Variables – Interview Questions – Part 1** 30](#_Toc12814685)

[**3B. Java Variables – Interview Questions – Part 2** 32](#_Toc12814686)

[**4. Java Data Types** 33](#_Toc12814687)

[**4A. Java Data Types – Interview Questions – Part 1** 35](#_Toc12814688)

[**4B. Java Data Types – Interview Questions – Part 2** 36](#_Toc12814689)

[**5. Java Operators - Arithmatic Operators and Assignment Operators** 39](#_Toc12814690)

[**5A. Java Operators - Auto Increment and Decrement Operators and Logical Operators** 40](#_Toc12814691)

[**5B. Java Operators - Comparison Operators and Ternary Operator** 41](#_Toc12814692)

[**5C. Java Operators - Interview Questions** 42](#_Toc12814693)

[**6. Java Methods - Declaring Methods, Passing Values and Returning Values** 45](#_Toc12814694)

[**6A. Java Methods - Passing integers and returning integers** 46](#_Toc12814695)

[**6B. Java Methods - Passing Strings and Returning Strings** 47](#_Toc12814696)

[**6C. Java Methods - Passing Arrays and returning Arrays** 48](#_Toc12814697)

[**6D. Java Methods - Passing and Returning an object** 49](#_Toc12814698)

[**6E. Java Methods - Interview Questions** 50](#_Toc12814699)

[**7. Java Basics - Method Overloading** 51](#_Toc12814700)

[**7A. Java Basics - Method Overloading - Interview Questions** 52](#_Toc12814701)

[**8. Java Constructors - Constructor Uses** 53](#_Toc12814702)

[**8A. Java Constructors - Interview Questions** 54](#_Toc12814703)

[**9. Java Basics - Properties Class** 56](#_Toc12814704)

[**10. Access Modifiers default, public, protected and Private** 57](#_Toc12814705)

[**10A. Access Modifiers - Interview Questions** 58](#_Toc12814706)

[**11. Java Non Access Modifiers - Static variables, methods, blocks and classes** 61](#_Toc12814707)

[**11A. Java Non Access Modifiers - Static Variables, Methods, Blocks and Classes - Interview Questions – Part 1** 64](#_Toc12814708)

[**11B. Java Non Access Modifiers - Static Variables, Methods, Blocks and Classes - Interview Questions – Part B** 67](#_Toc12814709)

[**12. Java Non Access Modifiers - Abstract Classes and Methods** 69](#_Toc12814710)

[**12A. Java Non Access Modifiers - Abstract Classes and Methods - Interview Questions** 71](#_Toc12814711)

[**13. Java Non Access Modifiers - Final Keyword** 73](#_Toc12814712)

[**13A. Java Non Access Modifiers - Final Keyword - Interview Questions** 74](#_Toc12814713)

[**14. Java Interfaces** 76](#_Toc12814714)

[**14A. Java Interfaces – Key Points** 77](#_Toc12814715)

[**14B. Java Interfaces - Interview Questions** 80](#_Toc12814716)

[**15. Java Inheritance** 82](#_Toc12814717)

[**15A. Constructors and Inheritance** 83](#_Toc12814718)

[**15B. Inheritance and Method Overriding** 84](#_Toc12814719)

[**15C. Different Types of Inheritance** 84](#_Toc12814720)

[**15D. Inheritance - Interview Questions** 86](#_Toc12814721)

[**16. Java Polymorphism** 87](#_Toc12814722)

[**16A. Java Polymorphism - Interview Questions** 89](#_Toc12814723)

[**17. Java Super Keyword** 90](#_Toc12814724)

[**17A. Java Basics - Accessing Parent Class Variables with Super Keyword** 90](#_Toc12814725)

[**17B. Java Basics - Accessing Parent Class No-Arg Constructor** 91](#_Toc12814726)

[**17C. Accessing Parent Class Argument Constructor** 92](#_Toc12814727)

[**17D. Super Keyword When Method Overriding** 93](#_Toc12814728)

[**17E. Accessing a Parent Method That is not Overridden** 93](#_Toc12814729)

[**17F. Java Super Keyword - Interview Questions** 94](#_Toc12814730)

[**18. This Keyword - Accessing Class Variables** 96](#_Toc12814731)

[**18A. This Keyword - Accessing Instance Variables** 97](#_Toc12814732)

[**18B. This Keyword - Accessing a Method from Another Method in the same Class** 98](#_Toc12814733)

[**18C. This Keyword - Accessing a Constructor from another Constructor in the same Class** 98](#_Toc12814734)

[**18D. Java This Keyword - Interview Questions** 99](#_Toc12814735)

[**19. Java If Else Conditions - Part 1** 100](#_Toc12814736)

[**19A. Java If Else - Using Different Comparison Operators** 102](#_Toc12814737)

[**19B. Java If Else - Using Different Logical Operators** 104](#_Toc12814738)

[**19C. Java Switch Case Statements** 106](#_Toc12814739)

[**19D. Java Switch Case Statements - Several Values Executing Same Code** 108](#_Toc12814740)

[**19E. Java Switch Case Statements - Without Break Statement** 109](#_Toc12814741)

[**19F. Java If Else and Switch Case - Interview Questions** 110](#_Toc12814742)

[**20. Selenium - Java For Loops and For-Each Loops** 110](#_Toc12814743)

[**20A. Java For Loops - Break and Labelled Break** 113](#_Toc12814744)

[**20B. Java For Loops - Continue and Labelled Continue** 114](#_Toc12814745)

[**20C. Java While Loops and Inner While Loops** 115](#_Toc12814746)

[**20D. Java While Loops - Break and Labelled Break** 116](#_Toc12814747)

[**20E. Java While Loops - Continue and Labelled Continue** 117](#_Toc12814748)

[**20F. Java Do While Loops and Inner Do While Loops** 118](#_Toc12814749)

[**20G. Java Do While Loops - Break and Labelled Break** 119](#_Toc12814750)

[**20H. Java Do While Loops - Continue and Labelled Continue** 120](#_Toc12814751)

[**20I. Java Loops - Difference between While Loop and Do While Loop** 122](#_Toc12814752)

[**20J. Java Loops - Interview Questions** 123](#_Toc12814753)

[**21. Java Strings - What are Strings and how do they work** 124](#_Toc12814754)

[**21A. Java Strings - String Methods** 125](#_Toc12814755)

[**21B. Java Strings - Comparing and Concatenating Two Strings** 127](#_Toc12814756)

[**21C. Java Strings - Checking if a String is Substring of other String** 128](#_Toc12814757)

[**21D. Java Strings - Interview Questions** 129](#_Toc12814758)

[**22. Java Arrays** 132](#_Toc12814759)

[**22A. Java Arrays - Passing an Array to a Method** 134](#_Toc12814760)

[**22B. Java Arrays - Returning an Array from a Method** 134](#_Toc12814761)

[**22C. Java Arrays - ArrayIndexOutOfBounds Exception** 134](#_Toc12814762)

[**22D. Java Arrays – ArrayStoreException** 135](#_Toc12814763)

[**22E. Java Arrays - Iterating Over an Array** 135](#_Toc12814764)

[**22F. Java Arrays - Sorting Elements in an Array** 136](#_Toc12814765)

[**22G. Java Arrays - Searching an Array** 137](#_Toc12814766)

[**22H. Java Arrays - Copying an Array using Object.clone()** 138](#_Toc12814767)

[**22I. Java Arrays - Copying an Array using System.arraycopy()** 139](#_Toc12814768)

[**22J. Java Arrays - Copying an Array using Arrays.copyOf()** 139](#_Toc12814769)

[**22K. Java Arrays - Copying an Array using Arrays.copyOfRange()** 140](#_Toc12814770)

[**22L. Java Arrays - Multi Dimensional Arrays** 140](#_Toc12814771)

[**22M. Java Arrays - Interview Questions** 141](#_Toc12814772)

[**23. Collections Framework – Introduction** 143](#_Toc12814773)

[**23A. Collections Framework – ArrayList** 145](#_Toc12814774)

[**23B. Collections Framework - Iterating through an ArrayList** 146](#_Toc12814775)

[**23C. Collections Framework - Adding a List to another List** 147](#_Toc12814776)

[**23D. Collections Framework - Removing a List from Another List** 148](#_Toc12814777)

[**23E. Collections Framework - retainAll()** 149](#_Toc12814778)

[**23F. Collections Framework - List of Lists** 149](#_Toc12814779)

[**23G. Collections Framework - Sorting a list** 150](#_Toc12814780)

[**23H. Collections Framework – LinkedList** 150](#_Toc12814781)

[**23I. Collections Framework – HashSet** 152](#_Toc12814782)

[**23J. Collections Framework – HashMap** 154](#_Toc12814783)

[**23K. Collections Framework - Map of Maps** 155](#_Toc12814784)

[**23L. Collections Framework – ConcurrentHashMap** 156](#_Toc12814785)

[**23M. Collections Framework - Interview Questions** 157](#_Toc12814786)

[**24. Exceptions - What are exceptions and how do we work with them** 163](#_Toc12814787)

[**24A. Exceptions - Try Catch Blocks** 165](#_Toc12814788)

[**24B. Exceptions - Throws Keyword** 165](#_Toc12814789)

[**24C. Exceptions - Throw Keyword** 166](#_Toc12814790)

[**24D. Exceptions - Interview Questions** 167](#_Toc12814791)

[**25. Java File Operations - Part 1 - Java Input Output and File Path Operations** 172](#_Toc12814792)

[**25A. Java File Operations - Part 2 - Java File and Directory Path Operations Example** 174](#_Toc12814793)

[**25B. Java File Operations - Part 3 - FileWriter and FileReader** 175](#_Toc12814794)

[**25C. Java File Operations - File Input and Output Interview Questions** 176](#_Toc12814795)

[**26. Java Database Connectivity - Installing XAMPP MySql Database On Local Machine** 178](#_Toc12814796)

[**26A. Java Database Connectivity - Basic SQL Knowledge** 184](#_Toc12814797)

[**26B. Java Database Connectivity - Connecting to Database and Execute Queries using Java** 187](#_Toc12814798)

[**27. Java Reflections - Loading classes, constructors and methods at run time.** 190](#_Toc12814799)

[**28. Java Date Functionality - Getting Current Date and Future Dates** 192](#_Toc12814800)

# **1. JAVA Basics – Starting with Java**

Java is a programming language and computing platform first released by Sun Microsystems in 1995.

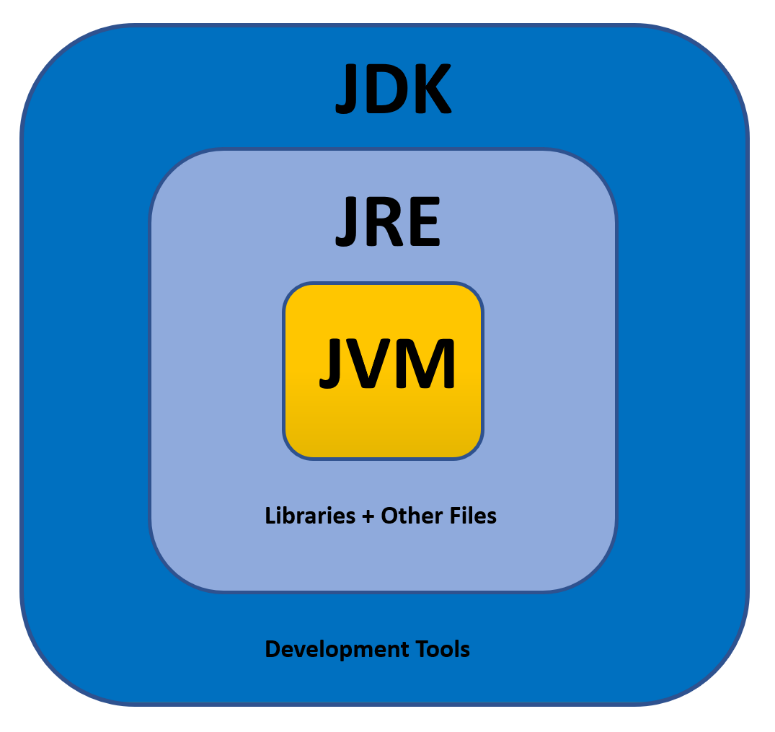
When we create java programs and compile them class files will be created. These class files contain byte code.

Java contains three things mainly.

JVM – Java Virtual Machine

JRE – Java Runtime Environment

JDK – Java Development Kit

****

JVM (Java Virtual Machine):

The JVM performs following main tasks:  
Loads code  
Verifies code  
Executes code

JRE (Java Runtime Environment):

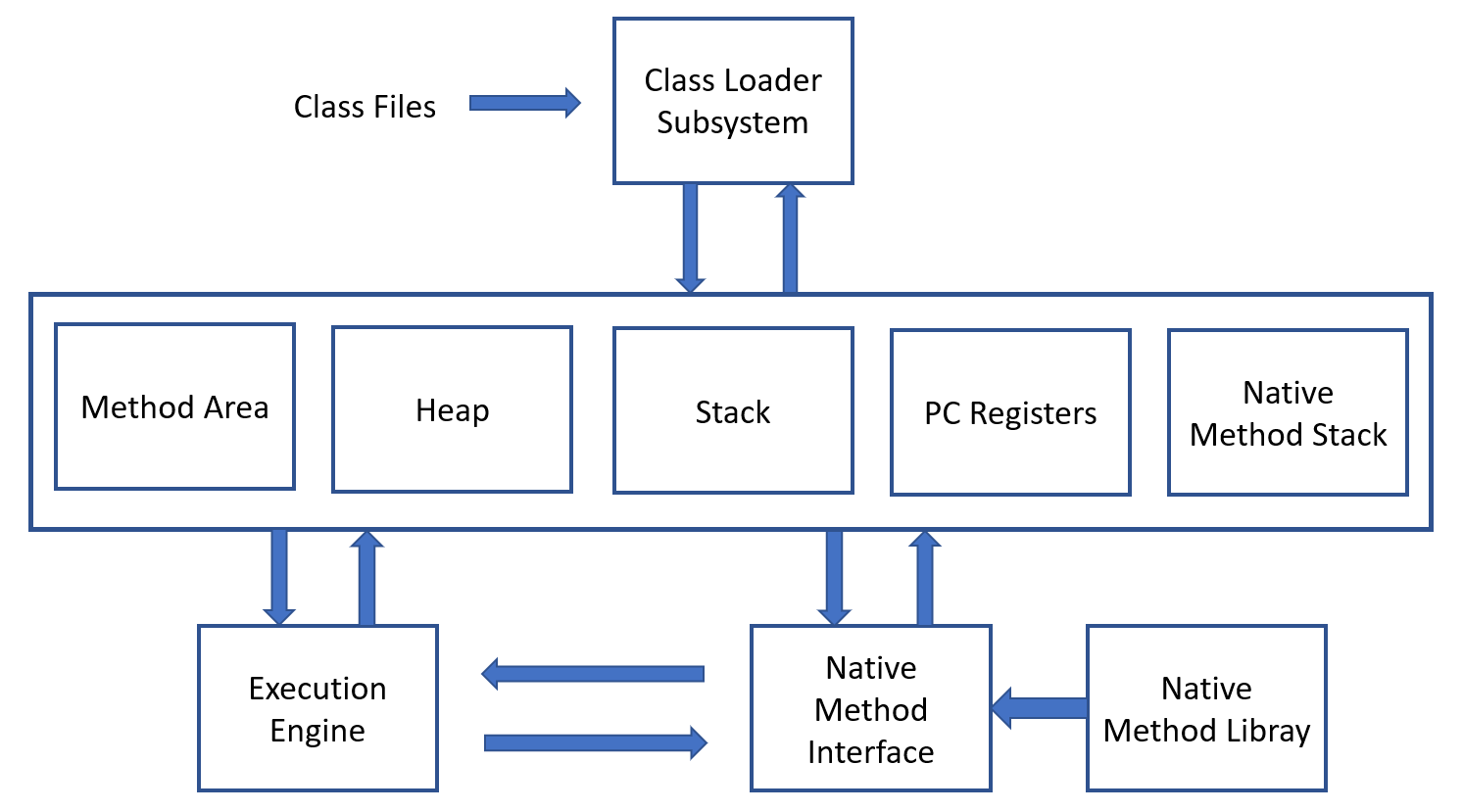
JRE is used to provide runtime environment. It contains set of libraries + other files JVM uses at runtime.

JDK (Java Development Kit):

JDK is java development kit. It contains JRE + development tools.

JDK, JRE and JVM are all platform dependent. But java is platform independent.

JVM Architecture:



The class Loader Subsystem takes a class file as input and perform the following operation.

Method Area: It stores all the class code, method code and class variables declared.

Heap Memory: It stores all the objects created. (Also the instance variables)

JVM stack Area: It stores the information of methods at run time. For example, the method variables initiated at run time. It stores local variables also. It stores references to the objects created in Heap memory.

PC Registers: It stores the address of next information that has to be executed.

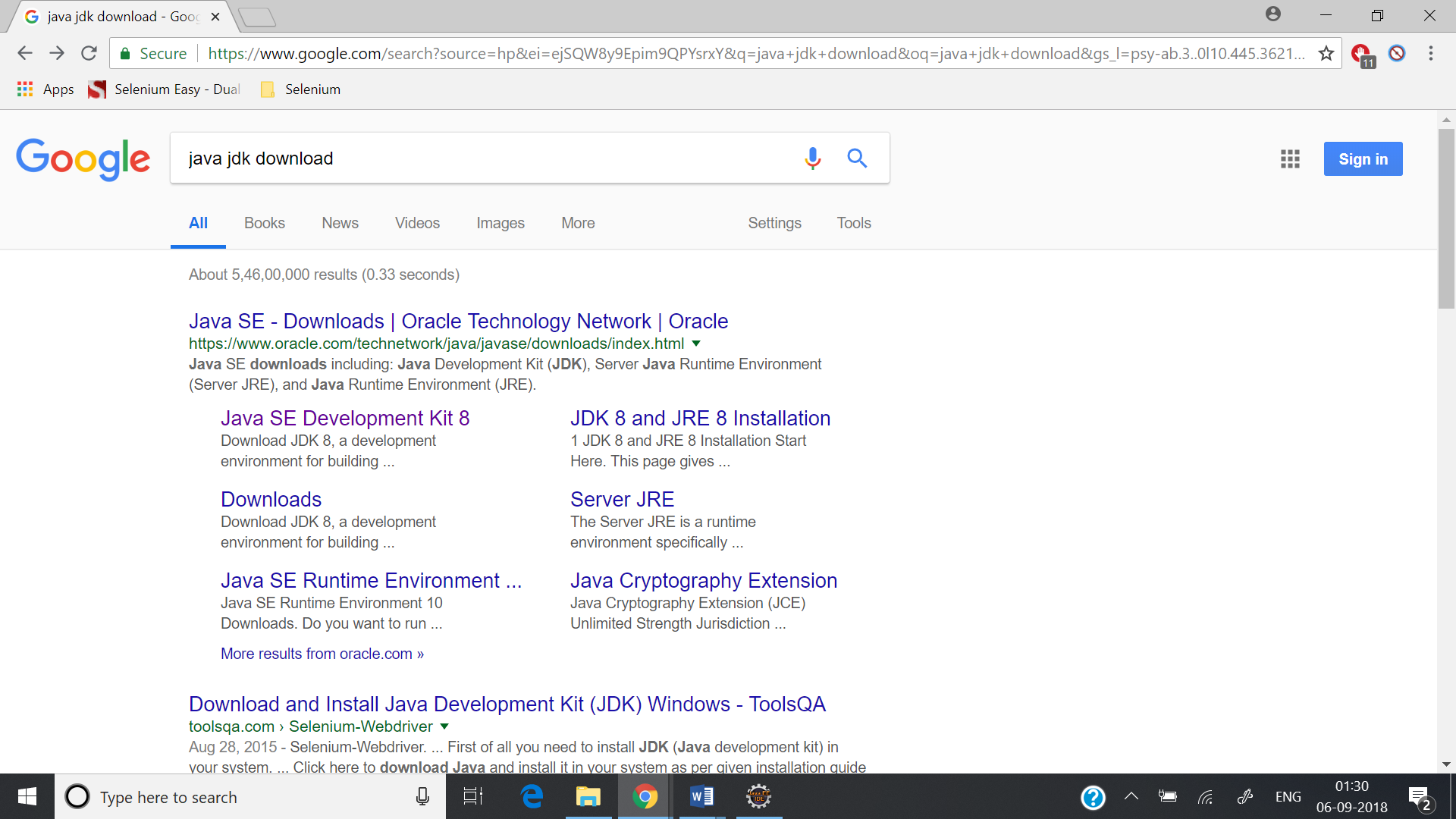
Java Native Stack: It store Non Java coding available in java code. Ex: Comments

Execution Engine: It is responsible for executing the program.

**Starting with JAVA:**

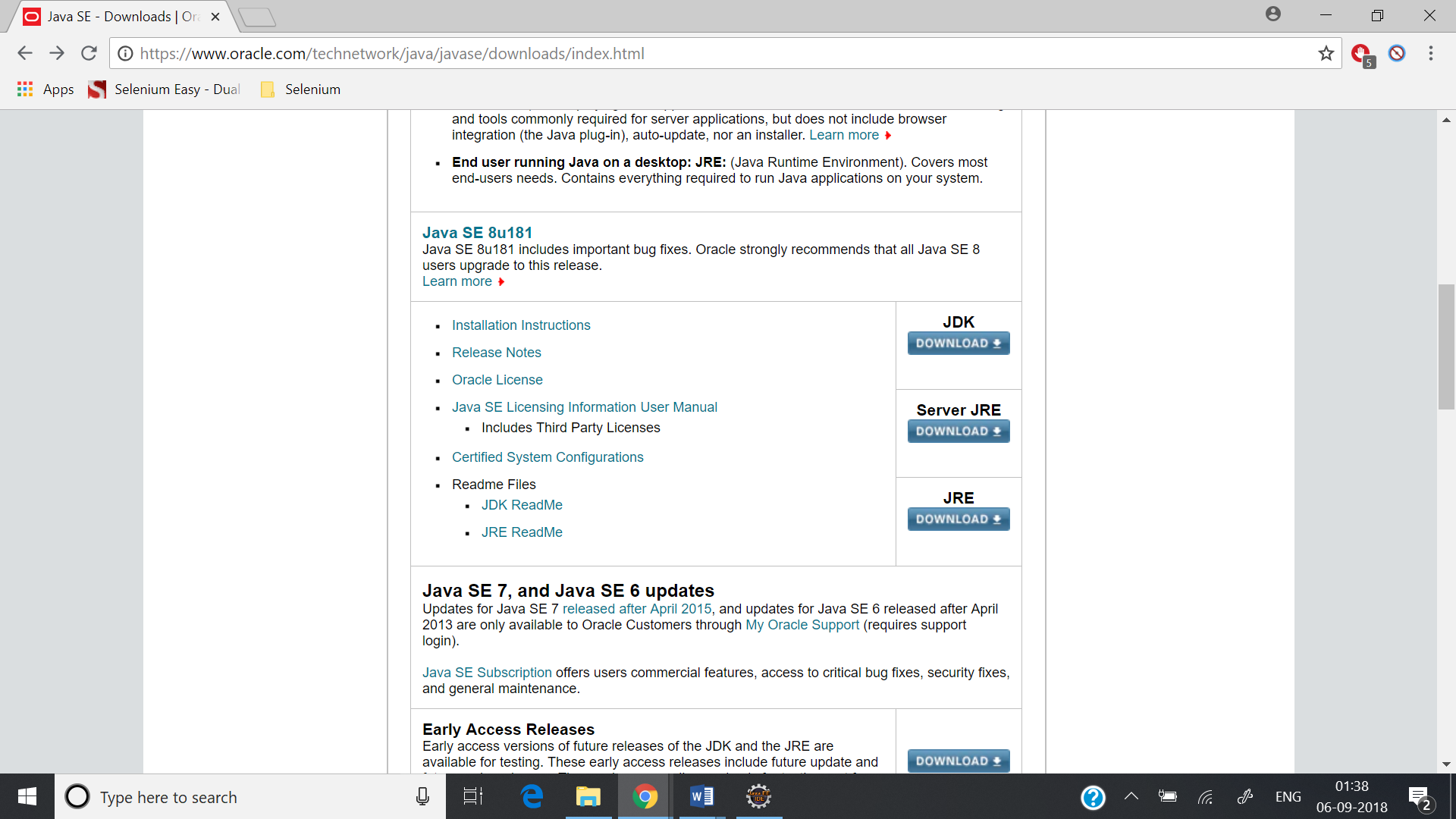
**Installing Java JDK:**

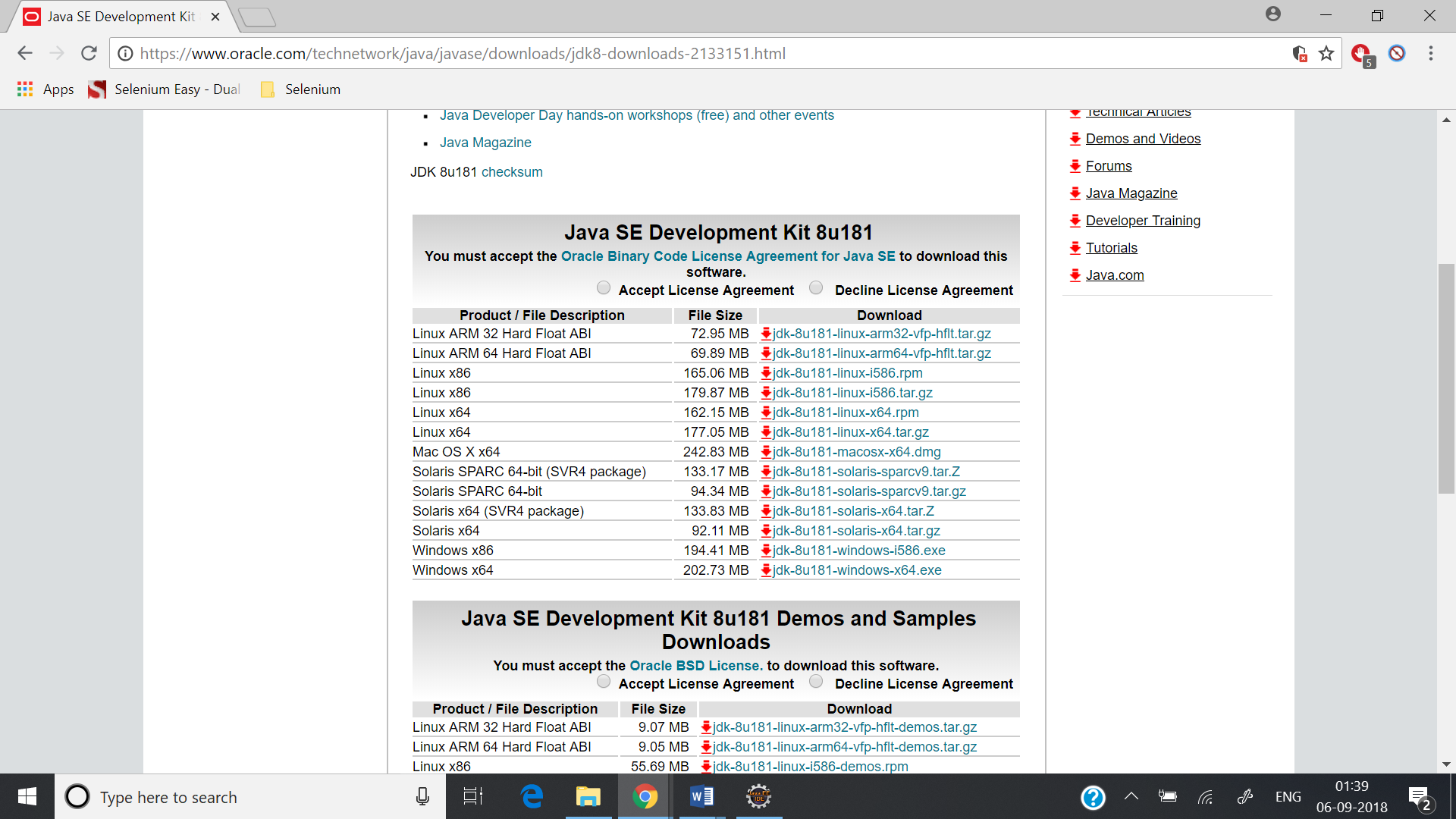
Go to google and search “Java JDK download”.



Click on the first link.

Download JDK 8 Version.



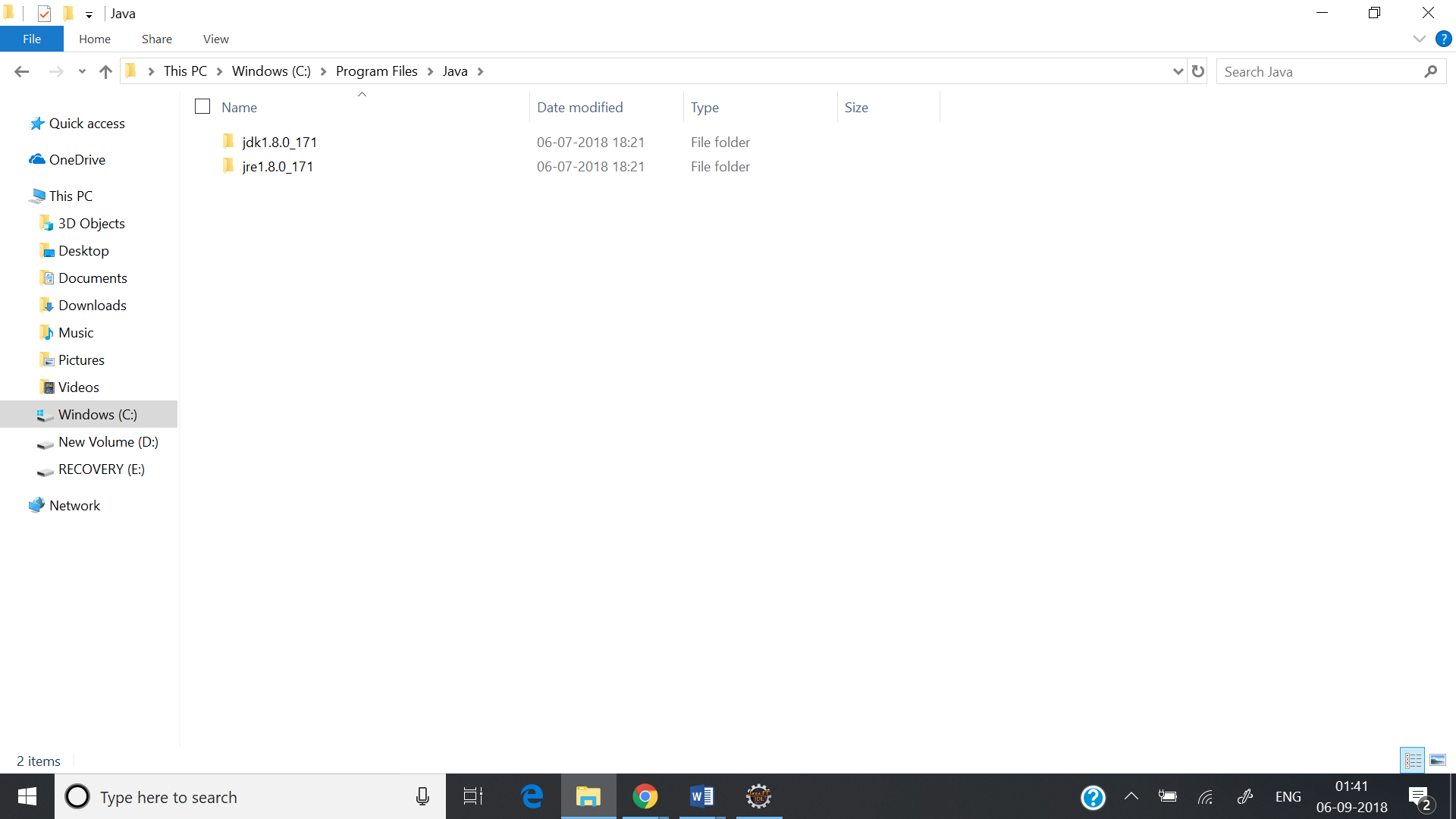


Download the corresponding version as per your operating system and install it.

Once installation is completed check where the JDK is installed. To check this go to

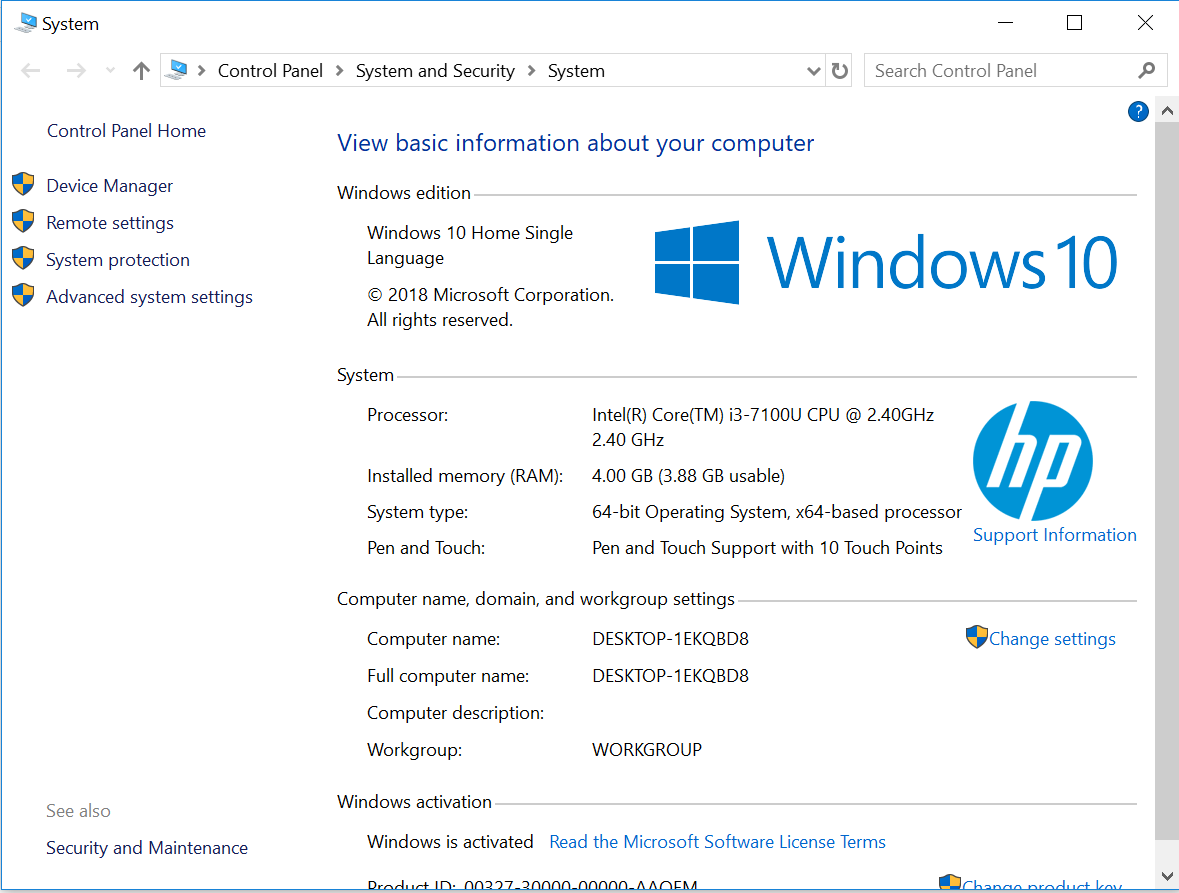
C Drive -> Program Files

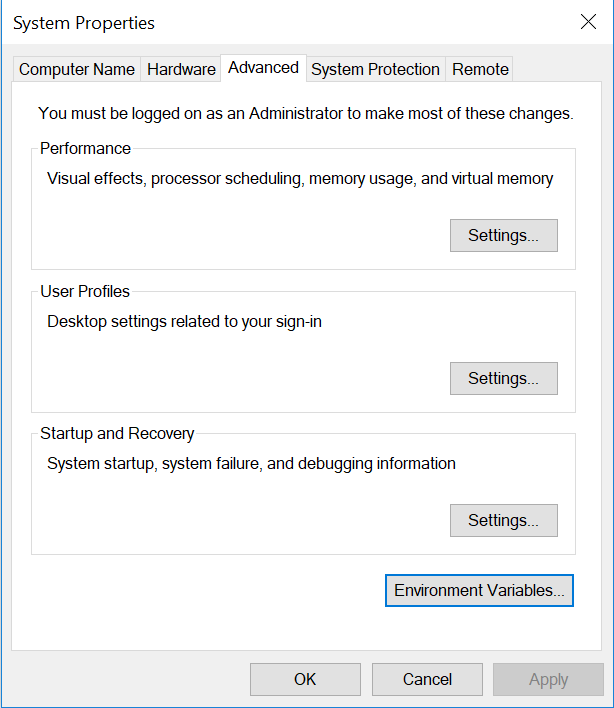
If Java folder is there, go to that folder and check if JDK is present.

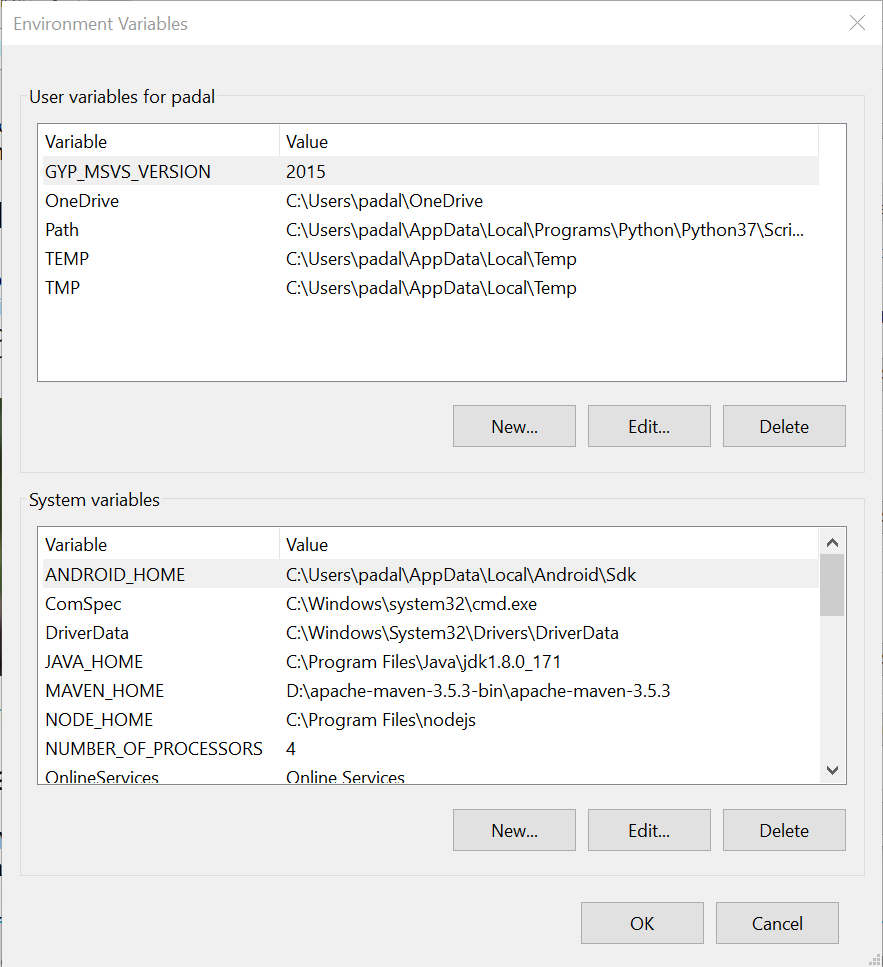


**Configure Java Path in System Variables:**

Right click on “This PC” and go to “Advanced System Settings”





Go to Environment Variables. 

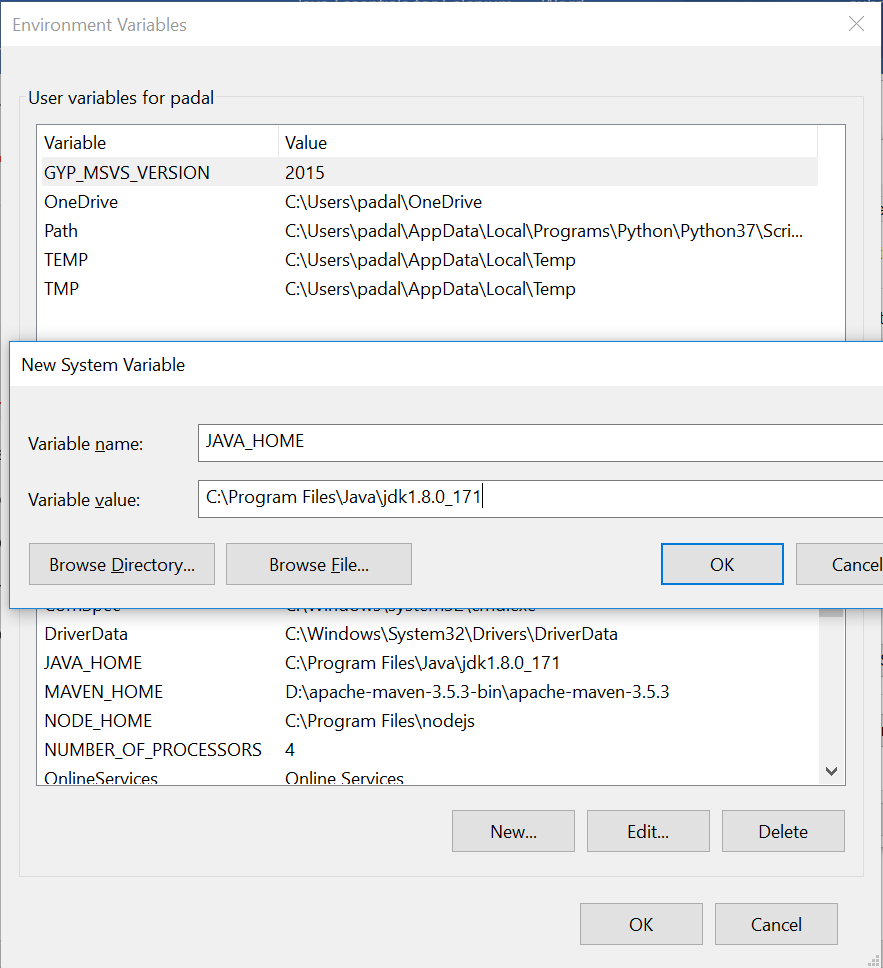
System Variables is the section where you have to add the paths.

There are two paths you have to have.

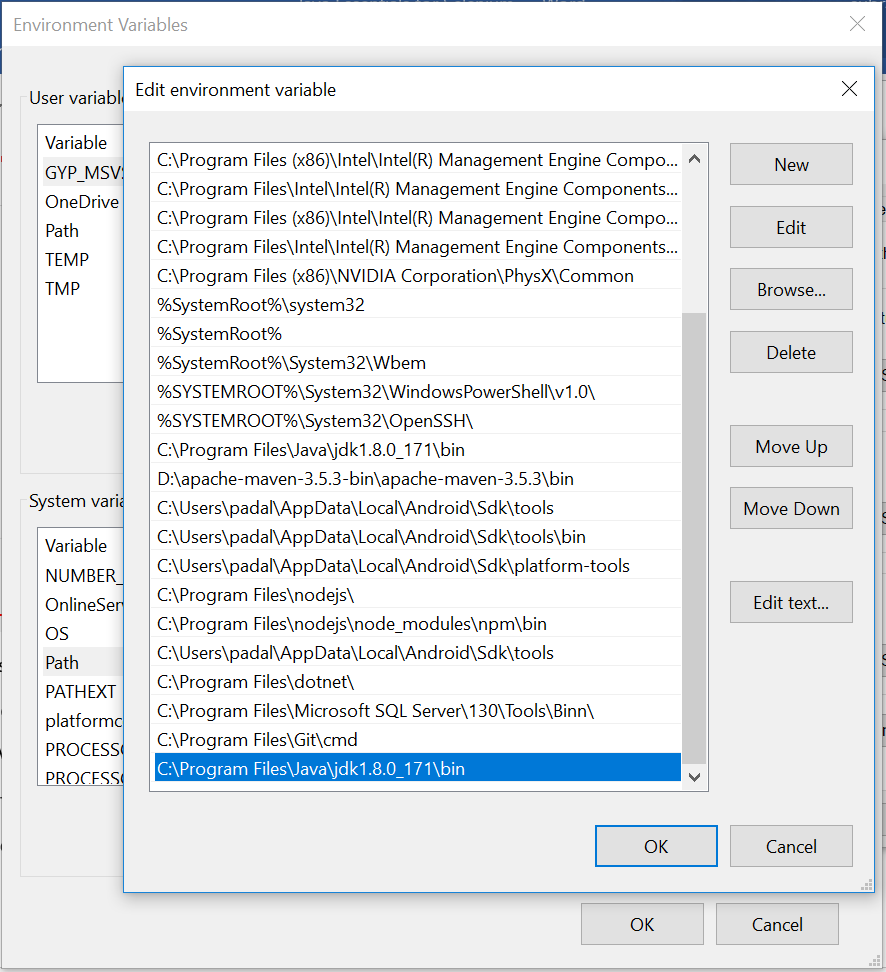
JAVA\_HOME – your jdk directory. (C:\Program Files\Java\jdk1.8.0\_171)

Path – Your bin folder in jdk directory. (C:\Program Files\Java\jdk1.8.0\_171\bin)

Click on New and add the JAVA\_HOME Variable.



Path variable should already be there. Edit it and add the path to the bin directory.

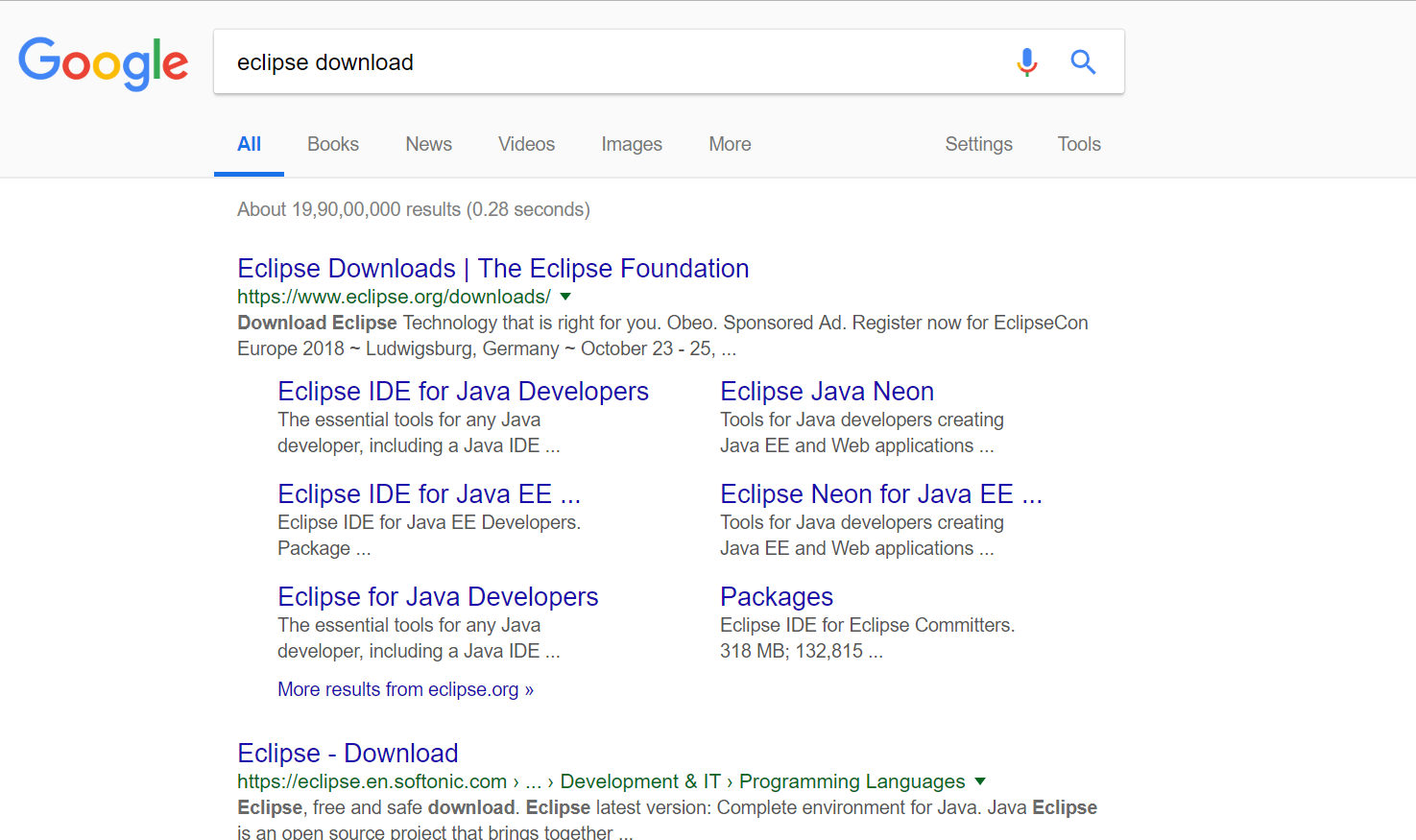


To check if java is set up correctly in your system, go to command prompt and type the command “java -version” and it should show the version.

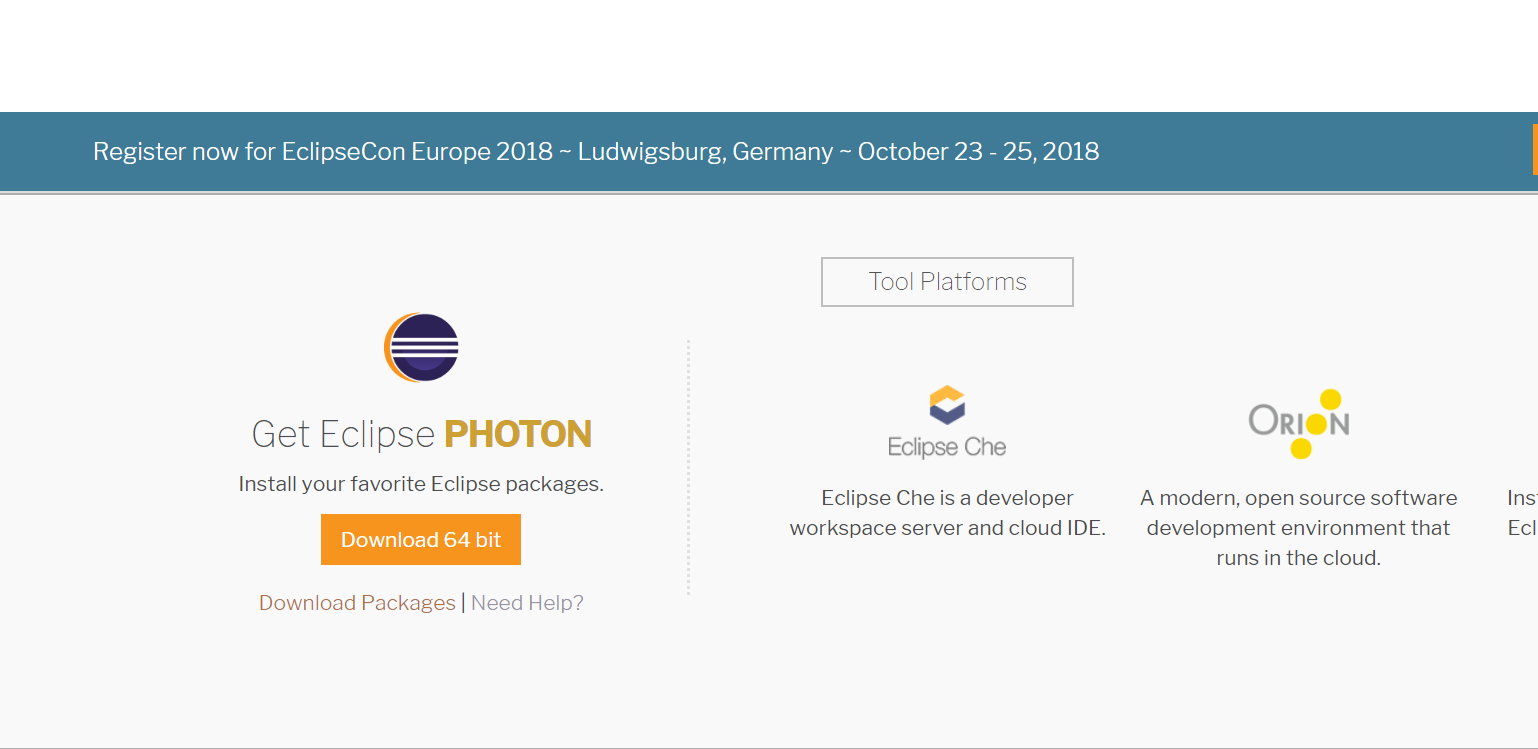


**Eclipse Installation:**

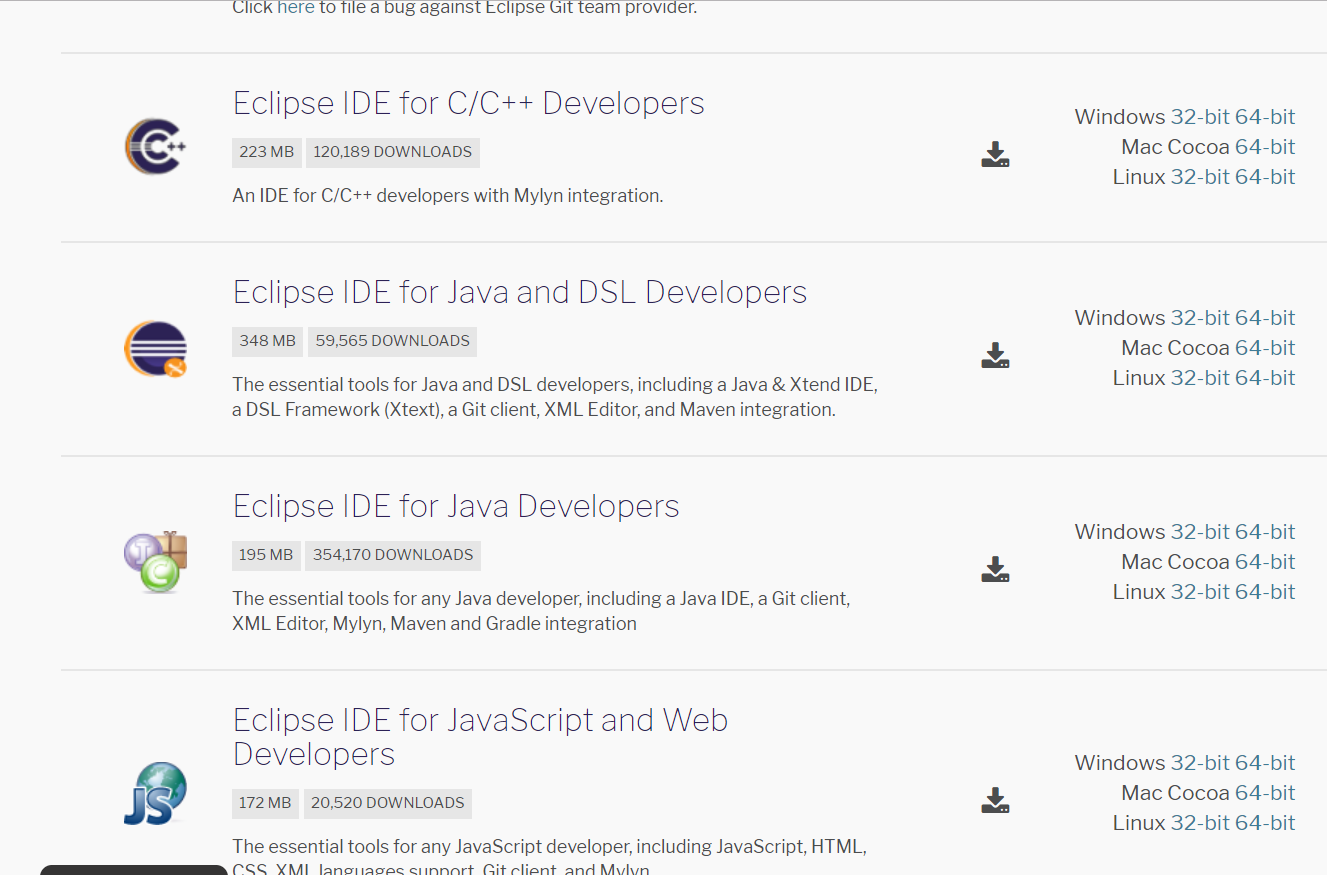
Go to google and type “Eclipse Download”



Go to the eclipse downloads and packages.

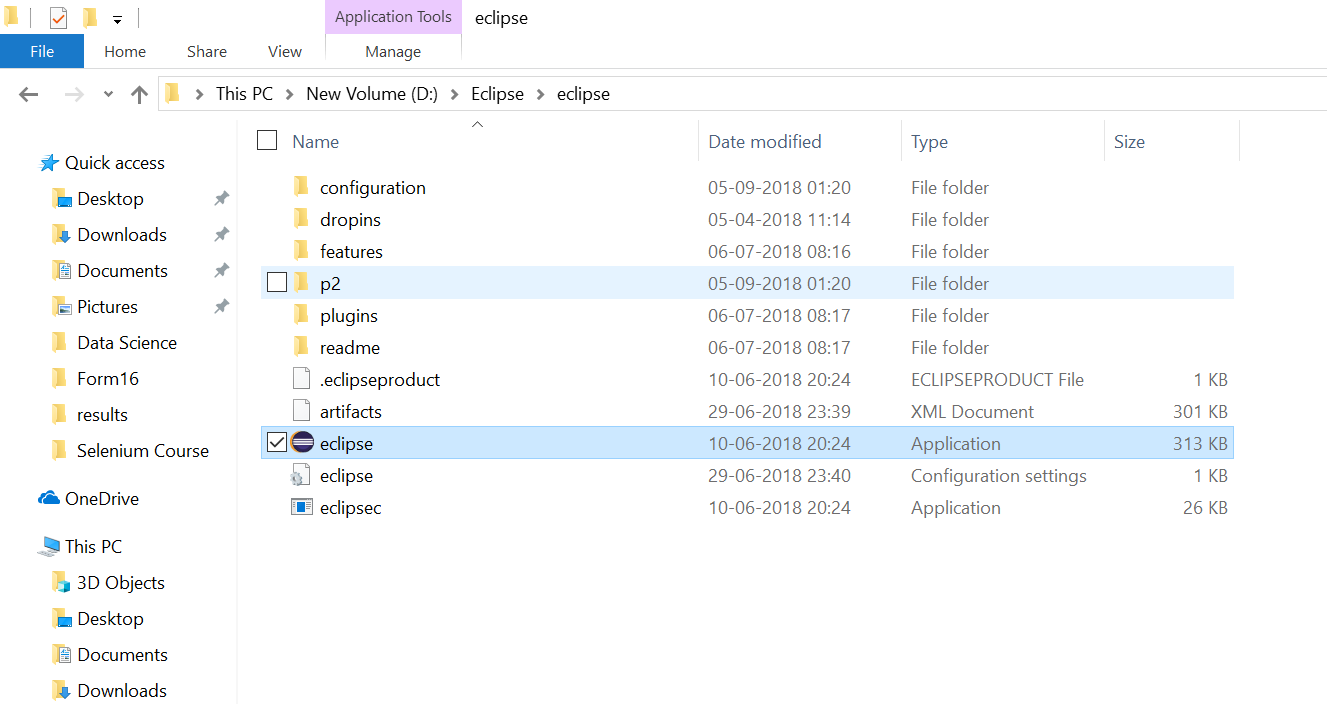


Go to download packages.



Download “Eclipse IDE for Java Developers”

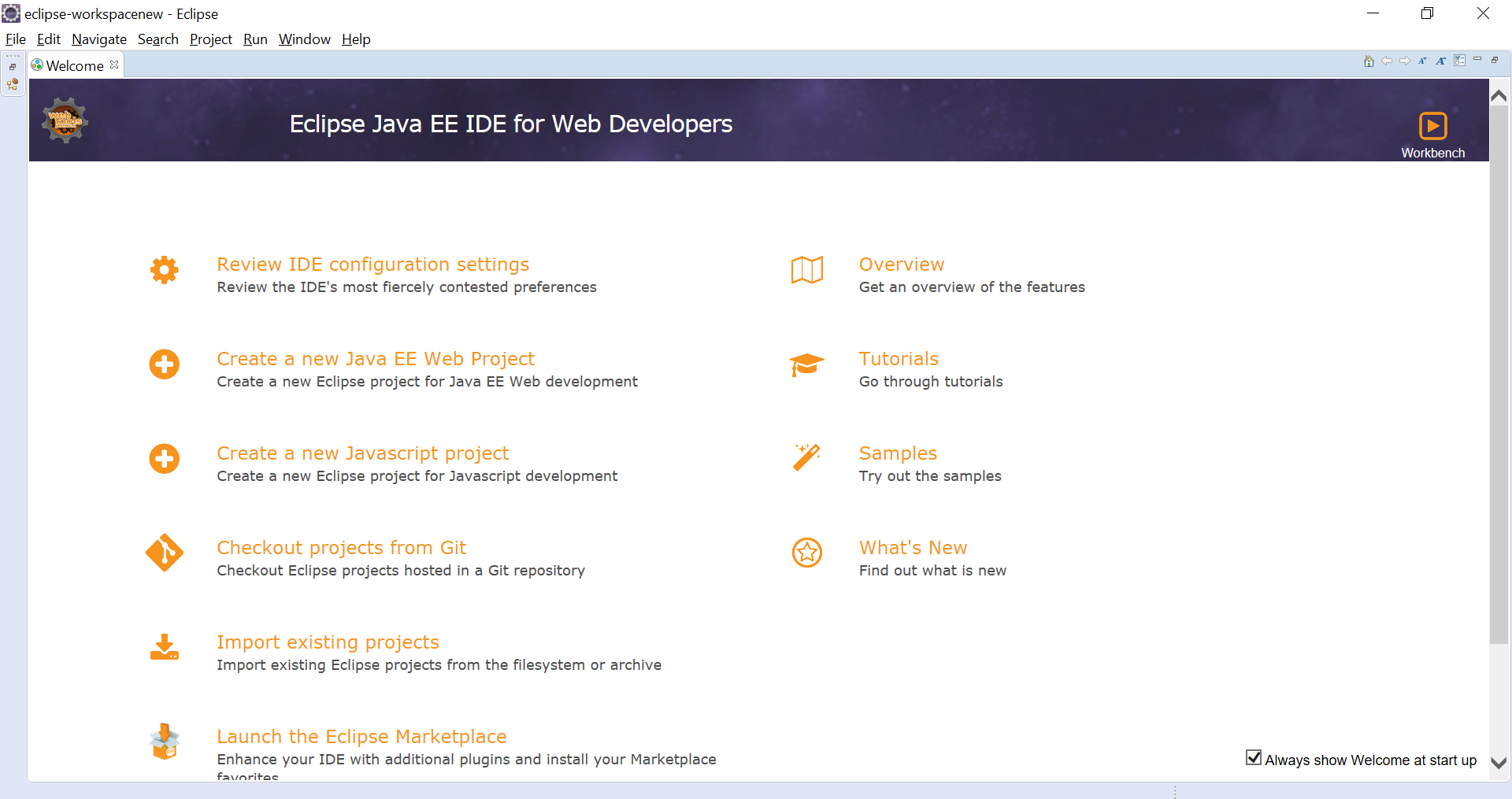
Once Eclipse is downloaded, unzip it and go to the eclipse folder.



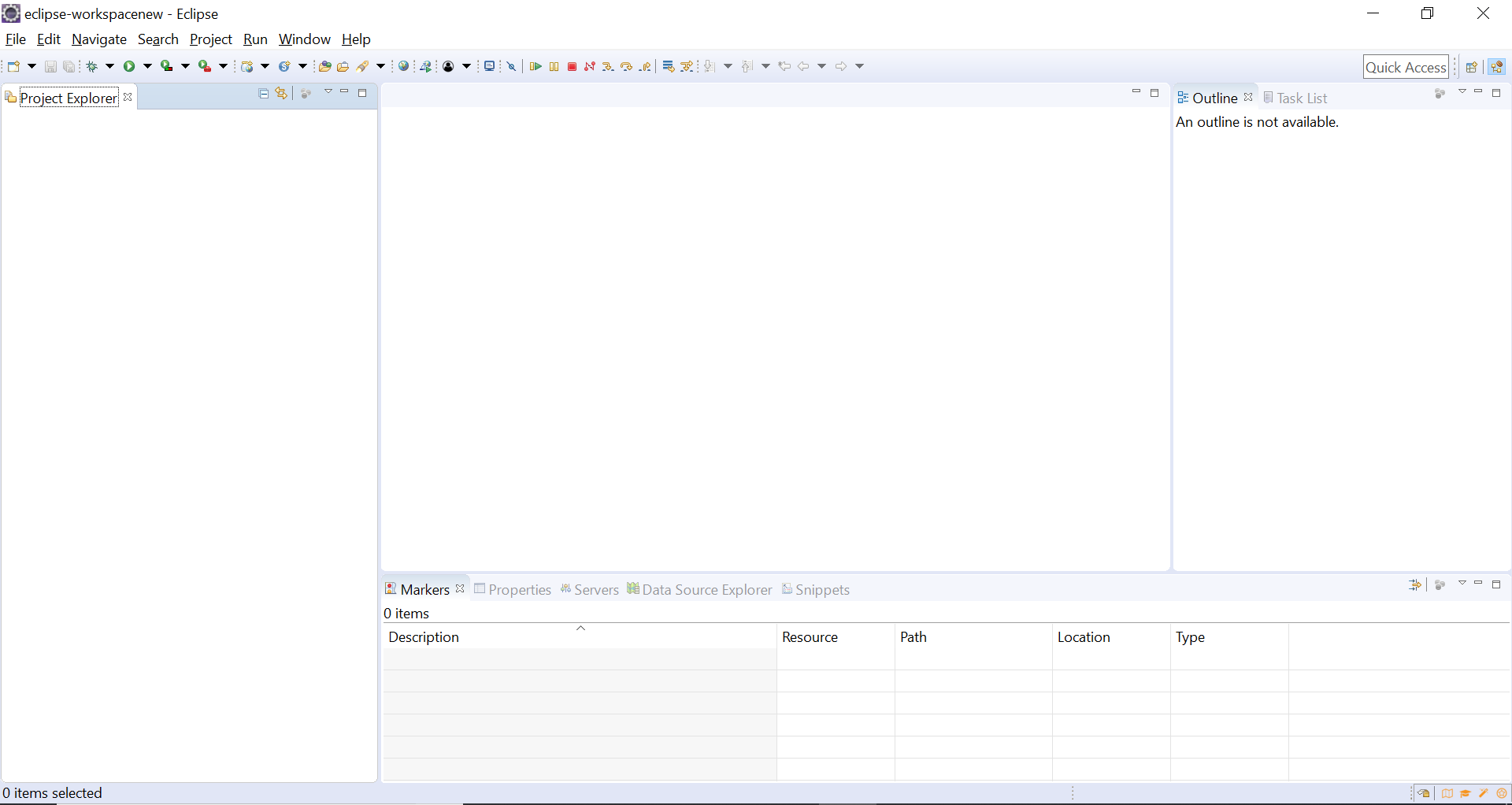
Click on Eclipse application link. It will ask you for the workspace. Provide a workspace folder and you are ready to start with your selenium scripts.



The first screen you see:

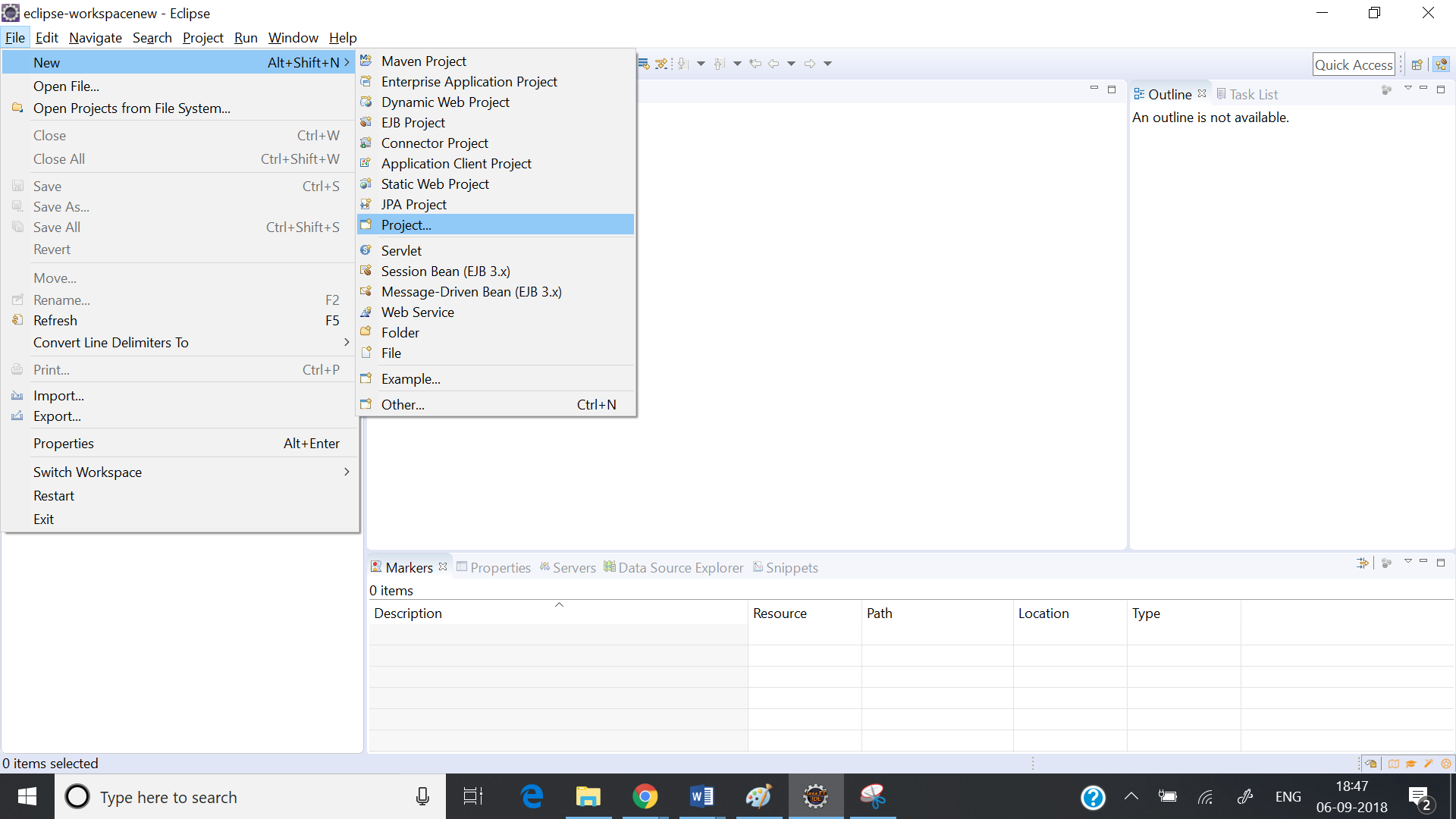


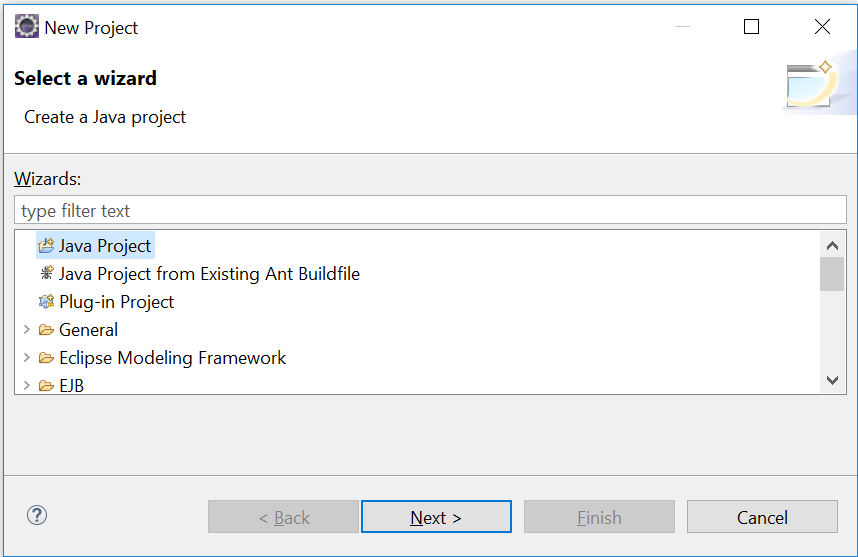
Click on Workbench:



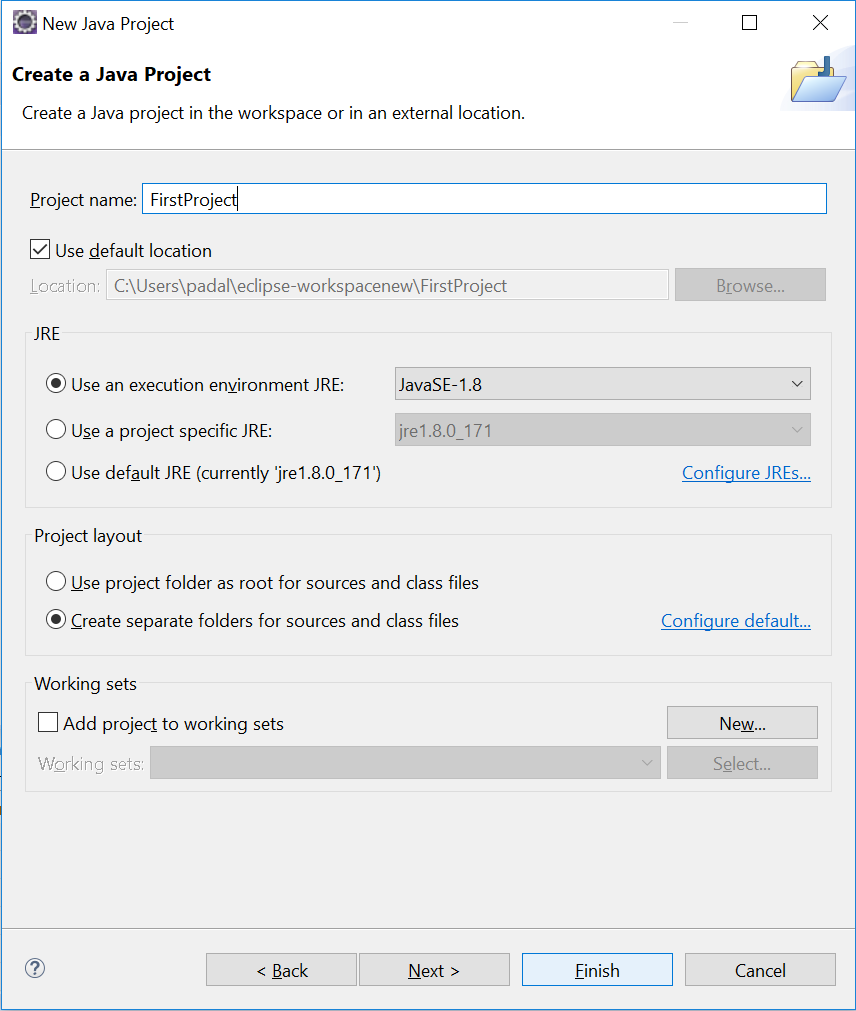
**Creating a Project:**

Go to File -> New -> Project

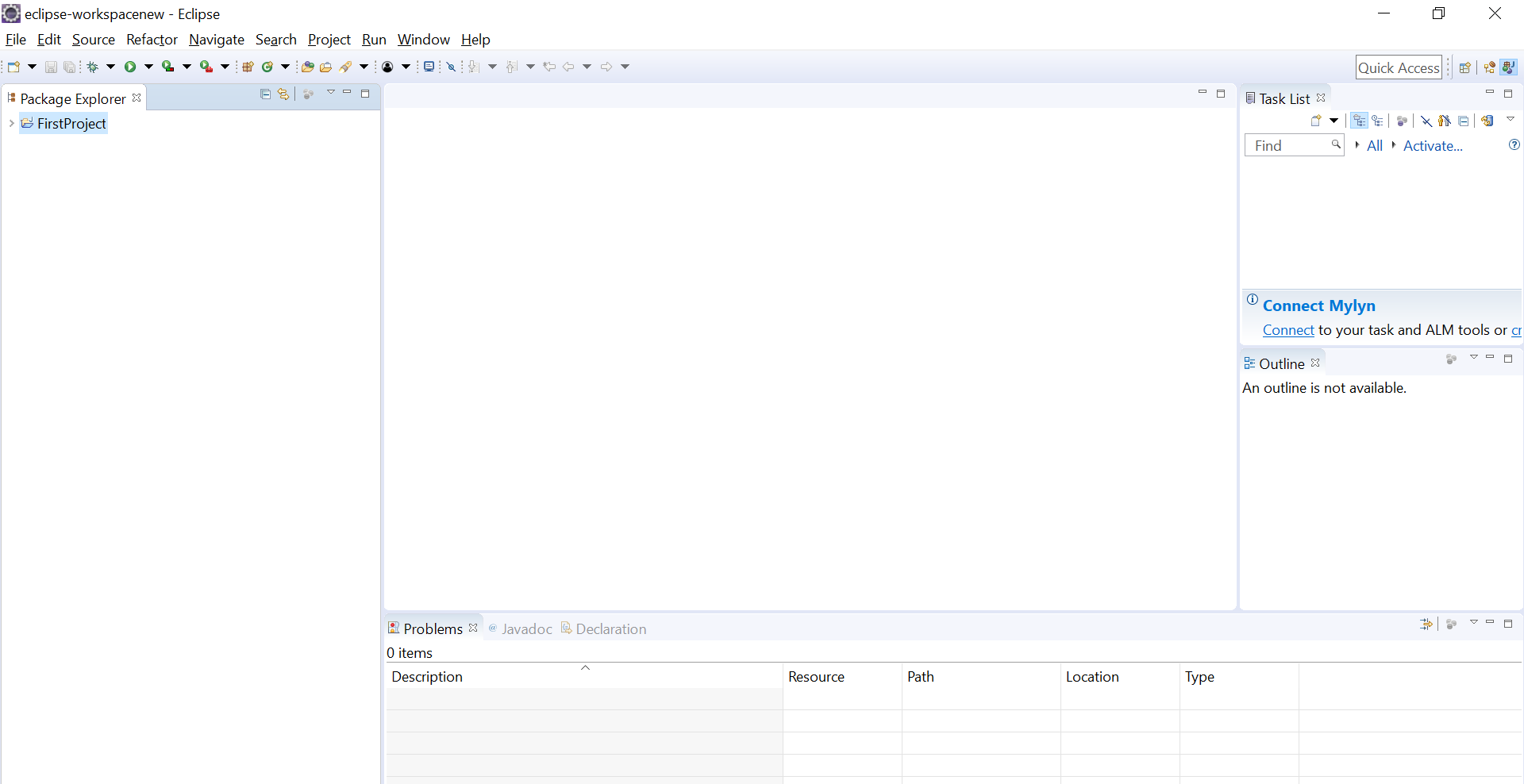




Select Java Project.

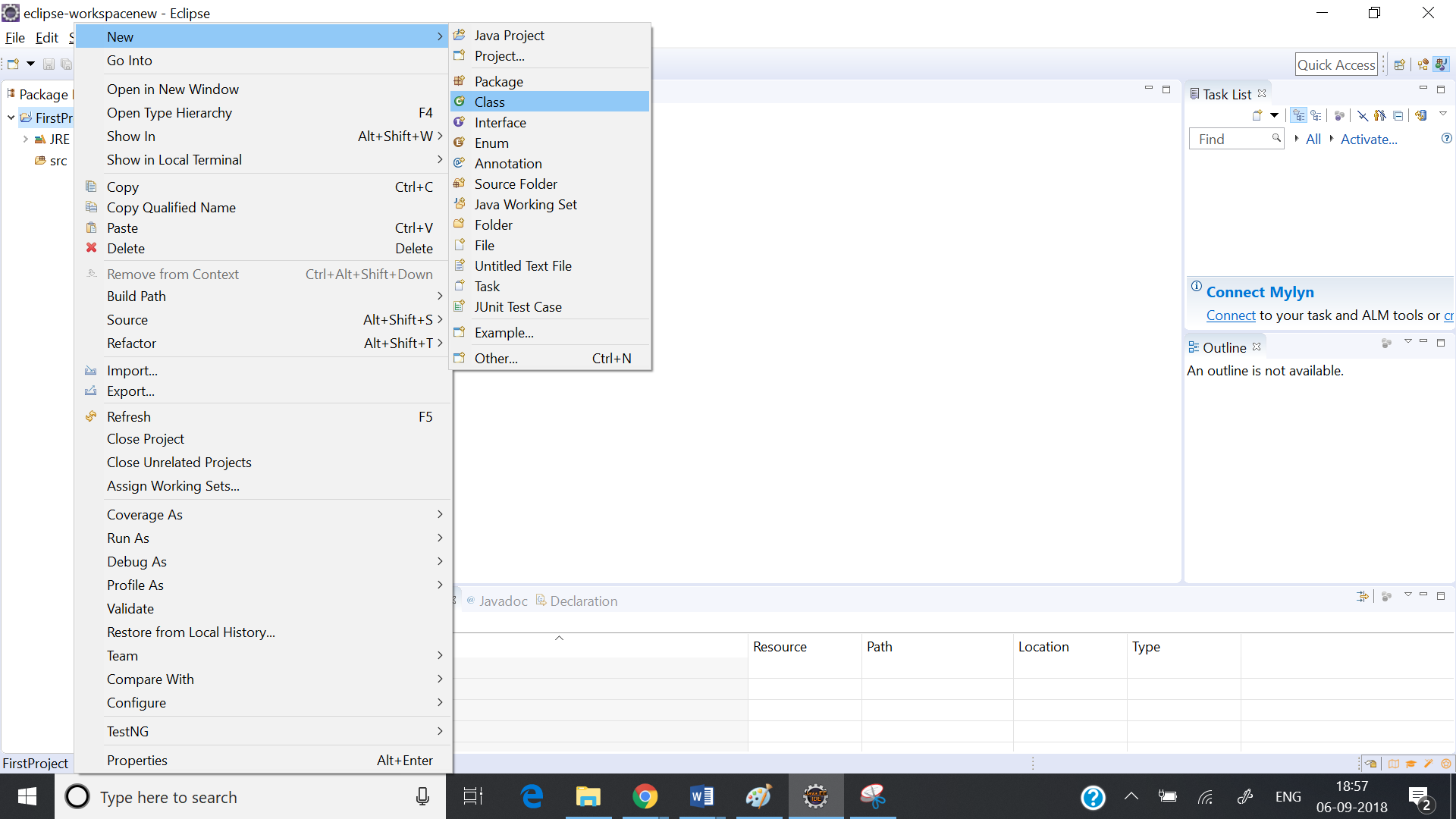


Give a name to the project and click on Finish.

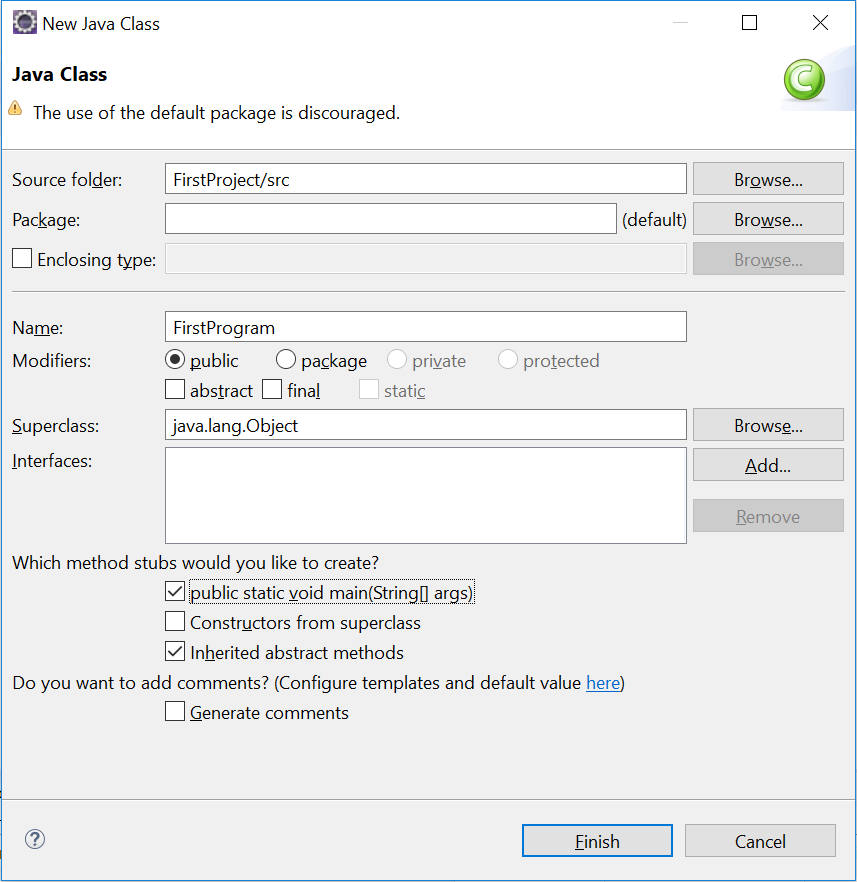


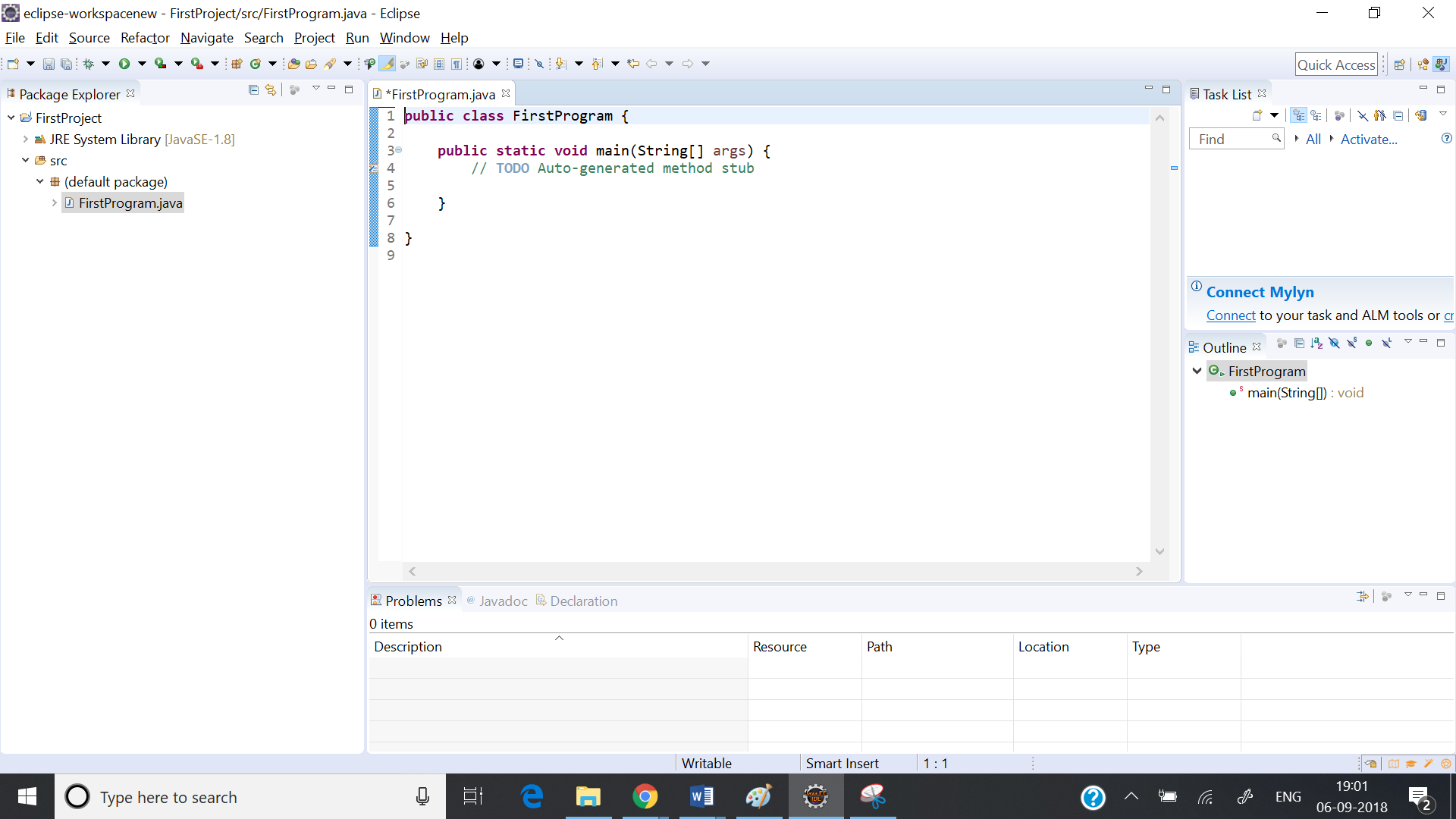
A new project with the name “First Project” is created.

**Creating a program (java file) under project:**



Right click on the project click on “New” and click on “Class”.

Give a name like “FirstProgram” and click on Finish.



This will create a program template and you can add your code to it.

# **1A. Java Basics – Interview Questions**

**What is the most important feature of Java?**

Java is a platform independent language.

**What do you mean by platform independence?**

Platform independence means that we can write and compile the java code in one platform (eg Windows) and can execute the class in any other supported platform eg (Linux,Solaris,etc).

**Why java is platform independent?**

The most unique feature of java is platform independent. In any programming language source code is compiled in to executable code. This cannot be run across all platforms. When javac compiles a java program it generates an executable file called .class file. class file contains byte codes. Byte codes are interpreted only by JVMs. Since these JVM’s are made available across all platforms, we can execute this byte code in any platform. Byte code generated in windows environment can also be executed in Linux environment. This makes java platform independent.

**What is a JVM?**

JVM is Java Virtual Machine which is a run time environment for the compiled java class files. JVM executes the class files produced by compilers.

**Are JVM's platform independent?**

JVM's are not platform independent. JVM's are platform specific run time implementation provided by the vendor.

**What is the difference between a JDK and a JVM?**

JDK is Java Development Kit which is for development purpose and it includes execution environment also. But JVM is purely a run time environment and hence you will not be able to compile your source files using a JVM.

**What is a pointer and does Java support pointers?**

Pointer is a reference handle to a memory location. Improper handling of pointers leads to memory leaks and reliability issues hence Java doesn't support the usage of pointers.

**What is jar?**

Jar stands for java archive file. Jars are created by using Jar.exe tool. Jar files contains .class files, other resources used in our application and manifest file.Manifest file contains class name with main method.jar contains compressed .class files. Jvm finds these .class files without uncompressing this jar.

**What are the different kinds of memory in java?**

here are two kinds of memory used in Java. These are called stack memory and heap memory. Stack memory stores primitive types and the addresses of objects. The object values are stored in heap memory. An object reference on the stack is only an address that refers to the place in heap memory where that object is kept.

Test test1 = new Test();

Test test2 = new Test();

test2 = test1;

What you're actually doing when you write this is assigning the address of the test1 object to the test2 object. Assume that test1's memory address was 0x33d444 and that test2's address was 0x99f775. After performing the above assignment, test2 now holds this address in stack memory:0x99f775, which refers to the same object as test1. The test2 object on the heap still exists, but it cannot be accessed. That's because this reassignment overwrote the old address that test2 was keeping on the stack. This kind of reassignment makes two stack references to the same object on the heap.

It is useful to know that these two different kinds of memory exist in Java. Stack memory is the program's memory, and heap memory resides outside of the program.

As a Java programmer, you do not have to directly address memory allocation and recovery of memory space, which is a common headache for C++ programmers. When you need a new object, Java allocates the required memory. When you are done with an object, the memory is reclaimed for you automatically via Java's garbage collection facility.

Garbage collection runs as a thread in the background, looking for objects that no longer have a usable reference. When it finds them, it destroys them and reclaims the memory.

# **2. What is Class and Object in Java?**

**First Program in Java:**

**public** **class** FirstProgram {

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

System.***out***.println("Hello World!");

}

}

Class: A program in java is written with class declaration. This is a basic plan using which we can create objects. All the program is enclosed in a class definition.

The body of the program is contained in a main method. In java the execution starts from main method.

"System" is a class in the java.lang package. "out" is a static member of the System class, and is an instance of java.io.PrintStream. "println" is a method of java.io.PrintStream. This method is overloaded to print message to output destination.

Object: An object is an instance of a class.

To create an object, we use the keyword “new”

**package** package1;

**public** **class** objectexample {

**int** a = 1;

**int** b = 2;

**public** **void** disp() {

objectexample obj = **new** objectexample();

System.***out***.println("b value is "+ obj.b);

}

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

objectexample ob = **new** objectexample();

System.***out***.println("a value is "+ob.a);

ob.disp();

}

}

In the above example we have created two objects.

objectexample obj = **new** objectexample();

objectexample ob = **new** objectexample();

**General Examples:**

If you want to construct a house, you will have a plan. The plan is nothing but a class. The constructed house is nothing but an object.

You can create any number of houses using the same plan. That means you can create any number of objects using a class.

Each and every house might differ in the number of rooms it has. That means objects will differ in their properties.

Example:

**package** package1;

**class** HousePlan {

**public** **int** bedrooms;

**public** **int** bathrooms;

}

**public** **class** House {

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

HousePlan a = **new** HousePlan();

a.bedrooms = 2;

a.bathrooms = 1;

HousePlan b = **new** HousePlan();

b.bedrooms = 3;

b.bathrooms = 2;

System.***out***.println("Bedrooms in House a "+a.bedrooms);

System.***out***.println("Bedrooms in House b "+b.bedrooms);

System.***out***.println("Bathrooms in House a "+a.bathrooms);

System.***out***.println("Bathrooms in House b "+b.bathrooms);

}

}

# **2A. Java Basics – Class and Object Interview Questions**

**What is a class?**

A class is simply a representation of a type of object. It is the blueprint/ plan/ template that describes the details of an object.

**What is the base class of all classes?**

java.lang.Object

**What is difference between Path and Classpath?**

Path and Classpath are operating system level environment variales. Path is used define where the system can find the executables(.exe) files and classpath is used to specify the location .class files.

**Explain about main() method in java?**

Main() method is starting point of execution for all java applications.

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

String args[] are array of string objects we need to pass from command line arguments. Every Java application must have at least one main method.

**What is the argument of main() method?**

main() method accepts an array of String object as argument.

**Does the order of public and static declaration matter in main() method?**

No. It doesn't matter but void should always come before main().

**Which class is extended by all other classes?**

The Object class is extended by all other classes.

**Should a main() method be compulsorily declared in all java classes?**

No, not required. main() method should be defined only if the source class is a java application.

**What is the return type of the main() method?**

Main() method doesn't return anything hence declared void.

**Can we have multiple classes in single file?**

Yes. We can have multiple classes in single file but people rarely do that and not recommended. We can have multiple classes in File but only one class can be made public. If we try to make two classes in File public we get following compilation error. “The public type must be defined in its own file”.

**package** intquestions;

**public** **class** test{ //Error: Public type must be defined in its file

**public** **static** **void** main() {

System.***out***.println("hello");

}

}

**public** **class** testclass

{

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

System.***out***.println("hi");

test.*main*();

}

}

**What are identifiers in java?**

Identifiers are names in java program. Identifiers can be class name, method name or variable name.

**What is classpath?**

The path where our .class files are saved is referred as classpath.

JVM searches for .class files by using the class path specified.

Class path is specified by using CLASSPATH environment variable.

CLASSPATH environment variable can contain more than one value.

CLASSPATH variable containing more than one value is separated by semicolon.

Example to set class path from command prompt : set CLASSPATH= C:Program FilesJavajdk1.6.0\_25bin;.;

only parent directories need to be added to classpath.Java compiler will look for appropriate packages and classes.

# **3. Java Variables**

Variables are value holders. To declare a value the syntax is

data\_type variable\_name = value;

int a = 9;

char c1 = ‘c’

Example:

**package** FPPackage;

**public** **class** variablesex {

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

**int** a = 10;

**int** b = 20;

**char** c1 = 'j';

System.***out***.println("a value is "+a);

System.***out***.println("b value is "+b);

System.***out***.println("C1 is "+c1);

**int** c = a \* b;

System.***out***.println("c value is "+c);

}

}

**Different Types of Variables:**

1. Class variable (Static fields) - Class variables are variables declared within the class body, outside of any methods or blocks, and declared with 'static' keyword.   
Class variables have the longest scope. They are created when the class is loaded, and remain in memory as long as the class remains loaded in JVM.  
  
2. Instance variables (Non-static fields) - Instance variable are variables declared within the class body, outside of any method or block, and declared without 'static' keyword.   
Instance variables have the second highest scope. Instance variables are created when a new class instance is created, and live until the instance is removed from memory.  
  
3. Local Variables - Local variables are variables declared within a method body. They live only as long as the method in which it is declared remains on the stack.  
  
4. Block variables - Block variables are variables declared within a block such as an init block or within a for loop. They live only during the execution of the block and are the shortest living variables.

Example:

**public** **class** VariablesExample {

**static** **int** *c* = 6; //Class Variable

**int** d = 8; //Instance Variable

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

**int** a = 5; //Local Variable

**int** b = a \* 5; //Local Variable

System.***out***.println("a value is "+a);

System.***out***.println("b value is "+b);

**for**(**int** i = 0; i<4; i++) {

**int** j = 2; //Block Variable

System.***out***.println(a\*j);

}

}

}

# **3A. Java Variables – Interview Questions – Part 1**

**What is meant by a variable?**

Variables are locations in memory that can hold values.

**How are the variables declared?**

Variables can be declared anywhere in the method definition and can be initialized during their declaration. They are commonly declared before usage at the beginning of the definition. Variables with the same data type can be declared together. Local variables must be given a value before usage.

**How do you assign values to variables?**

Values are assigned to variables using the assignment operator =.

**What are the kinds of variables in java and what are their uses?**

Java has four kinds of variables.

* Block Variables
* Local Variables
* Instance Variables
* Class Variables

Block Variables are used inside blocks like init blocks or for loops and are used to store information needed by a single block.

Local variables are used inside methods as temporary variables and are used to store information needed by a single method.

Instance variables (member variables) are used to define attributes or the state of a particular object and are used to store information needed by multiple methods in the objects.

Class variables are global to a class and to all the instances of the class and are useful for communicating between different objects of all the same class or keeping track of global states.

**What is a literal? How many types of literals are there?**

A literal represents a value of a certain type where the type describes how that value behaves. There are different types of literals namely:   
a) number literals (1,23)  
b) character literals (‘c’, ‘d’)  
c) boolean literals (true, false)  
d) string literals and etc. (“hello”)

**What are local variables?**

Local variables are those which are declared within methods. Local variables should be initialised before accessing them.

**What are instance variables?**

Instance variables are those which are defined at the class level without static keyword. Instance variables need not be initialized before using them as they are automatically initialized to their default values.

**What is the scope or life time of instance variables?**

When object is instantiated using new operator variables get allocated in the memory. Instance variables remain in memory till the instance gets garbage collected.

**Explain scope or life time of local variables in java?**

Local variables are variables which are defined inside a method. When the method is created local variables gets created in stack memory and this variable gets deleted from memory once the method execution is done.

**What are reference variables in java?**

Variables which are used to access objects in java are called reference variables.

Ex: Employee emp=new Employee();

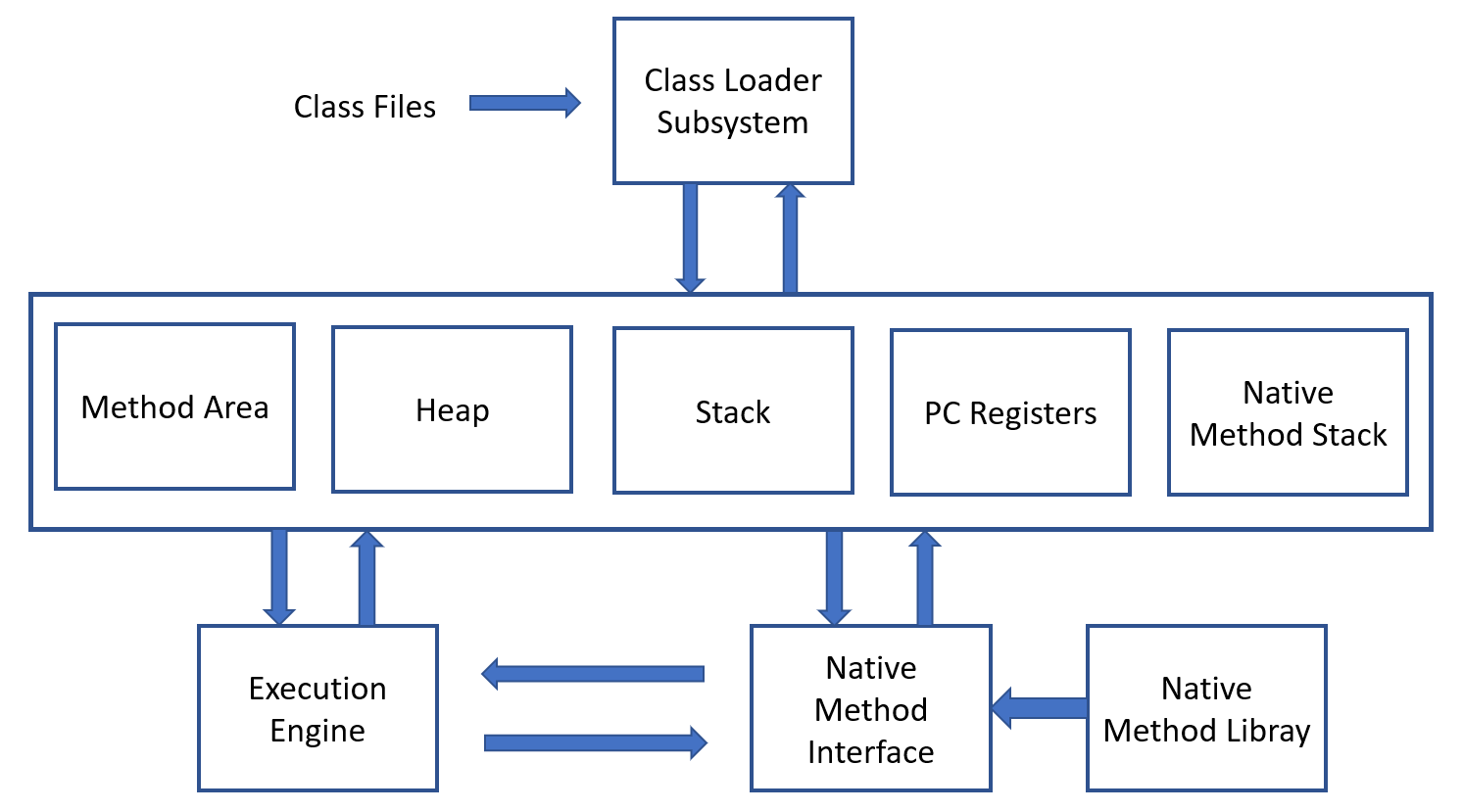
In the above example emp is reference variable.

**How long do primitive variables exist in memory? Define the various scopes of primitive variables?**

After a primitive variable is declared and initialized; how long it lives in memory is dependent on the scope of the variable. Scope of a variable is determined based on where it is declared within a java class. Following are the various scopes of a variable in a java program based on where they are declared.  
  
**1. Class variable (Static fields) -** Class variables are variables declared within the class body, outside of any methods or blocks, and declared with 'static' keyword.   
  
Class variables have the longest scope. They are created when the class is loaded, and remain in memory as long as the class remains loaded in JVM.  
  
**2. Instance variables (Non-static fields) -**Instance variable are variables declared within the class body, outside of any method or block, and declared without 'static' keyword.   
  
Instance variables have the second highest scope. Instance variables are created when a new class instance is created, and live until the instance is removed from memory.  
  
**3. Local Variables -**Local variables are variables declared within a method body. They live only as long as the method in which it is declared remains on the stack.  
  
**4. Block variables -**Block variables are variables declared within a block such as an init block or within a for loop. They live only during the execution of the block and are the shortest living variables.

# **3B. Java Variables – Interview Questions – Part 2**

**Explain where variables are created in memory?**



Method Area: It stores all the class code, method code and class variables declared.

Heap Memory: It stores all the objects created.

JVM stack Area: It stores the information of methods at run time. For example, the method variables initiated at run time. It also stores the references to the objects created.

**What is final variable?**

If a variable is declared as final variable, then you can not change its value. It becomes constant.

Example:

**public** **class** JavaExamples {

**static** **final** **int** ***a*** = 6;

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

***a*** = 7; //Error: The final field javaexamples.a cannot be assigned

System.***out***.println("a value is "+***a***);

}

}

**How to define a constant variable in Java?**

The variable should be declared as static and final. So only one copy of the variable exists for all instances of the class and the value can't be changed also.  
  
static final int MAX\_LENGTH = 50; is an example for constant.

# **4. Java Data Types**

Data types define the type of values that a variable can take.

**byte**, **short**, **int** and **long** data types are used for storing whole numbers.

**float** and **double** are used for fractional numbers.

**char** is used for storing characters(letters).

**boolean** data type is used for variables that holds either true or false.

Byte can store the whole number between -128 and 127

Short can store the whole number between -32,768 to 32767

Int can store the whole number between -2,147,483,648 to 2,147,483,647

Long can store the whole number between -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Double can store 15 decimal digits

Float can store 6 to 7 decimal digits

Boolean holds either true or false

Char holds characters.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Types** | **Size (bytes)** | **Minimum Value** | **Maximum Value** | **Precision** |
| Byte | 1 | -128 | 127 | From +127 to -128 |
| Short | 2 | -215 | 215-1 | From +32,767 to -32,768 |
| Int | 4 | -231 | 231-1 | From +2,147,483,647 to -2,147,483,648 |
| Long | 8 | -263 | 263-1 | From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808 |
| Float | 4 | 2-149 | (2-2-23)·2127 | From 3.402,823,5 E+38 to 1.4 E-45 |
| Double | 8 | 2-1074 | (2-2-52)·21023 | From 1.797,693,134,862,315,7 E+308 to 4.9 E-324 |
| Char | 2 | 0 | 216-1 | All Unicode characters |

Example:

**package** FPPackage;

**public** **class** variablesexample {

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

**byte** a = 9;

**int** b = 20;

**char** c1 = 'j';

System.***out***.println("a value is "+a);

System.***out***.println("b value is "+b);

System.***out***.println("C1 is "+c1);

**int** c = a \* b;

System.***out***.println("c value is "+c);

}

}

**Primitive Casting in JAVA:**

Primitive casting is used to convert primitive values from one data type to another data type. For example, an int value can be assigned to a float value, a double value can be assigned to an int value etc.

If we are assigning a small data type value to a large data type value no casting is needed. But if we are assigning a large data type value to a small data type value casting is needed.

Example:

Short takes 2 bytes in memory and int takes 4 bytes in memory. We can assign a short value to an int but for assigning an int value to a short we have to cast it.

**public** **class** JavaExamples {

**static** **short** *a* = 1;

**static** **int** *b*;

**static** **int** *c* = 2;

**static** **short** *d*;

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

*b* = *a*;

System.***out***.println("b value is "+*b*);

//d = c; //Error Can not convert from int to short

*d* = (**short**) *c*;

System.***out***.println("d value is "+*d*);

}

}

# **4A. Java Data Types – Interview Questions – Part 1**

**What are variable types?**

Variable types can be any data type that java supports, which includes the eight primitive data types, the name of a class or interface and an array.

**Name the eight primitive Java types.**

The eight primitive types are byte, char, short, int, long, float, double, and boolean.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Bits** | **Bytes** | **Min** | **Max** | **Default** |
| **Byte** | 8 | 1 | -29 | 28-01-2001 | 0 |
| **Short** | 16 | 2 | -217 | 216-1-1 | 0 |
| **Int** | 32 | 4 | -233 | 232-1-1 | 0 |
| **Long** | 64 | 8 | -265 | 264-1-1 | 0 |
| **Float** | 32 | 4 | >NA | NA | 0.0f |
| **Double** | 64 | 8 | NA | NA | 0.0d |
| **Boolean** | 1 | NA | NA | NA | FALSE |
| **Char** | 16 | NA | NA | NA | '' |

**What is casting?**

There are two types of casting, casting between primitive numeric types and casting between object references. Casting between numeric types is used to convert larger values, such as double values, to smaller values, such as byte values. Casting between object references is used to refer to an object by a compatible class, interface, or array type reference.

**Are arrays primitive data types?**

In Java, Arrays are objects.

**What are Primitive Literals?**

Primitive Literals are the code representation of values of primitive data types. For example 'a' is a char literal, 100 is an int literal, 'false' is a boolean literal and 2345.456 is a double literal.

**To what value is a variable of the boolean type automatically initialized?**

The default value of the boolean type is false.

**What is type conversion in java?**

Assigning a value of one type to variable of other type is called type conversion.

Example:

int a =10;

long b=a;

There are two types of conversion in java:

1) Widening conversion

2) Narrowing conversion

**Explain about Automatic type conversion in java?**

Java automatic type conversion is done if the following conditions are met :

1. When two types are compatible

Ex: int, float

int can be assigned directly to float variable.

1. Destination type is larger than source type.

Ex: int, long

Int can be assigned directly to long. Automatic type conversion takes place if int is assigned to long because long is larger datatype than int. Widening Conversion comes under Automatic type conversion.

**Explain about narrowing conversion in java?**

When destination type is smaller than source type we use narrowing conversion mechanism in java. Narrowing conversion has to be done manually if destination type is smaller than source type.

To do narrowing conversion we use cast. Cast is nothing but explicit type conversion.

Example:

long a;

byte b;

b=(byte)a;

Note: casting to be done only on valid types otherwise classcastexception will be thrown.

**Difference between double and float variables in Java.**

In java, float takes 4 bytes in memory while Double takes 8 bytes in memory. Float is single precision floating point decimal number while Double is double precision decimal number.

# **4B. Java Data Types – Interview Questions – Part 2**

**What is primitive casting in java?**

Primitive Casting is used to convert primitive values from one data type to another. For example, an int value can be assigned to a float data type, or a double value can be assigned to an int data type. Casting can be either implicit or explicit.

**Implicit Casting:** In implicit casting the conversion happens automatically, without writing specific code to do the conversion. Implicit casting happens when you convert or assign a smaller value, like a byte, to a larger data type such as an int.

**Explicit Casting:** In explicit casting code has to be specifically written to perform the conversion from one primitive type to another. Explicit casting is done by using the syntax (data\_type) where data\_type is the data type that the cast is being applied to. Explicit casting happens when you convert or assign a larger value to a smaller data type.

**package** intquestions;

**class** testclass

{

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

{

**byte** b = 2;

**int** i = b; //Implicit casting

**int** j = 3;

//byte k = j;//Throws error

**byte** k = (**byte**)j; //Explicit casting

}

}

**What are the default values of primitive data types?**

The default values are given by the following example.

**package** intquestions;

**public** **class** testclass

{

**byte** b;

**short** s;

**int** i;

**long** l;

**float** f;

**double** d;

**boolean** bo;

**char** ch;

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

testclass tc = **new** testclass();

System.***out***.println("byte default value is "+tc.b);

System.***out***.println("short default value is "+tc.s);

System.***out***.println("int default value is "+tc.i);

System.***out***.println("long default value is "+tc.l);

System.***out***.println("float default value is "+tc.f);

System.***out***.println("double default value is "+tc.d);

System.***out***.println("boolean default value is "+tc.bo);

System.***out***.println("char default value is "+tc.ch);

}

}

Result:

byte default value is 0

short default value is 0

int default value is 0

long default value is 0

float default value is 0.0

double default value is 0.0

boolean default value is false

char default value is

**How is rounding performed under integer division?**

The fractional part of the result is truncated. This is known as rounding toward zero.

Example:

**public** **class** JavaExamples {

**static** **int** *a* = 5;

**static** **int** *b* = 2;

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

System.***out***.println("a/b is "+5/2); //Rounding is performed by truncating the fractional part.

}

}

Result: 2

**Can you compare a boolean with an int variable in Java?**

No. you will get compilation error.

Example:

**public** **class** JavaExamples {

**static** **int** *a* = 5;

**static** **boolean** *b* = **false**;

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

**if**(*a* == *b*) { //Error: The operator == is undefined for the argument types int, boolean

System.***out***.println("Comparing boolean with int");

}

}

}

**What is the output of the following?**

System.out.println(**1.0**/**0**);

Most of us may expect ArithmeticException, however, in this case, there will be no exception instead it prints **Infinity**.

1.0 is a double literal and double datatype supports infinity.

Example:

**package** intquestions;

**public** **class** testclass {

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

System.***out***.println("value is "+(1.0/0));

}

}

Result:

value is Infinity

# **5. Java Operators - Arithmatic Operators and Assignment Operators**

**Operators in Java:**

Types of Operators in Java:

1. Basic Arithmetic Operators
2. Assignment Operators
3. Auto-increment and Auto-decrement Operators
4. Logical Operators
5. Comparison (relational) operators
6. Ternary Operator

**Basic Arithmetic Operators**

Basic arithmetic operators are: +, -, \*, /, %

**package** FPPackage;

**public** **class** ArithmaticOperators {

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

**int** a = 10;

**int** b = 9;

System.***out***.println(a+b);

System.***out***.println(a-b);

System.***out***.println(a\*b);

System.***out***.println((**float**)a/b);

System.***out***.println(a%b);

}

}

**Assignment Operators**

Assignments operators in java are: =, +=, -=, \*=, /=, %=

**num2 = num1** would assign value of variable num1 to the variable.

**num2+=num1** is equal to num2 = num2+num1

**num2-=num1** is equal to num2 = num2-num1

**num2\*=num1** is equal to num2 = num2\*num1

**num2/=num1** is equal to num2 = num2/num1

**num2%=num1** is equal to num2 = num2%num1

**package** FPPackage;

**public** **class** ArithmaticOperators {

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

**int** num1 = 10;

**int** num2 = num1;

System.***out***.println(num2);

num2 += num1;

System.***out***.println(num2);

num2 -= num1;

System.***out***.println(num2);

num2 \*= num1;

System.***out***.println(num2);

num2 /= num1;

System.***out***.println(num2);

num2 %= num1;

System.***out***.println(num2);

}

}

# **5A. Java Operators - Auto Increment and Decrement Operators and Logical Operators**

**Operators in Java:**

Types of Operators in Java:

1. Basic Arithmetic Operators
2. Assignment Operators
3. Auto-increment and Auto-decrement Operators
4. Logical Operators
5. Comparison (relational) operators
6. Ternary Operator

**Auto Increment and Auto Decrement Operators:**

++ and --

num1++ is num1 = num1+1

num1-- is num1 = num1-1

**package** FPPackage;

**public** **class** ArithmaticOperators {

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

**int** num1 = 10;

System.***out***.println(num1);

num1++;

System.***out***.println(num1);

num1--;

System.***out***.println(num1);

}

}

**Logical Operators:**

Logical operators: &&(logical AND), ||(logical OR) and !(Negation)

**package** FPPackage;

**public** **class** ArithmaticOperators {

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

**int** num1 = -10;

**int** num2 = -5;

**boolean** test = **false**;

**if**(num1 > 0 && num2 >0) {

System.***out***.println("Both values are greater than 0");

}

**else** **if**(num1 > 0 || num2 > 0) {

System.***out***.println("One of the values is greater than 0");

}

**else** {

System.***out***.println("Both values are less than 0");

}

**if**(test) {

System.***out***.println("It is true");

}

**if**(!test) {

System.***out***.println("It is false");

}

}

}

# **5B. Java Operators - Comparison Operators and Ternary Operator**

**Operators in Java:**

Types of Operators in Java:

1. Basic Arithmetic Operators
2. Assignment Operators
3. Auto-increment and Auto-decrement Operators
4. Logical Operators
5. **Comparison (relational) operators**
6. **Ternary Operator**

**Comparison (Relational) Operators:**

We have six relational operators in Java: ==, !=, >, <, >=, <=

**package** FPPackage;

**public** **class** ArithmaticOperators {

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

**int** num1 = 10;

**int** num2 = 10;

**if**(num1 == num2) {

System.***out***.println("Both values are equal");

}

**if**(num1 != num2) {

System.***out***.println("The values are not equal");

}

**if**(num1 > num2) {

System.***out***.println("Num1 is greater than Num2");

}

**if**(num1 < num2) {

System.***out***.println("Num1 is less than Num2");

}

**if**(num1 >= num2) {

System.***out***.println("Num1 is greater than or equal to Num2");

}

**if**(num1 <= num2) {

System.***out***.println("Num1 is less than or equal to Num2");

}

}

}

**Ternary Operator:**

This operator evaluates a boolean expression and assign the value based on the result.

Syntax:

variable num1 = (expression) ? value if true : value if false

**package** FPPackage;

**public** **class** ArithmaticOperators {

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

**int** num1 = 10;

**int** num2 = 1;

**int** num3;

num3 = (num1 == num2) ? 5 : 6;

System.***out***.println("Num3 is "+ num3);

}

}

# **5C. Java Operators - Interview Questions**

**What is the % operator?**

It is referred to as the modulo or remainder operator. It returns the remainder of dividing the first operand by the second operand.

Example:

**public** **class** JavaExamples {

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

**int** a = 5;

**int** b = 3;

System.***out***.println("% - Reminder is "+a%b);

}

}

Result: Reminder is 2

**What is the difference between the prefix and postfix forms of the ++ operator?**

The prefix form performs the increment operation and returns the value of the increment operation. The postfix form returns the current value all of the expression and then performs the increment operation on that value.

Example:

**public** **class** JavaExamples {

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

**int** a = 1;

**int** b;

b = a++;

System.***out***.println("b is "+b);

System.***out***.println("a is "+a);

**int** c = 1;

**int** d;

d = ++c;

System.***out***.println("c is "+c);

System.***out***.println("d is "+d);

}

}

Result:

b is 1

a is 2

c is 2

d is 2

**What is the difference between the Boolean & operator and the && operator?**

If an expression involving the Boolean & operator is evaluated, both operands are evaluated. Then the & operator is applied to the operand. When an expression involving the && operator is evaluated, the first operand is evaluated. If the first operand returns a value of true then the second operand is evaluated. The && operator is then applied to the first and second operands. If the first operand evaluates to false, the evaluation of the second operand is skipped.

Ex:

**public** **class** JavaExamples {

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

**int** a = 2;

**int** b = 3;

**int** c = 2;

**if**(((a==b) & (a==c))) {

System.***out***.println("equal values");

}

**else** {

System.***out***.println("values are not equal");

}

}

}

**What are compound assignment operators?**

The operator that combines Arithmetic operator with assignment is known as Compound Assignment operator.

For example, x+=10;

**What is the output of the following?**

1. **int** a, b;

a=10;

b=++a; //Result 11

1. **int** a, b;

a=10;

b=a++; //Result 10

1. **int** a=20;

a= ++a + 1; //Result 22

1. **int** a=20;

a= a++ + 1; //Result 21

1. **int** a=20;

a= ++a + ++a;//Result 43

1. **int** a=20;

a= a++ + a++; //Result 41

1. **int** a=20;

a= a++ + ++a; //Result 42

1. **int** a, b;

a=30;

b=a--; // Result 39

1. **int** a=20;

a= --a - 1; //Result 18

1. **int** a=20;

a= a-- - 1; //Result 19

1. **int** a=20;

a= --a - --a; //Result 1

1. **int** a=20;

a= a-- - a--; //Result 1

1. **int** a=20;

a= a-- - --a; //Result 2

1. **int** a=20;

a= a-- - ++a; //Result 40

# **6. Java Methods - Declaring Methods, Passing Values and Returning Values**

Methods allow us to reuse the code. For example, if you are adding two random numbers a lot of times, instead of writing it every time, you can write a method to add those number and call that method whenever you need to add two numbers.

**Defining a method:**

To define a method, you need to specify

Modifier – Defines access type of method

Return type – The data type of the value returned by the method

Method Name – The name of the method

Parameters list – Comma separated list of the input parameters.

Method body – The code you need to execute. It is enclosed in braces.

Example:

1. You can define a method as

public int add(int a, int b){

}

Here public is the modifier. Int is the return type. Add is the method name. int a and int b are parameters.

1. So, in the above method we are returning an integer.

If we return nothing from the method, we can define it as

public void add(int a, int b){

}

In the above, we don’t want to return anything so return type is void

1. Returning string

public String somemethod(String str1, String str2){

}

Here we are also passing strings.

1. Returning an array of integers

public int[] somemethod(int[] arr){

}

In the above method, we are returning an array of integers so mentioned as int[]

1. Returning an array of Strings

public String[] somemethod(String[] arr){

}

In the above method, we are returning an array of strings so mentioned as int[] and we are also passing an array of strings.

1. Passing and Returning an object

public obj change(obj o){

}

In the above method, if the class name of the method is obj, we have mentioned the return type as class name which means we are return an object of that class. Similarly in the parameters also we are passing an object.

**Calling a method:**

When you are calling a method, you need to capture the value returned by that method.

Example:

1. Calling a method that takes integers as inputs and returns an integer.

Method:

public int add(int a, int b){

}

Calling the method:

int a = add(2, 3);

# **6A. Java Methods - Passing integers and returning integers**

**Calling a Method that takes integer arguments and returns an integer:**

**package** package1;

**public** **class** AddingNumbers {

**public** **int** add(**int** a, **int** b) {

**int** c = a+b;

**return** c;

}

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

**int** a = 2;

**int** b = 2;

AddingNumbers an = **new** AddingNumbers();

System.***out***.println(an.add(a,b));

}

}

1. Calling a method that takes integers as arguments and returns nothing.

Method:

public void add(int a, int b){

}

Calling the method:

add(2, 3)

Example:

**package** package1;

**public** **class** AddingNumbers {

**public** **void** add(**int** a, **int** b) {

**int** c = a+b;

System.***out***.println(c);

}

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

**int** a = 2;

**int** b = 2;

AddingNumbers an = **new** AddingNumbers();

an.add(a,b);

}

}

# **6B. Java Methods - Passing Strings and Returning Strings**

1. Calling a method that returns a string by passing strings

Method:

public String somemethod(String str1, String str2){

}

Calling the method:

String str3 = somemethod(“Subbu”, “Selenium”)

Example:

**package** package1;

**public** **class** StringMethodExamples {

**public** String concatenate(String str1, String str2) {

String str3 = str1+" "+str2;

**return** str3;

}

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

String str1 = "Subbu";

String str2 = "Selenium";

StringMethodExamples sme = **new** StringMethodExamples();

String str3 = sme.concatenate(str1, str2);

System.***out***.println(str3);

}

}

# **6C. Java Methods - Passing Arrays and returning Arrays**

1. Calling a method that returns an array by passing an array

public int[] somemethod(int[] arr){

}

Calling the method:

int[] arr1 = {1, 2, 3};

int[] arr2 = somemethod(arr1)

Example:

**package** package1;

**public** **class** ExceptionExample {

**public** **static** **int**[] mth(**int**[] arr) {

**int**[] arr1 = arr;

**for** (**int** i=0; i<arr.length; i++) {

arr1[i] = arr[i]\*2;

}

**return** arr1;

}

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

**int**[] arr1 = {3,4,5};

**int**[] arr2 = *mth*(arr1);

System.***out***.println(arr2[0]);

System.***out***.println(arr2[1]);

System.***out***.println(arr2[2]);

}

}

1. Returning an array of Strings

public String[] concatenate(String[] arr){

}

Calling method:

String[] arr1 = {“Subbu”, “Selenium”};

String[] arr2 = concatenate(arr1);

**package** package1;

**public** **class** StringMethodExamples {

**public** String[] concatenate(String[] str) {

String[] str1 = str;

str1[0] = str[0]+" tutorials";

str1[1] = str[1]+" tutorials";

**return** str1;

}

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

String[] str = {"Subbu", "Selenium"};

StringMethodExamples sme = **new** StringMethodExamples();

String[] str1 = sme.concatenate(str);

**for**(**int** i=0; i<str1.length; i++) {

System.***out***.println(str1[i]);

}

}

}

# **6D. Java Methods - Passing and Returning an object**

1. Passing and Returning an object

public obj change(obj o){

}

Calling Method:

obj o1 = new obj();

obj o2 = change(o1);

Example:

**package** package1;

**public** **class** OPCA {

**public** **int** a;

**public** OPCA setVariable(OPCA opc) {

OPCA opc1 = **new** OPCA();

opc1.a = opc.a \* 10;

**return** opc1;

}

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

OPCA opc1 = **new** OPCA();

opc1.a = 2;

OPCA opc2 = **new** OPCA();

opc2 = opc2.setVariable(opc1);

System.***out***.println(opc2.a);

}

}

# **6E. Java Methods - Interview Questions**

**How primitive variables passed to methods - by value or by reference?**

In Java, primitive variables are passed to methods by value. If the passed value changes in the method, it does not change the original value.

Example:

**package** intquestions;

**public** **class** testclass {

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

{

**int** x = 5;

*change*(x);

System.***out***.println(x);

}

**public** **static** **void** change(**int** x)

{

x = 10;

}

}

Result: 5

**When we pass an object to a method in java, does it pass by value or reference?**

Java creates a copy of references and pass it to method, but they still point to same memory reference. Mean if set some other object to reference passed inside method, the object from calling method as well its reference will remain unaffected.

The changes are not reflected back if we change the object itself to refer some other location or object

**package** intquestions;

//A Java program to show that references are also passed

//by value.

**class** Test

{

**int** x;

Test(**int** i) { x = i; }

Test() { x = 0; }

}

**class** testclass

{

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

{

// t is a reference

Test t = **new** Test(5);

// Reference is passed and a copy of reference

// is created in change()

*change*(t);

// Old value of t.x is printed

System.***out***.println(t.x);

}

**public** **static** **void** change(Test t)

{

// We changed reference to refer some other location

// now any changes made to reference are not reflected

// back in main

t = **new** Test();

t.x = 10;

}

}

# **7. Java Basics - Method Overloading**

If you have more than one method with the same name but with different arguments or different argument types or different sequence of arguments it is called method overloading.

Three ways of Method Overloading:

1. Different no. of arguments

Ex: details(int age, String name)

details(int age, int sno, String name)

1. Different argument types

Ex: details(int a, int b)

details(String a, String b)

1. Different sequence

Ex: details(int age, String name)

Details(String name, int age)

Example:

Method Overloading – Different Arguments

**package** FPPackage;

**class** Demo {

**public** **static** **void** details(**int** age, String name) {

System.***out***.println("Age is "+age+" Name is "+name);

}

**public** **static** **void** details(**int** age, **int** sno, String name) {

System.***out***.println("Age is "+age+" S.No is "+sno+" Name is "+name);

}

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

*details*(20, "Venkat");

*details*(23, 123, "Ganesh");

}

}

Method Overloading – Different Argument Types

**package** FPPackage;

**class** Demo {

**public** **static** **void** details(**int** a, **int** b) {

System.***out***.println("a is "+a+" b is "+b);

}

**public** **static** **void** details(String a, String b) {

System.***out***.println("a is "+a+" b is "+b);

}

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

*details*(20, 30);

*details*("Venkat", "Ganesh");

}

}

Method Overloading – Different Sequence

**package** FPPackage;

**class** Demo {

**public** **static** **void** details(**int** age, String name) {

System.***out***.println("Age is "+age+" Name is "+name);

}

**public** **static** **void** details(String name, **int** age) {

System.***out***.println("Age is "+age+" Name is "+name);

}

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

*details*(20, "Venkat");

*details*("Ganesh", 22);

}

}

# **7A. Java Basics - Method Overloading - Interview Questions**

**What is method overloading in java ?**

A class having two or more methods with same name but with different arguments then we say that those methods are overloaded. Static polymorphism is achieved in java using method overloading. Method overloading is used when we want the methods to perform similar tasks but with different inputs or values. When an overloaded method is invoked java first checks the method name, and the number of arguments, type of arguments; based on this compiler executes this method. Compiler decides which method to call at compile time. By using overloading static polymorphism or static binding can be achieved in java.

Note: Return type is not part of method signature. we may have methods with different return types but return type alone is not sufficient to call a method in java.

**Can a main() method be overloaded?**

Yes. You can have any number of main() methods with different method signature and implementation in the class.

**Why does Java not support operator overloading?**

Operator overloading makes the code very difficult to read and maintain. To maintain code simplicity, Java doesn't support operator overloading.

**What restrictions are placed on method overloading?**

Two methods may not have the same name and argument list but different return types.

**Can a method be overloaded based on different return type but same argument type ?**

No, because the methods can be called without using their return type in which case there is ambiquity for the compiler.

# **8. Java Constructors - Constructor Uses**

A constructor is a block of code that is used to initialize the variables of a newly created object. A constructor resembles an instance method in java but it’s not a method as it doesn’t have a return type.

Constructor should be declared with the same name as that of class.

Example:

**package** ConstructorDemo;

**public** **class** ConstructorDemo1 {

**int** age;

String name;

ConstructorDemo1(){

**this**.age = 10;

**this**.name = "Subbu";

}

ConstructorDemo1(**int** age, String name){

**this**.age = age;

**this**.name = name;

}

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

ConstructorDemo1 cd1 = **new** ConstructorDemo1();

System.***out***.println("Age is "+cd1.age);

System.***out***.println("Name is "+cd1.name);

ConstructorDemo1 cd2 = **new** ConstructorDemo1(15, "Venkat");

System.***out***.println("Age is "+cd2.age);

System.***out***.println("Name is "+cd2.name);

}

}

# **8A. Java Constructors - Interview Questions**

**Define a constructor?**

A constructor is a method used to initialize the state of an object, and it gets invoked at the time of object creation. Rules for constructor are:

Constructor Name should be same as class name.

A constructor must have no return type.

**When does the compiler supply a default constructor for a class?**

The compiler supplies a default constructor for a class if no other constructors are provided.

**Will the compiler create a default constructor if I have a parameterized constructor in the class?**

No, compiler won’t create default constructor if there is parameterized constructor in the class. For example, if I have a class with no constructors, then compiler will create default constructor. For Example:

public class Car {

}

In the above Car class there are no constructors so compiler creates a default constructor.

public class Car {

Car(String name) {

}

}

In this example compiler won’t create any default constructor because already there is one constructor in the Car class.

**Can we have a method name same as class name in java?**

Yes. we can have method name same as class name it won’t throw any compilation error but it shows a warning message that method name is same as class name.

**Can we override constructors in java?**

Only methods can be overridden in java. Constructors can’t be inherited in java. So, there is no point of overriding constructors in java.

**How to call one constructor from the other constructor?**

* Within same class: It can be done using this() keyword for constructors in same class
* From base class: by using super() keyword to call constructor from the base class.

This is also called constructor chaining.

Example:

**package** intquestions;

//Java program to illustrate Constructor Chaining

//within same class Using this() keyword

**class** testclass

{

// default constructor 1

// default constructor will call another constructor

// using this keyword from same class

testclass()

{

// calls constructor 2

**this**(5);

System.***out***.println("The Default constructor");

}

// parameterized constructor 2

testclass(**int** x)

{

// calls constructor 3

**this**(5, 15);

System.***out***.println(x);

}

// parameterized constructor 3

testclass(**int** x, **int** y)

{

System.***out***.println(x \* y);

}

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

{

// invokes default constructor first

**new** testclass();

}

}

If we want to access parent class constructor, use super method.

**package** intquestions;

**class** baseclass{

String name;

baseclass(String name){

**this**.name = name;

System.***out***.println(name);

}

}

**class** testclass **extends** baseclass

{

testclass(){

**this**("constructor calling");

System.***out***.println("No argument constructor");

}

testclass(String str){

**super**(str);

}

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

testclass t = **new** testclass();

}

}

If we are calling a constructor from other constructor using this keyword, the keyword should be first. Otherwise it gives error.

**package** intquestions;

**class** testclass

{

testclass(){

System.***out***.println("No argument constructor");

**this**("constructor calling");

}

testclass(String str){

System.***out***.println(str);

}

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

testclass t = **new** testclass();

}

}

The above gives an error that “Constructor call must be the first statement in a constructor”.

# **9. Java Basics - Properties Class**

Properties class object “System” maintains a set of system properties for configuration. You can define your own system property as well.

Java itself maintains its own configuration by using Properties object.

Retrieving a system property:

System.getProperty(“java.class.path”);

Retrieving all system properties:

System.getProperties();

Setting up your own system property:

System.setProperty(“TestKey”,”Hello”);

Example:

**package** FPPackage;

**public** **class** PropertiesTest {

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

System.***out***.println(System.*getProperties*());

System.***out***.println(System.*getProperty*("java.class.path"));

System.*setProperty*("TestKey", "Hello");

System.***out***.println(System.*getProperty*("TestKey"));

}

}

# **10. Access Modifiers default, public, protected and Private**

access modifiers are used to control the visibility of a class or a method or a variable or a constructor. There are 4 different access modifiers are available in java. They are – default, public, private and protected.

**default:** default members or members with no access modifier are visible within the package. And they are inherited to only sub classes which reside in the same package. That means they are not inherited and visible outside the package.

**public:** public members are visible everywhere and they are inherited to any sub class.

**private:** private members of a class, whether it is a field or a method or a constructor, can not be accessed outside the class in which they are defined. private members are also not inherited to sub class.

**protected:** protected members have half the characteristics of public members and half the characteristics of default members i.e. protected members are visible within package like default members and they can be inherited to any sub class just like public members.

Class – Access modifiers for class of public and default. Private and Protected can’t be used for class.

Methods – Access modifiers for method are default, public, private, protected.

Variables – Access modifiers for variables are default, public, private and protected.

Create two pacakges FPPackage and SPPackage

Under FPPackage create two classes FirstProgram.java and SecondProgramInFPP.java

**FirstProgram.java:**

**package** FPPackage;

**public** **class** FirstProgram {

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

System.***out***.println("Hello World!");

}

**protected** **int** sum(**int** a, **int** b) {

**int** c = a+b;

**return** c;

}

}

**SecondProgramInFPP.java**

**package** FPPackage;

**public** **class** SecondProgramInFPP {

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

System.***out***.println("Calling First Program in same package");

FirstProgram fp = **new** FirstProgram();

**int** a = 5;

**int** b = 7;

**int** c;

c = fp.sum(a, b);

System.***out***.println("Sum is "+c);

}

}

Under SPPackage create a class SecondProgram.java

**package** SPPackage;

**import** FPPackage.FirstProgram;

**public** **class** SecondProgram **extends** FirstProgram {

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

System.***out***.println("Calling FirstProgram in FPPackage");

FirstProgram fp = **new** FirstProgram();

**int** c = fp.sum(5, 5);

System.***out***.println("Sum is "+c);

}

}

Now modify the access types and check them.

# **10A. Access Modifiers - Interview Questions**

**What access modifiers can be used for class?**

We can use only two access modifiers for class public and default.

public: A class with public modifier can be visible

1) In the same class

2) In the same package subclass

3) In the same package non-subclass

4) In the different package subclass

5) In the different package non-subclass.

default: A class with default modifier can be accessed

1) In the same class

2) In the same package subclass

3) In the same package non-subclass

4) In the different package subclass

5) In the different package non-subclass.

**Explain what access modifiers can be used for methods?**

We can use all access modifiers public, private, protected and default for methods.

public: When a method is declared as public it can be accessed

1) In the same class

2) In the same package subclass

3) In the same package non-subclass

4) In the different package subclass

5) In the different package non-subclass.

default: When a method is declared as default, we can access that method in

1) In the same class

2) In the same package subclass

3) In the same package non-subclass

We cannot access default access method in

1) Different package subclass

2) Different package non-subclass.

protected:

When a method is declared as protected it can be accessed

1) With in the same class

2) With in the same package subclass

3) With in the same package non-subclass

4) With in different package subclass

It cannot be accessed non-subclass in different package.

private: When a method is declared as private it can be accessed only in that class.

It cannot be accessed in

1) Same package subclass

2) Same package non-subclass

3) Different package subclass

4) Different package non-subclass.

**What all access modifiers are allowed for top class?**

For top level class only two access modifiers are allowed.

public and default.

If a class is declared as public it is visible everywhere. If a class is declared default it is visible only in same package.

If we try to give private and protected as access modifier to class we get the below compilation error.

Illegal Modifier for the class only public, abstract and final are permitted.

**package** intquestions;

**class** test{ //Error: Public type must be defined in its file

**public** **static** **void** main() {

System.***out***.println("hello");

}

}

**private** **class** testclass //Error

{

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

System.***out***.println("hi");

test.*main*();

}

}

**Explain what access modifiers can be used for variables?**

We can use all access modifiers public, private, protected and default for variables.

public: When a variable is declared as public it can be accessed

1) In the same class

2) In the same package subclass

3) In the same package nonsubclass

4) In the different package subclass

5) In the different package non subclass.

default: When a variable is declared as default, we can access that method in

1) In the same class

2) In the same package subclass

3) In the same package non subclass

We cannot access default access variables in

4) Different package subclass

5) Different package non subclass.

protected: When a variables is declared as protected it can be accessed

1) With in the same class

2) With in the same package subclass

3) With in the same package non subclass

4) With in different package subclass

It cannot be accessed non subclass in different package

private: When a variable is declared as private it can be accessed only in that class.

It cannot be accessed in

1) Same package subclass

2) Same package non subclass

3) Different package subclass

4) Different package non subclass.

**Can a class declared as private be accessed outside its package?**

Not possible.

**Can a class be declared as protected?**

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in an interface cannot be declared protected.

**What is the access scope of a protected method?**

A protected method can be accessed by the classes within the same package or by the subclasses of the class in any package.

**If a variable is declared as private, where may the variable be accessed?**

A private variable may only be accessed within the class in which it is declared.

**What do you understand by private, protected and public?**

These are accessibility modifiers. Private is the most restrictive, while public is the least restrictive. There is no real difference between protected and the default type (also known as package protected) within the context of the same package, however the protected keyword allows visibility to a derived class in a different package.

**What modifiers may be used with an inner class that is a member of an outer class?**

A (non-local) inner class may be declared as public, protected, private, static, final, or abstract.

**If a class is declared without any access modifiers, where may the class be accessed?**

A class that is declared without any access modifiers is said to have package access. This means that the class can only be accessed by other classes and interfaces that are defined within the same package.

**If a method is declared as protected, where may the method be accessed?**

A protected method may only be accessed by classes or interfaces of the same package or by subclasses of the class in which it is declared.

**What is the difference between a public and a non-public class?**

A public class may be accessed outside of its package. A non-public class may not be accessed outside of its package.

# **11. Java Non Access Modifiers - Static variables, methods, blocks and classes**

Java provides some other modifiers to provide the functionalities other than the visibility. These modifiers are called **Non-Access Modifiers**. There are many non-access modifiers available in java. Each modifier has it’s own functionality. Some of the most used non-access modifiers are listed below.

**static**: The members which are declared as static are common to all instances of a class. Static members are class level members which are stored in the class memory.

**final**: This modifier is used to restrict the further modification of a variable or a method or a class. The value of a variable which is declared as final can’t be modified once it gets a value. A final method can’t be overridden in the sub class and you can’t create a sub class to a final class. See [this](https://javaconceptoftheday.com/final-keyword-in-java/) post for more info on final keyword in java.

**abstract**: This modifier can be used either with a class or with a method. You can’t apply this modifier to variable and constructor. A method which is declared as abstract must be modified in the sub class. You can’t instantiate a class which is declared as abstract.

**Static:**

Static keyword can be used with class, variable, method and block. Static members belong to the class instead of a specific instance, this means if you make a member static, you can access it without object with in the same class/child class and with class name in a different class.

**Static Variables:**

**package** StaticPackage;

**public** **class** Static1 {

**int** count = 0;

**public** **void** count() {

count++;

}

**public** **int** returncount() {

**return** count;

}

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

System.***out***.println("Hello");

Static1 s1 = **new** Static1();

Static1 s2 = **new** Static1();

s1.count();

**int** c = s1.returncount();

**int** d = s2.returncount();

System.***out***.println("Count value is "+c);

System.***out***.println("Count value is "+d);

}

}

Remove the static keyword before the variable and observe the results.

Change the static keyword to final and observe the results.

**Static Methods:**

We can call a static method without any object because when we make a member static it becomes class level. If we remove the static keyword and make it non-static then we must need to create an object of the class in order to call it.

**package** FPPackage;

**class** StaticMethod

{

// This is a static method

//**void** myMethod()

static **void** myMethod()

{

System.***out***.println("myMethod");

}

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

{

/\* You can see that we are calling this

\* method without creating any object.

\*/

//StaticMethod s = **new** StaticMethod();

//s.myMethod();

myMethod();

}

}

In the above example, we have created an object to access the method “myMethod()”. If we declare this method as static, we don’t have to create an object.

**Static Block:**

Static block is used for initializing the static variables. This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

**package** FPPackage;

**public** **class** StaticBlock {

**static** **int** *num*;

**static** String *text*;

**static** {

System.***out***.println("Initiating variables in block 1");

*num* = 10;

*text* = "hello";

}

**static** {

System.***out***.println("Initiating variables in block 2");

*num* = 11;

*text* = "hello2";

}

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

System.***out***.println("Value of num is "+*num*);

System.***out***.println("Value of text is "+*text*);

}

}

**Static Class:**

A class can be made static only if it is a nested class.

Nested static class doesn’t need reference of Outer class

A static class cannot access non-static members of the Outer class.

**package** FPPackage;

**class** StaticClass{

**private** **static** String *str* = "Static Class Example";

//Static class

**static** **class** MyNestedClass{

//non-static method

**public** **void** disp() {

/\* If you make the str variable of outer class

\* non-static then you will get compilation error

\* because: a nested static class cannot access non-

\* static members of the outer class.

\*/

System.***out***.println(*str*);

}

}

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

{

/\* To create instance of nested class we didn't need the outer

\* class instance but for a regular nested class you would need

\* to create an instance of outer class first

\*/

StaticClass.MyNestedClass obj = **new** StaticClass.MyNestedClass();

obj.disp();

}

}

# **11A. Java Non Access Modifiers - Static Variables, Methods, Blocks and Classes - Interview Questions – Part 1**

**What is**[**static keyword in Java**](http://www.javainterviewpoint.com/use-of-static-keyword-in-java/)**?**

Static is a Non-Access Modifier. When we have something as static, that means it belongs to the class and not the object. Static can be applied to variable, method, nested class and initialization blocks (static block).

**What is a static variable?**

Static variables belong to the class and static variables are shared by all instances of a class.

A Static variable gets memory allocated only once during the time of class loading.

All the instance of the class share the same copy of the variable, a static variable can be accessed directly by calling “<<ClassName>>.<<VariableName>>” without need to create instance for the class.

value of a static variable will be common for all instances

public class StaticVariableExample

{

static int a =10;

public static void main(String args[]){

StaticVariableExample s1 = new StaticVariableExample();

StaticVariableExample s2 = new StaticVariableExample();

System.out.println("s1.a value :"+s1.a);

System.out.println("s2.a value :"+s2.a);

//Change s1 a value alone

s1.a=20;

System.out.println("s1.a value :"+s1.a);

System.out.println("s2.a value :"+s2.a);

}

}

**Output will be**  
s1.a value :10  
s2.a value :10  
s1.a value :20  
s2.a value :20

**Local variables** cannot be assigned as static it will throw compile time error **“illegal start of expression”**, as the memory cannot be assigned during class load.

**How do we access static members in java?**

Instance variables and instance methods can be accessed using reference variable. But to access static variables or static methods we use Class name in java.

**What is a static method?**

A static method belongs to class rather than object. It can be called directly by using the classname “<<ClassName>>.<<MethodName>>”

A static method can access static variables directly and it cannot access non-static variables and can only call a static method directly and it cannot call a non-static method from it.

Only the main() method which is static will be called by the JVM automatically, Not all the static method will be called automatically.

**Can Static methods access instance variables in java?**

No. Instance variables can’t be accessed in static methods. When we try to access instance variable in static method we get compilation error. The error is as follows:

“Cannot make a static reference to the non static field name”

Though Static methods cannot access the instance variables directly, they can access them using instance handler.

**When are static variables loaded in memory?**

They are loaded at runtime when the respective Class is loaded.

**Can a static block exist without a main() method ?**

Yes. You can have static block alone in the class without a main method.

**Can we**[**Overload**](http://www.javainterviewpoint.com/java-method-overloading-example/)**static methods in Java?**

**Yes**, you can overload a static method in Java.

**Can we**[**Override**](http://www.javainterviewpoint.com/what-is-method-overriding-in-java/)**static methods in Java**

No, you cannot override a static method in [Java](https://www.javainterviewpoint.com/category/core-java/) as there will not be any [Run-time Polymorphism](http://www.javainterviewpoint.com/run-time-polymorphism-and-compile-time-in-java/) happening.

**Why main() method is declared as static ?**

main() method is called by the JVM even before the instantiation of the class hence it is declared as static.

If our main() method is not declared as static then the JVM has to create object first and call which causes the problem of having extra memory allocation.

**What is a static block ?**

A static block, is a block of code inside a Java class that will be executed when a class is first loaded in to the JVM. Mostly the static block will be used for initializing the variables.

Static block will be called only once while loading and it **cannot have any return type**, or any keywords (**this** or **super**).

class test

{

int val;

static{

val = 100;

}

}

**Can we have multiple static blocks in our code ?**

Yes, we can have more than one static block in our code. It will be executed in the same order it is written.

**What is a static class?**

In Java only nested classes are allowed to be declared as static, a top level class cannot be declared as static.

Even though static classes are nested inside a class, they don’t need the reference of the outer class they act like outer class only.

**Can constructors be static in Java?**

In general, a static method means that “The Method belong to class and not to any particular object” but a constructor is always invoked with respect to an object, so it makes no sense for a constructor to be **static**.

**Why**[**abstract method**](http://www.javainterviewpoint.com/abstract-class-java/)**cannot be static in Java ?**

Suppose when you have a concrete method in an abstract class then that method can be static. Suppose we have a class like below

public class AbstractTest

{

static void disp()

{

System.out.println("disp of static method");

}

}

Then the disp() can be access by “AbstractTest.disp()”  
However, for the same reason cannot be applied when you declare a static method to be abstract. Since static method can be called directly, making it abstract would make it possible to call an undefined method which is of no use, hence it is not allowed.

**Can**[**Interface in Java**](http://www.javainterviewpoint.com/interface-java/)**have static methods in it?**

No, Interface cannot have static methods in it because all methods are [implicitly abstract](http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.4). This is why an interface cannot have a static method.

**Can abstract class have static variable in it ?**

Yes, an abstract class can have static variables in it.

# **11B. Java Non Access Modifiers - Static Variables, Methods, Blocks and Classes - Interview Questions – Part B**

**Can abstract class have static variable in it ?**

Yes, an abstract class can have static variables in it.

**When will you define a method as static?**

When a method needs to be accessed even before the creation of the object of the class then we should declare the method as static.

**What is the restriction imposed on a static method or a static block of code?**

A static method should not refer to instance variables without creating an instance and cannot use "this" operator to refer the instance.

**non-static method cannot be referenced from a static context?**

public class Test

{

/\*\* Non Static main method with String[] args\*\*/

public static void main(String[] args)

{

welcome();

}

void welcome()

{

System.out.println("Welcom to JavaInterviewPoint");

}

}

The welcome() method which we tried calling is an instance-level method, we do not have an instance to call it . static methods belong to the class, non-static methods belong to instances of the class and hence it throws the error ”non-static method cannot be referenced from a static context“.

**Can a class be declared as static?**

We can not declare top level class as static, but only inner class can be declared static.

public class Test

{

static class InnerClass

{

public static void InnerMethod()

{ System.out.println("Static Inner Class!"); }

}

public static void main(String args[])

{

Test.InnerClass.InnerMethod();

}

}

//output: Static Inner Class!

**I want to print "Hello" even before main() is executed. How will you acheive that?**

Print the statement inside a static block of code. Static blocks get executed when the class gets loaded into the memory and even before the creation of an object. Hence it will be executed before the main() method. And it will be executed only once.

**package** intquestions;

**class** testclass

{

**static** {

System.***out***.println("hello");

}

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

}

}

This will print the result as “hello”.

**What is the importance of static variable?**

static variables are class level variables where all objects of the class refer to the same variable. If one object changes the value then the change gets reflected in all the objects.

**Can we declare a static variable inside a method?**

Static variables are class level variables and they can't be declared inside a method. If declared, the class will not compile.

**What is the difference between a static and a non-static inner class?**

A non-static inner class may have object instances that are associated with instances of the class's outer class. A static inner class does not have any object instances.

**What happens to a static variable that is defined within a method of a class ?**

Can't do it. You'll get a compilation error.

**How many static initializers can you have?**

As many as you want, but the static initializers and class variable initializers are executed in textual order and may not refer to class variables declared in the class whose declarations appear textually after the use, even though these class variables are in scope.

**Why main() method is public, static and void in java ?**

public : “public” is an access specifier which can be used outside the class. When main method is declared public it means it can be used outside class.

static : To call a method we require object. Sometimes it may be required to call a method without the help of object. Then we declare that method as static. JVM calls the main() method without creating object by declaring keyword static.

void : void return type is used when a method does’nt return any value . main() method does’nt return any value, so main() is declared as void.

**Explain the scope or life time of class variables or static variables?**

Static variables do not belong to instances of the class. We can access static fields even before instantiating the class. Static variable remain in memory till the life time of application.

**Explain about static imports in java?**

From Java 5.0 we can import static variables in to source file. Importing static member to source file is referred as static import. The advantage of static import is we can access static variables without class or interface name.

Syntax :

import static packagename.classname.staticvariablename;

Ex : import static com.abc.Employee.eno;

To import all static variables from a class in to our source file we use \*.

import static com.abc.Employee.\*

# **12. Java Non Access Modifiers - Abstract Classes and Methods**

**Abstract Class**

A class that is declared using “abstract” keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (regular methods with body). A normal class (non-abstract class) cannot have abstract methods.

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

Why we need an abstract class?

Let’s say we have a class bird. A bird color changes from bird to bird so we declare the color() method as abstract and we don’t need to implement this in parent class. But birds will fly so this can be implemented as a normal method in parent class.

Parent Class:

**package** FPPackage;

**public** **abstract** **class** Bird {

**public** **abstract** **void** color();

**public** **static** String fly() {

**return** "I fly";

}

}

Child Class:

**package** FPPackage;

**public** **class** Parrot **extends** Bird{

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

// **TODO** Auto-generated method stub

String flystring = *fly*();

Parrot p = **new** Parrot();

p.color();

System.***out***.println(flystring);

}

**public** **void** color() {

System.***out***.println("My color is green");

}

}

Note 1: As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Note 2: Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods (those that were abstract in parent but implemented in child class).

Note 3: If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

**Why can’t we create the object of an abstract class?**

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen? There would be no actual implementation of the method to invoke.  
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

**Abstract class vs Concrete class**

A class which is not abstract is referred as **Concrete class**.

**Key Points:**

An abstract class has no use until unless it is extended by some other class.

If you declare an abstract method in a class then you must declare the class abstract as well. you can’t have abstract method in a concrete class. It’s vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.

It can have non-abstract method (concrete) as well.

# **12A. Java Non Access Modifiers - Abstract Classes and Methods - Interview Questions**

**How To Describe Abstraction In Interview?**

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

A method that is declared as abstract and does not have implementation is known as abstract method.

There are two ways to achieve abstraction in java

1- By Abstract class (0 to 100%)  , 2- By Interface (100%)

**What is abstraction and abstract class in Java?**

**Abstraction:**

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

Abstract class (0 to 100%)  
Interface (100%)

**Abstract class in Java:**

A class that is declared as abstract is known as abstract class. It needs to be extended and its method should be implemented. It cannot be instantiated.

Example abstract class

abstract class A{}

The important features of abstract classes are :

1) Abstract classes cannot be instantiated.

2) An abstract classe contains abstract methods, concrete methods or both.

3) Any class which extends abstract class must override all methods of abstract class.

4) An abstract class can contain either 0 or more abstract methods.

**Abstract method:**

A method that is declared as abstract and does not have implementation is known as abstract method. It’s implementation is deferred to the child class.  
Example abstract method

abstract void printStatus();//no body and abstract

**Can abstract class have constructors in Java?**

Yes, abstract class can declare and define constructor in Java. Since you can not create instance of abstract class, constructor can only be called during constructor chaining, i.e. when you create instance of concrete implementation class.

**Can abstract class implements interface in Java? does they require to implement all methods?**

Yes, abstract class can implement interface by using implements keyword. Since they are abstract, they don’t need to implement all methods.

It’s good practice to provide an abstract base class, along with an interface to declare Type.

**Can abstract class be final in Java?**

No, abstract class cannot be final in Java. Making them final will stop abstract class from being extended, which is the only way to use abstract class.

They are also opposite of each other, abstract keyword enforces to extend a class, for using it, on the other hand, final keyword prevents a class from being extended.

In real world also, abstract signifies incompleteness, while final is used to demonstrate completeness. Bottom line is, you can not make your class abstract and final in Java, at same time, it’s a compile time error.

**Can you create instance of abstract class? Can you create an object of abstract class?**

No, you can not create instance of abstract class in Java, they are incomplete.

Even though, if your abstract class don’t contain any abstract method, you can not create instance of it.

By making a class abstract, you told compiler that, it’s incomplete and should not be instantiated. Java compiler will throw error, when a code tries to instantiate abstract class.

**Is it necessary for abstract class to have abstract method?**

No, It is not mandatory for an abstract class to have any abstract method. We can make a class abstract in Java, by just using abstract keyword in its declaration.

**What is abstract method in Java?**

An abstract method is a method without body.  It  uses abstract keyword in method declaration. All method declared inside Java Interface are by default abstract. Here is an example of abstract method in Java

public void abstract printVersion();

Now, In order to implement this method, We need to extend this abstract class and override its method.

**Can abstract class contains main method in Java?**

Yes, abstract class can contain main method, it just another static method and you can execute Abstract class with main method, until you don’t create any instance.

**What is use of a abstract variable?**

Variables can't be declared as abstract. only classes and methods can be declared as abstract.

**Can a abstract class be defined without any abstract methods?**

Yes, it's possible. This is basically to avoid instance creation of the class.

**What does it mean that a method or class is abstract?**

An abstract class cannot be instantiated. Abstract methods may only be included in abstract classes. However, an abstract class is not required to have any abstract methods, though most of them do. Each subclass of an abstract class must override the abstract methods of its superclasses or it also should be declared abstract.

# **13. Java Non Access Modifiers - Final Keyword**

**Final**

Final is a non-access modifier and can be applied to variables, methods and classes.

Final Variable – Cannot change the variable or assign new values.

Final Methods – Cannot do method overriding for these.

Final Classes – Cannot inherit these classes.

Final Variables: Final variables cannot be changed and if you try to change them you will get a compilation error “The final field cannot be assigned”.

Example:

**package** FPPackage;

**public** **class** FinalKeywordEx {

**final** String str = "Subbu's Selenium Tutorials";

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

FinalKeywordEx fke = **new** FinalKeywordEx();

System.***out***.println("Final variable a is "+ fke.str);

fke.str = "Java and selenium tutorials";//Error: The final field FinalKeywordEx.str cannot be assigned

}

}

Final Methods: Final methods cannot be overridden. It gives an error that “Cannot override the final method from parent”.

Example:

**package** FPPackage;

**class** ParentClass {

**public** **final** **void** methodA() {

System.***out***.println("This is final method");

}

}

**public** **class** FinalKeywordEx **extends** ParentClass{

**public** **final** **void** methodA() { //Error: Cannot override the final method from ParentClass

System.***out***.println("Final method can't be overridden");

}

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

}

}

Final Class: Final classes cannot be inherited. If you try to inherit them, you will get the error “The type child class cannot subclass the final class parent class”

Example:

**package** FPPackage;

**final** **class** ParentClass {

}

**public** **class** FinalKeywordEx **extends** ParentClass{ //Error: The type FinalKeywordEx cannot subclass the final class ParentClass

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

}

}

# **13A. Java Non Access Modifiers - Final Keyword - Interview Questions**

**Can a main() method be declared final?**

Yes. Any inheriting class will not be able to have it's own default main() method. Because final method can’t be overridden.

**package** FPPackage;

**class** ParentClass {

**public** **static** **final** **void** main(String[] args) {

}

}

**public** **class** FinalKeywordEx **extends** ParentClass{

**public** **static** **void** main(String[] args) { //Error: Can not override the final method from parent class

}

}

**What is the purpose of declaring a variable as final?**

A final variable's value can't be changed. final variables should be initialized before using them.

A general example is constructing a house. When you are constructing a house, the total sq. foot available is constant. So, this is a final variable.

**What is the impact of declaring a method as final?**

A method declared as final can't be overridden. A sub-class can't have the same method signature with a different implementation.

**I don't want my class to be inherited by any other class. What should i do?**

You should declared your class as final. But you can't define your class as final, if it is an abstract class. A class declared as final can't be extended by any other class.

**Can you give few examples of final classes defined in Java API?**

java.lang.String, java.lang.Math are final classes.

**How is final different from finally and finalize()?**

final is a modifier which can be applied to a class or a method or a variable. final class can't be inherited, final method can't be overridden and final variable can't be changed.   
  
finally is an exception handling code section which gets executed whether an exception is raised or not by the try block code segment.   
  
finalize() is a method of Object class which will be executed by the JVM just before garbage collecting object to give a final chance for resource releasing activity.

**What does it mean that a class or member is final?**

A final class cannot be inherited. A final method cannot be overridden in a subclass. A final field cannot be changed after it's initialized, and it must include an initializer statement where it's declared.

**What are constants and how to create constants in java?**

Constants are fixed values whose values cannot be changed during the execution of program. We create constants in java using final keyword.

Ex : final int number =10;

final String str=”Subbus Selenium Tutorials”

# **14. Java Interfaces**

Abstract class is used for partial abstraction. Interface is used for full abstraction.

Abstraction is a process where you show only relevant data and hide unnecessary details of an object from the user.

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default.

Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However, you can implement more than one interfaces in your class.

Syntax:

interface DemoInterface{

public void testmethod1();

public void testmethod2();

}

Example:

TestInterface Interface:

**package** FPPackage;

**public** **interface** TestInterface {

**public** **void** test1();

**public** **void** test2();

}

ClassForInterface Class:

**package** FPPackage;

**public** **class** ClassForInterface **implements** TestInterface{

**public** **void** test1() {

System.***out***.println("Implementing test1");

}

**public** **void** test2() {

System.***out***.println("Implementing test2");

}

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

TestInterface cl = **new** ClassForInterface();

cl.test1();

cl.test2();

}

}

An interface cannot implement another interface. It has to extend another interface.

Example:

Interface TestInterface:

**package** FPPackage;

**public** **interface** TestInterface {

**public** **void** test1();

**public** **void** test2();

}

Interface TestInterface1:

**package** FPPackage;

**public** **interface** TestInterface1 **extends** TestInterface{

**public** **void** test3();

**public** **void** test4();

}

Class ClassForInterface:

**package** FPPackage;

**public** **class** ClassForInterface **implements** TestInterface1{

**public** **void** test1() {

System.***out***.println("Implementing test1");

}

**public** **void** test2() {

System.***out***.println("Implementing test2");

}

**public** **void** test3() {

System.***out***.println("Implementing test3");

}

**public** **void** test4() {

System.***out***.println("Implementing test4");

}

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

TestInterface cl = **new** ClassForInterface();

cl.test1();

cl.test2();

cl.test3();

cl.test4();

}

}

Even though the class is implementing only TestInterface1 it has to implement all the methods of the TestInterface also as TestInterface1 is extending TestInterface.

# **14A. Java Interfaces – Key Points**

**Key points:**

1) We can’t instantiate an interface in java. That means we cannot create the object of an interface

2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.

3) implements keyword is used by classes to implement an interface.

4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.

5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.

6) Interface cannot be declared as private, protected or transient.

7) All the interface methods are by default abstract and public.

8) Variables declared in interface are public, static and final by default.

interface Try

{

int a=10;

public int a=10;

public static final int a=10;

final int a=10;

static int a=0;

}

All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

interface Try

{

int x;//Compile-time error

}

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static final by default and final variables cannot be re-initialized.

class Sample implements Try

{

public static void main(String args[])

{

x=20; //compile time error

}

}

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A class can implement any number of interfaces.

13) If there are two or more same methods in two interfaces and a class implements both interfaces, implementation of the method once is enough.

interface A

{

public void aaa();

}

interface B

{

public void aaa();

}

class Central implements A,B

{

public void aaa()

{

//Any Code here

}

public static void main(String args[])

{

//Statements

}

}

14) A class cannot implement two interfaces that have methods with same name but different return type.

interface A

{

public void aaa();

}

interface B

{

public int aaa();

}

class Central implements A,B

{

public void aaa() // error

{

}

public int aaa() // error

{

}

public static void main(String args[])

{

}

}

15) Variable names conflicts can be resolved by interface name.

interface A

{

int x=10;

}

interface B

{

int x=100;

}

class Hello implements A,B

{

public static void Main(String args[])

{

/\* reference to x is ambiguous both variables are x

\* so we are using interface name to resolve the

\* variable

\*/

System.out.println(x);

System.out.println(A.x);

System.out.println(B.x);

}

}

# **14B. Java Interfaces - Interview Questions**

**Define interface in java?**

Interface is collection of abstract methods and constants. An interface is also defined as pure or 100 percent abstract class. Interfaces are implicitly abstract whether we define abstract access modifier or not. A class implementing interface overrides all the abstract methods defined in interface. Implements keyword is used to implement interface.

**What is the purpose of interface?**

Interface is a contract. Interface acts like a communication between two objects. When we are defining an interface, we are defining a contract what our class should do but not how it does. An interface doesn’t define what a method does. The power of interface lies when different classes that are unrelated can implement interface. Interfaces are designed to support dynamic method resolution at run time.

**Explain features of interfaces in java?**

1) All the methods defined in interfaces are implicitly abstract even though abstract modifier is not declared.

2) All the methods in interface are public whether they are declared as public or not.

3) variables declared inside interface are by default public, static and final.

4) Interfaces cannot be instantiated.

5) we cannot declare static methods inside interface.

6) ‘implements’ keyword is used to implement interface.

7) Unlike class, interface can extend any number of interfaces.

8) We can define a class inside interface and the class acts like inner class to interface.

9) An interface can extend a class and implement an interface

10) Multiple inheritance in java is achieved through interfaces.

**Class C implements Interface I containing method m1 and m2 declarations. Class C has provided implementation for method m2. Can i create an object of Class C?**

No not possible. Class C should provide implementation for all the methods in the Interface I. Since Class C didn't provide implementation for m1 method, it has to be declared as abstract. Abstract classes can't be instantiated.

**Can a method inside a Interface be declared as final?**

No not possible. Doing so will result in compilation error. public and abstract are the only applicable modifiers for method declaration in an interface.

**Can an Interface implement another Interface?**

Interfaces don't provide implementation hence an interface cannot implement another interface. Interface extends another interface.

**Can an Interface extend another Interface?**

Yes an Interface can inherit another Interface, for that matter an Interface can extend more than one Interface.

**Can a Class extend more than one Class?**

Not possible. A Class can extend only one class but can implement any number of Interfaces.

**Why is an Interface be able to extend more than one Interface but a Class can't extend more than one Class?**

Basically Java doesn't allow multiple inheritance, so a Class is restricted to extend only one Class. But an Interface is a pure abstraction model and doesn't have inheritance hierarchy like classes(do remember that the base class of all classes is Object). So an Interface is allowed to extend more than one Interface.

**Can an Interface be final?**

Not possible. Doing so so will result in compilation error.

**Can we define private and protected modifiers for variables in interfaces?**

No.

**What modifiers are allowed for methods in an Interface?**

Only public and abstract modifiers are allowed for methods in interfaces.

**When can an object reference be cast to an interface reference?**

An object reference be cast to an interface reference when the object implements the referenced interface.

**Can we define static methods inside interface?**

We can’t declare static methods inside interface. Only instance methods are permitted in interfaces. Only public and abstract modifiers are permitted for interface methods. If we try to declare static methods inside interface we get compilation error saying “Illegal modifier for the interface method Classname.methodName(); only public & abstract are permitted”.

**Difference between abstract class and interface?**

|  |  |
| --- | --- |
| **Interface** | **Abstract Class** |
| Interface contains only abstract methods | Abstract class contains abstract methods, concrete methods or both |
| Access Specifiers for methods in interface must be public | Except private we can have any access specifier for methods in abstract class. |
| Variables defined must be public , static , final | Except private variables can have any access specifiers |
| Multiple Inheritance in java is implemented using interface | We cannot achieve multiple inheritance using abstract class. |
| To implement an interface we use implements keyword | To extend an abstract class we use extends keyword. |

# **15. Java Inheritance**

The process by which one class acquires the properties (data members) and functionalities(methods) of another class is called inheritance. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from another class.

Child Class:  
The class that extends the features of another class is known as child class, sub class or derived class.

Parent Class:  
The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

Example:

ParentClass.java

**package** FPPackage;

**public** **class** ParentClass {

String name = "Subbu";

String role = "Manager";

String company = "IBM";

**public** **static** **void** displayDetails() {

ParentClass pc = **new** ParentClass();

System.***out***.println("Name is "+pc.name);

System.***out***.println("Role is "+pc.role);

System.***out***.println("Company is "+pc.company);

}

}

ChildClass.java

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

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

ChildClass cl = **new** ChildClass();

System.***out***.println("Name is "+cl.name);

System.***out***.println("Role is "+cl.role);

System.***out***.println("Company is "+cl.company);

}

}

# **15A. Constructors and Inheritance**

**Constructors and Inheritance:**

When the constructor of a child class is invoked, it by default invokes the constructor of the parent class.

**Parent Class:**

**package** FPPackage;

**public** **class** ParentClass {

String name = "Subbu";

String role = "Manager";

String company = "IBM";

ParentClass(){

System.***out***.println("Parent Class Constructor");

}

**public** **static** **void** displayDetails() {

ParentClass pc = **new** ParentClass();

System.***out***.println("Name is "+pc.name);

System.***out***.println("Role is "+pc.role);

System.***out***.println("Company is "+pc.company);

}

}

**Child Class:**

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

ChildClass(){

System.***out***.println("Child Class Constructor");

}

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

ChildClass cl = **new** ChildClass();

System.***out***.println("Name is "+cl.name);

System.***out***.println("Role is "+cl.role);

System.***out***.println("Company is "+cl.company);

}

}

# **15B. Inheritance and Method Overriding**

**Inheritance and Method Overriding:**

If we declare the same method of parent class in child class, it is called method overriding.

We can call the child method class using child class object.

We can call the parent class method using super keyword.

Example:

ParentClass:

**package** FPPackage;

**public** **class** ParentClass {

ParentClass(){

System.***out***.println("Parent Class Constructor");

}

**public** **void** displayDetails() {

System.***out***.println("I am parent class");

}

}

ChildClass:

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

ChildClass(){

System.***out***.println("Child Class Constructor");

}

**public** **void** displayDetails() {

System.***out***.println("I belong to child class");

**super**.displayDetails();

}

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

ChildClass cl = **new** ChildClass();

cl.displayDetails();

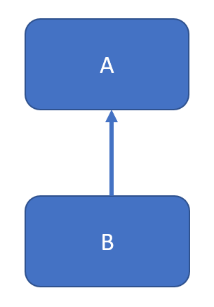
}

}

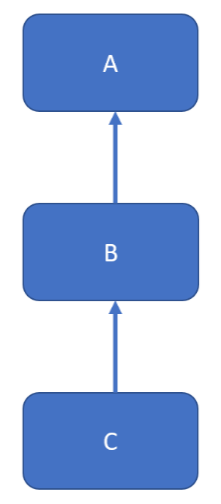
# **15C. Different Types of Inheritance**

**Types of Inheritance:**

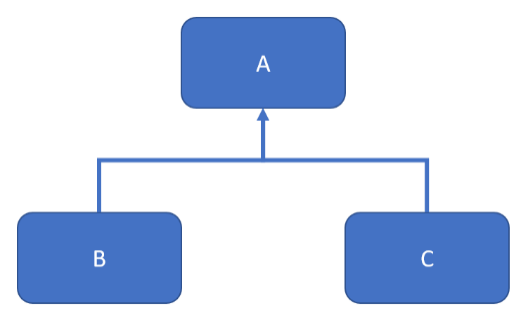
**Single Inheritance:** When a class extends to another one class only, then it is called single inheritance.



**Multilevel Inheritance:** In Multilevel inheritance, we have a base class and we extend this class from a child class and this class will be extended from another child class.

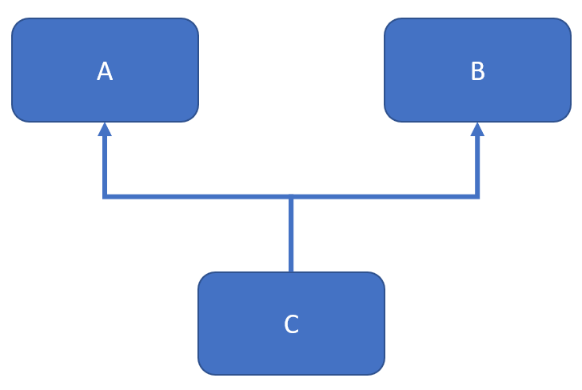
****

**Hierarchical Inheritance:** In hierarchical inheritance a single class will be extended by multiple classes.

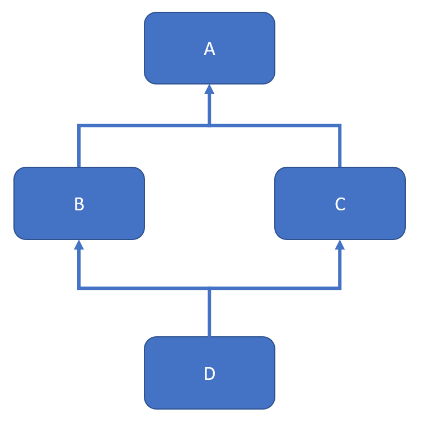


**Multiple Inheritance:** In multiple inheritance a single class extends multiple classes.

Java doesn’t support multiple inheritance for classes. It is supported for interfaces only.



**Hybrid Inheritance:** It’s a combination of single and multiple inheritance.



# **15D. Inheritance - Interview Questions**

**What is Inheritance?**

Inheritance is a concept where one class shares the structure and behaviour defined in another class. If inheritance applied on one class is called Single Inheritance, and if it depends on multiple classes, then it is called multiple Inheritance.

**Explain importance of inheritance in java?**

Reusability: The major advantage of inheritance is code reuse. We can avoid duplicating code by using inheritance. We can place all common state and behaviour in that class, by extending that class we can

Extendability : We can add new functionality to our application without touching the existing code. For example if we take Ms word we came across number of versions of msword such as word 2003,2007. Everytime they won’t write new code they reuse the existing code and some more features.

**Does Java support multiple inheritance?**

Java doesn't support multiple inheritance.

**Does a class inherit the constructors of its superclass?**

A class does not inherit constructors from any of its super classes.

**Explain about field hiding in java?**

If superclass and subclass have same fields subclass cannot override superclass fields. In this case subclass fields hides the super class fields. If we want to use super class variables in subclass we use super keyword to access super class variables.

**What are the different types of inheritances supported by Java?**

Java supports:

Single Inheritance

Multilevel Inheritance

Hierarchical Inheritance

Java doesn’t support Multiple Inheritance and Hybrid Inheritance.

**What is the difference between Inheritance and Encapsulation?**

Inheritance is an object oriented concept which creates a parent-child relationship. It is one of the ways to reuse the code written for parent class but it also forms the basis of Polymorphism. On the other hand, Encapsulation is an object oriented concept which is used to hide the internal details of a class e.g. HashMap encapsulate how to store elements and how to calculate hash values.

**What is the difference between Inheritance and Abstraction?**

Abstraction is an object oriented concept which is used to simply hide things by abstracting details. It helps in the designing system. On the other hand, Inheritance allows code reuse. You can reuse the functionality you have already coded by using Inheritance.

**Can we override a private method in Java?**

No, you cannot override a private method in Java because the private method is not inherited by the subclass in Java, which is essential for overriding. In fact, a private method is not visible to anyone outside the class and, more importantly, a call to the private method is resolved at compile time by using Type information as opposed to runtime by using the actual object.

**Can a class implement more than one interface in Java?**

Yes, A class can implement more than one interface in Java e.g. A class can be both Comparable and Serializable at the same time. This is why the interface should be the best use for defining Type as described in Effective Java. This feature allows one class to play a polymorphic role in the program.

**Can a class extends more than one class in Java?**

No, a class can only extend just one more class in Java.  Though Every class also, by default extend the java.lang.Object class in Java.

**Can an interface extend more than one interface in Java?**

Yes, unlike classes, an interface can extend more than one interface in Java.

**How to call a method of a subclass, if you are holding an object of the subclass in a reference variable of type superclass?**

You can call a method of the subclass by first casting the object hold by reference variable of superclass into the subclass. Once you hold the object in subclass reference type, you can call methods from the subclass. See how type casting works in Java for more details.

# **16. Java Polymorphism**

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

**Types of Polymorphism:**

1. Static Polymorphism / Compile time polymorphism
2. Dynamic Polymorphism / Run time polymorphism

Method overloading is static polymorphism.

Method overriding is dynamic polymorphism.

**Example:**

**Man.java**

**package** FPPackage;

**public** **class** Man {

**public** **void** action() {

System.***out***.println("Crawl, Walk and Run");

}

}

**Child.java**

**package** FPPackage;

**public** **class** Child **extends** Man{

**public** **void** action() {

System.***out***.println("Crawl");

}

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

Man m = **new** Child();

m.action();

}

}

**Youngster.java**

**package** FPPackage;

**public** **class** Youngster **extends** Man{

**public** **void** action() {

System.***out***.println("Walk and Run");

}

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

Man m = **new** Youngster();

m.action();

}

}

**Oldman.java**

**package** FPPackage;

**public** **class** Oldman **extends** Man{

**public** **void** action() {

System.***out***.println("Walk");

}

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

Man m = **new** Oldman();

m.action();

}

}

# **16A. Java Polymorphism - Interview Questions**

**Which object-oriented concept is achieved by using overloading and overriding?**

Polymorphism.

**What is polymorphism and what are the types of it?**

Single task can be done in different way.

Method overloading (compile time polymorphism),method overriding(run time polymorphism)

**What is method overriding?**

Specific implementation of a method for child class.

**What is method overloading?**

If a class have multiple methods by same name but different parameters, it is known as Method Overloading.

**Difference between method overloading and overriding?**

|  |  |
| --- | --- |
| **Method overloading** | **Method overriding** |
| In case of method overloading, signature of method changes | While in case of method overriding it remain same. |
| Can overload method in one class | Can only be done on subclass. |
| Can overload static, final or private method in Java | Can not override static, final and private method in Java |
| Overloaded method in Java is bonded by static binding | Overridden methods are subject to dynamic binding. |

**What is run time polymorphism and compile time polymorphism?**

**Compile time polymorphism:**

it is nothing but the method overloading in java. In simple terms we can say that a class can have more than one methods with same name but with different number of arguments or different types of arguments or both.

**Runtime polymorphism:**

Runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time.

In this process, an overridden method is called through the reference variable of a super class. The determination of the method to be called is based on the object being referred to by the reference variable.

**Can we achieve polymorphism through data member?**

No, Polymorphism is always achieved via behavior of an object only properties of an object do not play any role in case of polymorphism.

**can we overload main() method?**

Yes, we can have many main() methods in a class by overloading the main method.

**package** FPPackage;

**public** **class** MainOverloading {

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

System.***out***.println("Normal main method");

*main*(3);

*main*("Subbu","Selenium");

}

**public** **static** **void** main(**int** args) {

System.***out***.println("Normal main method with one argument");

}

**public** **static** **void** main(String args1, String args2) {

System.***out***.println("Normal main method with two arguments");

}

}

When you run the above program it always prints the first main method regardless how many arguments you pass as this is the entry point for a class.

So, you have to call the other main methods from the program.

# **17. Java Super Keyword**

The super keyword refers to the objects of immediate parent class.

Super keyword is used for:

1) To access the data members of parent class when both parent and child class have member with same name  
2) To explicitly call the no-arg and parameterized constructor of parent class  
3) To access the method of parent class when child class has overridden that method.

Super Keyword to Access the Parent Class Variables:

# **17A. Java Basics - Accessing Parent Class Variables with Super Keyword**

**Accessing Parent class variables using Super keyword:**

ParentClass.java:

**package** FPPackage;

**public** **class** ParentClass {

**int** age = 10;

String name = "Subbu";

}

ChildClass.java

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

**int** age = 15;

String name = "Venkat";

**public** **void** displayParentDetails() {

System.***out***.println(**super**.age);

System.***out***.println(**super**.name);

}

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

ChildClass cl = **new** ChildClass();

cl.displayParentDetails();

System.***out***.println(cl.age);

System.***out***.println(cl.name);

}

}

# **17B. Java Basics - Accessing Parent Class No-Arg Constructor**

**Accessing the Parent Class Argument Constructor:**

Parent Class:

**package** FPPackage;

**public** **class** ParentClass {

ParentClass(){

System.***out***.println("Parent class no-arg constructor");

}

ParentClass(**int** value){

System.***out***.println("Parent class constructor with argument "+value);

}

}

Child Class:

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

ChildClass(){

**super**(100);

System.***out***.println("I am child class constructor");

}

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

ChildClass cl = **new** ChildClass();

}

}

**Super Keyword When Method Overriding:**

Example:

ParentClass.java:

**package** FPPackage;

**public** **class** ParentClass {

**public** **void** methodoverriding() {

System.***out***.println("This method will be overriden");

}

}

ChildClass.java:

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

**public** **void** methodoverriding() {

System.***out***.println("This method is overridden");

}

**public** **void** parentmethod() {

**super**.methodoverriding();

}

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

ChildClass cl = **new** ChildClass();

cl.methodoverriding();

cl.parentmethod();

}

}

# **17C. Accessing Parent Class Argument Constructor**

**Accessing the Parent Class Argument Constructor:**

Parent Class:

**package** FPPackage;

**public** **class** ParentClass {

ParentClass(){

System.***out***.println("Parent class no-arg constructor");

}

ParentClass(**int** value){

System.***out***.println("Parent class constructor with argument "+value);

}

}

Child Class:

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

ChildClass(){

**super**(100);

System.***out***.println("I am child class constructor");

}

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

ChildClass cl = **new** ChildClass();

}

}

# **17D. Super Keyword When Method Overriding**

**Super Keyword When Method Overriding:**

Example:

ParentClass.java:

**package** FPPackage;

**public** **class** ParentClass {

**public** **void** methodoverriding() {

System.***out***.println("This method will be overriden");

}

}

ChildClass.java:

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

**public** **void** methodoverriding() {

System.***out***.println("This method is overridden");

}

**public** **void** parentmethod() {

**super**.methodoverriding();

}

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

ChildClass cl = **new** ChildClass();

cl.methodoverriding();

cl.parentmethod();

}

}

# **17E. Accessing a Parent Method That is not Overridden**

**How to access the parent method that is not overridden?**

When a parent method is not overridden you can just access it with the child class object.

Example:

ParentClass.java:

**package** FPPackage;

**public** **class** ParentClass {

**public** **void** methodoverriding() {

System.***out***.println("This method will be overriden");

}

**public** **void** methodnotoverriding() {

System.***out***.println("I am not overridden");

}

}

ChildClass.java:

**package** FPPackage;

**public** **class** ChildClass **extends** ParentClass{

**public** **void** methodoverriding() {

System.***out***.println("This method is overridden");

}

**public** **void** parentmethod() {

**super**.methodoverriding();

}

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

ChildClass cl = **new** ChildClass();

cl.methodoverriding();

cl.parentmethod();

cl.methodnotoverriding();

}

}

# **17F. Java Super Keyword - Interview Questions**

**What is the super keyword?**

Super keyword is used to invoke the overridden method which overrides one of its superclass methods. This keyword allows to access overridden methods and also to access hidden members of the superclass.

It also forwards a call from a constructor to a constructor in the superclass.

**Why can’t we use super or this keyword in main method?**

Static keyword is mainly used in java for efficient memory management. Static keyword can be applied to variables, methods, blocks and nested classes.

Static keyword belongs to the class rather than the instance of the class. Whenever a method is declared as static, it can only be accessed by its Class name. Static method can only have access to its static data members and methods. It cannot access non - static members i.e. instance variables.

Super keyword:  It has the following uses,

* It’s a keyword which is a reference variable used to refer to the parent class object.
* It is used to invoke parent class methods.
* super() is used to invoke parent class constructor.
* super keyword is used to access the parent class’s instance variables.

For the above reasons, super keyword cannot be used under static method, it can only be used in non-static context.

This keyword: It has the following uses,

* It can be used to refer to the current class instance variable.
* It is used to refer to the current class object, (the object whose constructor or method is getting called). You can refer to any member of the current object from within an instance method or constructor using this.

Now since, static methods are related to the class and not to the objects, this cannot be used inside static methods.

**Write some points about super keyword in java?**

These are some uses of super keyword in java.

Super keyword can be used to refer parent class instance variables.

Super keyword can be used to invoke parent class method.

Super keyword can be used to invoke parent class constructor.

**Can we access parent class variables in child class by using super keyword?**

Yes, we can access parent class variable in child class by using java super keyword.

**Can we call parent class method in sub class by using super keyword?**

Yes, we can access parent class method in child class by using super keyword.

**Can we call parent class constructor in sub class constructor by using super keyword?**

Yes, we can call parent class constructor in sub class constructor by using super keyword but super keyword must be the first statement in sub class constructor.

**Can we use both "this" and "super" in constructor?**

No, It is not possible in java we cannot use both this and super keyword together in java constructor because this and super should be the first statement in any java constructor.

**What is difference between this() and super()?**

There are some difference between java this() constructor and super() constructor.

Java this()

It is used to access current class constructor.

It can be used inside another constructor of same class.

It can be used to remove ambiguity error when we have data members and local are same name.

For example: Remove ambiguity error

Java super()

It is used to access parent class constructor from sub class.

It can be used to only inside child class constructor.

It doesn't remove any ambiguity error from programs.

**Can we use super keyword in static method of a sub class for calling parent class method?**

No, we cannot use super keyword in static methods because it belongs to the immediate parent object and static belongs to the class level.

# **18. This Keyword - Accessing Class Variables**

This is a keyword in java. The keyword “This” can be used with

1. Can be used to refer any member of the current object from within an instance method or a constructor
2. **“this”** keyword can also be used inside Methods to call another Method from same Class.
3. **“this”** keyword can be used inside the constructor to call another overloaded constructor in the same Class. It is called the Explicit Constructor Invocation.

**Accessing the class variables:**

Example:

**package** FPPackage;

**public** **class** ThisKeyword {

**static** **int** *a*;

**static** **int** *b*;

**public** **void** assign(**int** c, **int** d) {

*a* = c;

*b* = d;

}

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

ThisKeyword tk = **new** ThisKeyword();

tk.assign(5,6);

System.***out***.println("A value is "+*a*);

System.***out***.println("B value is "+*b*);

}

}

Result:

A value is 5

B value is 6

In the above example we have created a method “assign(int c, int d)” to assign values to the instance variables a and b.

If we change the method to assign(int a, int b) then the program confuses as to whether it has to assign the values to the local variables or instance variables.

Example:

**package** FPPackage;

**public** **class** ThisKeyword {

**static** **int** *a*;

**static** **int** *b*;

**public** **void** assign(**int** a, **int** b) {

a = a;

b = b;

}

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

ThisKeyword tk = **new** ThisKeyword();

tk.assign(5,6);

System.***out***.println("A value is "+*a*);

System.***out***.println("B value is "+*b*);

}

}

Result:

A value is 0

B value is 0

So, we use this keyword to assign the values to the instance variables.

Example:

**package** FPPackage;

**public** **class** ThisKeyword {

**static** **int** *a*;

**static** **int** *b*;

**public** **void** assign(**int** a, **int** b) {

**this**.*a* = a;

**this**.*b* = b;

}

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

ThisKeyword tk = **new** ThisKeyword();

tk.assign(5,6);

System.***out***.println("A value is "+*a*);

System.***out***.println("B value is "+*b*);

}

}

Result:

A value is 5

B value is 6

# **18A. This Keyword - Accessing Instance Variables**

**Accessing the Instance Variables:**

**package** FPPackage;

**public** **class** ThisExample {

**int** a = 10;

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

ThisExample te = **new** ThisExample();

te.method1();

}

**public** **void** method1() {

**int** a = 30;

System.***out***.println("a value is "+**this**.a);

}

}

# **18B. This Keyword - Accessing a Method from Another Method in the same Class**

**Accessing a method from another method in the same class:**

**package** FPPackage;

**public** **class** ThisExample {

**int** a = 10;

**public** **void** method1() {

System.***out***.println("I am in method 1");

**this**.method2();

}

**public** **void** method2() {

System.***out***.println("I am method 2");

}

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

ThisExample te = **new** ThisExample();

te.method1();

}

}

# **18C. This Keyword - Accessing a Constructor from another Constructor in the same Class**

**Accessing a constructor from another constructor in the same class:**

**package** FPPackage;

**public** **class** ThisExample {

**int** a = 10;

ThisExample(){

**this**("Subbus Selenium Tutorials");

System.***out***.println("Normal Constructor");

}

ThisExample(String str){

System.***out***.println("Parameterized constructor and paramenter is "+str);

}

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

ThisExample te = **new** ThisExample();

}

}

# **18D. Java This Keyword - Interview Questions**

**How are this() and super() used with constructors?**

this() is used to invoke a constructor of the same class. super() is used to invoke a superclass constructor.

**What are the differences between this() and super()?**

this() is used to access one constructor from another with in the same class while super() is used to access superclass constructor. Either this() or super() exists it must be the first statement in the constructor.

**Can we use This keyword in static method?**

No. We can’t use This keyword in static method because static method belongs to the class.

**Can we use This keyword to access static method?**

Yes. We can use the keyword if the method we are calling from is not static.

**Can we use This method in main() method?**

No. We can’t use This keyword in main() method as main() method is static.

**Can we call method on this keyword from constructor?**

Yes. We can call non-static method using this keyword from constructor.

**Is it possible to assign a reference to “this” in java?**

No. We can’t assign a value to “this” because its always points to current object and it’s always final reference in java.

If we try to change or assign a reference to “this”, it will throw a compile time error.

**What are the uses of This keyword?**

this must be used to access instance variable if both instance and local variable names are same.

We can use this keyword in constructor overloading, to call one constructor from another we need this(); and this(); call should be first statement of the constructor.

This can be used to call other non static methods from with in methods.

**Can we pass this as parameter of method?**

Yes, we can pass this as parameter in a method

**Can we use this to refer static members?**

Yes, it’s possible to access static variable of a class using this but its discouraged and as per best practices this should be used on non static reference.

**Is it possible to use this in static blocks?**

No its not possible to use this keyword in static block.

**Can we return “this” from a method?**

Yes. We can return this as current class object.

**package** FPPackage;

**public** **class** ThisExample {

**int** a;

**public** **void** method1(**int** a) {

**this**.a = a;

}

**public** **int** getA() {

**return** a;

}

**public** ThisExample show() {

**return** **this**;

}

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

ThisExample te = **new** ThisExample();

te.method1(10);

ThisExample te1 = **new** ThisExample();

te1 = te.show();

System.***out***.println(te1.getA());

}

}

# **19. Java If Else Conditions - Part 1**

**Java If – Else:**

a) if statement  
b) nested if statement  
c) if-else statement  
d) if-else-if statement

**if Statement:**

Structure:

If(condition){

Statement;

}

Example:

**package** FPPackage;

**public** **class** ifelse {

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

**int** num1 = 5;

**int** num2 = 3;

**if**(num1 > num2) {

System.***out***.println("True");

}

}

}

**Nested If Statement:**

Structure:

If(condition){

If(condition){

Statement;

}

}

Example:

**package** FPPackage;

**public** **class** ifelse {

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

**int** num1 = 5;

**int** num2 = 3;

**int** num3 = 2;

**if**(num1 > num2) {

**if**(num2 > num3) {

System.***out***.println("True");

}

}

}

}

**if-else statement:**

if(condition) {

Statement;

}

else {

statement;

}

Example:

**package** FPPackage;

**public** **class** ifelse {

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

**int** num1 = 5;

**int** num2 = 6;

**if**(num1 > num2) {

System.***out***.println("True");

}

**else** {

System.***out***.println("False");

}

}

}

**if-else-if statement:**

if(condition) {

Statement;

}

else if {

statement;

}

Example:

**package** FPPackage;

**public** **class** ifelse {

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

**int** num1 = 5;

**int** num2 = 6;

**if**(num1 == num2) {

System.***out***.println("num1 is equal to num2");

}

**else** **if**(num1 > num2){

System.***out***.println("num1 is greater than num2");

}

**else** {

System.***out***.println("num1 is less than num2");

}

}

}

# **19A. Java If Else - Using Different Comparison Operators**

**Using Different Comparison Operators:**

The different comparison operators are

==, !=, >, <, >=, <=

We have already seen == operator above.

**Example with != :**

**public** **class** JavaExamples {

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

**int** a = 5;

**int** b = 8;

**if**(a != b) {

System.***out***.println("a and b are not equal");

}

**else** {

System.***out***.println("both the values are equal");

}

}

}

Result: a and b are not equal

**Example with > :**

**public** **class** JavaExamples {

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

**int** a = 5;

**int** b = 8;

**if**(a > b) {

System.***out***.println("a is greater than b");

}

**else** {

System.***out***.println("a is less than b");

}

}

}

Result: a is less than b

**Example with <:**

**public** **class** JavaExamples {

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

**int** a = 5;

**int** b = 8;

**if**(a < b) {

System.***out***.println("a is less than b");

}

**else** {

System.***out***.println("a is greater than b");

}

}

}

Result: a is less than b

**Example with >= :**

**public** **class** JavaExamples {

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

**int** a = 9;

**int** b = 8;

**if**(a >= b) {

System.***out***.println("a is greater than or equal to b");

}

**else** {

System.***out***.println("a is less than b");

}

}

}

Result: a is greater than or equal to b

**Example with <= :**

**public** **class** JavaExamples {

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

**int** a = 5;

**int** b = 8;

**if**(a <= b) {

System.***out***.println("a is less than or equal b");

}

**else** {

System.***out***.println("a is greater than b");

}

}

}

Result: a is less than or equal to b

# **19B. Java If Else - Using Different Logical Operators**

**Using Different Logical Operators:**

Different logical operators that we have are:

&& (Logical AND), || (Logical OR) and ! (Negation)

**Using && (Logical AND):**

&& (Logical AND) is used if we have to verify two or more conditions and if we want all the conditions to be true. If all the conditions are true then only the block of code under the condition will execute.

Example 1 with Two Conditions:

**public** **class** JavaExamples {

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

**int** a = 5;

**int** b = 8;

**int** c = 10;

**if**(a < b && a < c) {

System.***out***.println("a is less than b and c");

}

**else** {

System.***out***.println("a may be greater than b or c");

}

}

}

**Example 2 with three conditions:**

**public** **class** JavaExamples {

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

**int** a = 5;

**int** b = 8;

**int** c = 10;

**int** d = 20;

**if**(a < b && a < c && a < d) {

System.***out***.println("a is less than b and c and d");

}

**else** {

System.***out***.println("a may be greater than b or c or d");

}

}

}

**Using || (Logical OR):**

|| (Logical OR) is used if we have to verify two or more conditions and if we want at least one of the conditions to be true. If at least one of the conditions is true then only the block of code under the condition will execute.

Example 1 with two conditions:

**public** **class** JavaExamples {

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

**int** a = 5;

**int** b = 8;

**int** c = 10;

**if**(a < b || a < c) {

System.***out***.println("a is smaller than one of b or c");

}

**else** {

System.***out***.println("a is greater than b or c");

}

}

}

Example 2 with three conditions:

**public** **class** JavaExamples {

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

**int** a = 21;

**int** b = 8;

**int** c = 10;

**int** d = 20;

**if**(a < b || a < c || a < d) {

System.***out***.println("a is less than b and c and d");

}

**else** {

System.***out***.println("a is greater than b and c and d");

}

}

}

Using ! (Negation):

**public** **class** JavaExamples {

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

**boolean** test = **true**;

**if**(!test) {

System.***out***.println("test is false");

}

**else** {

System.***out***.println("test is true");

}

}

}

# **19C. Java Switch Case Statements**

**Switch – Case Statement:**

Switch case is used when we have a number of options and we have to do a separate operation for each one.

Structure:

Switch(expression)

{

Case constant:

Statements;

break;

Case constant:

Statements;

break;

Default:

Statements;

break;

}

Example:

**package** FPPackage;

**public** **class** javaexamples {

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

**int** num1 = 2;

**switch** (num1) {

**case** 1:

System.***out***.println("I am 1");

**break**;

**case** 2:

System.***out***.println("I am 2");

**break**;

**case** 3:

System.***out***.println("I am 3");

**break**;

**default**:

System.***out***.println("I am not 1 or 2 or 3");

}

}

}

Result: I am 2

Earlier java used to support only integers in switch case statements. But now strings are also supported.

**public** **class** JavaExamples {

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

String a = "two";

**switch** (a) {

**case** "one":

System.***out***.println("I am 1");

**break**;

**case** "two":

System.***out***.println("I am 2");

**break**;

**case** "three":

System.***out***.println("I am 3");

**break**;

**case** "four":

System.***out***.println("I am 4");

**break**;

**case** "five":

System.***out***.println("I am 5");

**break**;

**case** "six":

System.***out***.println("I am 6");

**break**;

**default**:

System.***out***.println("I am not 1 or 2 or 3 or 4 or 5 or 6");

}

}

}

Result: I am 3

# **19D. Java Switch Case Statements - Several Values Executing Same Code**

**Switch – Case Statement:**

Switch Case can also be used when several values execute the same code.

**public** **class** JavaExamples {

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

**int** month = 4;

**switch** (month) {

**case** 1:

**case** 3:

**case** 5:

**case** 7:

**case** 8:

**case** 10:

**case** 12:

System.***out***.println("No. of days are "+31);

**break**;

**case** 2:

System.***out***.println("No. of days are "+28);

**break**;

**default**:

System.***out***.println("No. of days are "+30);

}

}

}

# **19E. Java Switch Case Statements - Without Break Statement**

**Switch – Case Statement:**

**Break statement in Switch – Case:**

You have to provide break statement after each case statement otherwise all the further case statements will also be executed.

For example, in the following program I have taken out all the break statements.

**public** **class** JavaExamples {

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

String a = "three";

**switch** (a) {

**case** "one":

System.***out***.println("I am 1");

**case** "two":

System.***out***.println("I am 2");

**case** "three":

System.***out***.println("I am 3");

**case** "four":

System.***out***.println("I am 4");

**case** "five":

System.***out***.println("I am 5");

**case** "six":

System.***out***.println("I am 6");

**default**:

System.***out***.println("I am not 1 or 2 or 3 or 4 or 5 or 6");

}

}

}

Once the condition in case “three” is met, it will execute not only case “three” but everything else after that.

Result:

I am 3

I am 4

I am 5

I am 6

I am not 1 or 2 or 3 or 4 or 5 or 6

# **19F. Java If Else and Switch Case - Interview Questions**

**What is the difference between an if statement and a switch statement?**

An *if-then-else* statement is preferable when we need to check ranges of values or multiple conditions.

A *switch* statement is better suited when testing a single variable against many single values. If you have to use relational or logical operators then you can’t use switch case.

**Describe the *if-then* and *if-then-else* statements. What types of expressions can be used as conditions?**

Both statements tell our program to execute the code inside of them only if a particular condition evaluates to *true*. However, the *if-then-else* statement provides a secondary path of execution in case the if clause evaluates to *false*.

**Describe the *switch* statement. What object types can be used in the *switch* clause?**

Switch allows the selection of several execution paths based on a variables’ value.

Each path is labelled with *case* or *default*, the *switch* statement evaluates each *case* expression for a match and executes all statements that follow the matching label until a *break* statement is found. If it can’t find a match, the *default* block will be executed instead

**What happens when we forget to put a *break* statement in a *case* clause of a *switch*?**

This means that it will continue the execution of all *case* labels until if finds a *break* statement, even though those labels don’t match the expression’s value.

**Can we use Switch statement with Strings?**

Prior to Java 7 we can use only int values and enum constants in Switch. Starting with Java 7 we can use strings in Switch statement. If we use strings in switch statement prior to Java 7 we will get compile time error “only int and enum constants are permitted”.

# **20. Selenium - Java For Loops and For-Each Loops**

**Java For Loops and For Each Loops:**

Looping in programming languages is a feature which facilitates the execution of a set of statements repeatedly.

Syntax:

for(initialization; condition; increment/decrement)

{

Statements;

}

Example:

**package** FPPackage;

**public** **class** ifelse {

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

**for**(**int** i=0; i<10; i++) {

System.***out***.println(i);

}

System.***out***.println("Completed");

}

}

**For Loops inside For Loops:**

We can also create for loops inside other for loops.

**package** package1;

**public** **class** JavaExamples {

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

**for** (**int** i=0; i<10; i++) {

System.***out***.println("I am outside loop");

**for**(**int** j=0; j<5; j++) {

System.***out***.println(" -- I am inside loop");

}

}

}

}

**For Each Loop:**

For each loops are mainly used for Arrays and Collections.

For example if we have an array, we can access the elements of array using the following way.

**package** package1;

**public** **class** JavaExamples {

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

String[] a = {"sunday","monday","tuesday", "wednesday", "thursday", "friday", "saturday"};

**for** (**int** i=0; i<a.length; i++) {

System.***out***.println(a[i]);

}

}

}

We can use a normal for loop with array length in it and loop through the array.

The same can be achieved using for each loop as well.

**package** package1;

**public** **class** JavaExamples {

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

String[] a = {"sunday","monday","tuesday", "wednesday", "thursday", "friday", "saturday"};

**for** (String item : a) {

System.***out***.println(item);

}

}

}

In the above program we used

for (String item: a)

Here each value of the array will be stored in item which will be looped through.

The syntax for for each loop is

For (datatype of array/collection variable : array/collection)

**Another program to find the sum of elements in array using for each loop:**

**package** package1;

**public** **class** JavaExamples {

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

**int**[] a = {1,2,3,4,5,6,7,8,9,10};

**int** sum = 0;

**for** (**int** item : a) {

sum = sum + item;

}

System.***out***.println("Sum of array values is "+sum);

}

}

# **20A. Java For Loops - Break and Labelled Break**

**Java For Loop – Break and Labelled Break Statements:**

**Break Statements:**

If you want to come out of the current loop, you can use break statement. Normally break statements are used with conditions.

Example:

**package** package1;

**public** **class** JavaExamples {

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

**for** (**int** i=0; i<10; i++) {

System.***out***.println("i value is "+i);

**for**(**int** j=50; j<53; j++) {

**if** (i == 5) {

**break**;

}

System.***out***.println(" -- j value is "+j);

}

}

}

}

**Labelled Break Statements:**

An unlabelled *break* statement terminates the innermost *switch*, *for*, *while* or *do-while* statement, whereas a labelled *break* ends the execution of an outer statement.

**package** package1;

**public** **class** JavaExamples {

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

outer: **for** (**int** i=0; i<10; i++) {

System.***out***.println("i value is "+i);

**for**(**int** j=50; j<53; j++) {

**if** (i == 5) {

**break** outer;

}

System.***out***.println(" -- j value is "+j);

}

}

}

}

# **20B. Java For Loops - Continue and Labelled Continue**

**Java For Loop - Continue and Labelled Continue Statements:**

**Continue Statements:**

A continue statement skips to the end of the current iteration in the innermost for, while and do-while loops.

**Ex:**

**package** package1;

**public** **class** JavaExamples {

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

**for** (**int** i=0; i<10; i++) {

System.***out***.println("i value is "+i);

**for**(**int** j=50; j<53; j++) {

**if** (i == 5) {

**continue**;

}

System.***out***.println(" -- j value is "+j);

}

}

}

}

**Labelled Continue Statements:**

An unlabelled *continue* statement skips to the end of the current iteration in the innermost *for*, *while*, or *do-while* loop, whereas a labelled *continue* skips to an outer loop marked with the given label.

**package** package1;

**public** **class** JavaExamples {

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

outer: **for** (**int** i=0; i<10; i++) {

System.***out***.println("i value is "+i);

**for**(**int** j=50; j<53; j++) {

**if** (j == 51) {

**continue** outer;

}

System.***out***.println(" -- j value is "+j);

}

}

}

}

# **20C. Java While Loops and Inner While Loops**

**While Loops and Inner While Loops:**

Structure:

while(condition){

Statements;

}

Example:

**package** FPPackage;

**public** **class** ifelse {

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

**int** i = 0;

**while**(i<10) {

System.***out***.println(i);

i++;

}

System.***out***.println("Completed");

}

}

**Inner While Loops:**

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

**while** (j<3) {

System.***out***.println(" j value is "+j);

**int** k = 0;

**while** (k<3) {

System.***out***.println(" -- k value is "+k);

k++;

}

j++;

}

}

}

# **20D. Java While Loops - Break and Labelled Break**

**While Loops – Break and Labelled Break Statements:**

**Break Statement:**

Break statements are used to break the current loop during execution.

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

**while** (j<3) {

System.***out***.println(" j value is "+j);

**int** k = 0;

**while** (k<3) {

System.***out***.println(" -- k value is "+k);

**if**(k==1)

**break**;

k++;

}

j++;

}

}

}

**Labelled Break Statements:**

Labelled break statements are used to break the outer loop.

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

outer: **while** (j<3) {

System.***out***.println(" j value is "+j);

**int** k = 0;

**while** (k<3) {

System.***out***.println(" -- k value is "+k);

**if**(k==1)

**break** outer;

k++;

}

j++;

}

}

}

# **20E. Java While Loops - Continue and Labelled Continue**

**While Loop – Continue and Labelled Continue Statements:**

**Continue Statement:**

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

**while** (j<3) {

System.***out***.println(" j value is "+j);

**int** k = 0;

**while** (k<3) {

**if**(k>=1) {

k++;

**continue**;

}

System.***out***.println(" -- k value is "+k);

k++;

}

j++;

}

}

}

**Labelled Continue Statement:**

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

outer: **while** (j<3) {

System.***out***.println(" j value is "+j);

**int** k = 0;

**while** (k<3) {

System.***out***.println(" -- k value is "+k);

**if**(k>=0) {

j++;

**continue** outer;

}

//

//k++;

}

//j++;

}

}

}

# **20F. Java Do While Loops and Inner Do While Loops**

**Do – While Loops and Inner Do - While Loops:**

Structure:

do {

Statements;

} while(Condition)

Example:

**package** FPPackage;

**public** **class** ifelse {

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

**int** i = 12;

**do** {

System.***out***.println(i);

i++;

} **while**(i <10);

System.***out***.println("Completed");

}

}

**Inner do – while loops:**

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

**do** {

System.***out***.println(" j value is "+j);

**int** k = 0;

**do** {

System.***out***.println(" -- k value is "+k);

k++;

}**while** (k<3);

j++;

}**while** (j<3);

}

}

# **20G. Java Do While Loops - Break and Labelled Break**

**Do – While Loops – Break and Labelled Break Statements:**

**Break Statement:**

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

**do** {

System.***out***.println(" j value is "+j);

**int** k = 0;

**do** {

System.***out***.println(" -- k value is "+k);

k++;

**if** (k==1)

**break**;

}**while** (k<3);

j++;

}**while** (j<3);

}

}

**Labelled Break Statement:**

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

outer: **do** {

System.***out***.println(" j value is "+j);

**int** k = 0;

**do** {

System.***out***.println(" -- k value is "+k);

k++;

**if** (k==1)

**break** outer;

}**while** (k<3);

j++;

}**while** (j<3);

}

}

# **20H. Java Do While Loops - Continue and Labelled Continue**

**Do – While Loops – Continue and Labelled Continue Statements:**

**Continue Statement:**

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

**do** {

System.***out***.println(" j value is "+j);

**int** k = 0;

**do** {

**if** (k>=1) {

k++;

**continue**;

}

System.***out***.println(" -- k value is "+k);

k++;

}**while** (k<3);

j++;

}**while** (j<3);

}

}

**Labelled Continue Statement:**

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 0;

outer: **do** {

System.***out***.println(" j value is "+j);

**int** k = 0;

**do** {

**if** (k>=1) {

k++;

j++;

**continue** outer;

}

System.***out***.println(" -- k value is "+k);

k++;

}**while** (k<3);

j++;

}**while** (j<3);

}

}

# **20I. Java Loops - Difference between While Loop and Do While Loop**

**What is the difference between while loop and do-while loop?**

While loop doesn’t execute the code inside the loop if the condition is not met. But do-while loop executes it at least once.

Example:

while loop:

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 5;

**while**(j<3) {

System.***out***.println("Value of j is "+j);

}

}

}

In the above while loop value of j is greater than 3. So it won’t print anything.

do-while loop:

**package** package1;

**public** **class** JavaExamples {

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

**int** j = 5;

**do** {

System.***out***.println("Value of j is "+j);

}**while** (j<3);

}

}

In the above do-while loop, even though the value of j is greater than 3, it still executes once as the condition is mentioned at the bottom.

# **20J. Java Loops - Interview Questions**

**What types of loops are supported by java?**

Java offers three different types of loops: for, while, and do-while.

A for loop provides a way to iterate over a range of values. It’s most useful when we know in advance how many times a task is going to be repeated.

A *while* loop can execute a block of statements while a particular condition is *true*.

A *do-while* is a variation of a *while* statement in which the evaluation of the *Boolean* expression is at the bottom of the loop. This guarantees that the code will execute at least once.

**What is an enhanced loop?**

Enhanced loop is another syntax of the *for* statement designed to iterate through all the elements of a collection, array, enum or any object implementing the *Iterable* interface.

Syntax:

for(datatype variable : array/collection){

//code here

}

**What is the difference between a break statement and a continue statement?**

A break statement results in the termination of the statement to which it applies (switch, for, do, or while). A continue statement is used to end the current loop iteration and return control to the loop statement.

**How can you exit anticipatedly from a loop?**

Using break statement.

**What is the difference between a labelled and an unlabelled break statement?**

An unlabelled *break* statement terminates the innermost *switch*, *for*, *while* or *do-while* statement, whereas a labelled *break* ends the execution of an outer statement.

**What is the difference between an unlabelled and a labelled continue statement?**

An unlabelled *continue* statement skips to the end of the current iteration in the innermost *for*, *while*, or *do-while* loop, whereas a labelled *continue* skips to an outer loop marked with the given label.

**Can a for statement loop indefinitely?**

Yes, a for statement can loop indefinitely. For example, consider the following: for(;;);

**package** package1;

**public** **class** JavaExamples {

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

**for**(;;)

{

System.***out***.println("hello");

System.***out***.println("how are you?");

}

}

}

The above will continuously iterate and prints the statements.

# **21. Java Strings - What are Strings and how do they work**

**Strings:**

A string is a series of characters (It’s an array of characters).

Strings can be mainly created in three ways.

1. Using Character Array

Ex:

char ch = {‘H’,’e’,’l’,’l’,’o’};

String str = new String(ch);

1. Using String Literal

String str1 = “Hello”;

1. Using new keyword

String str2 = new String(“Hello”);

Example:

**package** package1;

**public** **class** JavaExamples {

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

//First way

**char**[] ch = {'H','e','l','l','o','1'};

String str1 = **new** String(ch);

//Second way

String str2 = "Hello2";

//Third way

String str3 = **new** String("Hello3");

System.***out***.println(str1);

System.***out***.println(str2);

System.***out***.println(str3);

}

}

**What is the difference when we create a string with string literal and strings with new keyword?**

When we create a string with string literal, it first searches the memory if there is already an existing string with the same value. If there is an existing string, it just points to that memory.

When we create a string with new keyword, that means we are creating a string object. Even if the string values are same, it always creates a new memory space for the strings and store the string values there.

**Strings are immutable**

Strings are immutable that means they can’t be changed. But they can be assigned to new strings.

This means that when you create a string it will be there in the memory until the object is garbage collected. You can’t change the value of the string in the memory. But when you try to assign it to a new value, that new value will be created in a new memory space. But the old string remains there without any reference.

# **21A. Java Strings - String Methods**

**Different Methods of Strings:**

Length of a string – length()

Getting the index of character or a substring – indexOf(Character or substring)

Getting the character a particular position – charAt(index)

Checking if a string contains a substring – contains()

Checking if a string ends with something – endsWith()

Replacing substrings in a string – Replace, ReplaceAll, ReplaceFirst

Converting a string to lower case – toLowerCase()

Converting a string to Upper Case – toUpperCase()

**package** FPPackage;

**public** **class** StringsDemo {

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

String strtest1 = "hello hi";

//Length of string

System.***out***.println("Lenght of string is "+strtest1.length());

//Index of a character or a substring

System.***out***.println("Index of l is "+strtest1.indexOf("l"));

System.***out***.println("Index of hi is "+strtest1.indexOf("hi"));

//Character at a particular index

System.***out***.println("Character at 3rd index "+strtest1.charAt(3));

//Check if a string contains a substring

System.***out***.println("Does the string contain hi? "+strtest1.contains("hi"));

//Check if a string ends with something

System.***out***.println(("Does the string ends with i? " + strtest1.endsWith("i")));

System.***out***.println(("Does the string ends with hi? " + strtest1.endsWith("hi")));

//Replace, ReplaceAll, ReplaceFirst

String strtest2 = "hi hi hi hi";

System.***out***.println("Replacing hi with hello: "+strtest2.replace("hi", "hello"));

System.***out***.println("Replacing hi with hello: "+strtest2.replaceFirst("hi", "hello"));

System.***out***.println("Replacing hi with hello: "+strtest2.replaceAll("hi", "hello"));

//tolowercase and to uppercase

String strtest3 = "HelLo Hi";

System.***out***.println("All Upper Case Letters: "+strtest3.toUpperCase());

System.***out***.println("All Lower Case Letters: "+strtest3.toLowerCase());

}

}

**Converting a String to an Integer:**

To convert a string to an integer it should be a number string.

Integer.parseInt(string);

Example:

**package** FPPackage;

**public** **class** StringsDemo {

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

String strtest1 = "53";

//int k = strtest1 \* 100;

**int** i = Integer.*parseInt*(strtest1);

**int** j = i\*100;

System.***out***.println("j value is "+j);

}

}

**NumberFormatException:**

If you try to convert a non number string to an integer, you will get this exception.

**Substrings in Java:**

In java you can get the substring of a string using substring() method.

If you pass only starting index for this method, you get a substring starting from the index you provided till the end of the string.

If you pass the starting index and ending index for this method, you get a substring from the starting index to the ending index not including the ending index.

**public** **class** StringsEx {

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

String str = **new** String("Selenium Tutorials");

System.***out***.println(str.substring(9));

System.***out***.println(str.substring(9,14));

}

}

# **21B. Java Strings - Comparing and Concatenating Two Strings**

**Comparing two Strings:**

Strings are compared with equals method.

Str1.equals(Str2);

== is used to compare others.

For example, integer values are compared with == method.

If you use == to compare strings, it compares the reference points but not the actual values.

**package** FPPackage;

**public** **class** StringsDemo {

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

//Test 1

String strtest1 = "hi";

String strtest2 = "hi";

**if**(strtest1 == strtest2) {

System.***out***.println("Test1: Both are equal");

}

**else** {

System.***out***.println("Test1: Both are not equal");

}

//Test 2

String strtest3 = **new** String("hi");

String strtest4 = **new** String("hi");

**if**(strtest3==strtest4) {

System.***out***.println("Test2: Both are equal");

}

**else** {

System.***out***.println("Test2: Both are not equal");

}

//Test 3

String strtest5 = **new** String("hi");

String strtest6 = **new** String("hi");

**if**(strtest5.equals(strtest6)) {

System.***out***.println("Test3: Both are equal");

}

**else** {

System.***out***.println("Test3: Both are not equal");

}

}

}

**Concatenating two strings:**

For concatenating two strings we use + operator.

**package** FPPackage;

**public** **class** StringsDemo {

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

String strtest1 = "hi";

String strtest2 = "hello";

String strtest3 = strtest1 + strtest2;

System.***out***.println(strtest3);

String strtest4 = strtest1.concat(strtest2);

System.***out***.println(strtest4);

}

}

# **21C. Java Strings - Checking if a String is Substring of other String**

**Checking if a String is Substring of Other String:**

We can use the following two methods to check if a string is substring of other string.

1. Contains()
2. IndexOf()

Example:

**package** substring;

**public** **class** SubstringExample {

**static** String *str1* = "subbu";

**static** String *str2* = "subbu selenium tutorials";

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

**if**(*str2*.indexOf(*str1*) != -1) {

System.***out***.println("1st Method: str2 is substring of str1");

}

**else** {

System.***out***.println("1st Method: str2 is NOT substring of str1");

}

**if**(*str2*.contains(*str1*)) {

System.***out***.println("2nd Method: str2 is substring of str1");

}

**else** {

System.***out***.println("2nd Method: str2 is NOT substring of str1");

}

}

}

This will result in:

1st Method: str2 is substring of str1

2nd Method: str2 is substring of str1

# **21D. Java Strings - Interview Questions**

**To what value is a variable of the String type automatically initialized?**

The default value of a String type is null.

**Difference between Character Constant and String Constant in java?**

Character constant is enclosed in single quotes. String constants are enclosed in double quotes. Character constants are single digit or character. String Constants are collection of characters. Ex :’2’, ‘A’ Ex : “Hello World”

**What are the different ways to create Strings in java?**

Strings can be mainly created in three ways.

1. Using Character Array

Ex:

char[] ch = {‘H’,’e’,’l’,’l’,’o’};

String str = new String(ch);

1. Using String Literal

String str1 = “Hello”;

1. Using new keyword

String str2 = new String(“Hello”);

**Write a method that will remove given character from the String?**

We can use replaceAll method to replace all the occurrences of a String with another String. The important point to note is that it accepts String as argument, so we will use Character class to create String and use it to replace all the characters with empty String.

str.replaceAll(Character.toString(c), “”);

**public** **class** StringsEx {

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

String str = **new** String("Selenium Tutorials");

//removing e from the above

**char** ch = 'e';

String str1 = str.replaceAll(Character.*toString*(ch), "");

System.***out***.println(str1);

}

}

**How can we make a string to upper case or lower case?**

Using toLowerCase() and toUpperCase() methods.

**How can we compare strings ignoring case?**

Using equalsIgnoreCase(String str) method.

Example:

**public** **class** StringsEx {

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

String str1 = **new** String("SELENIUM TUTORIALS");

String str2 = **new** String("Selenium Tutorials");

**if**(str1.equalsIgnoreCase(str2)) {

System.***out***.println("Both strings are equal");

}

**else** {

System.***out***.println("The strings are not equal");

}

}

}

**How to split a string in java?**

Using split(string regex) method

Example:

**public** **class** StringsEx {

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

String str1 = **new** String("Selenium Tutorials by subbu");

String[] arr1 = str1.split(" ");

System.***out***.println("length of array is "+arr1.length);

**for**(**int** i=0; i<arr1.length; i++) {

System.***out***.println(arr1[i]);

}

}

}

Result:

length of array is 4

Selenium

Tutorials

by

subbu

**How many objects will be created by the following?**

**String str = “Selenium Tutorials”**

**String str1 = “Selenium Tutorials”**

Only one object.

**What is the difference between equals() method and == operator?**

The equals() method matches content of the strings whereas == operator matches object or reference of the strings.

**Is String class final?**

Yes.

**How to check if a string is empty?**

We can check the length of the string to find if a string is empty. We can also use isEmpty() method to check the length of string.

**public** **class** StringsEx {

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

String str1 = **new** String("");

**if**(str1.isEmpty()) {

System.***out***.println("usign isEmpty() - The string is empty");

}

**if**(str1.length() == 0) {

System.***out***.println("using length() - The string is empty");

}

}

}

**How to convert a string into int?**

We can convert a string into int only if the string is a numbered string. Otherwise it will throw an exception “NumberFormatException”

Example:

**public** **class** StringsEx {

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

String str1 = **new** String("67");

**int** no = Integer.*parseInt*(str1);

System.***out***.println(no);

}

}

**How to convert a string into double?**

We convert a string into double using Double.parseDouble() method.

**public** **class** StringsEx {

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

String str1 = **new** String("67.89");

**double** no = Double.*parseDouble*(str1);

System.***out***.println(no);

}

}

# **22. Java Arrays**

**Java Arrays:**

An array is a collection of similar type of elements.

Java array is an object which contains elements of a similar data type. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

**Advantages:**

Code Optimization

Random Access

**Disadvantages:**

Only fixed size of elements can be stored in an array.

**Types of Array:**

Single Dimensional

Multi-Dimensional

**Declaring Single Dimensional Array:**

datatype[] arr;

datatype []arr;

datatype arr[];

**How to instantiate an array:**

ArrayRefVar = new datatype[size];

Example:

**package** FPPackage;

**public** **class** ArraysDemo {

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

**int** testarray[] = **new** **int**[5];

testarray[0] = 1;

testarray[1] = 2;

testarray[2] = 3;

testarray[3] = 4;

testarray[4] = 5;

System.***out***.println(testarray[0]);

System.***out***.println(testarray[1]);

System.***out***.println(testarray[2]);

System.***out***.println(testarray[3]);

System.***out***.println(testarray[4]);

}

}

Array length can be found with .length

Ex: testarray.length

**package** FPPackage;

**public** **class** ArraysDemo {

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

**int** testarray[] = **new** **int**[5];

testarray[0] = 1;

testarray[1] = 2;

testarray[2] = 3;

testarray[3] = 4;

testarray[4] = 5;

System.***out***.println("Array length is "+testarray.length);

**for**(**int** i=0; i<testarray.length; i++) {

System.***out***.println(testarray[i]);

}

}

}

We can declare, instantiate and initialize an array as following.

int a[] = {1,2,3,4,5}

Example:

**package** FPPackage;

**public** **class** ArraysDemo {

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

**int** testarray[] = {1,2,3,4,5};

System.***out***.println("Array length is "+testarray.length);

**for**(**int** i=0; i<testarray.length; i++) {

System.***out***.println(testarray[i]);

}

}

}

# **22A. Java Arrays - Passing an Array to a Method**

**Java Arrays:**

**Passing an Array to a Method:**

**Finding the Max. Value in an Array:**

**package** FPPackage;

**public** **class** ArraysDemo {

**public** **static** **int** findMax(**int** arr[]) {

**int** max = arr[0];

**for**(**int** i = 0; i<arr.length; i++) {

**if**(max < arr[i]) {

max = arr[i];

}

}

**return** max;

}

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

**int** testarray[] = {5,33,65,43,67,34};

**int** maxvalue = *findMax*(testarray);

System.***out***.println("Max. Value is "+maxvalue);

}

}

# **22B. Java Arrays - Returning an Array from a Method**

**Java Arrays:**

**Returning an array from a method:**

**package** FPPackage;

**public** **class** ArraysDemo {

**public** **static** **int**[] returnArray() {

**int**[] arr = {1,2,3,4,5};

**return** arr;

}

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

**int** arr[] = *returnArray*();

**for**(**int** i=0; i<arr.length; i++) {

System.***out***.println(arr[i]);

}

}

}

# **22C. Java Arrays - ArrayIndexOutOfBounds Exception**

**Java Arrays:**

**ArrayIndexOutOfBoundsException:**

When you try to access an element, which is not in the index of the array then you an ArrayIndexOutOfBoundsException.

**package** FPPackage;

**public** **class** ArraysDemo {

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

**int**[] arr = {1,2,3,4,5};

**for**(**int** i=0; i<=arr.length; i++) {

System.***out***.println(arr[i]);

}

}

}

This gives “ArrayIndexOutOfBoundsException”.

# **22D. Java Arrays – ArrayStoreException**

**Java Arrays:**

**What is ArrayStoreException?**

ArrayStoreException comes when you have stored an element of type other than the type of array.

Example:

**package** package1;

**public** **class** ArraysExample {

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

Object[] arrobj = **new** String[3];

arrobj[0] = **new** Integer(3);

}

}

Result: Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer

at package1.ArraysExample.main(ArraysExample.java:9)

# **22E. Java Arrays - Iterating Over an Array**

**Java Arrays:**

**How can you iterate over an array in java?**

You can iterate over an array using a for loop or a for each loop.

Example: Using for loop

**package** package1;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "selenium", "tutorials"};

**for** (**int** i=0; i<arr1.length; i++) {

System.***out***.println(arr1[i]);

}

}

}

Example: Using for each loop

**package** package1;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "selenium", "tutorials"};

**for** (String item : arr1) {

System.***out***.println(item);

}

}

}

# **22F. Java Arrays - Sorting Elements in an Array**

**Java Arrays:**

**How to sort elements in an array?**

Array elements can be sorted using Arrays.sort() method.

**package** package1;

**import** java.util.Arrays;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "selenium", "java", "tutorials"};

Arrays.*sort*(arr1);

**for** (**int** i=0; i<arr1.length; i++) {

System.***out***.println(arr1[i]);

}

**int** arr2[] = {5,3,6,4,8,7,2,1,10,9};

Arrays.*sort*(arr2);

System.***out***.println(Arrays.*toString*(arr2));

}

}

Result:

[java, selenium, subbu, tutorials]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# **22G. Java Arrays - Searching an Array**

**Java Arrays:**

**How to search an array to check if an element exists there?**

You can search for an element using a for loop. You can also sort the array first and then use Arrays.binarySearch() method to check for the element.

**Using For loop:**

**package** package1;

**import** java.util.Arrays;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "java", "selenium", "tutorials"};

String str\_to\_check = "Subbu";

**boolean** check = **false**;

**for**(**int** i=0; i<arr1.length; i++) {

**if**(arr1[i].equalsIgnoreCase(str\_to\_check)) {

check = **true**;

**break**;

}

}

**if**(check) {

System.***out***.println(str\_to\_check+" is found in the array");

}

**else** {

System.***out***.println(str\_to\_check+" is not found in the array");

}

}

}

**Using Arrays.binarySearch() method:**

**package** package1;

**import** java.util.Arrays;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "java", "selenium", "tutorials"};

Arrays.*sort*(arr1);

System.***out***.println(Arrays.*binarySearch*(arr1,"subbu"));

}

}

You can also convert the arrays to List or Set and use contains() method to check for the element.

# **22H. Java Arrays - Copying an Array using Object.clone()**

**Java Arrays:**

**You can copy an array using Object.clone() method.**

**package** package1;

**import** java.util.Arrays;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "selenium", "java", "tutorials"};

String arr2[] = **new** String[5];

arr2 = arr1.clone();

arr2[2] = "hello";

System.***out***.println(Arrays.*toString*(arr1));

System.***out***.println(Arrays.*toString*(arr2));

}

}

# **22I. Java Arrays - Copying an Array using System.arraycopy()**

**Java Arrays:**

**You can copy an array using System.arraycopy() method.**

System.arraycopy() takes the following arguments.

First argument: Source array

Second argument: Index from where copying should start in the source array

Third argument: Destination array

Fourth argument: Index from where the elements should be copied to, in the destination array

Fifth argument: No. of elements to be copied.

**package** package1;

**import** java.util.Arrays;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "selenium", "java", "tutorials"};

String arr2[] = **new** String[4];

System.*arraycopy*(arr1, 0, arr2, 0, 4);

System.***out***.println(Arrays.*toString*(arr1));

System.***out***.println(Arrays.*toString*(arr2));

}

}

# **22J. Java Arrays - Copying an Array using Arrays.copyOf()**

**Java Arrays:**

**You can copy an array using Arrays.copyOf() method.**

Arrays.copyOf() method takes two arguments.

First Argument: This is the source array

Second Argument: No. of elements that should be copied from source array starting from 0 index

**package** package1;

**import** java.util.Arrays;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "selenium", "java", "tutorials"};

String arr2[] = **new** String[4];

arr2 = Arrays.*copyOf*(arr1, 3);

System.***out***.println(Arrays.*toString*(arr1));

System.***out***.println(Arrays.*toString*(arr2));

}

}

# **22K. Java Arrays - Copying an Array using Arrays.copyOfRange()**

**Java Arrays:**

**You can copy an array using Arrays.copyOfRange() method.**

Arrays.copyOfRange() method takes three arguments.

First Argument: Source Array

Second Argument: The starting index from where elements should be copied

Third Argument: The ending index of the elements that should be copied. This is not included when the elements are copied.

**package** package1;

**import** java.util.Arrays;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "selenium", "java", "tutorials"};

String arr2[] = **new** String[4];

arr2 = Arrays.*copyOfRange*(arr1, 1, 3);

System.***out***.println(Arrays.*toString*(arr1));

System.***out***.println(Arrays.*toString*(arr2));

}

}

# **22L. Java Arrays - Multi Dimensional Arrays**

**Java Arrays:**

**Multidimensional Array:**

Syntax:

* dataType[][] arrayRefVar;
* dataType [][]arrayRefVar;
* dataType arrayRefVar[][];
* dataType []arrayRefVar[];

Instantiating a Multidimensional Array:

int[][] testarray = new int[2][2];

Initializing Multidimensional Array:

testarray[0][0] = 1;

testarray[0][1] = 2;

testarray[1][0] = 3;

testarray[1][1] = 4;

You can define, instantiate and initialize a multidimensional array the following way.

Int[][] testarray = {{1,2},{3,4}};

Example:

**package** FPPackage;

**public** **class** ArraysDemo {

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

**int**[][] arr = {{1,2,3},{4,5,6},{7,8,9}};

**for**(**int** i=0; i<3; i++) {

**for**(**int** j=0; j<3; j++) {

System.***out***.println(arr[i][j]);

}

}

}

}

# **22M. Java Arrays - Interview Questions**

**What is difference between length and length() method in java ?**

length() : In String class we have length() method which is used to return the number of characters in string. Ex : String str = “Hello World”; System.out.println(str.length()); Str.length() will return 11 characters including space.

length : we have length instance variable in arrays which will return the number of values or objects in array. For example : String days[]={” Sun”,”Mon”,”wed”,”thu”,”fri”,”sat”}; Will return 6 since the number of values in days array is 6.

**Can you change the size of an array once you created?**

No. You can’t change it.

**How to access array elements in java?**

Using index and index starts from 0.

**Where does array stored in memory?**

Array is created in heap space of JVM memory. Since array is object in Java, even if you create array locally inside a method or block, object is always allocated memory from heap.

**Can you store a string into an array of integer in java?**

No. You will get a compile time error.

**package** package1;

**public** **class** ArraysExample {

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

**int** arr1[] = **new** **int**[10];

arr1[0] = **new** Integer(9);

arr1[1] = "subbu"; //Error: Can not convert from string to int

}

}

**What is ArrayIndexOutOfBounds Exception?**

ArrayIndexOutOfBounds exception comes when your code tries to access an invalid index from a given array.

**What is a two dimensional array in java?**

It’s an array of array in java. You can declare a two dimensional array with three rows and three columns as

int[][] primes = new int[3][3]

**What is ArrayStoreException?**

ArrayStoreException comes when you have stored an element of type other than the type of array.

**How can you iterate over an array in java?**

You can iterate over an array using a for loop or a for each loop.

**How to sort elements in an array?**

Array elements can be sorted using Arrays.sort() method.

**How to search an array to check if an element exists there?**

You can search for an element using a for loop. You can also sort the array first and then use Arrays.binarySearch() method to check for the element.

**Can we use = to copy an array into another array?**

We can’t use = to copy an array to another array. Arrays are mutable and when we use = to copy an array, it won’t really copy the array but the second array points to the same memory area of the first array. Since arrays are mutable, if we change a value in an array it changes the value in second array also.

Example:

**package** package1;

**import** java.util.Arrays;

**public** **class** ArraysExample {

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

String arr1[] = {"subbu", "selenium", "java", "tutorials"};

String arr2[] = **new** String[5];

arr2 = arr1;

arr2[2] = "hello";

System.***out***.println(Arrays.*toString*(arr1));

System.***out***.println(Arrays.*toString*(arr2));

}

}

Result:

[subbu, selenium, hello, tutorials]

[subbu, selenium, hello, tutorials]

**How to copy an array to another array?**

1. Object.clone()
2. System.arraycopy()
3. Arrays.copyOf()
4. Arrays.copyOfRange()

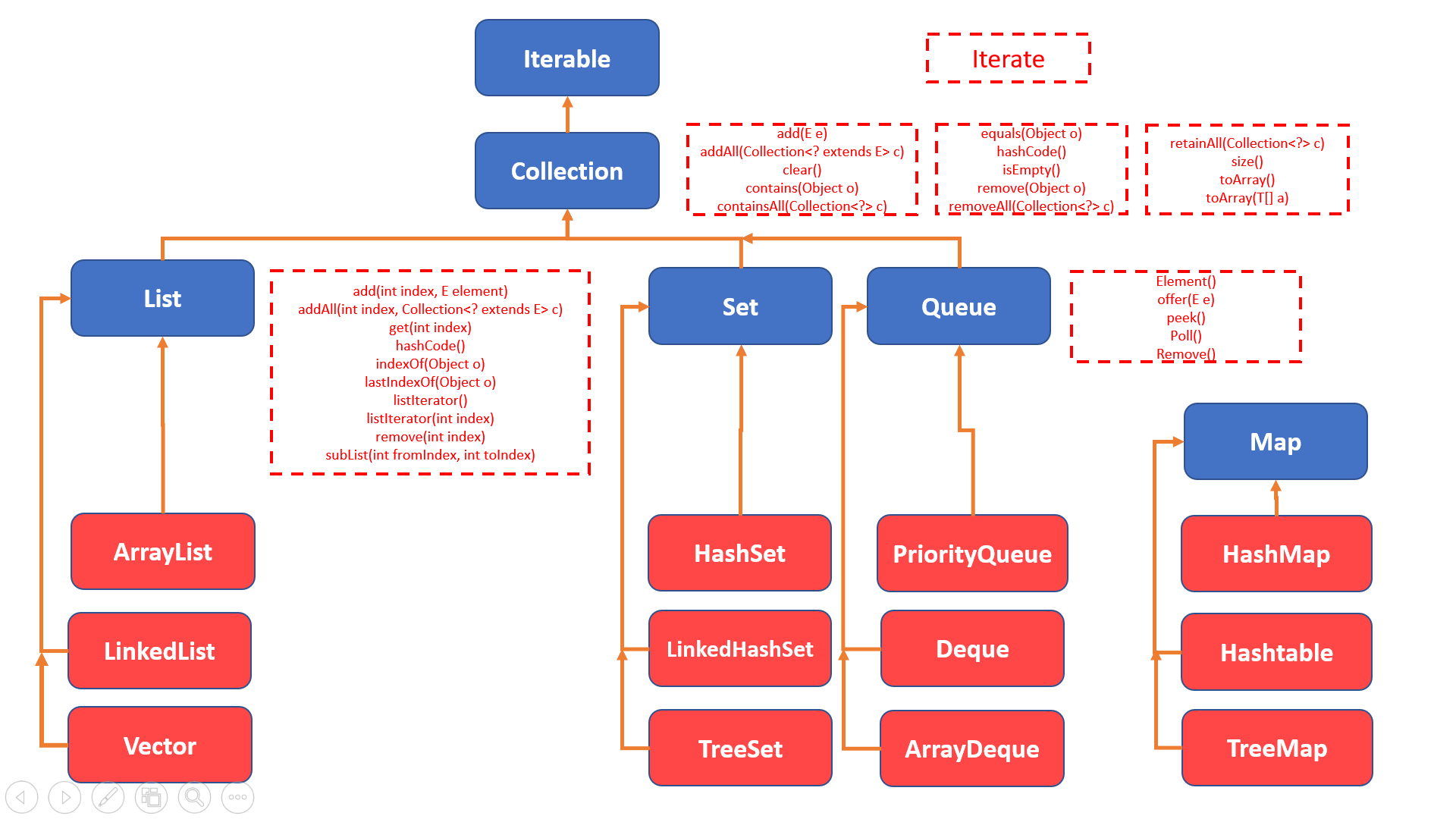
# **23. Collections Framework – Introduction**

**Java Collections Framework:**

The main disadvantage with Arrays is, they are of fixed size. If we declare an array of length 3, we can add only 3 elements into it. We can’t change the size of the array. We can’t increase it or decrease it.

If we want to have more or less elements into it, we need to have an extendable or shrinkable arrays and Collections framework is an answer for that.

List, Set, Queue and Map are four important interfaces of Collections Framework. Map doesn’t implement Collection interface but it is a part of Collections Framework.



The first interface in collections framework is “Iterable” interface and it defined a method “Iterate” which is to iterate over the collections.

The second one is “Collection” interface which has defined additional methods.

The next interfaces are List, Set and Queue and all of these are extending the “Collection” framework and defined some more additional methods.

Different classes like ArrayList, HashSet etc., were developed which implement these interfaces.

Map doesn’t extend Collection interface but it is still under collections framework.

You can have a look at the following link (oracle docs) for more information about the interfaces and classes developed.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html>

**Differences between List, Set, Queue and Map:**

List in java provides ordered and indexed collection which may contain duplicates.

The Set interface provides an unordered collection of unique objects ie Set doesn’t allow duplicates.

Queue works on FIFO (First In First Out) basis. In FIFO the first element is removed frist and the last element is last.

Map provides a data structure based on key value pair.

**List – Differences between ArrayList and LinkedList**

**ArrayList:**

1. ArrayList internally uses dynamic array to store the elements
2. Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.
3. ArrayList is better for storing and accessing data.

**LinkedList:**

1. LinkedList internally uses doubly linked list to store the elements.
2. Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3. LinkedList is better for manipulating data.

# **23A. Collections Framework – ArrayList**

**ArrayList:**

ArrayList is a data structure that can be stretched to accommodate additional elements within itself and shrink back to a smaller size when elements are removed. In other words, it’s a dynamic array.

Example: Methods add(), size() and remove()

**package** FPPackage;

**import** java.util.ArrayList;

**public** **class** ArrayListDemo {

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

//Declaring an arraylist

ArrayList a = **new** ArrayList();

//Adding elements to array list

a.add("Subbu");

a.add("Raj");

a.add("Krishna");

a.add("Vishnu");

a.add("Raghu");

//Getting elements from arraylist

System.***out***.println(a.get(0));

System.***out***.println(a.get(1));

System.***out***.println(a.get(2));

//Getting the number of elments in an arraylist

System.***out***.println(a.size());

//Printing an arraylist

System.***out***.println(a);

//Removing an elements from an array list

a.remove("Vishnu");

System.***out***.println(a.size());

System.***out***.println(a);

//Removing an elements by index

a.remove(2);

System.***out***.println(a.size());

System.***out***.println(a);

//Checking if an array list contains a value

System.***out***.println("Contains Vishnu "+a.contains("Vishnu"));

}

}

The above array list can accept anything and even integers as well.

Ex: ArrayList with different datatypes

**package** FPPackage;

**import** java.util.ArrayList;

**public** **class** ArrayListDemo {

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

//Declaring an arraylist

ArrayList a = **new** ArrayList();

//Adding elements to array list

a.add("Subbu");

a.add("Raghu");

a.add(1);

//Printing an arraylist

System.***out***.println(a);

}

}

Here we are adding an integer value of 1 with the previous strings values. This doesn’t cause any problem and it produces the result as

[Subbu, Raghu, 1]

But if we want to restrict the arraylist to accept only a particular datatype we can do it the following way.

ArrayList<String> a = new ArrayList<String>();

Now the above problem produces a compilation error as it won’t accept integers.

# **23B. Collections Framework - Iterating through an ArrayList**

**Iterating through an arraylist:**

**Two ways:**

1. **Using Iterator**

**package** FPPackage;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** ArrayListDemo {

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

//Declaring an arraylist

ArrayList<String> a = **new** ArrayList<String>();

//Adding elements to array list

a.add("Subbu");

a.add("Raghu");

a.add("Venu");

a.add("Rajesh");

//Iterating through the list

Iterator itr = a.iterator();

**while**(itr.hasNext()) {

System.***out***.println(itr.next());

}

}

}

1. **Using For Loop:**

**package** FPPackage;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** ArrayListDemo {

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

//Declaring an arraylist

ArrayList<String> a = **new** ArrayList<String>();

//Adding elements to array list

a.add("Subbu");

a.add("Raghu");

a.add("Venu");

a.add("Rajesh");

//Iterating through the list

**for**(String s:a) {

System.***out***.println(s);

}

}

}

# **23C. Collections Framework - Adding a List to another List**

**addAll method:**

**package** FPPackage;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** ArrayListDemo {

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

//Declaring an arraylist

ArrayList<String> a = **new** ArrayList<String>();

//Adding elements to array list

a.add("Subbu");

a.add("Raghu");

a.add("Venu");

a.add("Rajesh");

ArrayList<String> b = **new** ArrayList<String>();

b.add("Ganesh");

b.add("Vamshi");

//Adding b list to a

a.addAll(b);

//Iterating through the list

**for**(String s:a) {

System.***out***.println(s);

}

}

}

# **23D. Collections Framework - Removing a List from Another List**

**removeAll() Example:**

**package** FPPackage;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** ArrayListDemo {

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

//Declaring an arraylist

ArrayList<String> a = **new** ArrayList<String>();

//Adding elements to array list

a.add("Subbu");

a.add("Raghu");

a.add("Venu");

a.add("Rajesh");

ArrayList<String> b = **new** ArrayList<String>();

b.add("Raghu");

b.add("Rajesh");

b.add("Vamshi");

//Adding b list to a

a.removeAll(b);

//Iterating through the list

**for**(String s:a) {

System.***out***.println(s);

}

}

}

# **23E. Collections Framework - retainAll()**

**retainAll() Example:**

**package** FPPackage;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** ArrayListDemo {

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

//Declaring an arraylist

ArrayList<String> a = **new** ArrayList<String>();

//Adding elements to array list

a.add("Subbu");

a.add("Raghu");

a.add("Venu");

a.add("Rajesh");

ArrayList<String> b = **new** ArrayList<String>();

b.add("Raghu");

b.add("Rajesh");

b.add("Vamshi");

//Adding b list to a

a.retainAll(b);

//Iterating through the list

**for**(String s:a) {

System.***out***.println(s);

}

}

}

# **23F. Collections Framework - List of Lists**

**List of Lists:**

**package** Selenium;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ListofLists {

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

List<List<String>> lls = **new** ArrayList<List<String>>();

List<String> ls = **new** ArrayList<String>();

ls.add("subbu");

ls.add("selenium");

lls.add(ls);

List<String> ls1 = **new** ArrayList<String>();

ls1.add("java");

ls1.add("tutorials");

lls.add(ls1);

System.***out***.println(lls);

**for**(**int** i=0; i<lls.size(); i++) {

System.***out***.println(lls.get(i).get(0));

System.***out***.println(lls.get(i).get(1));

}

}

}

# **23G. Collections Framework - Sorting a list**

**Sorting a list in java:**

**package** Selenium;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** ListofLists {

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

List<String> ls = **new** ArrayList<String>();

ls.add("subbu");

ls.add("selenium");

ls.add("java");

ls.add("tutorials");

System.***out***.println(ls);

//In Ascending order

Collections.*sort*(ls);

//In descending order

//Collections.sort(ls, Collections.reverseOrder());

System.***out***.println(ls);

}

}

# **23H. Collections Framework – LinkedList**

**Linked List:**

Linked list is an unordered list.

LinkedList can be declared as follows.

1. LinkedList object = new LinkedList();

It accepts any data type

1. LinkedList<String> object = new LinkedList<String>();

It accepts only String type.

Example:

In the following example we will work with the following operations.

1. Adding elements to LinkedList
2. Getting size of LinedList
3. Getting the elements of the LinkedList by using for each loop
4. Getting the elements of the LinkedLIst by using Iterator
5. Removing an element by value
6. Removing an element by index
7. Removing first element of the list
8. Removing last element of the list
9. Checking if the Linked List contains a particular value
10. Getting a value using index
11. Setting a value using index

**package** package1;

**import** java.util.Iterator;

**import** java.util.LinkedList;

**public** **class** LinkedListExample {

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

//Creating an object of linked list

LinkedList<String> ll = **new** LinkedList<String>();

//Adding elements to the linked list

ll.add("Subbu");

ll.add("Selenium");

ll.add("Java");

ll.add("Tutorials");

ll.add("Subscribe");

ll.add("Youtube");

ll.add("Channel");

//Printing linked list

System.***out***.println(ll);

//Size of linked list

System.***out***.println("Size of linked list is "+ll.size());

//Printing each element using for each loop

System.***out***.println("Printing values using for each loop:");

**for**(String str : ll) {

System.***out***.println(str);

}

//Iterating through a list and printing values

System.***out***.println("Printing values by iterating through the list:");

Iterator itr = ll.iterator();

**while**(itr.hasNext()) {

System.***out***.println(itr.next());

}

//Removing an element using value

ll.remove("Channel");

System.***out***.println(ll);

//Removing an element using index

ll.remove(5);

System.***out***.println(ll);

//Removing first element

ll.removeFirst();

System.***out***.println(ll);

//Removing last element

ll.removeLast();

System.***out***.println(ll);

//checking if the list contains a particular element

**boolean** check = ll.contains("Java");

**if**(check) {

System.***out***.println("The list contains Java");

}

**else** {

System.***out***.println("It doesn't contain Java");

}

//Get an element from the list

String str = ll.get(1);

System.***out***.println("The string at 1st index is "+str);

//Setting an element in the list

ll.set(2, "Practice");

System.***out***.println(ll);

}

}

# **23I. Collections Framework – HashSet**

**Set – HashSet:**

HashSet is an unordered collection. It doesn’t maintain the order in which elements are inserted.

It doesn’t allow duplicate values. It allows null values.

Example:

The following example shows

1. How to declare HashSet
2. Check if the HashSet is empty
3. Adding elements to the hashset
4. Adding duplicate elements to the set and check if it allows duplicate elements
5. Check if the set allows null value
6. Check the size of the HashSet
7. Print the values using for each loop
8. Print the values using Iterator
9. Removing an element by value
10. Removing an element by index
11. Check the list to see if it contains a particular element

**package** package1;

**import** java.util.HashSet;

**import** java.util.Iterator;

**public** **class** HashSetExample {

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

//Creating an object of linked list

HashSet<String> hh = **new** HashSet<String>();

//Checking if the HashSet is Empty

**if**(hh.isEmpty())

System.***out***.println("The HashSet is empty");

//Adding elements to the linked list

hh.add("Subbu");

hh.add("Selenium");

hh.add("Java");

hh.add("Tutorials");

hh.add("Subscribe");

hh.add("Youtube");

hh.add("Channel");

//Printing linked list

System.***out***.println(hh);

//Check if it allows duplicate values

hh.add("Subbu");

System.***out***.println(hh);

//Check if it allows null values

hh.add("");

System.***out***.println(hh);

//Size of Hash Set

System.***out***.println("Size of HashSet is "+hh.size());

//Printing each element using for each loop

System.***out***.println("Printing values using for each loop:");

**for**(String str : hh) {

System.***out***.println(str);

}

//Iterating through a set and printing values

System.***out***.println("Printing values by iterating through the set:");

Iterator itr = hh.iterator();

**while**(itr.hasNext()) {

System.***out***.println(itr.next());

}

//Removing an element using value

hh.remove("Channel");

System.***out***.println(hh);

//Removing an element using index

hh.remove(5);

System.***out***.println(hh);

//checking if the list contains a particular element

**boolean** check = hh.contains("Java");

**if**(check) {

System.***out***.println("The set contains Java");

}

**else** {

System.***out***.println("It doesn't contain Java");

}

}

}

# **23J. Collections Framework – HashMap**

**Maps – HashMap:**

HashMap stores the data in Key Value pairs. It implements the Map Interface. To know a value of HashMap, one must know the key for it. It uses a technique called Hashing internally.

The following example shows:

1. How to declare a HashMap
2. Check if the HashMap is empty
3. Adding key value pairs to the map
4. Printing the map
5. Check if the map contains a key
6. Get a value using a key
7. Check to see if the map accepts duplicate key value pairs
8. Remove a key value pair using a key
9. Check if the map contains a value
10. Removing all key value pairs using clear()

**package** package1;

**import** java.util.HashMap;

**public** **class** HashMapExample {

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

// **TODO** Auto-generated method stub

HashMap<String, String> hm = **new** HashMap<String, String>();

//Check if the map is empty

**if**(hm.isEmpty()) {

System.***out***.println("Map is empty");

}

//Add key value pairs to map

hm.put("name", "Subbu");

hm.put("role","Test Manager");

hm.put("company", "IBM");

//Printing map

System.***out***.println(hm);

//check if the map contains a key

**if**(hm.containsKey("name")) {

System.***out***.println("Map contains key name");

}

//Getting a value using key

System.***out***.println(hm.get("name"));

//Checking to see if map accepts duplicate key value pairs

hm.put("name", "Subbu");

System.***out***.println(hm);

//Removing a key value pair (Company:IBM)

hm.remove("company");

System.***out***.println("Removed company IBM");

System.***out***.println(hm);

//Check to see if map contains a particular value

**if**(hm.containsValue("Subbu")) {

System.***out***.println("Map contains value Subbu");

}

//Removing all the key value pairs from a map

hm.clear();

System.***out***.println("Removed all key value pairs");

System.***out***.println("Size of map is "+hm.size());

}

}

# **23K. Collections Framework - Map of Maps**

**Maps of Maps:**

**package** Selenium;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** MapofMaps {

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

Map<String, Map<String, String>> mpmp= **new** HashMap<String, Map<String, String>>();

Map<String, String> mp = **new** HashMap<String, String>();

mp.put("name", "Subbu");

mp.put("role", "test manager");

mpmp.put("1", mp);

System.***out***.println(mpmp);

Map<String, String> mp1 = **new** HashMap<String, String>();

mp1.put("name", "Venkat");

mp1.put("role", "test lead");

mpmp.put("2", mp1);

System.***out***.println(mpmp);

System.***out***.println(mpmp.get("1"));

System.***out***.println(mpmp.get("2"));

System.***out***.println(mpmp.get("1").get("name"));

System.***out***.println(mpmp.get("1").get("role"));

System.***out***.println(mpmp.get("2").get("name"));

System.***out***.println(mpmp.get("2").get("role"));

}

}

# **23L. Collections Framework – ConcurrentHashMap**

**Maps – ConcurrentHashMap:**

ConcurrentHashMap stores the data in Key Value pairs. It implements the Map Interface. To know a value of ConcurrentHashMap, one must know the key for it. It uses a technique called Hashing internally. ConcurrentHasMap is fail safe that means it is thread safe.

If we run selenium tests parallelly and if we have to use maps use ConcurrentHashMap.

The following example shows:

1. How to declare a ConcurrentHashMap
2. Check if the ConcurrentHashMap is empty
3. Adding key value pairs to the map
4. Printing the map
5. Check if the map contains a key
6. Get a value using a key
7. Check to see if the map accepts duplicate key value pairs
8. Remove a key value pair using a key
9. Check if the map contains a value
10. Removing all key value pairs using clear()

**package** package1;

**import** java.util.concurrent.ConcurrentHashMap;

**public** **class** ConcurrentHashMapExample {

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

// **TODO** Auto-generated method stub

ConcurrentHashMap<String, String> hm = **new** ConcurrentHashMap<String, String>();

//Check if the map is empty

**if**(hm.isEmpty()) {

System.***out***.println("Map is empty");

}

//Add key value pairs to map

hm.put("name", "Subbu");

hm.put("role","Test Manager");

hm.put("company", "IBM");

//Printing map

System.***out***.println(hm);

//check if the map contains a key

**if**(hm.containsKey("name")) {

System.***out***.println("Map contains key name");

}

//Getting a value using key

System.***out***.println(hm.get("name"));

//Checking to see if map accepts duplicate key value pairs

hm.put("name", "Subbu");

System.***out***.println(hm);

//Removing a key value pair (Company:IBM)

hm.remove("company");

System.***out***.println("Removed company IBM");

System.***out***.println(hm);

//Check to see if map contains a particular value

**if**(hm.containsValue("Subbu")) {

System.***out***.println("Map contains value Subbu");

}

//Removing all the key value pairs from a map

hm.clear();

System.***out***.println("Removed all key value pairs");

System.***out***.println("Size of map is "+hm.size());

}

}

# **23M. Collections Framework - Interview Questions**

**What is collections framework?**

A framework is set of classes and interfaces to build a functionality.

Java collections framework provides set of interfaces and classes for storing and manipulating collections. Collections framework contains classes and interfaces in java.util package and java.util.concurrent packages.

Advantages or benefits of Collections framework :

1) High performance

2) Using this framework we can create different types of collections

3) We can create our own collection and we can extend a collection.

4) Reduces programming effort.

5) Increases speed and quality: Collections framework provides high performance, implementations of useful data structures and algorithms.

**What is collection?**

A collection is a container which holds group of objects. Collection provides a way to manage objects easily. Collections manages group of objects as single unit. Examples include list of strings, integers etc.

Here are few basic operations we do on collections:

1) Adding objects to collection.

2) Removing or deleting objects from collection.

3) Retrieving object from collection.

4) Iterating collection.

**Explain about Collection interface in java?**

Collection is the fundamental and root interface in Collections framework. Collection extends Iterable interface and inherits iterator method which returns Iterator object.

Signature: public interface Collection extends Iterable { }

Methods in Collection interface:

|  |  |
| --- | --- |
| boolean add(E e); | Adds an element to the collection. Returns true if element is added. |
| boolean remove(Object o); | Removes an object from collection if that object is present in collection. Return true if matching object is removed from collection. |
| boolean addAll(Collection<? extends E> c); | Adds all the elements specified in the collection to this collection. Returns true if all elements are added. |
| boolean removeAll(Collection<?> c); | Removes all the elements from this collection that are specified in other collection. Returns true if all the elements are removed. |
| int size(); | Returns number of elements in collection. |
| boolean isEmpty(); | Checks whether collection contains elements or not. If no elements are present it returns false. |
| boolean contains(Object o); | Checks whether specified object is in collection or not. Return true if object is in collection. |
| Iterator<E> iterator(); | Used to iterator over collection. No guarantee on order of elements iterated. |
| boolean retainAll(Collection<?> c); | Removes all the elements which are not in specified collection. Returns only elements specified in collection removing other elements. |
| Object[] toArray(); | Returns an array of elements in collection. |

**What are the basic interfaces of Collections framework?**

The root interface is Iterable and then we have Collection framework. Then we have List, Set and Queue. These are the interfaces of Collections framework.

**Why Map interface doesn’t extend Collection interface?**

Maps are not collections even though they belong to the Collections framework. Maps provide functionality based on key value pairs. Collections are a group of elements. The way Maps and Collections work is different so Map interface doesn’t extend collection interface.

**What is iterable?**

Iterable is the interface which is implemented by Collection interface. It provides a method called iterator() which is used to iterate over a collection.

**What are the different ways to iterate over a list?**

We can iterate over a list in two ways.

1. Iterator
2. For – each loop

**List the interfaces which extends collection interface?**

1) List

2) Set

3) Queue

4) Deque

**List implementations of List Interface?**

1) ArrayList

2) Vector

3) LinkedList

**Explain List interface?**

List interface extends collection interface used to store sequence of elements in collection. We can even store duplicate elements in list. We can insert or access elements in list by using index as we do in arrays. List is an ordered collection.

Some of the operations we can perform on List:

1) Adding an element at specified index.

2) Removing an element at specified index.

3) To get the index of element List contains some specific methods apart from Collection interface methods.

**Explain methods specific to List interface?**

|  |  |
| --- | --- |
| boolean addAll(int index, Collection<? extends E>c); | This method inserts all the elements in specified collection to the list at specified position. |
| E get(int index); | This method returns an element at specified position in the list. |
| E set(int index, E element); | This method replaces the element at specified position in the list with the specified element. |
| void add(int index, E element); | This method inserts the specified element with the index specified. |
| E remove(int index); | This method removes the element at specified index and returns the element removed. |
| int indexOf(Object o); | indexOf() method returns the index of last occurrence of specified element. If there is no element in the list it removes the element. |
| ListIterator<E> listIterator(); | ListIterator<E> listIterator(); Returns a list iterator of elements in list. |
| List<E> subList(int fromIndex, int toIndex); | This method returns list of elements between indexes specified. |

**Explain about ArrayList?**

ArrayList is an ordered collection which extends List interface implements collections interface. We use ArrayList mainly when we need faster access and fast iteration of elements in list. We can insert nulls in to ArrayList. Arraylist is nothing but a growable array.

Advantages:

1) Faster and easier access.

2) Used for Random access of elements.

Drawbacks:

1) We cannot insert or delete elements from middle of list.

**Difference between Array and ArrayList?**

Arrays are used to store primitives or objects of same type or variables that are subclasses of same type.

ArrayList: It is an ordered collection which grows dynamically. In list we can insert nulls values and list allows duplicate elements.

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| While creating array we have to know the size. | But it is not required to know size while creating ArrayList, because arraylist grows dynamically. |
| To put an element in to array we use the following syntax: String array[] = newString[5];array[1] = “java”; We must know specific location to insert an element in to array. If we try to put element in index which is out of range we get arrayIndexOutOfBounds Exception. | We can add element to arraylist with following syntax: List<String> stringList = new ArrayList<String>();stringList.add(“java”); |
| Arrays are static | ArrayList is dynamic |
| We can store objects and primitives | We can store only primitives prior to 1.5 . From 1.5 we can store even objects also. |
| ) We have to manually write logic for inserting and removing elements. | Just a method call would add or remove elements from list. |
| Arrays are faster | Arraylist is slower. |
|  | Arraylist is implemented using arrays |

**What is vector?**

Vector is similar to arraylist used for random access.

Vector is a dynamic array like arraylist.

Size increases or decreases when elements are added and removed.

Vector is synchronized.

**Difference between arraylist and vector?**

Both ArrayList and vector grows dynamically.

The differences between arraylist and vector are:

1) Arraylist is not synchronized and vector is synchronized.

2) Performance wise it is recommended to use arraylist rather than vector because by default vector is synchronized which reduces performance if only one thread accesses it.

**Explain about Sets?**

A set is a collection which does not allow duplicates. Set internally implements equals() method which doesn’t allow duplicates. Adding a duplicate element to a set would be ignored. Set interface is implemented in java.util.set package.

Set interface does not have any additional methods. It has only collection methods. A set can contain atmost one null value. ArrayList is an ordered collection. In arraylists order remains same in which they are inserted. But coming to set it is an unordered collection.

Important operations that can be performed on set:

1) Adding an element to set.

2) Removing an element from set.

3) Check if an element exists in set.

4) Iterating through set.

**Implementations of Set interface?**

1) HashSet

2) Linked HashSet

3) TreeSet

**Explain about Map interface in java?**

A map is an association of key-value pairs. Both keys and values in map are objects.

Features of map:

1. Maps cannot have duplicate keys but can have duplicate value objects.
2. In order to get a value from a map, you need to know the key

Implementations of Map Interface:

1. HashMap
2. HashTable
3. TreeMap

**What is fail-fast system in java?**

When a problem occurs, a fail-fast system fails immediately. In java, we can find this with behaviour with iterators. If you call an iterator on a collection object on a collection object and other thread tries to modify collection object, then concurrent modification exception will be thrown. This is call fail-fast.

Ex: ArrayList, Vector, HashMap

**What is fail-safe system in java?**

Fail-Safe iterators don’t throw any exceptions if the collection is modified while iterating over it. Because they iterate on the clone of the collection and not on the actual collection.

Ex: ConcurrentHashMap

# **24. Exceptions - What are exceptions and how do we work with them**

**Exceptions:**

There are two types of errors in java.

Compile time errors and Runtime errors.

Compile time errors are errors which will be reported during compilation.

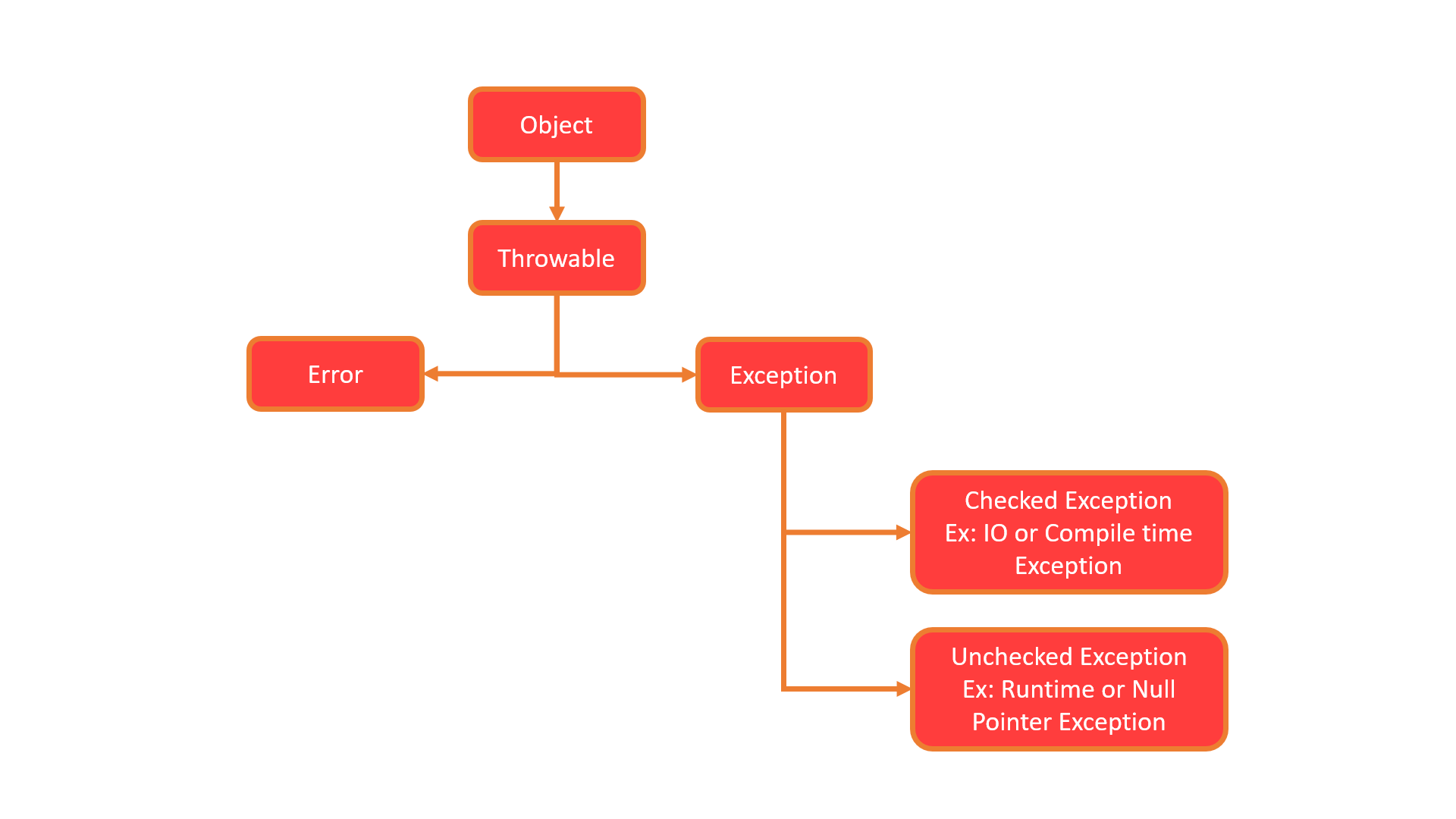
Example:

When you are declaring an integer variable, if you declare “in a” instead of “int a”, this will be a compile time error.

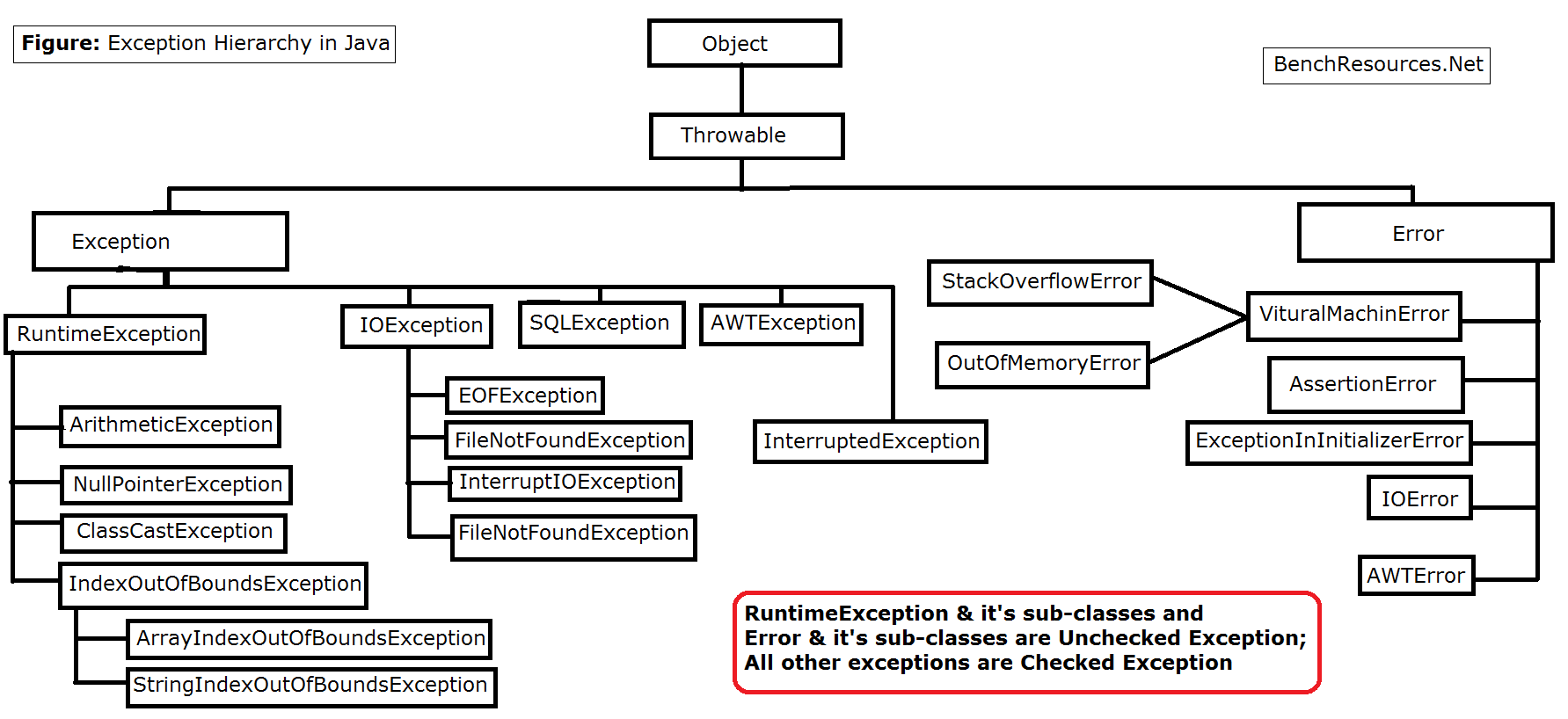
Run time errors are called exceptions.

Example:

Dividing something by zero will throw ArithmaticException.



Error and Exception are sub classes of class Throwable, which is the base class.



**java.lang.Throwable:**

* Throwable is the root class for exception & it’s sub-type and error & it’s sub-types
* In other words, it is super class for exception & error
* java.lang.Throwable class extends java.lang.Object class (as shown in the above figure)
* It defines 2 sub classes i.e.; Exception and Error

**java.lang.Exception:**

* java.lang.Exception is super class for all types of Exception
* It extends java.lang.Throwable class
* Exception are due to programmatic logic
* And it is recoverable
* Exception are categorized into checked exception and unchecked exception
* Example: RuntimeException, SQLException, IOException, FileNotFoundException, ArithmeticException, NullPointerException

**java.lang.Error:**

* java.lang.Error is super class for all types of Error
* It extends java.lang.Throwable class
* Error are due to lack of system resources
* And it is non-recoverable
* All error fall into unchecked exception category, as it is raised due to lack of system resources at runtime
* It is out of programming scope as such type of error can’t predicted, may be well planned care can be taken to avoid these kind of Error
* Example: VirtualMachineError, AssertionError, ExceptionInInitializerError, StackOverflowError, OutOfMemoryError, LinkageError, InstantiationError

**How to handle exceptions in Java:**

Java Exception handling is managed via five keywords – try, catch, finally, throws and throw.

try – catch – finally:

If we think a particular piece of code throws an exception we put it in try block and we write the code to catch the exception in catch block.

Any code that must be executed after a try catch block completes is put in a finally block.

The exceptions we handle using try catch block are called unchecked exceptions.

Ex: ArithmaticException, ArrayIndexOutOfBoundsException etc

Throws:

Sometimes, we know that some methods throw some exceptions and these can be handled by using throws keyword when we define a class. These are checked exceptions.

Ex: IOException, FileNotFoundException, SQLException etc.

Throw:

In java we can define our own exceptions using Throw keyword. These are known as user-defined or custom exceptions.

**How to Handle Exceptions:**

Using Try Catch Block:

Syntax:

try{

Statements;

}

catch(Exception e){

Statements;

}

finally{

Statements;

}

You can use finally block which will be executed irrespective whether an exception raised or not. This is optional.

**Example:**

**package** FPPackage;

**public** **class** StringsDemo {

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

**int** arr1[] = {10,20,30};

**int** arr2[] = {2,5,10};

**for**(**int** i=0; i<3; i++) {

**for**(**int** j=0; j<3; j++) {

System.***out***.println(arr1[i]/arr2[j]);

}

}

}

}

In the above example we are dividing each element of arr1 with each element of arr2 and printing the results. But if any values of arr2 is 0 it will throw ArithmaticException.

Ex:

**package** FPPackage;

**public** **class** StringsDemo {

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

**int** arr1[] = {10,20,30};

**int** arr2[] = {2,5,0};

**for**(**int** i=0; i<3; i++) {

**for**(**int** j=0; j<3; j++) {

System.***out***.println(arr1[i]/arr2[j]);

}

}

}

}

It will print 5 and 2 and then it reports arithmetic exception.

We use try catch blocks to handle these kinds of exceptions and we will continue further.

# **24A. Exceptions - Try Catch Blocks**

**Try Catch Block:**

Try catch blocks are used to catch exceptions.

**package** package1;

**public** **class** ExceptionExample {

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

**int** arr1[] = {10,20,30};

**int** arr2[] = {2,0,10};

**for**(**int** i=0; i<3; i++) {

**for**(**int** j=0; j<3; j++) {

**try** {

System.***out***.println(arr1[i]/arr2[j]);

}

**catch**(Exception e) {

System.***out***.println(e);

}

}

}

}

}

# **24B. Exceptions - Throws Keyword**

**Throws Keyword:**

Throws keyword is used to declare that a method may throw an exception.

For example, when we are creating a file, IOException may occur.

These are checked exceptions. The compiler knows that an exception might occur so we have to catch them and compiler throws an error that we have to catch these exceptions and these exceptions are called checked exceptions. Checked exceptions can be handled by try catch block or throws keyword.

**package** FPPackage;

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** StringsDemo {

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

**try**{

FileWriter file = **new** FileWriter("K:\\Data1.txt");

file.write("Subbu");

file.close();

}

**catch**(IOException e){

System.***out***.println("Some error occured.");

}

}

}

I don’t have K drive in my system so it will throw IOException. But we are catching it with try and catch blocks.

If there are a lot of exceptions like these then we have to write a lot of catch blocks and the code will become cumbersome so in order to avoid this we use throws keyword and mention all the exceptions we expect.

**package** FPPackage;

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** StringsDemo {

**public** **static** **void** main(String[] args) **throws** IOException {

FileWriter file = **new** FileWriter("K:\\Data1.txt");

file.write("Guru99");

file.close();

}

}

# **24C. Exceptions - Throw Keyword**

**Throw Keyword:**

Throw keyword in java is used to explicitly throw an exception from a method or a block of code. We can throw a checked exception or unchecked exception.

Ex:

Throw new ArithmeticException(“Sorry! Not possible”);

Example:

**public** **class** exceptionex {

**static** **void** validate(**int** age) {

**if**(age < 18) {

**try** {

**throw** **new** ArithmeticException("not valid");

}

**catch**(ArithmeticException e) {

System.***out***.println(e);

}

}

**else**

System.***out***.println("welcome to vote");

}

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

*validate*(8);

System.***out***.println("rest of the code");

}

}

# **24D. Exceptions - Interview Questions**

**What is an exception in java?**

An exception is a run time error. In java exception is an object. Exceptions are created when an abnormal situations are arised in our program. Exceptions can be created by JVM or by our application code. All Exception classes are defined in java.lang.

**State some situations where exceptions may arise in java?**

1) Accessing an element that does not exist in array.

2) Dividing an integer by zero (Arithmeticexception)

2) Invalid conversion of number to string and string to number. (NumberFormatException)

3) Invalid casting of class (Class cast Exception)

4) Trying to create object for interface or abstract class (Instantiation Exception)

**What is Exception handling in java?**

Exception handling is a mechanism what to do when some abnormal situation arises in program. When an exception is raised in program it leads to termination of program when it is not handled properly. The significance of exception handling comes here in order not to terminate a program abruptly and to continue with the rest of program normally. This can be done with help of Exception handling.

**In how many ways we can do exception handling in java?**

We can handle exceptions in either of the two ways:

1) By specifying try catch block where we can catch the exception.

2) Declaring a method with throws clause.

**List out five keywords related to Exception handling?**

1) Try

2) Catch

3) throw

4) throws

5) finally

**Explain try and catch keywords in java?**

In try block we define all exception causing code. In java try and catch forms a unit. A catch block catches the exception thrown by preceding try block. Catch block cannot catch an exception thrown by another try block. If there is no exception causing code in our program or exception is not raised in our code jvm ignores the try catch block.

Syntax:

try {

}

Catch(Exception e) {

}

**Can we have try block without catch block?**

Each try block requires at least one catch block or finally block. A try block without catch or finally will result in compiler error. We can skip either of catch or finally block but not both.

**Can we have multiple catch block for a try block?**

In some cases our code may throw more than one exception. In such case we can specify two or more catch clauses, each catch handling different type of exception. When an exception is thrown jvm checks each catch statement in order and the first one which matches the type of exception is execution and remaining catch blocks are skipped.

**Can we have any code between try and catch blocks?**

We shouldn’t declare any code between try and catch block. Catch block should immediately start after try block.

**Explain importance of finally block in java?**

Finally, block is used for cleaning up of resources such as closing connections, sockets etc. if try block executes with no exceptions then finally is called after try block without executing catch block. If there is exception thrown in try block finally block executes immediately after catch block. If an exception is thrown, finally block will be executed even if the no catch block handles the exception.

**Can we catch more than one exception in single catch block?**

From Java 7, we can catch more than one exception with single catch block. This type of handling reduces the code duplication.

Note: When we catch more than one exception in single catch block , catch parameter is implicity final. We cannot assign any value to catch parameter.

Ex: catch(ArrayIndexOutOfBoundsException | ArithmeticException e) {

}

In the above example e is final we cannot assign any value or modify e in catch statement.

Example:

**package** package1;

**public** **class** ExceptionExample {

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

**int** arr1[] = {10,20,30};

**int** arr2[] = {2,0,10};

**for**(**int** i=0; i<3; i++) {

**for**(**int** j=0; j<4; j++) {

**try** {

System.***out***.println(arr1[i]/arr2[j]);

}

**catch**(ArrayIndexOutOfBoundsException | ArithmeticException e) {

System.***out***.println(e);

//}

//catch(ArithmeticException e) {

// System.out.println(e);

}

}

}

}

}

**What are checked Exceptions?**

1) All the subclasses of Throwable class except error, Runtime Exception and its subclasses are checked exceptions.

2) Checked exception should be thrown with keyword throws or should be provided try catch block, else the program would not compile.

We do get compilation error.

Examples:

1) IOException,

2) SQlException,

3) FileNotFoundException,

4) InvocationTargetException,

5) CloneNotSupportedException

6) ClassNotFoundException

7) InstantiationException

**What are unchecked exceptions in java?**

All subclasses of RuntimeException are called unchecked exceptions. These are unchecked exceptions because compiler does not check if a method handles or throws exceptions. Program compiles even if we do not catch the exception or throws the exception. If an exception occurs in the program, program terminates. It is difficult to handle these exceptions because there may be many places causing exceptions.

Example :

1) Arithmetic Exception

3) ArrayIndexOutOfBoundsException

4) ClassCastException

5) IndexOutOfBoundException

6) NullPointerException

7) NumberFormatException

8) StringIndexOutOfBounds

9) UnsupportedOperationException

**Explain differences between checked and Unchecked exceptions in java?**

|  |  |
| --- | --- |
| **Unchecked Exceptions** | **Checked Exceptions** |
| All the subclasses of RuntimeException are called unchecked exception. | All subclasses of Throwable class except RuntimeException are called as checked exceptions |
| Unchecked exceptions need not be handled at compile time | Checked Exceptions need to be handled at compile time. |
| These exceptions arise mostly due to coding mistakes in our program. |  |
| ArrayIndexOutOfBoundsException, ClassCastException, IndexOutOfBoundException | SqlException, FileNotFoundException,ClassNotFoundException |

**What is default exception handling in Java?**

When JVM detects exception causing code, it constructs a new exception handling object by including the following information.

1) Name of Exception

2) Description about the Exception

3) Location of Exception.

After creation of object by JVM it checks whether there is exception handling code or not. If there is exception handling code then exception handles and continues the program. If there is no exception handling code JVM give the responsibility of exception handling to default handler and terminates abruptly. Default Exception handler displays description of exception, prints the stacktrace and location of exception and terminates the program.

Note: The main disadvantage of this default exception handling is program terminates abruptly.

Example:

**package** package1;

**public** **class** ExceptionExample {

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

**int** arr1[] = {10,20,30};

**int** arr2[] = {2,0,10};

**for**(**int** i=0; i<3; i++) {

**for**(**int** j=0; j<3; j++) {

System.***out***.println(arr1[i]/arr2[j]);

}

}

}

}

**Result:**

Exception in thread "main"

java.lang.ArithmeticException: / by zero

at package1.ExceptionExample.main(ExceptionExample.java:11)

**Can we nested try statements in java?**

Yes try statements can be nested. We can declare try statements inside the block of another try statement.

**Explain the importance of throwable class and its methods?**

Throwable class is the root class for Exceptions. All exceptions are derived from this throwable class. The two main subclasses of Throwable are Exception and Error.

The three methods defined in throwable class are :

1. void printStackTrace() :

This prints the exception information in the following format : Name of the exception, description followed by stack trace.

1. getMessage()

This method prints only the description of Exception.

1. toString():

It prints the name and description of Exception.

# **25. Java File Operations - Part 1 - Java Input Output and File Path Operations**

Java I/O is used to process input and produce output. An input stream reads the file and stores the data in process. The output stream reads from the process and writes to the destination file.

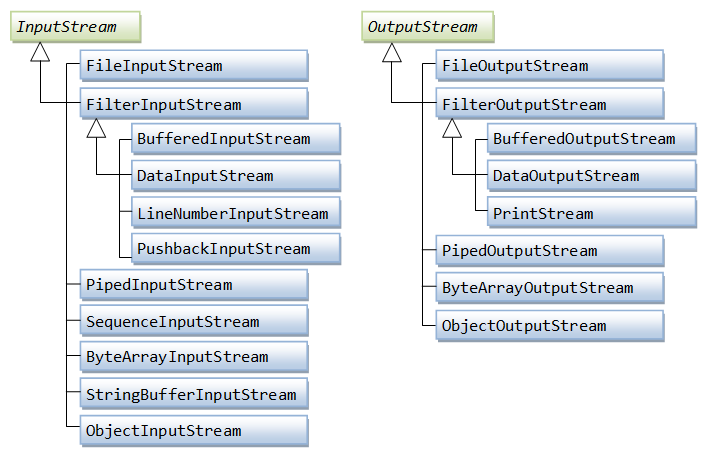
There are two types of streams – Byte Streams and Character Streams

[Byte streams](http://way2java.com/io/byte-streams-vs-character-streams/) were introduced with JDK 1.0 and operate on the files containing ASCII characters. We know Java supports other language characters also known as Unicode characters. To read the files containing Unicode characters, the designers introduced character streams with JDK 1.1. As ASCII is a subset of Unicode, for the files of English characters, we can go with either byte streams or [character streams](http://way2java.com/io/byte-streams-and-character-streams/).

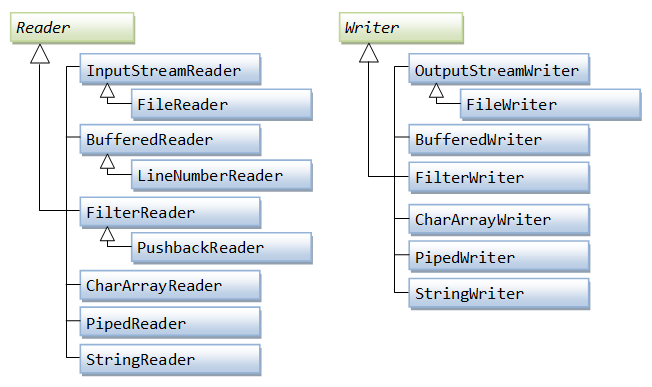
All Byte streams can be divided into two types – InputStream and OutputStream

All Character streams can be divided into two types – Reader and Writer.

Byte I/O Streams:



Character I/O Streams:



**File/Directory Operations in Java:**

File/Directory Operations are operations other than reading or writing to files like checking if a file exists in directory, creating a new directory, creating a new file etc.

These operations are provided by java.io.File class.

These methods can be divided into following.

1. Methods that query the file system
2. Methods that modify the file system

Some of the methods that query the file system are

1. canRead()
2. canWrite()
3. exists()
4. isDirectory()
5. isFile()
6. isHidden()
7. getAbsolutePath()
8. lastModified()
9. length()
10. listFiles()
11. listRoots()

Some of the methods that modify the file system are

1. createNewFile()
2. mkdir()
3. renameTo()
4. delete()
5. deletOnExit()
6. setReadOnly()
7. setLastModified()

# **25A. Java File Operations - Part 2 - Java File and Directory Path Operations Example**

Example on checking the File/Directory operations:

**package** FPPackage;

**import** java.io.File;

**import** java.io.IOException;

**public** **class** CheckingFiles {

**public** **static** **void** main(String[] args) **throws** IOException {

File tmpDir = **new** File("D:\\Selenium Course\\FileOperations");

File tmpFile = **new** File(tmpDir+"\\Sample.txt");

**if**(tmpDir.isDirectory()) {

System.***out***.println("This is a directory");

}

**if**(tmpDir.exists()) {

System.***out***.println("The directory exists");

}

**if**(tmpFile.isFile()) {

System.***out***.println("This is file");

}

**if**(tmpFile.exists()) {

System.***out***.println("The file exists");

}

**if**(tmpFile.canRead()) {

System.***out***.println("The file can be read");

}

**if**(tmpFile.canWrite()) {

System.***out***.println("The file can be written");

}

**if**(tmpFile.isHidden()) {

System.***out***.println("The file is hidden");

}

**else** {

System.***out***.println("The file is not hidden");

}

File path = tmpFile.getAbsoluteFile();

System.***out***.println(path);

File file = **new** File(tmpDir+"\\Sample1.txt");

**if**(file.createNewFile()) {

System.***out***.println("File created successfully.");

}

File dir = **new** File(tmpDir+"\\test");

**if**(dir.mkdir()) {

System.***out***.println("Directory Created Successfully");

}

}

}

# **25B. Java File Operations - Part 3 - FileWriter and FileReader**

**FileWriter:**

Java FileWriter class is used to write character-oriented data to a [file](https://www.javatpoint.com/java-file-class). It is character-oriented class which is used for file handling in [java](https://www.javatpoint.com/java-tutorial).

Unlike FileOutputStream class, you don't need to convert string into byte [array](https://www.javatpoint.com/array-in-java) because it provides method to write string directly.

Example:

**package** FPPackage;

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** FileWriterExample {

**public** **static** **void** main(String[] args) **throws** IOException {

FileWriter fw = **new** FileWriter("C:\\JavaExamples\\test.txt");

String s = "Hello Hello Hello";

fw.write(s);

fw.close();

System.***out***.println("Success!");

}

}

**FileReader:**

Java FileReader class is used to read data from the file. It returns data in byte format like [FileInputStream](https://www.javatpoint.com/java-fileinputstream-class) class.

It is character-oriented class which is used for [file](https://www.javatpoint.com/java-file-class) handling in [java](https://www.javatpoint.com/java-tutorial).

Example:

**package** FPPackage;

**import** java.io.FileReader;

**public** **class** FileReaderExample {

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

**try** {

FileReader fr = **new** FileReader("C:\\JavaExamples\\test.txt");

**int** i;

**while**((i = fr.read())!= -1){

System.***out***.print((**char**)i);

}

fr.close();

}

**catch**(Exception e) {

System.***out***.println(e);

}

}

}

# **25C. Java File Operations - File Input and Output Interview Questions**

**What is a stream and what are the types of Streams and classes of the Streams?**

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow. There are two types of Streams :

Byte Streams: Provide a convenient means for handling input and output of bytes.

Character Streams: Provide a convenient means for handling input & output of characters.

Byte Streams classes: Are defined by using two abstract classes, namely InputStream and OutputStream.

Character Streams classes: Are defined by using two abstract classes, namely Reader and Writer.

**What is an IO stream?**

It is a stream of data that flows from source to destination. Good example is file copying. Two streams are involved – input stream and output stream. An input stream reads from the file and stores the data in the process (generally in a temporary variable). The output stream reads from the process and writes to the destination file.

**What is the necessity of two types of streams – byte streams and character streams?**

Byte streams were introduced with JDK 1.0 and operate on the files containing ASCII characters. We know Java supports other language characters also known as Unicode characters. To read the files containing Unicode characters, the designers introduced character streams with JDK 1.1. As ASCII is a subset of Unicode, for the files of English characters, we can go with either byte streams or character streams.

**What are the super most classes of all streams?**

All the byte stream classes can be divided into two categories (input stream classes and output stream classes) and all character streams classes into two (reader classes and writer classes). There are four abstract classes from which all these streams are derived. The super most class of all byte stream classes is java.io.InputStream and for all output stream classes, java.io.OutputStream. Similarly for all reader classes is java.io.Reader and for all writer classes is java.io.Writer.

**What is the difference between the Reader/Writer class hierarchy and the InputStream/OutputStream class hierarchy?**

The Reader/Writer class hierarchy is character-oriented, and the InputStream/OutputStream class hierarchy is byte-oriented

**What value does read() return when it has reached the end of a file?**

The read() method returns -1 when it has reached the end of a file.

**What is the package name for ObjectInputStream class?**

java.io

**Which is the Parent Class of StringBufferInputStream class?**

InputStream

Note: Anything that ends with InputStream are subclasses of InputStream.

Anything that ends with OutputStream are subclasses of OutputStream.

Anything that ends with Reader are subclasses of Reader.

Anything that ends with Writer are subclasses of Writer.

**What are FileInputStream and FileOutputStream?**

These two are general purpose classes used by the programmer very often to copy file to file. These classes work well with files containing less data of a few thousand bytes as by performance these are very poor. For larger data, it is preferred to use BufferedInputStream (or BufferedReader) and BufferedOutputStream (or BufferedWriter).

**Which you feel better to use – byte streams or character streams?**

I feel personally to go with character streams as they are the latest. Many features exist in character streams that do not in byte streams like a) using BufferedReader in place of BufferedInputStreams and DataInputStream (one stream for two) and b) using newLine() method to go for next line and for this effect we must go for extra coding in byte streams etc.

**What is System.out.println()?**

"println()" is a method of PrintStream class. "out" is a static object of PrintStream class defined in "System" class. System is a class from java.lang package used to interact with the underlying operating system by the programmer.

**What are filter streams?**

Filter streams are a category of IO streams whose responsibility is to add extra functionality (advantage) to the existing streams like giving line numbers in the destination file that do not exist int the source file or increasing performance of copying etc.

**Name the filter streams available?**

There are four filter streams in java.io package – two in byte streams side and two in character streams side. They are FilterInputStream, FilterOutputStream, FilterReader and FilterWriter. These classes are abstract classes and you cannot create of objects of these classes.

**Name the filter stream classes on reading side of byte stream?**

There are four classes – LineNumberInputStream (the extra functionality is it adds line numbers in the destination file), DataInputStream (contains special methods like readInt(), readDouble() and readLine() etc that can read an int, a double and a string at a time), BufferedInputStream (gives buffering effect that increases the performance to the peak) and PushbackInputStream (pushes the required character back to the system).

**What is the functionality of SequenceInputStream?**

It is very useful to copy multiple source files into one destination file with very less code.

**What is PrintStream and PrintWriter?**

Functionally both are same but belong to two different categories – byte streams and character streams. println() method exists in both classes.

**Which streams are advised to use to have maximum performance in file copying?**

BufferedInputStream and BufferedOutputStream on byte streams side and BufferedReader and BufferedWriter on character streams side.

**What are piped streams?**

There are four piped streams – PipedInputStream, PipedOutputStream, PipedReader and PipedWriter. These streams are very useful to pass data between two running threads (say, processes).

**What is File class?**

It is a non-stream (not used for file operations) class used to know the properties of a file like when it was created (or modified), has read and write permissions, size etc.

# **26. Java Database Connectivity - Installing XAMPP MySql Database On Local Machine**

**Installing MySql database locally for practice:**

You can download and install mysql from mysql.com. But it is little complex. So instead of downloading mysql, you can download a local server with which mysql comes. There are a few local servers you can download and some of them are XAMPP, WAMPP etc.

For installing MySql, we will download and install XAMPP. With this we get Apache server and MySql database.

**Note: We also use this when we do a project on selenium. So, I recommend you to install this.**

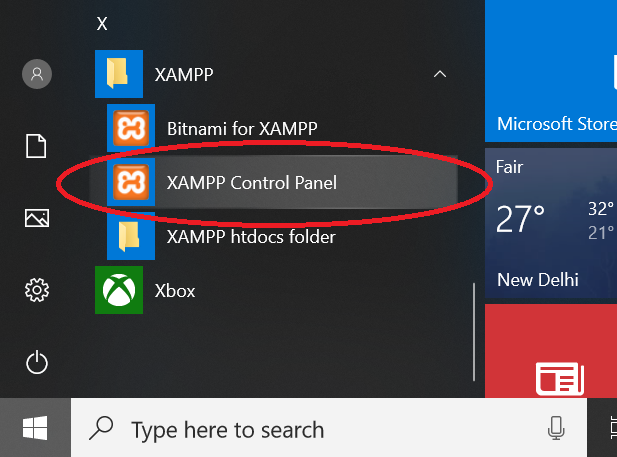
Go to google and search for XAMPP download.

You will get many sources from where you can download XAMPP. The following is one of them.

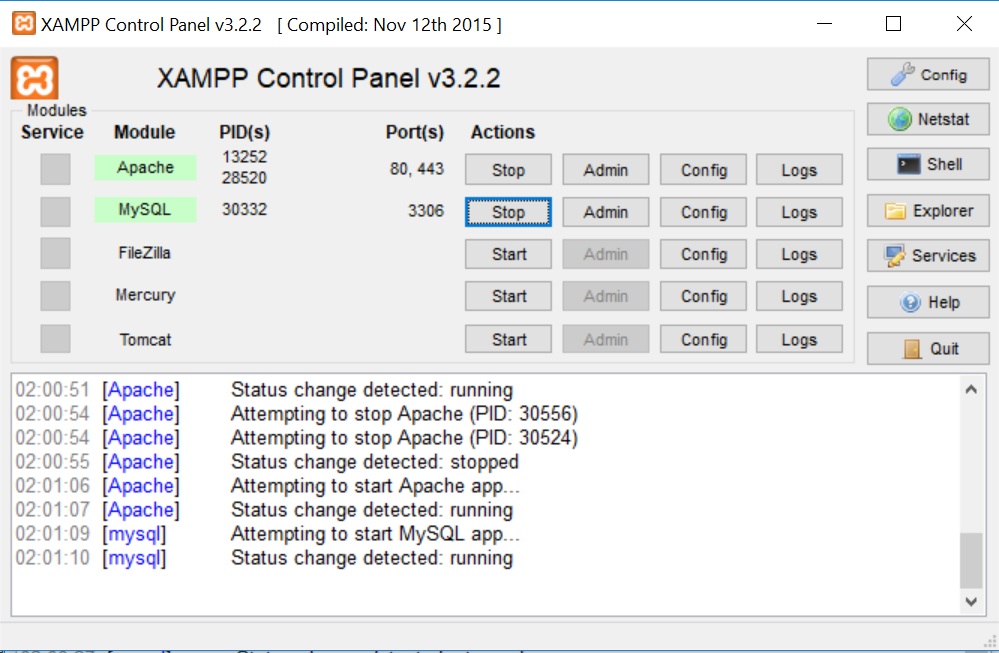
<https://xampp-windows.en.softonic.com/download>



Once you download XAMPP, install it on your system.

****

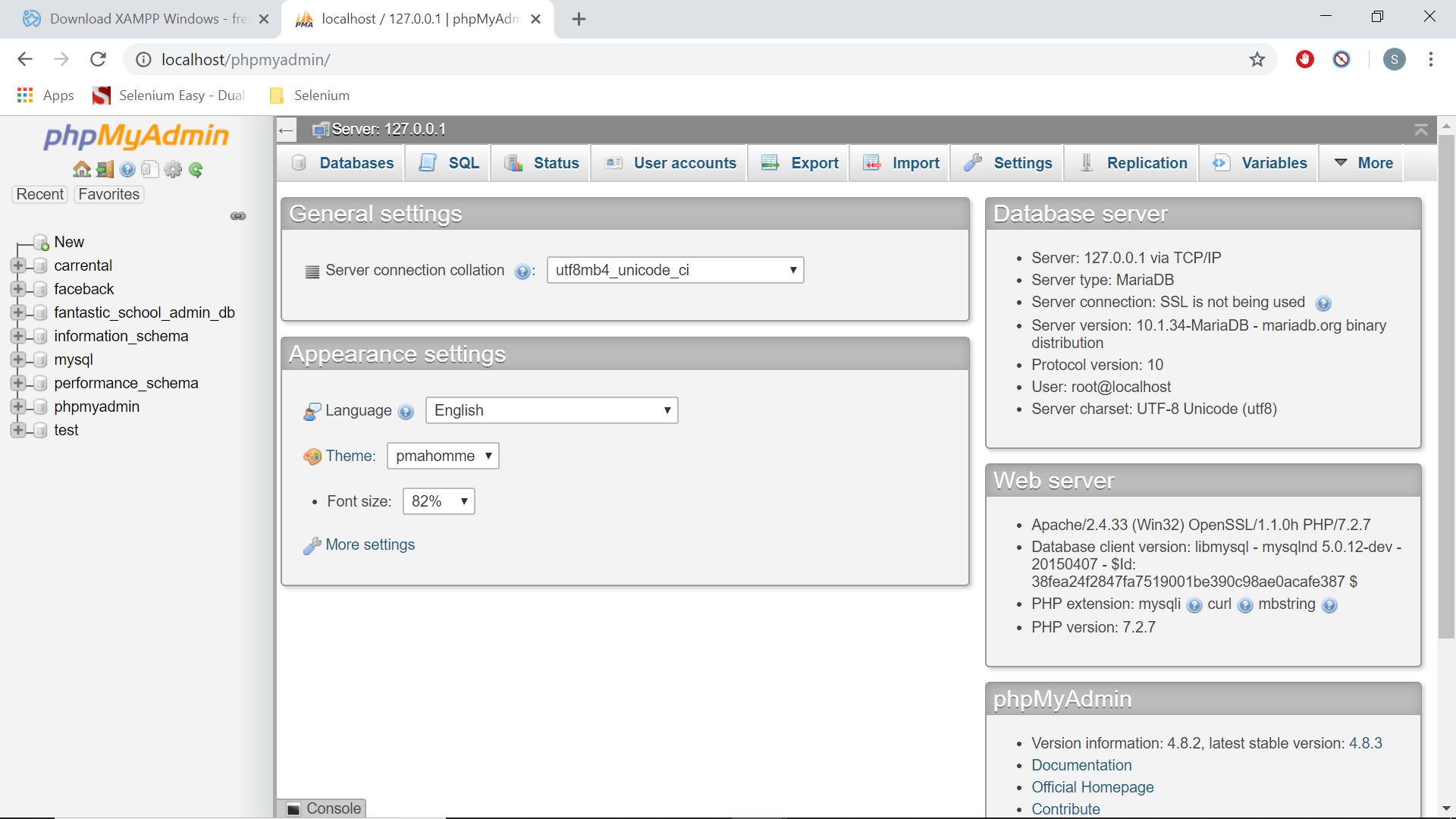
Once it is installed successfully, go to windows start button and go to folder XAMPP and you will see XAMPP Control Panel. Open the XAMPP control panel.



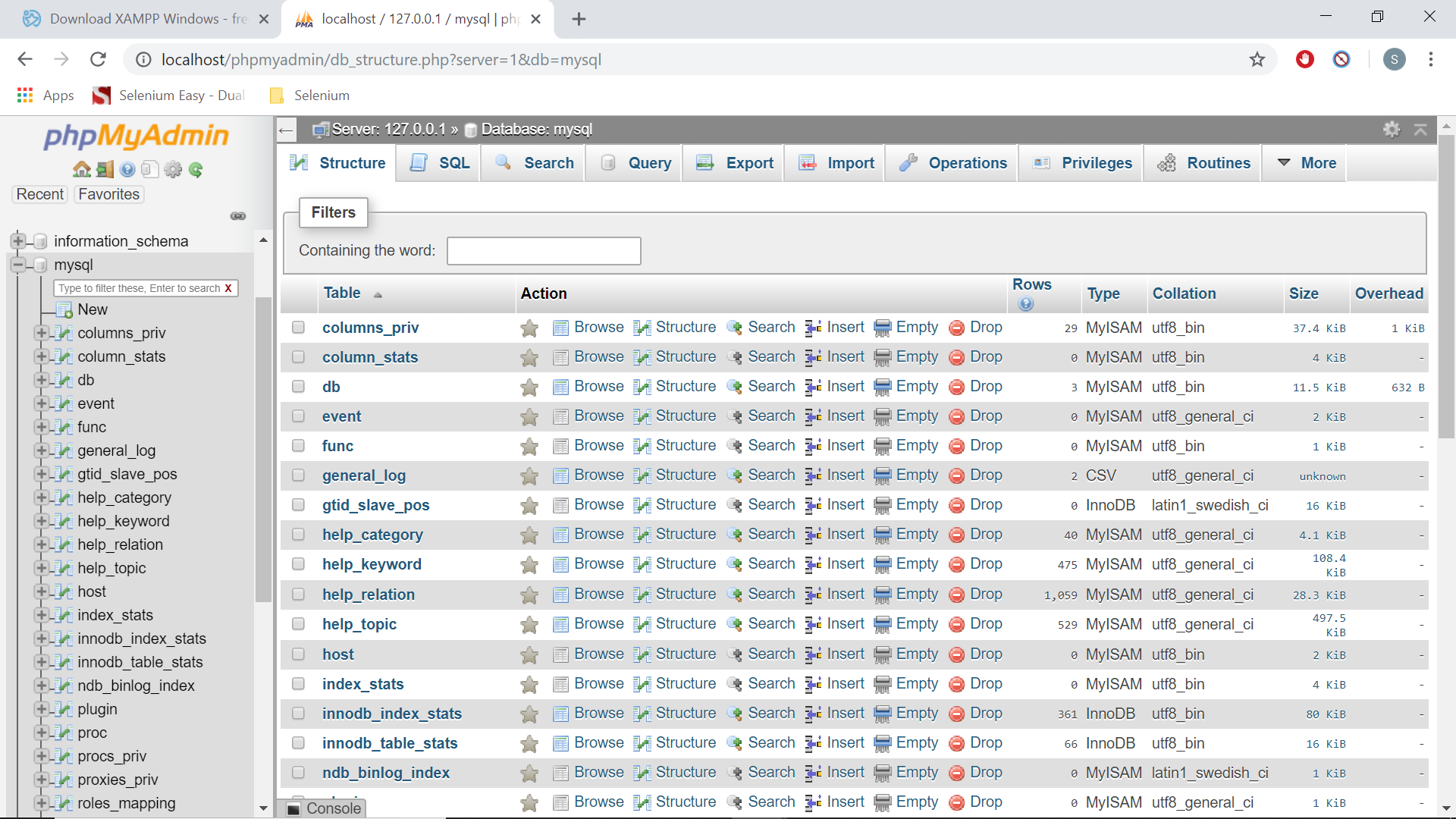
Start the Apache Server and MySQL server by clicking on start buttons. You should see that both are started without any errors.

Now you can access the database using the following url.

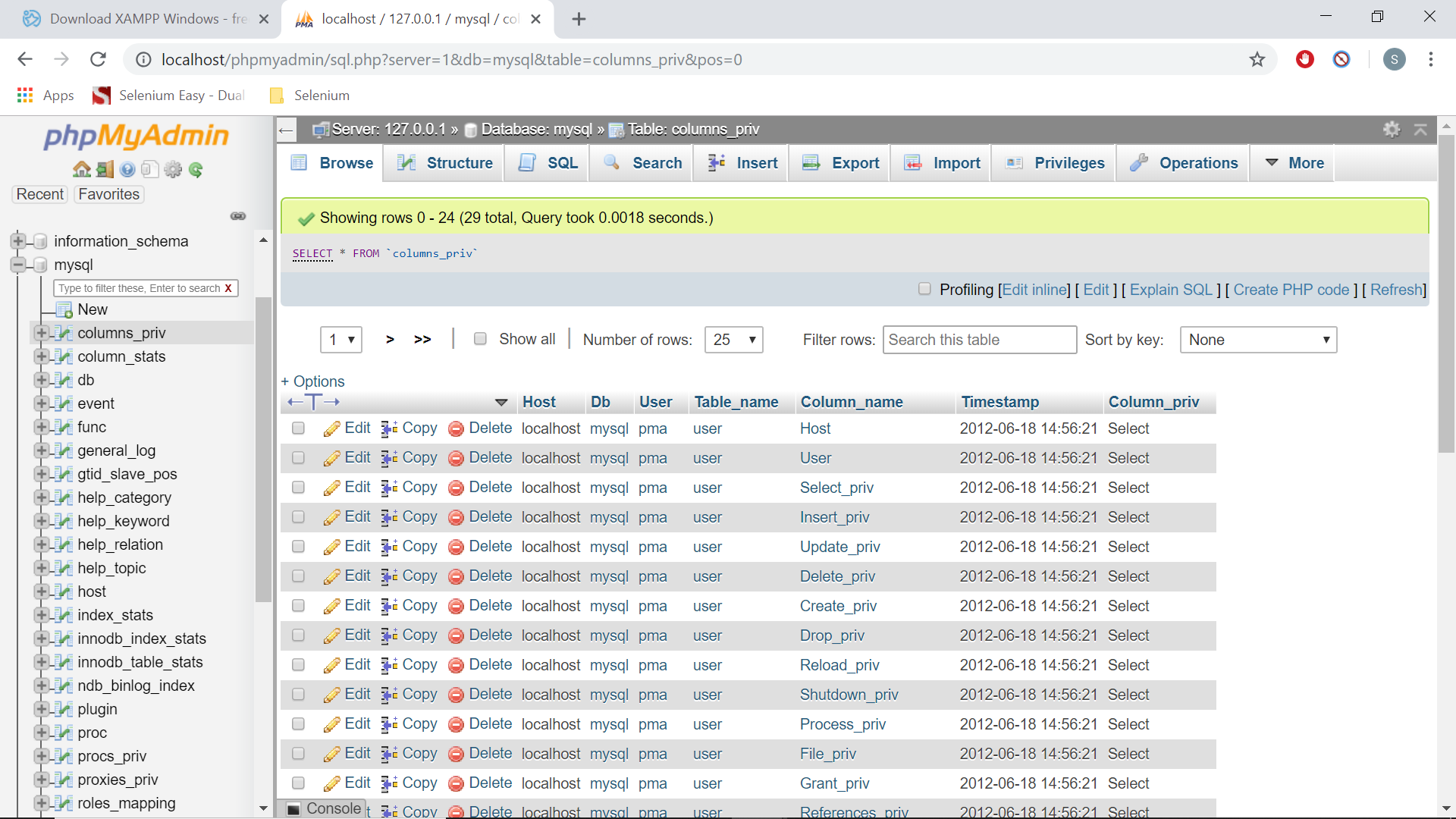
<http://localhost/phpmyadmin>



On the left-hand side, you will see different databases.



Once you click on any database, you will see the different tables associated with it.



Once you click on any table you will see the data of the tables.

**Setting up a User Name and Password:**

Once you install XAMPP, the default user name and password for phpMyAdmin are

User name: root

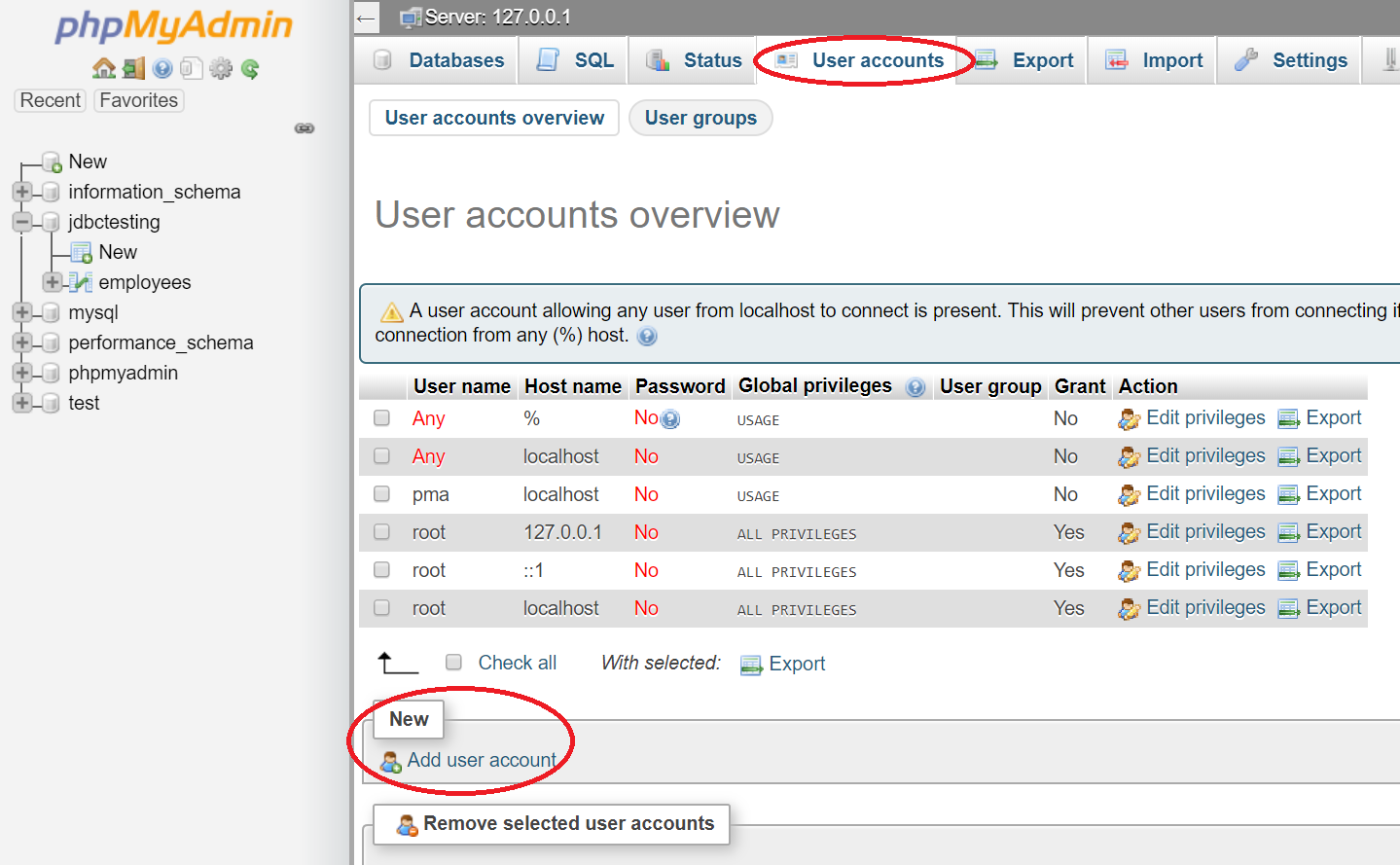
Password: (no password).

You can use the above details for connecting to any database using JDBC. But let’s create a new username and password.

Go to the home page by clicking on the home button or phpMyAdmin image on the top left side. Now click on “User Accounts” and you should be able to see all the user names here.

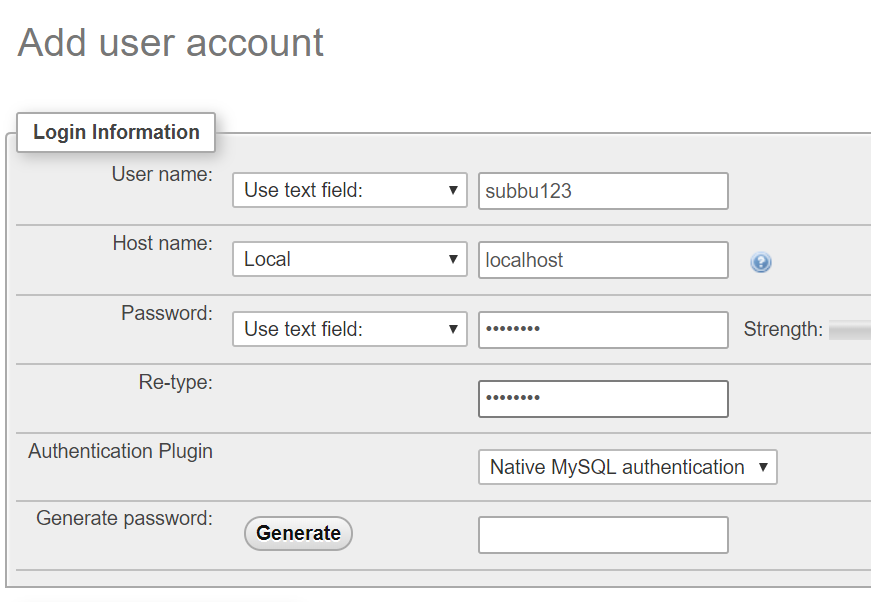
Click on “Add User Account” link.

Enter the user name and password you like.

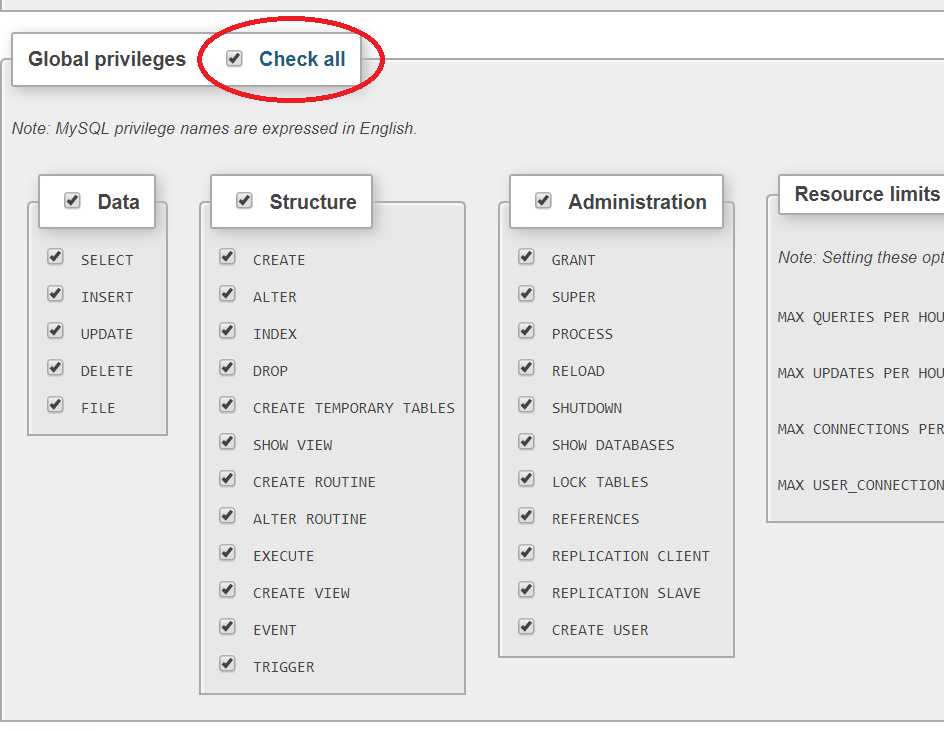


Enter the user name and password you like. (I am adding username: subbu123 and password: subbu123)

Select the host as localhost. (This is important otherwise it won’t work)



Under the Global Privileges section, select the checkbox “Check All” and save it and a new user should be created with all the privileges.



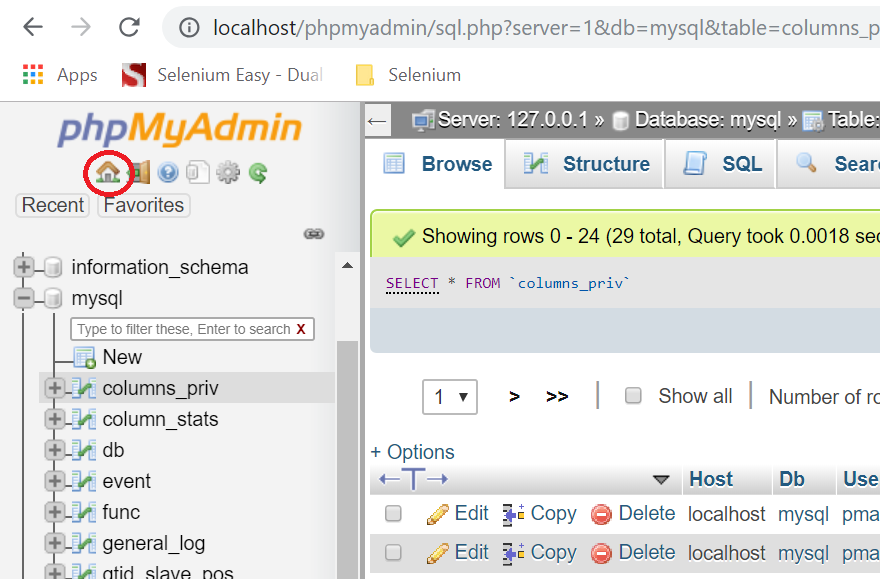
This step is important as if you don’t select this check box the new user will not get all the privileges.

# **26A. Java Database Connectivity - Basic SQL Knowledge**

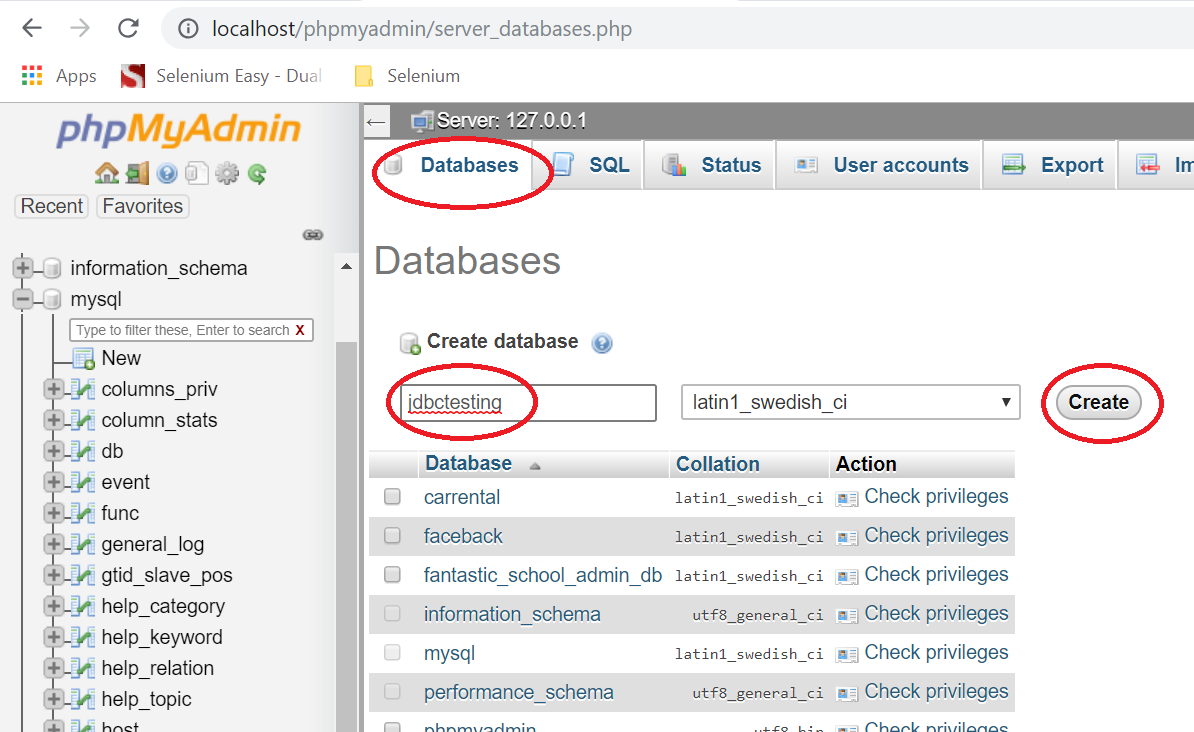
**Basic SQL Knowledge for Database Testing:**

You need to have minimum SQL knowledge to work with databases. So in this section I cover those sql statements.

First on the phpMyAdmin page click on the home button on the top left hand side.



Click on Databases link and enter the name of the database as “jdbctesting” and click on “Create” button.



Now the database should be created and you should see it in the left pane.

**Basic SQL Queries:**

We will be creating a table “employees” with no. of 6 columns and they are

1. id (auto increment)
2. firstname (for capturing employee first name)
3. lastname (for capturing employee last name)
4. age (for capturing the age of the employee)
5. company (for capturing the company name of the employee)
6. salary (for capturing the salary of employee)

**Creating Tables and Data:**

Now click on the SQL link.



In this page you can practice SQL queries.

**Creating Table:**

CREATE TABLE `jdbctesting`.`employees` (

`id` INT NOT NULL primary key AUTO\_INCREMENT ,

`firstname` VARCHAR(50) NOT NULL ,

`lastname` VARCHAR(50) NOT NULL ,

`age` INT NOT NULL ,

`company` VARCHAR(50) NOT NULL ,

`salary` INT NOT NULL

) ENGINE = InnoDB;

**Note: In phpMyAdmin, when executing the above query use backquotes(“ ` “) instead of single quotes (“ ‘ “ ). Otherwise the above query won’t work.**

**Inserting Data into Table:**

Query 1:

insert into jdbctesting.employees ( firstname, lastname, age, company, salary)

VALUES('james', 'smith', 35, 'IBM', 1250000)

Query 2:

insert into jdbctesting.employees ( firstname, lastname, age, company, salary)

VALUES('michael', 'smith', 35, 'HONDA', 1800000)

**Retrieving Data from Table:**

If you want to retrieve all the data use select query.

SELECT \* FROM employees

If you want to retrieve specific rows use select query with where condition.

SELECT \* FROM employees WHERE id = 1

**Updating Data in the Table:**

UPDATE employees SET firstname = 'Robert' WHERE id = 1

Make sure that you use the where condition otherwise it will update the entire data.

**Deleting Data from Table:**

DELETE FROM employees WHERE id = 1

Make sure that you use the where condition otherwise it will delete the entire table.

The most important queries that you use are SELECT and UPDATE.

# **26B. Java Database Connectivity - Connecting to Database and Execute Queries using Java**

**Java Database Connectivity:**

JDBC is java database connectivity.

Java JDBC is a Java API to connect and execute queries with the database. JDBC API uses jdbc drivers to connect with the database.

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

* Register the Driver class
* Create connection
* Create statement
* Execute queries
* Close connection

1. Register the Driver Class

The forName() method of class Class is used to register the driver class.

For MySQL it is:

Class.forName("com.mysql.jdbc.Driver");

For Oracle it is:

Class.forName("oracle.jdbc.driver.OracleDriver");

1. Create a connection object

For MySQL:

Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/dbname","root","root");

For Oracle:

Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","password");

1. Create the statement object

For MysQL/Oracle:

Statement stmt=con.createStatement();

1. Execute Query

ResultSet rs=stmt.executeQuery("select \* from emp");

While(rs.next()){

System.out.println(rs.getInt(1)+” “+rs.getString(2));

}

**Note: For selecting data we use stmt.executeQuery**

**For inserting, updating, deleting data we use stmt.executeUpdate**

1. Close the connection

con.close()

**Example for Connecting to MySQL:**

**package** FPPackage;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.Statement;

**public** **class** JDBCTest {

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

**try** {

Class.*forName*("com.mysql.cj.jdbc.Driver");

//Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/carrental?useSSL=false","root","vinayaka123");

Connection con = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/jdbctesting","subbu123","subbu123");

Statement stmt=con.createStatement();

//stmt.executeUpdate("insert into jdbctesting.employees ( firstname, lastname, age, company, salary)VALUES('robert', 'smith', 30, 'HONDA', 1250000)");

ResultSet rs=stmt.executeQuery("select \* from jdbctesting.employees");

**while**(rs.next())

System.***out***.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3)+" "+rs.getInt(4)+" "+rs.getString(5)+" "+rs.getInt(6));

con.close();

//stmt.executeUpdate("UPDATE jdbctesting.employees SET firstname='Mary', age=30, company='Infosys', salary=1800000 where id=2");

//stmt.executeUpdate("DELETE FROM jdbctesting.employees where id=3");

}

**catch**(Exception e) {

System.***out***.println(e);

}

}

}

Results:

1 Maruti 2017-06-18 21:54:34

2 BMW 2017-06-18 21:54:50

3 Audi 2017-06-18 21:55:03

4 Nissan 2017-06-18 21:55:13

5 Toyota 2017-06-18 21:55:24

7 Marutiu 2017-06-19 11:52:13

**Example for connecting to Oracle:**

**package** FPPackage;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.Statement;

**public** **class** JDBCTest {

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

**try** {

Class.*forName*("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:carrental","root","password");

Statement stmt=con.createStatement();

ResultSet rs=stmt.executeQuery("select \* from carrental.tblbrands");

**while**(rs.next())

System.***out***.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));

con.close();

}

**catch**(Exception e) {

System.***out***.println(e);

}

}

}

# **27. Java Reflections - Loading classes, constructors and methods at run time.**

**Java Reflections – Loading Classes, Constructors and Methods at Run Time:**

Java reflections package is used if we want to load any class, it’s constructor or methods at run time. In this one we only see what is needed for our selenium tutorials.

Here we created a package “actions” and a class under it called HomePage. We have created two static variables and a constructor to initialize those variables. We have also created four methods as well.

We have created another package “drivers” and class under it “ContactActions”. In this class we don’t create any object for HomePage class but we dynamically access this class and it’s constructor and methods at run time.

**HomePage.java of actions package:**

**package** actions;

**public** **class** HomePage {

**public** **static** **int** *globala*;

**public** **static** String *globalstr*;

**public** HomePage(**int** a, String str) {

*globala* = a;

*globalstr* = str;

}

**public** **int** getSumofTwoWithGlobal(**int** a) {

**return**(*globala*+a);

}

**public** **int** getSumofThreeWithGlobal(**int** a, **int** b) {

**return**(*globala*+a+b);

}

**public** String concatTwoWithGlobal(String str1) {

**return**(*globalstr*+str1);

}

**public** String concatThreeWithGlobal(String str1, String str2) {

**return**(*globalstr*+str1+str2);

}

}

For accessing a class at run time, we use Class.forName() method. We have to pass the package name followed by “.” followed by class name. It returns a Class object. By writing Class<?>, you're declaring a Class object which can be of any type (? is a wildcard). The Class type is a type that contains meta information about a class.

Yu can get all the methods of the class by using getDeclaredMethods() which returns an array of public methods the class has. So we save all these in Method[].

Now you can print all the methods using Arrays.toString() method or you can loop through the Method[] array and print one by one.

To invoke any method first we have to load the method. This we do it with

Method mc = cls.getDeclaredMethod("getSumofTwoWithGlobal",**int**.**class**);

For loading a method we use getDeclaredMethod() and we have to pass the name of the method and type of variables the methoduse. If we use integer parameters in the method we use int.class and if we use string parameters it is String.class.

Next before invoking the method, we need to load the constructor. We use getConstructor() method and the variable types the constructor use.

Constructor<?> constructor = cls.getConstructor(**int**.**class**, String.**class**);

Next, we invoke the method using invoke() method. We first pass the constructor.newInstance() as parameter and then the actual parameters the method use. Since this method returns an integer, we capture that in variable sum.

**int** sum = (**int**) mc.invoke(constructor.newInstance(1, "Selenium"), 2);

**ContactActions.java of drivers package:**

**package** drivers;

**import** java.lang.reflect.Constructor;

**import** java.lang.reflect.InvocationTargetException;

**import** java.lang.reflect.Method;

**import** java.util.Arrays;

**public** **class** ContactActions {

**public** **static** **void** main(String[] args) **throws** ClassNotFoundException, NoSuchMethodException, SecurityException, IllegalAccessException, IllegalArgumentException, InvocationTargetException, InstantiationException {

Class<?> cls = Class.*forName*("actions.HomePage");

//Print all public methods

Method[] publicMethods = cls.getDeclaredMethods();

System.***out***.println(Arrays.*toString*(publicMethods));

**for**(**int** i=0; i<publicMethods.length; i++) {

System.***out***.println(publicMethods[i]);

}

//Invoke method getSumofTwoWithGlobal

Method mc = cls.getDeclaredMethod("getSumofTwoWithGlobal",**int**.**class**);

Constructor<?> constructor = cls.getConstructor(**int**.**class**, String.**class**);

**int** sum = (**int**) mc.invoke(constructor.newInstance(1, "Selenium"), 2);

System.***out***.println("Sum for getSumofTwoWithGlobal is "+sum);

//Invoke method getSumofThreeWithGlobal

mc = cls.getDeclaredMethod("getSumofThreeWithGlobal",**int**.**class**,**int**.**class**);

constructor = cls.getConstructor(**int**.**class**, String.**class**);

sum = (**int**) mc.invoke(constructor.newInstance(1, "Selenium"), 2, 3);

System.***out***.println("Sum for getSumofThreeWithGlobal is "+sum);

//Invoke method concatTwoWithGlobal

mc = cls.getDeclaredMethod("concatTwoWithGlobal", String.**class**);

constructor = cls.getConstructor(**int**.**class**, String.**class**);

String cnt = (String) mc.invoke(constructor.newInstance(1, "Selenium"), " Automation");

System.***out***.println(cnt);

//Invoke method concatThreeWithGlobal

mc = cls.getDeclaredMethod("concatThreeWithGlobal", String.**class**, String.**class**);

constructor = cls.getConstructor(**int**.**class**, String.**class**);

cnt = (String) mc.invoke(constructor.newInstance(1, "Selenium"), " Automation", " Testing");

System.***out***.println(cnt);

}

}

# **28. Java Date Functionality - Getting Current Date and Future Dates**

**Getting Current Date and Future Dates:**

**import** java.text.SimpleDateFormat;

**import** java.util.Calendar;

**import** java.util.Date;

**public** **class** JavaDateExamples {

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

Date d1 = **new** Date();

System.***out***.println("Current Date is "+d1);

SimpleDateFormat sdf = **new** SimpleDateFormat("dd/MM/yyyy");

String d2 = sdf.format(d1);

System.***out***.println("Formatted date is "+d2);

Calendar c = Calendar.*getInstance*();

c.setTime(**new** Date()); // Now use today date.

c.add(Calendar.***DATE***, 1); // Adding 1 days

String fromdate = sdf.format(c.getTime());

System.***out***.println(fromdate);

}

}

# **Enumerations in Java**

Enum in java is a datatype which contains a fixed set of constants.

It can be used for days of week, seasons, colors etc.

Since these are constants, we should declare the enum constants in capital letters.

* Enums improve type safety
* Enums can be easily used in switch
* Enums can be traversed
* Enums can have fields, constructors and methods
* Enums may implement many interfaces but cannot extend any class because it internally extends Enum class

Enums can be declared outside of class, inside of class or as a separate program.

Enum constants can be retrieved by “enum name”.”enum constant”.

If a value is assigned by a constructor, we can retrieve the value by using “enum name”. “enum constant”.value. (value is the variable name we used)

**Enum Outside Class:**

The following example is for an enum outside of a class.

**package** enumerations;

**enum** color {

***RED***, ***GREEN***, ***BLUE***;

}

**public** **class** EnumOutsideClass1 {

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

**color c1 = color.*RED*;**

**System.*out*.println(c1);**

}

}

Result:

WINTER

**Enum Inside Class:**

The following example is for an enum inside a class.

**package** enumerations;

**public** **class** EnumInsideClass2 {

**public** **enum** season{

***WINTER***, ***SPRING***, ***SUMMER***, ***FALL***;

}

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

//Printing an enum constant

season s = season.***WINTER***;

System.***out***.println(s);

}

}

Result: WINTER

**Looping Through Enum:**

The following example shows how to loop through an enum. Values() gives an array of enum constants.

**package** enumerations;

**public** **class** LoopingThroughEnum3 {

**public** **enum** season{

***WINTER***, ***SPRING***, ***SUMMER***, ***FALL***;

}

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

//Iterating over an enum

**for(season s1:season.*values*()) {**

**System.*out*.println(s1);**

**}**

}

}

Result:

WINTER

SPRING

SUMMER

FALL

The java compiler internally adds the values() method when it creates an enum. The values() method return an array containing all the values of the enum.

**Printing the values and index:**

We use valueOf() method to print the value of an enum. To print the index we first use valueOf() method and then use ordinal() method.

**package** enumerations;

**public** **class** ValuesAndIndex4 {

**public** **enum** season{

***WINTER***, ***SPRING***, ***SUMMER***, ***FALL***;

}

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

//Printing the values

System.***out***.println(season.*valueOf*("WINTER"));

System.***out***.println(season.*valueOf*("SPRING"));

System.***out***.println(season.*valueOf*("SUMMER"));

//Printing the index

System.***out***.println(season.*valueOf*("WINTER").ordinal());

System.***out***.println(season.*valueOf*("SPRING").ordinal());

System.***out***.println(season.*valueOf*("SUMMER").ordinal());

}

}

Result:

WINTER

SPRING

SUMMER

0

1

2

***Note: Semicolon at the end of enum constants is optional.***

Only Enum With Main Method:

You can also define an Enum only with main method and execute it.

**package** enumerations;

**public** **enum** OnlyEnumWithMain5 {

***WINTER***, ***SPRING***, ***SUMMER***, ***FALL***;

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

System.***out***.println(OnlyEnumWithMain5.***WINTER***);

}

}

Result:

WINTER

**Initializing Specific values to Enum Constants:**

The enums will have initial values of 0, 1, 2, 3 etc. But we can initialize specific values to enum constants. But we should use a constructor to do it. If we don’t initialize, it will use the constants as values.

**Initializing Values to Enum Constants:**

We need to define a constructor to initialize values to enum constants.

**package** enumerations;

**public** **class** InitializingValuesToEnumConstants6 {

**enum** season{

***WINTER***(5), ***SPRING***(10), ***SUMMER***(20), ***FALL***(30);

**private** **int** value1;

**private** season(**int** value1) {

**this**.value1 = value1;

}

}

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

**for**(season s:season.*values*()) {

System.***out***.println(s+" "+s.value1);

}

}

}

Result:

WINTER 5

SPRING 10

SUMMER 20

FALL 30

**Enums in Switch Case:**

Enums are very useful in switch case statements.

**package** enumerations;

**public** **class** EnumsInSwitchCase {

**enum** season {

***WINTER***, ***SUMMER***, ***FALL***, ***SPRING***;

}

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

season s = season.***WINTER***;

**switch**(s) {

**case** ***WINTER***:

System.***out***.println("It's Winter");

**break**;

**case** ***SUMMER***:

System.***out***.println("It's Summer");

**break**;

**case** ***FALL***:

System.***out***.println("It's Fall");

**break**;

**case** ***SPRING***:

System.***out***.println("It's Spring");

**break**;

}

}

}

Result:

It's Winter

**Enums in a separate file:**

You can create enums in a separate file and access them in a different class.

**EnumInSeperateFile8.java**

**package** enumerations;

**public** **enum** EnumInSeperateFile8 {

***SPRINT***(10), ***WINTER***(20), ***SUMMER***(30), ***FALL***(40);

**public** **int** value;

**private** EnumInSeperateFile8(**int** value){

**this**.value = value;

}

}

**AccessEnumInSeperateFile8.java**

**package** enumerations;

**public** **class** AccessEnumInSeperateFile8 {

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

EnumInSeperateFile8 s = EnumInSeperateFile8.***WINTER***;

System.***out***.println(s+" "+s.value);

**for**(EnumInSeperateFile8 s1:EnumInSeperateFile8.*values*()) {

System.***out***.println(s1+" "+s1.value);

}

}

}

Interview Questions:

**What is the purpose of values() method in enum?**

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

**What is the purpose of valueOf() method in enum?**

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

**What is the purpose of ordinal() method in enum?**

The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

**What are the initial values of enums?**

0, 1, 2, 3 etc.

**What is the access type of enum constructor?**

It is private. If you don’t declare private constructor it internally creates a private constructor.

**Can we create an instance of an Enum by using new keyword?**

No. Because Enums contain private constructors only.

**Can we have abstract methods in Enums?**

Yes, we can have abstract methods in enums and can provide implementation of these methods.