

## Java Notes

Java notes by Ajit Sir...

Jspiders/Qspiders Btm

Contact : 9980500900

---

### What is a platform ?

→ Any software or hardware environment where a program runs is called a platform.

### What is a software ?

→ It is a set of instructions or a program.

→ It is an automated solution to a real world challenges or problems.

### What is a program ?

→ Program is a set of instructions or code.

### What is programming language ?

→ It is a language using which a software can be developed.

### What is Java ?

→ Java is a high level object oriented programming language.

### What are the applications that can be developed using java ?

- Desktop applications( Media player, antivirus etc)
- Web Applications (eg : Facebook, irctc.com)
- Enterprise Applications (eg : banking application)
- Mobiles
- Games etc

### Who developed Java ?

- Java was developed by James Gosling in 1996. (JDK version - 1.0) (JDK - Java Development Kit)
- The latest version of JDK is 11.
- It was developed by Sun Microsystem and now it is overtaken by Oracle corporation.

### Why the language name is Java ?

→ Java is the name of an island in Indonesia, where coffee was first produced. They have given the Java name in the name of that island.

### What are the features of Java ?

- Simple :-- it is very simple to learn and implement
- Robust - Strong
- High Performance

- Portable
- Object Oriented :
  - Everything in java is an object.
  - Few concepts of OOPs (Object Oriented Programming ) are below.
    - Class
    - Object
    - Inheritance
    - Polymorphism
    - Abstraction
    - Encapsulation
- Multi Threaded
- **Platform Independent : Why java is platform independent ?**
- Java is a platform independent, because once we write java program, we can execute it in any platform. (WORA -- Write Once, Run Anywhere). Java code will be compiled into bytecode (.class file ) by the compiler and these byte code is platform independent. Means : it can be executed in any specific platform provided we have a platform specific Java runtime environment.

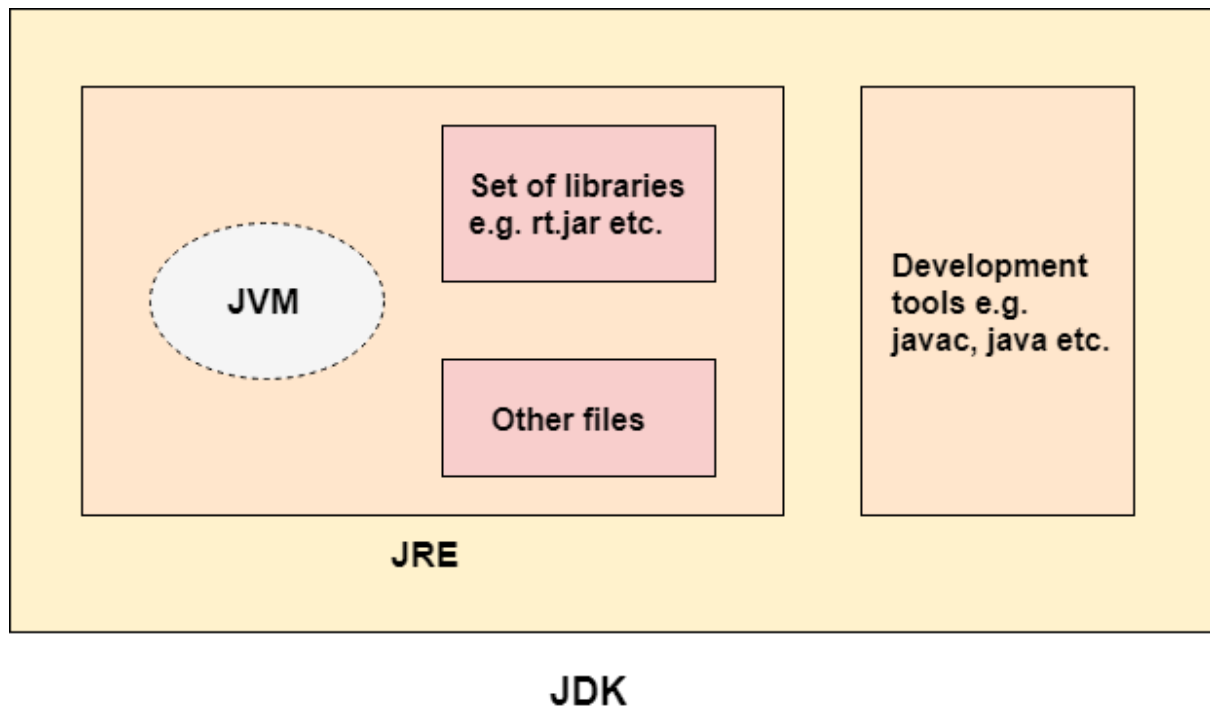
Note : JRE of windows system will be different from JRE of MAC system.

#### **Difference between C++ and Java ? No**

- Java is platform independent whereas C++ is not.
- Java does not support Multiple Inheritance with classes but C++ supports.

#### **What is JDK, JRE and JVM ?**

- JDK contains both compiler (javac) and Java RunTime environment (**JRE**).
- JDK is a java development kit which is used to develop an application.
- JRE further contains Java Virtual Machine (JVM) and a few other resource
- JRE is the physical implementation of JVM and it provides a runtime environment to execute the java programs.
- JRE is platform dependent. Windows JRE is different from MAC JRE.
- JVM stands for Java Virtual Machine. It is the specification of JRE.
- JVM is responsible to convert the byte code in to system specific instructions.



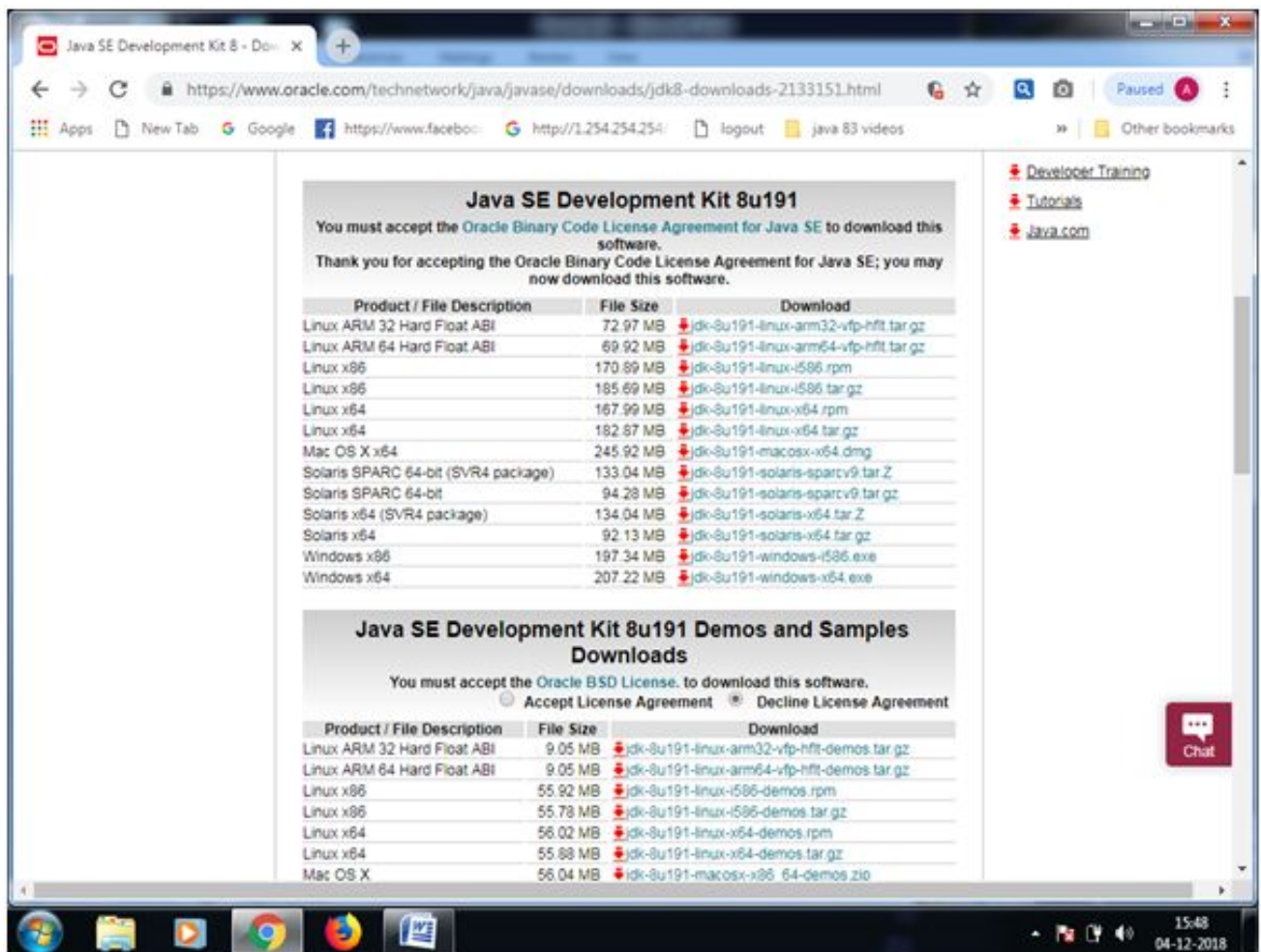
-----  
Q : How do you install Java in your system ?

#### **JDK Installation Steps : no**

1. Go to this url..

<https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

2. Select radio button -- Accept Licence Agreement



3. You can see the file downloaded in download folder.
4. Double click on the .exe file and click on Next , next till finish.
5. Verify that JDK is installed successfully in the below location

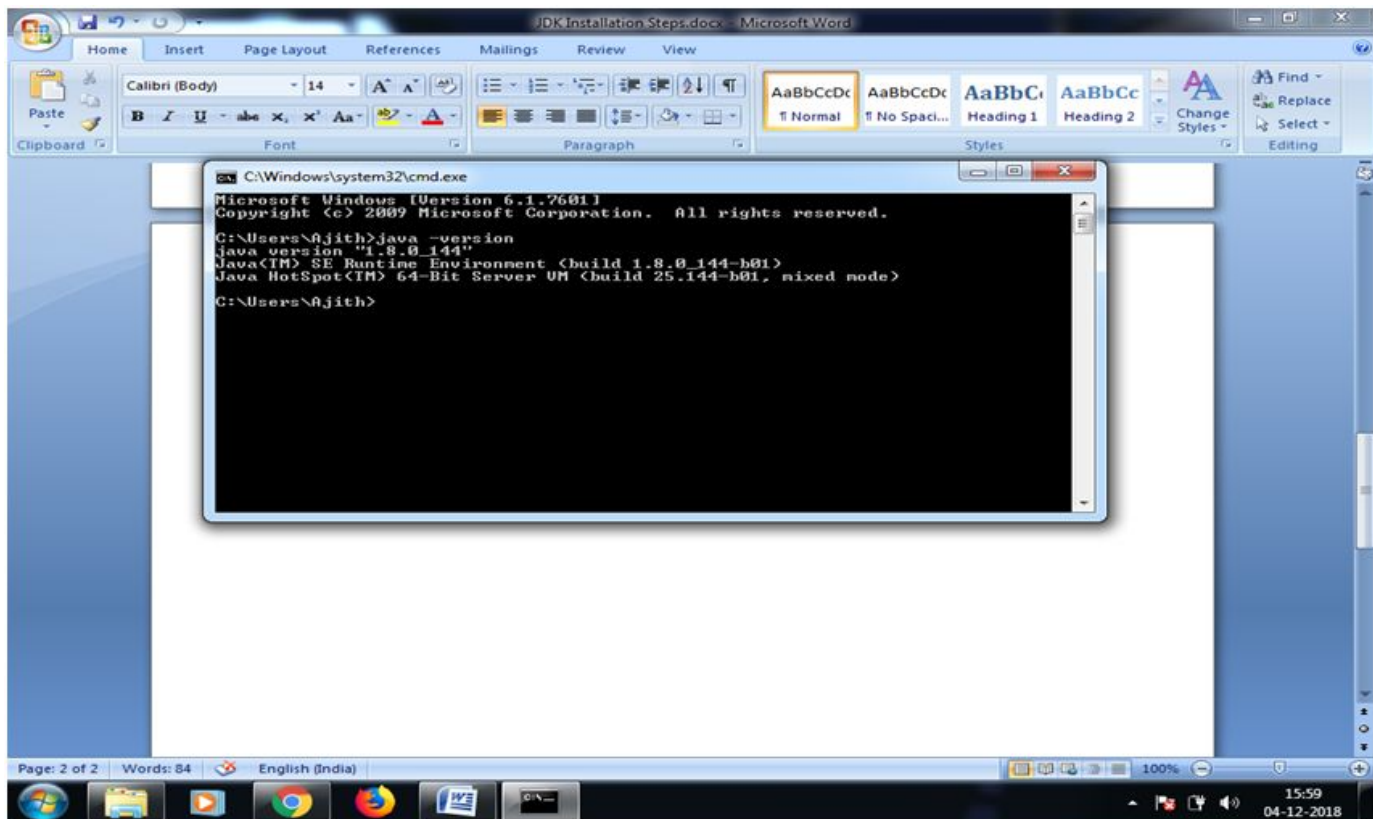
**C:\Program Files\Java\jdk1.8.0\_144\bin**

**Question : What is the shortcut to verify that JAVA is installed in your system ?**

Ans : Open command Prompt and hit this command.

**Java -version**

Check below snapshot:



Download Eclipse Editor and to launch it, set the jdk path in system environment variable.

**Question : How to set the path of JDK in System environment variable?**

Answer :

Right click on Computer à click on Properties à click on Advanced System Settings à Under Advanced Tab ( this tab is selected by default), click on Environmental Variables à Under System variable section, select path (Variable name) and click on EDIT

In Variable Value textbox, go to the end, add a semicolon and copy and paste the below path.

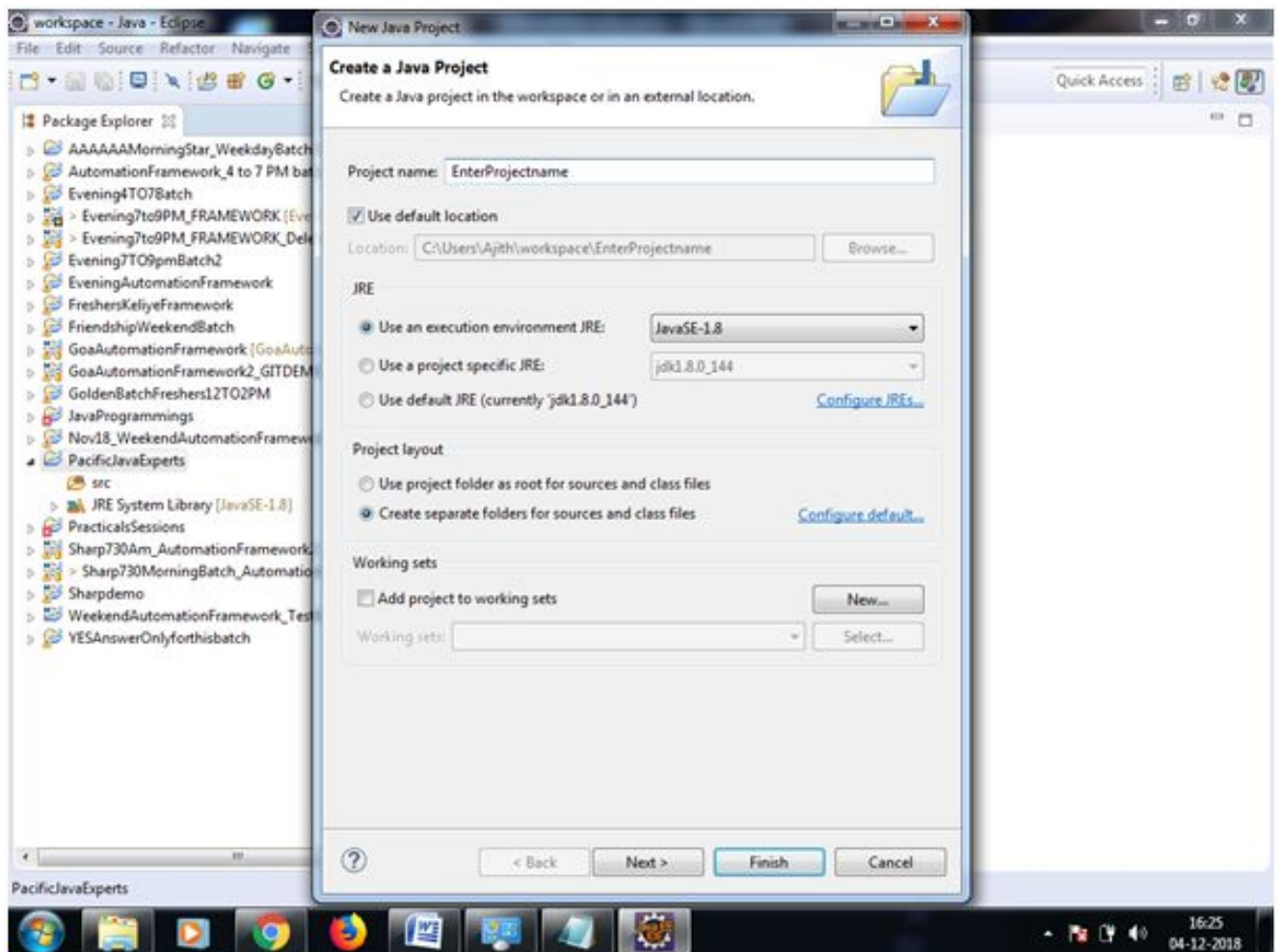
**C:\Program Files\Java\jdk1.8.0\_144\bin**

-----  
**How to create a JAVA project ?**

1. Launch eclipse

2. File → New → Java Project

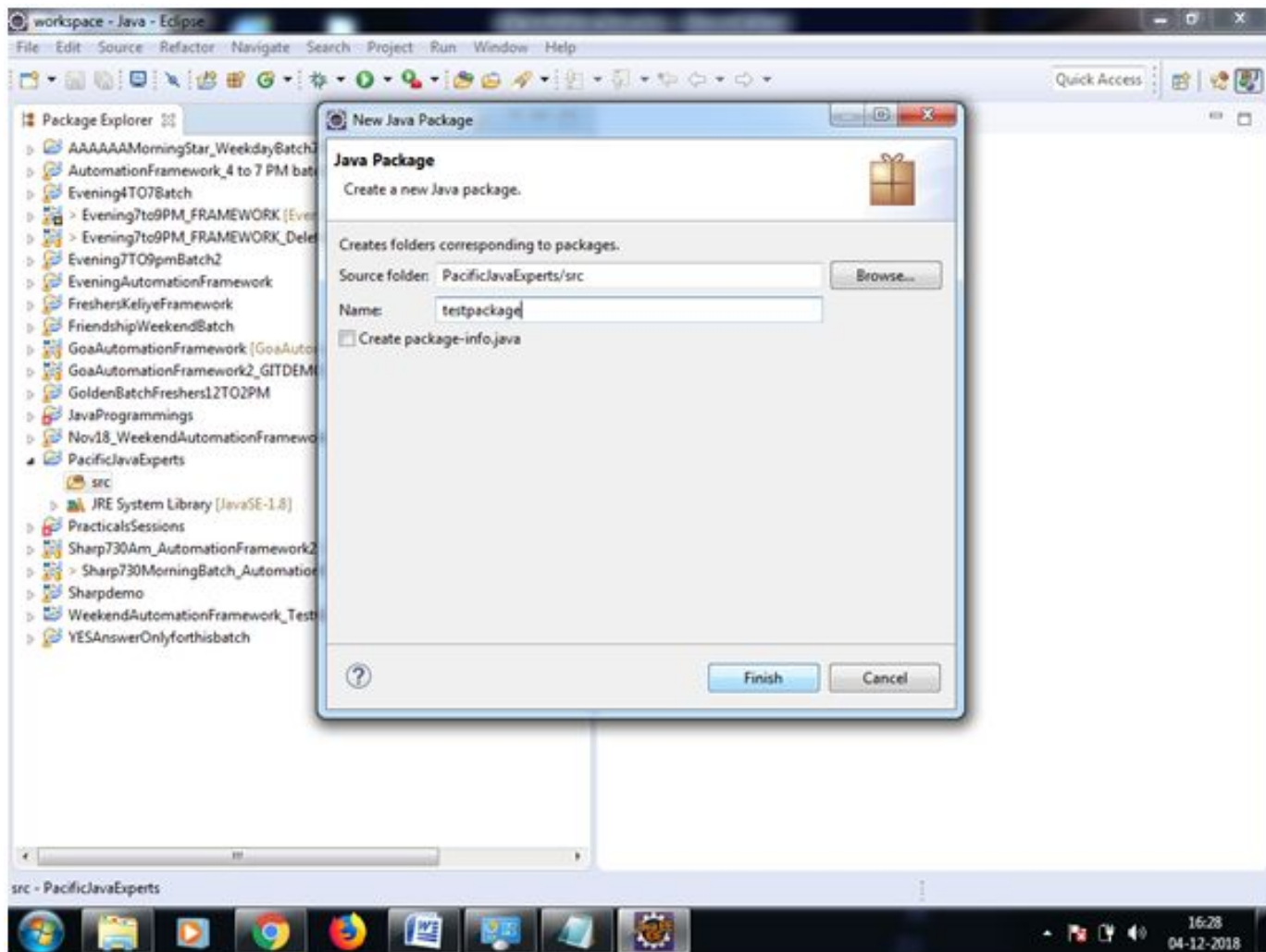
Enter a project name as shown below and click on Finish button.



How to create a package under SRC folder ?

Right click on src → new → package

Enter a package name and click on Finish as shown below.

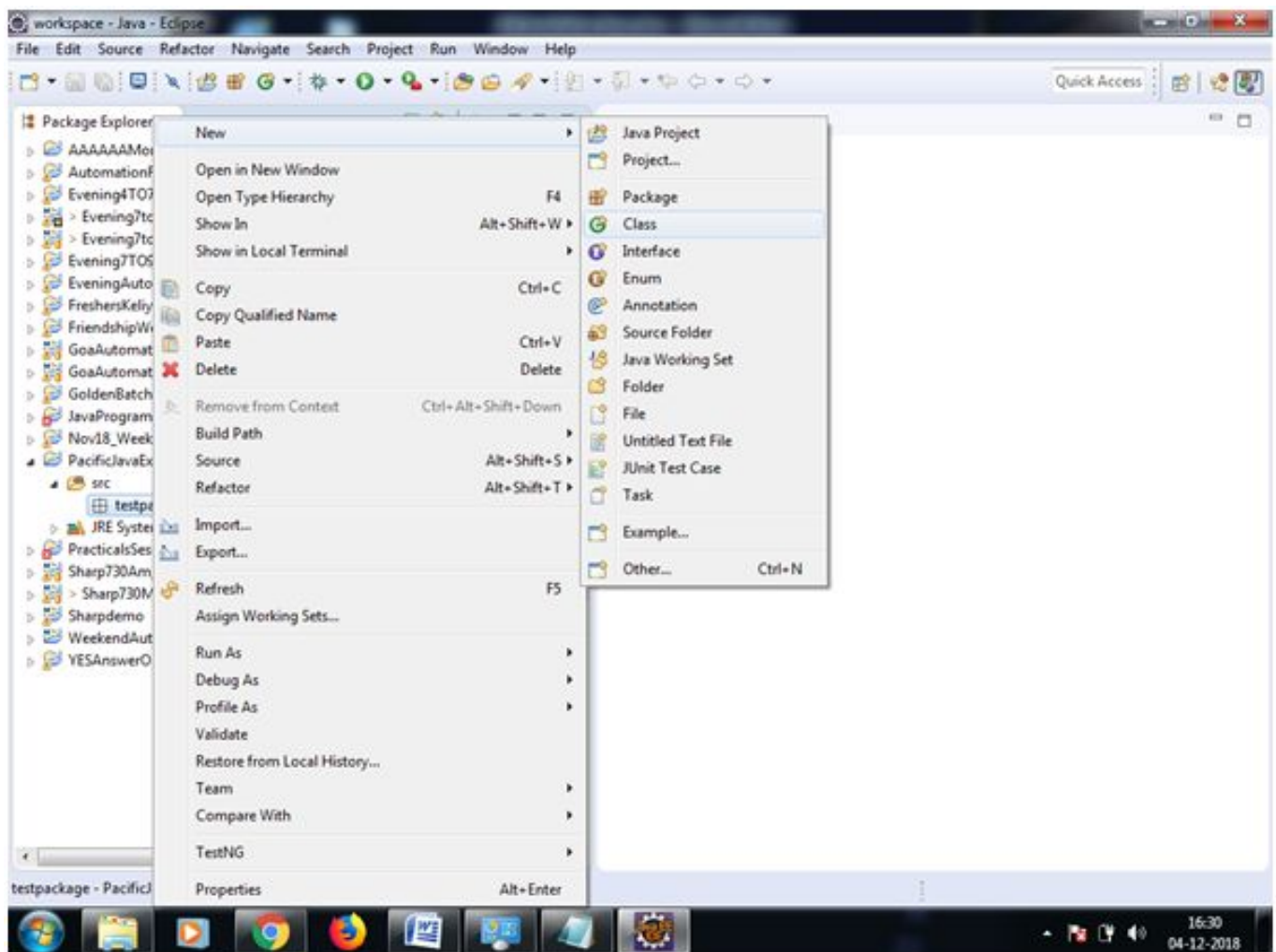


### How to create a class ?

Right Click on the package → new → class

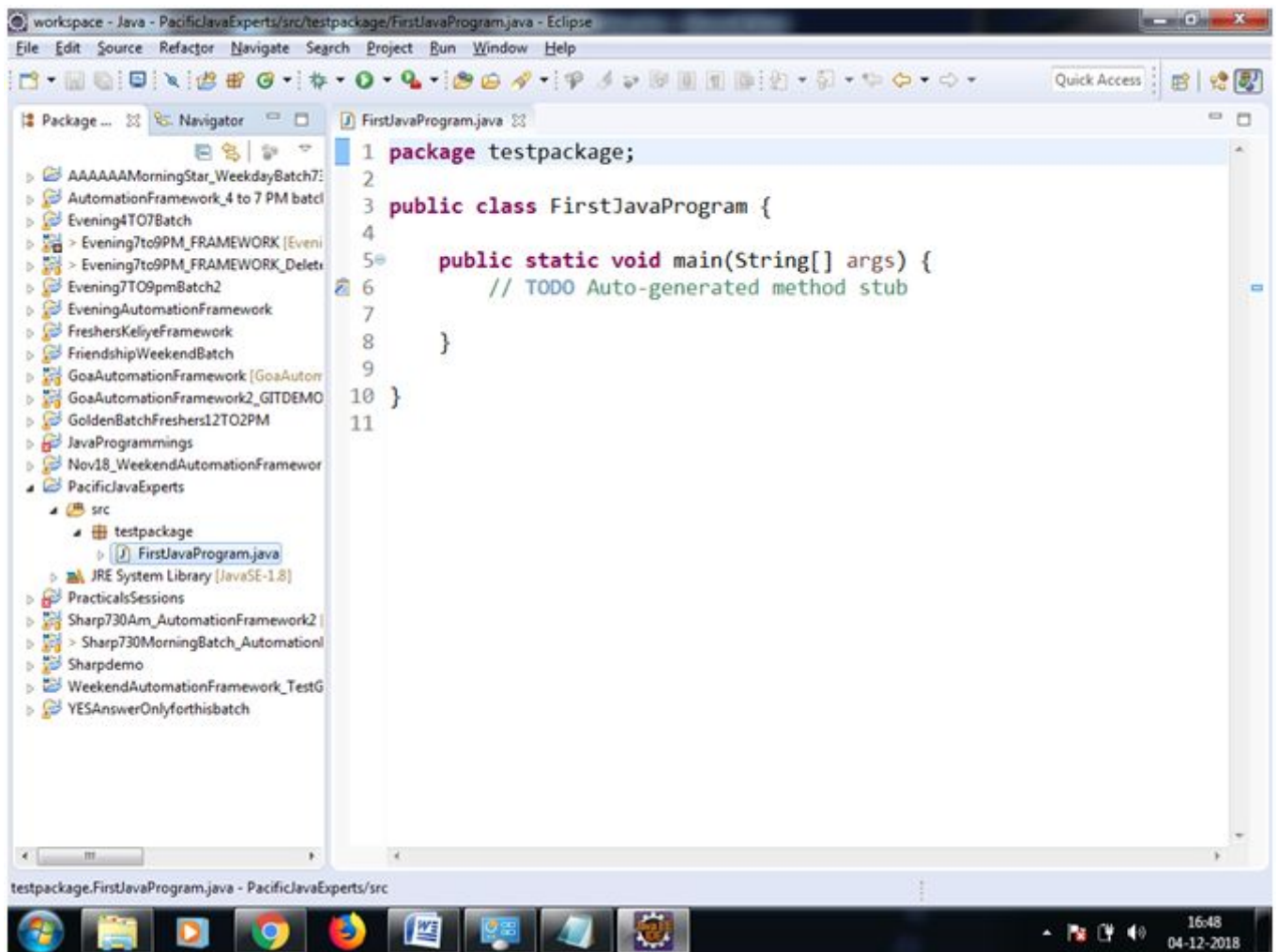
In the below window, enter a project name, select the checkbox – public static void main(Strings[] args) and click on Finish button.





It will create a class as shown below.

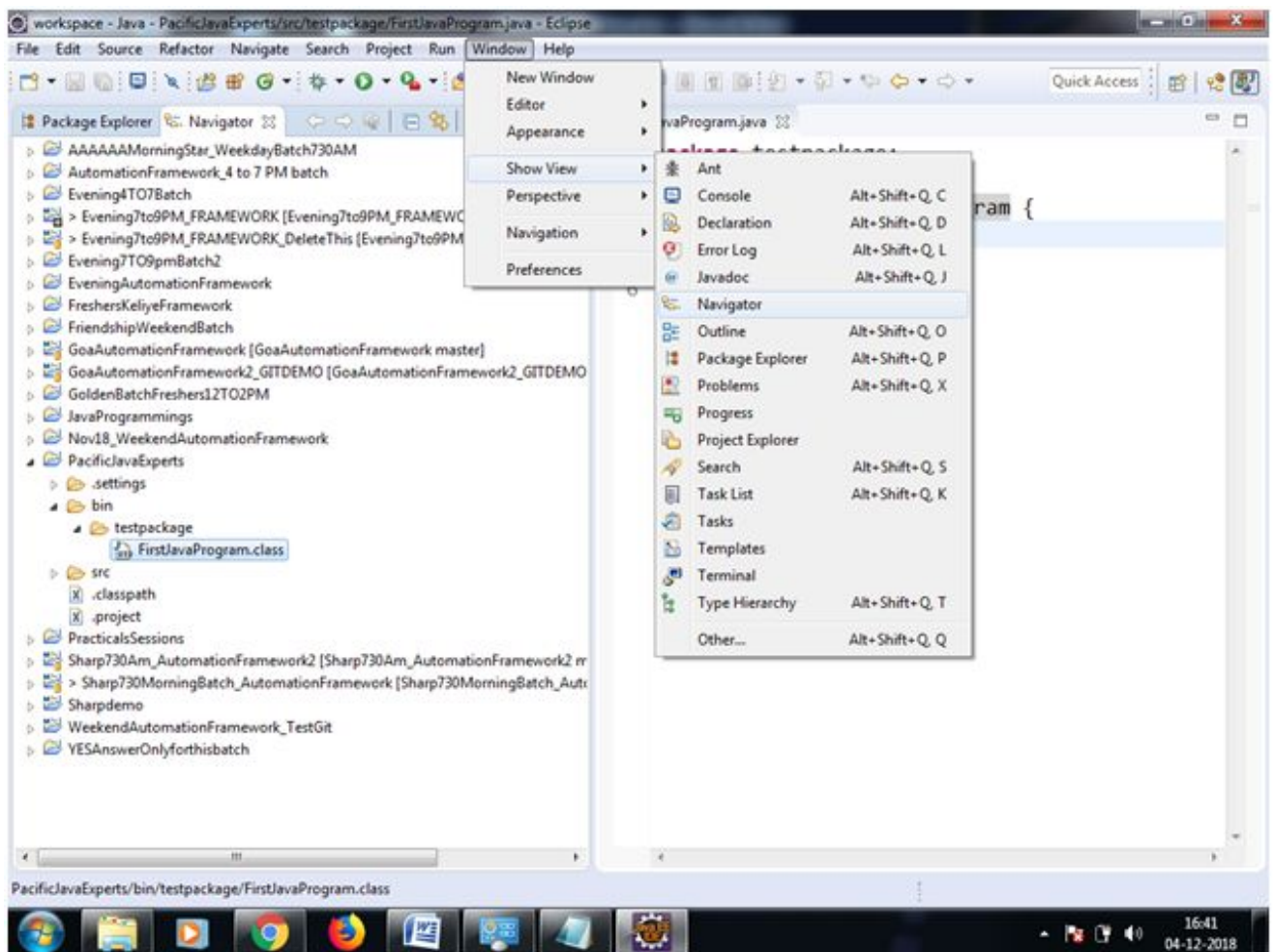




Where do we see the compiled .class file ?

Ans :

Under Navigator window pane → under the project → bin folder → package --> class file is present as shown below.



### Question : What is a Class ?

- A class is a blueprint of a plan or a design.
- It is a logical entity which can't be touched or experienced. It doesn't have any physical existence.

### Question : What is an Object ?

- Object is a physical entity which can be touched or experienced.
- Object is also called as an instance of a class.
- Object can be created out of a class.
- Multiple objects can be created out of a same class, we call these objects as identical objects or similar objects.

### Association :

→ it is a phenomenon in java where in multiple objects are associated with a single owner object.

### How many types of association do we have in java ?

Association are of 2 types:

- Composition ( is also known Strong HAS-A relationship)
- Aggregation ( is also known Weak HAS-A relationship)

### Question : What is composition in java ?

→ It is a special form of association where in the associated objects can't exists independently without the owner object. That is, if the owner object is destroyed, the associated objects will also be destroyed.

**Eg : 1. Water and the fish,**

Here, Water is the owner object, and fish is the associated object, Without water, fish can't survive.

- 2. Webpage ( Owner object ) with buttons ( associated objects).**
- 3, Heart ( associated object ) and the human body ( Owner object).**
- 4, Tree ( Owner object) and the leaves ( associated object ) ....Etc etc**

### Question : What is Aggregation in java ?

→ It is a special form of association where in the associated objects can exists independently without the owner object. That is, if the owner object is destroyed, the associated objects will still be alive.

**Eg : Shoe ( Owner object ) and a lace ( associated object ),**

**Company ( Owner object ) and the employee ( associated object )**

**Car ( Owner object ) and the driver ( associated object )etc etc..**

---

### Java Identifier :

- Identifier in java is the one which is used to identify any java member,
- Java member can be the name of the class, the name of the variable or the name of the method.

### Rules of Identifier :

- Identifier should not have any space.

```
Eg : public class Smart Phone {  
    // compilation error, class name can't have a space.  
}
```

```
class SmartPhone {  
    // No compilation error  
}
```

→ Identifier should not have any special characters except "\$" ( dollar ) and  
"\_"(underscore)

Eg :

```
public class Smart-Phone {  
    CE : // hyphen ( - ) is a special character which is not allowed  
}
```

```
class Smart_Phone {  
    // Underscore ( _ ) can be used  
}
```

3. Identifier should not start with a number.

```
Eg : public class 123SmartPhone {  
    // CE : Identifier should not start with numbers  
}
```

```
class SmartPhone123 {  
    // We can have numbers as part of the identifier but not in the beginning.  
}
```

4. Java reserved keywords should not be used as an identifier.

Eg :

```
public class int {  
    // CE : int is a reserved keyword, can't be used as an identifier.  
}  
class try {  
    // CE : try is a reserved keyword, can't be used as an identifier.  
}
```

**Interview Question :**

**Can we use a predefined class name or interface name as an identifier ?**

→ Name of a predefined class or interface in java can be used as an identifier, but we don't use because it is not a good practice. Why ??

Because,

- it creates confusion to the programmer,
- it increases the complexity of the program,
- it decreases the code readability.

**Eg :**

```
public class Collection {  
    // Collection is a predefined interface in java  
}  
class String {  
    // String is a predefined class in java  
}
```

-----  
**Keywords :**

**What are keyword in java ?**

→ Keywords in java are something that has a special meaning.

For eg :

If we want to store a numeric data, we use a keyword called "int"

If we want to store a character value, we use char keyword.

If we want to store a decimal value, we use either float or double keyword.

How many keywords we have in java ?

We have total 53 keywords in java.

→ Out of which, 3 are literals or constant value ( true, false, and null)

→ Out of remaining 50, we don't use 2 keywords in java ( goto and const)

The list of 48 keywords are as follows.

Not necessary

1. abstract -Specifies that a class or method will be implemented later, in a subclass
2. assert -Assert describes a predicate (a true-false statement) placed in a Java program to indicate that the developer thinks that the predicate is always true

at that place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort.

3. `boolean` – A data type that can hold `True` and `False` values only
4. `break` – A control statement for breaking out of loops
5. `byte` – A data type that can hold 8-bit data values
6. `case` – Used in switch statements to mark blocks of text
7. `catch` – Catches exceptions generated by try statements
8. `char` – A data type that can hold unsigned 16-bit Unicode characters
9. `class` -Declares a new class
10. `continue` -Sends control back outside a loop
11. `default` -Specifies the default block of code in a switch statement
12. `do` -Starts a do-while loop
13. `double` – A data type that can hold 64-bit floating-point numbers
14. `else` – Indicates alternative branches in an if statement
15. `enum` – A Java keyword used to declare an enumerated type. Enumerations extend the base class.
16. `extends` -Indicates that a class is derived from another class or interface
17. `final` -Indicates that a variable holds a constant value or that a method will not be overridden
18. `finally` -Indicates a block of code in a try-catch structure that will always be executed
19. `float` -A data type that holds a 32-bit floating-point number
20. `for` -Used to start a for loop
21. `if` -Tests a true/false expression and branches accordingly
22. `implements` -Specifies that a class implements an interface
23. `import` -References other classes
24. `instanceof` -Indicates whether an object is an instance of a specific class or implements an interface
25. `int` – A data type that can hold a 32-bit signed integer
26. `interface` – Declares an interface
27. `long` – A data type that holds a 64-bit integer

28. native -Specifies that a method is implemented with native (platform-specific) code
29. new – Creates new objects
30. package – Declares a Java package
31. private -An access specifier indicating that a method or variable may be accessed only in the class it's declared in
32. protected – An access specifier indicating that a method or variable may only be accessed in the class it's declared in (or a subclass of the class it's declared in or other classes in the same package)
33. public – An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible)
34. return -Sends control and possibly a return value back from a called method
35. short – A data type that can hold a 16-bit integer
36. static -Indicates that a variable or method is a class method (rather than being limited to one particular object)
37. strictfp – A Java keyword used to restrict the precision and rounding of floating point calculations to ensure portability.
38. super – Refers to a class's base class (used in a method or class constructor)
39. switch -A statement that executes code based on a test value
40. synchronized -Specifies critical sections or methods in multithreaded code
41. this -Refers to the current object in a method or constructor
42. throw – Creates an exception
43. throws -Indicates what exceptions may be thrown by a method
44. transient -Specifies that a variable is not part of an object's persistent state
45. try -Starts a block of code that will be tested for exceptions
46. void -Specifies that a method does not have a return value
47. volatile -Indicates that a variable may change asynchronously
48. while -Starts a while loop



-----  
**Question : How do you create an object of a class ?**

Ans : By using new keyword and the constructor of the class.

**Syntax :**

**<Class\_Name> objectReferenceVariable = new <class\_name>();**

**Eg :**

**//Class declaration :**

**public class Student {**

**}**

**Object Creation :**

**Student stdObj = new Student();**

Here,

**stdObj is the object reference variable.**

**What is object reference variable ?**

→ A variable which refers to an object in the heap memory is called object Reference variable.

→ **new** is a keyword, using which, we can create an object of a class.

→ **Student()** -- is the constructor of the class.

**Question: who is responsible to create objects in java and where these objects get created ?**

→ JVM is responsible to create the object in java. These objects are created in HEAP memory of RAM.

**Question : what are identical objects ?**

→ Multiple objects created out of a same class are known as identical objects.

-----  
**Java Variables :**

→ A variable is something that holds a value in the memory.

→ it refers to the name of the memory location.

→ it is always tagged with some or the other datatype.

Eg : int age = 30;

→ Here, age is the variable holding 30, and int is one of the primitive data type.

There are 3 types of variables :

→ local variable

→ instance variable

→ static variable

**Local Variable :**

- A variable which is declared inside the method body is called local variable.
- The scope of local variable is within the same method in which it is declared and initialized.
- It is recommended to initialize the local variables at the time of declaration.

### **Instance Variable :**

- A variable declared in the class and outside the method is called instance variable.
- Instance variable are specific to an instance of the class. That means, the value of instance variables will not be shared with other instance/object of the same class.
- Instance variables are not declared as static.

### **Static Variable :**

- A variable declared as static is called static variable.
- Static variable is a single copy created in memory and it can be shared by any number of instances/object of the same class.

```
Eg : public class VariableDemo
{
    Int age = 27; // instance variable
    Static String country = "India"; // Static variable
    Public void m1()
    {
        Private double bankBalance = 99999.99; // local variable
    }
}
```

---

Eg 2 :

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

        int a=10;

        int b=10;

        int c=a+b;

        System.out.println(c);

    }}

```

---

### **Java Datatypes:**

- It defines what type of data and the size of data that is stored in a variable.
- There are 2 types of DataType in java :
- Primitive datatype

→ Non primitive datatype

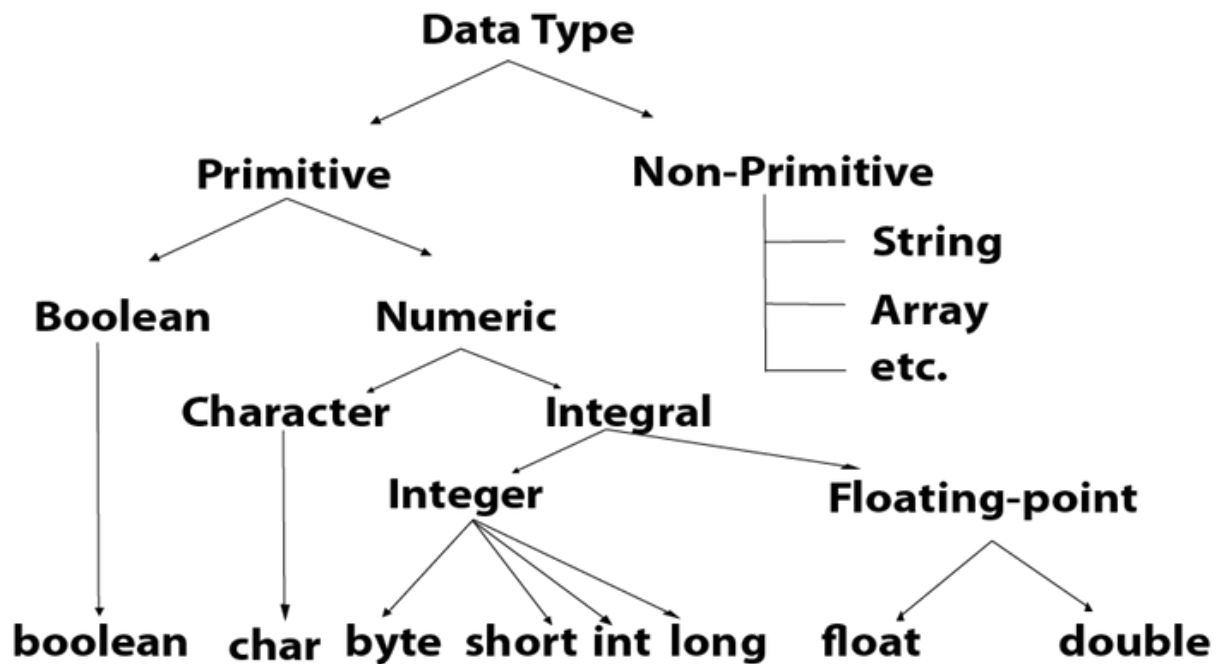
### Primitive DataType :

→ There are 8 primitive datatype in java.

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

### Non Primitive Datatype :

→ These datatypes are of Class type, interface type or array type.



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte

float	0.0f	4 byte
double	0.0d	8 byte

### **'boolean datatype :**

→ It allows only 2 values :: true and false

Eg: boolean isMarried = true;

### **'byte DataType :**

→ Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

**Example:** byte a = 10, byte b = -20

### **'Short DataType:**

→ Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

**Example:** short s = 10000, short r = -5000

### **'Int DataType :**

→ Its value-range lies between - 2,147,483,648 ( $-2^{31}$ ) to 2,147,483,647 ( $2^{31} - 1$ ) (inclusive). Its minimum value is - 2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

**Example:** int a = 100000, int b = -200000

### **'Long DataType :**

→ Its value-range lies between -9,223,372,036,854,775,808 ( $-2^{63}$ ) to 9,223,372,036,854,775,807 ( $2^{63} - 1$ ) (inclusive). Its minimum value is - 9,223,372,036,854,775,808 and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

**Example:** long a = 100000L, long b = -200000L

### **'float DataType :**

→ Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0f.

**Example:** float f1 = 234.5f

### **'double Datatype :**

→ Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

**Example:** double d1 = 12.3

### **'char datatype :**

→ Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

**Example:** char letterA = 'A'

## **Java Fundamentals**

### **1. Identifiers**

→ It refers to a name in java program.. It can be name of a class, or name of a variable or name of a method as well.

#### **Rules for defining the identifiers :**

→ Identifier should not start with digit, it should always start with characters.

→ Identifiers are case sensitive.

→ Only 2 special characters can be used in the identifier. ( \_ and \$)

→ Reserved words can't be used as identifiers.

Eg: int if = 20;

→ All java classes and interfaces name can be used as the name of the identifier, but it is not recommended, because it will decrease the readability of the code.

Eg : int String = 100; // No Compilation error

int Runnable = 200; // No Compilation error

### **Interview questions :**

**Which one are the valid java identifiers ?**

**Total\_number (valid)**

**total# ( Not valid - special character not allowed)**

**123total ( Not valid)**

**Total123 (valid)**

**ca\$h (valid)**

**\_\$\$\_\$\$\_\$ ( valid - these 2 special characters are allowed)**

**all@hands ( not valid - special character not allowed)**

**Java2share (valid)**

**Integer (valid but not recommended to use because these are class name)**

**Int (Valid)**

**int ( Not valid - keyword not allowed)**

---

### **Instance Variables:**

1. Instance Variables are the variables which should be created inside the class but outside the method, or block or constructor.
2. Instance variable gets loaded in the Heap memory once an object is created.
3. Instance variables are also known as non static variables or object level variables.
4. Instance variables are object specific. That is..If the data is different for different objects, then we declare instance variables.
5. Instance variables can't be accessed directly from static area (eg : static method). But, these variables can be accessed by using object reference variable from static area.
6. Instance variables can be directly accessed from instance area (eg - instance method)
7. If the instance variables in one object is modified, it will not impact any other object, and hence these objects are independent of each other.
8. The scope of instance variable is same as the scope of the object. That is, once the object is destroyed, the life of instance variable will also be expired.

---

### **Static Variables :**

1. Variables which are declared with static keyword are known as static variables.
2. These static variables must be created within the class but outside the method body or block or constructor.
3. Static variables get loaded in the memory (class area or static pool) once the class (.class file ) is loaded by JVM
4. Static variables are also known as class variables.
5. Static variables can be directly accessed from both static area and instance area.
6. Static variables should be accessed by using the class name. However, these variables can also be accessed by using the object reference variable, but it is not recommended.
7. If the static variable is modified, then all the referring objects will have the updated value, because, static means one single copy in the memory.

8. The scope of static variable is same as the scope of the class. That is, once the class is unloaded (--> due to JVM shutdown), the life of static variable will also be expired.
  9. If we want common data to be shared across multiple objects, then we declare these data members as static variables.
- 

### **Local Variable :**

- A variable which is declared inside the method body is called local variable.
  - The scope of local variable is within the same method in which it is declared and initialized.
  - It is recommended to initialize the local variables at the time of declaration.
- 

### **Program Eg:**

```
package testpackage;
public class Student{

    String name;
    double percentage;
    static String institue = "Jspiders";

    public static void main(String[] args) {

        System.out.println("****Create object for student 1****");
        Student studentObj1 = new Student();

        System.out.println("*****B4 assinging the value or default values****");
        System.out.println(studentObj1.name);//Null
        System.out.println(studentObj1.percentage);//0.0
        System.out.println(Student.institue);//Jspiders

        studentObj1.name = "Deepu Neeta";
        studentObj1.percentage = 36;
        System.out.println("*****After assinging the value ****");
        System.out.println(studentObj1.name);//deepu neeta
        System.out.println(studentObj1.percentage);//36.0
        System.out.println(Student.institue);//Jspiders
        System.out.println(studentObj1.institue);//Jspiders

        System.out.println("*****for arpita*****");
        Student studentObj2 = new Student();
        studentObj2.name = "Arpita";
        studentObj2.percentage = 35;
        System.out.println(studentObj2.name);//Arpita
```



```

System.out.println(studentObj2.percentage);//35.0
System.out.println(Student.institute);//Jspiders
System.out.println(studentObj2.institute);//Jspiders

System.out.println("*****changing the value of static
variables*****");
institute = "Qspiders";
System.out.println(Student.institute);//Qspiders
System.out.println(studentObj1.institute);//Qspiders
System.out.println(studentObj2.institute);//Qspiders
}
}

```

---

### Question : Explain System.out.println("welcome to java") ?

- > It is used to print a custom value on the console.
- Here, System is a predefined class in java.
- > 'out' is a static object reference variable in System class, which is pointing to an object of PrintStream class.
- > Println() is one of the non static method of PrintStream class which is actually responsible to print any custom value on the console.

---

### Array :

- It is an index based collection of similar type of data.
- Can store only homogenous data.
- Data present in array are known as elements.
- Array index starts with zero.
- Arrays are fixed, that is, when we declare an array, we must define the size.
- We can store duplicate values.
- We can also store null values.
- We can store both the address of an object and the actual object itself.

### Array Declaration :

→ Array can be declared in 2 ways.

#### 1<sup>st</sup> Approach :

```

int[] arr = new int[4]; (--> int type array)
String[] arr = new String[4]; (→ String type array)

```

### Int Array Initialisation :

```

arr[0] = 10;
arr[1] = 20;
arr[2] = 30;

```

```
'arr[3] = 40;
'arr[4] = 50; [Exception here :- ArrayIndexOutOfBoundsException]
```

### String Array Initialisation :

```
'arr[0] = "java";
'arr[1] = "python";
'arr[2] = "c#";
'arr[3] = "sql";
'arr[4] = "oracle"; [Exception here :- ArrayIndexOutOfBoundsException]
```

### 2<sup>nd</sup> Approach of Array Declaration and Initialisation:

```
'int[] arr = {10,20,30,40};
String[] arr = {"java","python","c#","sql"};
```

### Question : How do you iterate in an array object ?

--> We can iterate in an array object by using either **for loop** or **enhanced for loop**.

*Eg1 : Using for loop, we can traverse in an array in forward direction.*

```
'for(int i = 0; i < arr.length; i++)
{
    System.out.println(arr[i]);
}
```

---

*Eg2 : Using for loop, we can also traverse in an array in backward direction.*

```
'for(int i = arr.length-1; i >= 0; i--)
{
    System.out.println(arr[i]);
}
```

---

*Eg3 : Using for loop, we can traverse partially in an array.*

```
'for(int i = 3; i < arr.length-5; i++)
{
    System.out.println(arr[i]);
}
```

---

### Array iteration using Enhanced for loop :

Eg : Using enhanced for loop, we can traverse in an array **ONLY** in forward direction. Enhanced for loop doesn't support backward and partial search.

```
for(int i : arr)
{
    System.out.println(i);
}
```

//Here,  
arr --> is the array object reference,  
int → denotes what type of data does the array object holds  
i --> is a variable

---

## **Method :**

### **What is a method ?**

Ans :

- A method is a set of instructions or block of code which performs a specific task.
  - Method can return either primitive data or non primitive data.
  - Method can't return multiple primitive data types (eg : method can't return both int and String at the same time).
  - But, It can return collection of homogeneous data type. ( eg : int[] array or String[] array etc..)
- 

### **Demo program : Method returning a primitive data type**

```
public class CalculatorDemo {  
    //method to add 2 numbers  
    double add(double a, double b){  
        double c = a + b;  
        return c;  
    }  
    public static void main(String[] args) {  
        CalculatorDemo calc = new CalculatorDemo();  
        double res = calc.add(10.10, 20.20);  
        System.out.println("The addition of 2 number is : "+res);  
    }  
}
```

---

### **Demo program : Method returning a non primitive data type [ a class type data]**

```
public class Water {  
    void drink(){  
        System.out.println("drink water..and stay happy..");  
    }  
}  
  
public class Tap {  
    public Water openTap(){  
        Water w = new Water();  
        return w;  
    }  
}
```

```

}
public class Person {
    public static void main(String[] args) {
        //Create an object of Tap class and call openTab() method.
        Tap t = new Tap();
        Water openTap = t.openTap();
        openTap.drink();
    }
}

```

---

### **Demo Program : Method returning non primitive Array object**

**Eg1 : A shop selling multi color pencils.[method returning an array of pencils..]**

**Program :**

**Pencil.java :**

```

public class Pencil {
    public String color;
    void write(){
        System.out.println("pencil writes in " + color);
    }
}

```

**Shop.java**

```

public class Shop {
    Pencil[] sell(){
        //Creating an object of Pencil with Red color
        Pencil p1 = new Pencil();
        p1.color = "Red";
        //Creating an object of Pencil with Blue color
        Pencil p2 = new Pencil();
        p2.color = "Blue";
        //Creating an object of Pencil with Black color
        Pencil p3 = new Pencil();
        p3.color = "Black";
        //Creating an Pencil array and storing the address of pencil objects
        Pencil[] pArr = {p1,p2,p3};
        //returning the pencil array object reference
        return pArr;
    }
}

```

**Kid.java**

```

public class Kid {

```

```

public static void main(String[] args) {
    //Kids goes to the shop and buy pencils - create an shop object here.
    Shop s = new Shop();
    Pencil[] allPencils = s.sell();
    //use Enhanced foreach loop to iterate with in the array elements.
    for (Pencil pencil : allPencils) {
        pencil.write();
    }
}

```

#### Output :

```

pencil writes in Red color
pencil writes in Blue color
pencil writes in Black color

```

---

**Eg2 : A shop selling multi color flowers[method returning an array of flowers..]**

Program :

```

/*
Do this as an assignment
*/

```

---

#### Method Syntax :

```

<Access_modifier> <Return_Type> methodName(parameter1, parameter2,...)
{
    //code -- method body or implementation or definition or logic
}

```

---

#### Why we use methods ?

Ans :

- To achieve code reusability
- For code optimisation

#### What is Method invocation or method call ?

Ans : Once we declare a method, it will not get executed on its own. We need to explicitly call or invoke the method, this process is known as method invocation or method call.

---

#### Program Demo : Initialisation of instance variables using methods

```

public class Account {
    int acc_no;

```

```

String accHolderName;
double amount;
void openAccount(int i, String s, double d){
    acc_no = i;
    accHolderName = s;
    amount = d;
}
void displayAccountInfo(){
    System.out.println("Account is created below");
    System.out.println("Account number --> "+acc_no + " \n " + "Account holder
Name -->" + accHolderName+ " \n " + "Initial Amount -->" + amount);
}
void checkBalance(){
    System.out.println("Available balance is : "+amount);
}
void deposit(double depositAmt){
    amount = amount + depositAmt;
    System.out.println("Amount Deposited : " + depositAmt);
}
void withdraw(double withdrawAmt){
    if (withdrawAmt > amount) {
        System.out.println("Insufficient balance...");
    } else {
        amount = amount - withdrawAmt;
        System.out.println("Amount Withdrawn : " + withdrawAmt);
    }
}
public static void main(String[] args) {
    Account acc = new Account();
    acc.openAccount(111, "Smrithi", 1000);
    acc.displayAccountInfo();
    acc.checkBalance();
    acc.deposit(9000);
    acc.checkBalance();
    acc.withdraw(10000);
    acc.checkBalance();
}
}

```

### Output :

*Account is created below*

*Account number --> 111*

Account holder Name -->Smrithi

Initial Amount -->1000.0

Available balance is : 1000.0

Amount Deposited :9000.0

Available balance is : 10000.0

Amount Withdrawn : 10000.0

Available balance is : 0.0

---

### **Conditional statement :**

If we want to execute a set of statement based on a condition, then we go for conditional statements.

### **If -else if :--**

Problem Statement :

- If the age of a person is less than 18 -- treat him as a kid
- If the age of a person is (greater than or equal to 18 ) AND (less than or equal to 59) -- treat him as a Adult
- If the age of a person is (greater than or equal to 60) AND (less than or equal to 120) -- treat him as a Senior Citizen
- If the age of person is greater than 120, he is expired.

```
public class If_Elself_Example {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter the age..");  
        int age = sc.nextInt();  
        if (age<18) {  
            System.out.println("Kid's Age is :--> "+age);  
        } else if(age >=18 && age<=59)  
        {  
            System.out.println("Adult age is :--> " + age);  
        }else if(age>=60 && age <=120){  
            System.out.println("Senior Citizen age is :-->"+age);  
        } else{  
            System.out.println("He is expired...");  
        }  
    }  
}
```

---

### **Problem Statement :**



Print the month based on user input.

eg : 1 -- january, 2 -- february .. 12 -- dec, greater than 12 - month doesn't exists

using if-elseif :

```
public class If_Elself_Example {  
    public static void main(String[] args) {  
        //Alt + Shift + R -- short cut to rename a variable at multiple places  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter the month..");  
        int month = sc.nextInt();  
        if (month==1) {  
            System.out.println(month + " --> January");  
        } else if(month==2)  
        {  
            System.out.println(month + " --> February");  
        }else if(month==3){  
            System.out.println(month + " --> March");  
        } else if(month==4)  
        {  
            System.out.println(month + " --> Apr");  
        }else if(month==5){  
            System.out.println(month + " --> May");  
        } else if(month==6)  
        {  
            System.out.println(month + " --> June");  
        }else if(month==7){  
            System.out.println(month + " --> July");  
        } else if(month==8)  
        {  
            System.out.println(month + " --> August");  
        }else if(month==9){  
            System.out.println(month + " --> Sept");  
        } else if(month==10)  
        {  
            System.out.println(month + " --> Oct");  
        }else if(month==11){  
            System.out.println(month + " --> Nov");  
        } else if(month==12)  
        {  
            System.out.println(month + " --> Dec");  
        }else{  
            System.out.println("Month doesn't exists..");  
        }  
    }  
}
```

```
    }  
}  
}
```

### Drawback of If-else-if ladder :

→ If we deal with lots of conditions, if else if ladder is not recommended, because it doesn't provide code readability. So, in this case, we should go for switch case which provides code readability.

---

### using switch- case :

```
public class SwitchCaseExample {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter month number.. 1 -- Jan , 2 -- Feb...so on");  
        int monthNum = sc.nextInt();//CTRL + 1 + ENTER  
        switch(monthNum){  
            case 1:  
                System.out.println(monthNum + " --> January");  
                break;  
            case 2:  
                System.out.println(monthNum + "--> Feb");  
                break;  
            case 3:  
                System.out.println(monthNum + "--> Mar");  
                break;  
            case 4:  
                System.out.println(monthNum + "--> Apr");  
                break;  
            case 5:  
                System.out.println(monthNum + "--> May");  
                break;  
            case 6:  
                System.out.println(monthNum + "--> June");  
                break;  
            case 7:  
                System.out.println(monthNum + "--> July");  
                break;  
            case 8:  
                System.out.println(monthNum + "--> Aug");  
                break;  
            case 9:
```

```

        System.out.println(monthNum + "--> Sept");
        break;
    case 10:
        System.out.println(monthNum + "--> Oct");
        break;
    case 11:
        System.out.println(monthNum + "--> Nov");
        break;
    case 12:
        System.out.println(monthNum + "--> Dec");
        break;
    default:
        System.out.println("Month doesn't exists.. enter a valid month");
        break;
    }
}
}

```

---

## **While loop:**

- when we want a set of statement to be executed multiple times based on a condition, we use while loop.
- If the condition is true, then only the set of statements inside the while loop will be executed.
- Control will come out of the while loop, if the condition turned out to be false.
- If the condition is never true, it will never execute the while loop statements.

## **Why we go for loop ?**

- to optimize the code
- to increase the code readability.
- to decrease the code complexity

## **Syntax :**

```

while(condition)
{
    //code
}

```

## **Program example1 :**

**Problem statement : Print 1 to 10 number.**

```

public class WhileLoopDemo {
    public static void main(String[] args) {

```

```

        int x = 1;
        while(x <= 10){
            System.out.println(x);
            x++;
        }
    }
}

```

---

### Program example2 :

**Problem statement :** Iterate using while loop for 10 times, if the number is 5, come out of the loop.

```

public class WhileLoopDemo {
    public static void main(String[] args) {
        int x = 1;
        while(x <= 10){
            if (x == 5) {
                System.out.println(x);
                break;
            }
            x++;
            System.out.println("****inside the while loop****");
        }
        System.out.println("****outside the while loop****");
    }
}

```

---

### do while loop :

→ Whether the condition in do while loop is true or false, it will execute the statements atleast once.

→ it will execute the statements till the condition is true. and will come out of the loop, once the condition becomes false.

---

-

### Syntax :

```

do
{
    // code or statements
}while(condition);

```

---

-

### Program Example 1:

```
public class DoWhileLoopDemo {  
    public static void main(String[] args) {  
        int x = 1;  
        do {  
            System.out.println(x);//1  
            x = x + 1;  
        } while (x > 10);  
    }  
}
```

O/p == 1

---

### Program Example 2 :

```
public class DoWhileLoopDemo {  
    public static void main(String[] args) {  
        int x = 1;  
        do {  
            System.out.print(x + " ");  
            x = x + 1;  
        } while (x <= 10);  
    }  
}
```

O/p --> 1 2 3 4 5 6 7 8 9 10

---

### Question : When do we go for while loop and do while loop ?

→ We go for while loop, when we want to execute a set of statements only and only if the condition is true.

→ And, we go for do while loop, if we want to execute a set of statements atleast once, irrespective of the condition being true or false.

---

### What is method overloading ?

The concept of having multiple methods in a class with the same name but different in Parameters is called method overloading.

**OR**

Performing similar tasks with different inputs is called method overloading.

---

### Rules of method overloading :-

1. The method name must be same.
  2. The number of parameters must be different.
  3. The datatype of parameters must be different.
  4. The sequence of parameters must be different
- 

**Question : Does the method return type play any role in Method Overloading ?**

No. Method return type doesn't play any role in method overloading. It can be same or different.

**Question : What is method signature ?**

Ans : Method signature includes method name and parameter list.

**Question : Why method overloading ?**

Ans:

1. To reduce the complexity of the program.
2. To increase the readability of the program.
3. To achieve compile time polymorphism.

**Question : What is compile time polymorphism ?**

→ The process of compiler taking a decision as to which method implementation to be executed during compile time is called compile time polymorphism.

→ It is also known as Early binding or static binding.

**Question: Why compile time polymorphism is known as early binding ?**

Ans :

Because compiler binds a method call with a method definition at an early stage right before the program execution, and hence, we call compile time polymorphism as early binding.

**Example 1:**

```
public class MethodOverloadingDemo {  
    void display(int i){  
        System.out.println(i);  
    }  
    void display(String s){  
        System.out.println(s);  
    }  
    void display(double d){  
        System.out.println(d);  
    }  
    void display(boolean b){
```

```

        System.out.println(b);
    }
    public static void main(String[] args) {
        MethodOverloadingDemo m = new MethodOverloadingDemo();
        m.display(10.50);
        m.display(10);
        m.display("java");
        m.display(true);
    }
}

```

#### Output

10.5

10

java

true

#### Example 2 :

```

public class MethodOverloadingDemo {
    void add(int a, int b){
        System.out.println("Sum of 2 int values --> " +(a+ b));
    }
    void add(double a, double b){
        System.out.println("Sum of 2 double values --> " +(a + b));
    }
    void add(int a, double b){
        System.out.println("Sum of 1 int and 1 double value --> " +(a+ b));
    }
    void add(double a, int b){
        System.out.println("Sum of 1 double and 1 int value --> " +(a + b));
    }
    public static void main(String[] args) {
        MethodOverloadingDemo m = new MethodOverloadingDemo();
        m.add(10, 20);
        m.add(10.10, 20.20);
        m.add(10, 10.10);
        m.add(10.10, 20);
    }
}

```

#### Output :

Sum of 2 int values --> 30

Sum of 2 double values --> 30.299999999999997

Sum of 1 int and 1 double value --> 20.1



*Sum of 1 double and 1 int value --> 30.1*

---

-

### **Type Promotion : (In case of primitive data type)**

The process of automatic conversion of lower data type in to higher data type by the compiler is called type promotion.

### **Type promotion - program example :**

```
public class TypePromotionExample {  
    void add(double a, double b){  
        System.out.println("Type promotion by the compiler --> " + (a + b));  
    }  
    public static void main(String[] args) {  
        TypePromotionExample m = new TypePromotionExample();  
        m.add(10, 20);  
    }  
}
```

### **Output:**

*Type promotion by the compiler --> 30.0*

---

### **Type casting (In case of primitive data type)**

The process of conversion of higher data type in to lower data type explicitly by the user is called type casting.

### **Program example:**

```
public class TypeCastingExample {  
    public static void main(String[] args) {  
        int a = (int) 10.10;  
        System.out.println(a);  
    }  
}
```

### **Output**

*10*

---

### **Question : What is method dispatch or method resolution ?**

Ans :

The process of binding a method call with a method definition by the compiler is called method dispatch or method resolution.

---

### **What is a constructor ?**

A constructor is a special method which is used to initialise the instance variables at the time of object creation.

**Question : When does the constructor gets called ?**

Ans : Constructor gets called at the time of object creation.

**Constructor Syntax :**

```
<Class_Name>()  
{  
}
```

- Constructor name must be same as the class name.
- Constructor doesn't have a return type, not even void.

**Question : How many type of constructor we have ?**

There are 2 types of constructors.

- Default Constructor
- Custom Constructor

**Question : Explain Default Constructor ?**

Ans:

- When user don't define any constructor, compiler provides a default constructor.
- Default constructor is used to initialize the instance variable to its default values based on the data type.
- Default constructor will not have any parameter. It is a zero-param constructor or no-arg Constructor.
- The access modifier of default constructor is same as that of the class.

**Question : Explain Custom Constructor ?**

Ans:

- User defined constructor is also known as custom constructor.
- Custom constructor can be either zero-param constructor or parameterised constructor.
- Custom constructor is used to initialize the instance variable to user defined values.
- The access modifier of custom constructor can be either public or protected or default or Private.

**Question : What is a zero-param constructor or no-arg constructor ?**

Ans:

A constructor with no parameter is called a zero-param constructor.

**Question : What is a parameterised constructor ?**

Ans:

An user defined constructor with any number of parameters is called a parameterised constructor.

**Question : What does a constructor returns ?**

Ans : It returns an instance of the same class. (However, it does not have any return type.)

**Question : What is constructor overloading ?**

The concept of having more than one constructor in a class with different parameter list is called constructor overloading.

**Question : Why constructor overloading ?**

It provides a flexibility to create different objects based on the requirement.

**Rules for constructor Overloading :**

1. Constructor name must be same as the class name.
2. Number of parameters must be different.
3. Data type of the parameters must also be different.
4. Sequence of parameters must be different.

**Example of Overloaded Constructor :**

Program :

```
public class Car {
    String brand;
    double price;
    int model;
    Car(){
    }
    Car(String b, double p){
        brand = b;
        price = p;
    }
    Car(String b, double p, int m) {
        brand = b;
        price = p;
        model = m;
    }
    public void display(){
        System.out.println(brand + " : " + price + " : " + model);
    }
    public static void main(String[] args) {
        Car c1 = new Car("Audi", 9999999.99);
        Car c2 = new Car();
        Car c3 = new Car("Rolls Royce", 999, 2018);
    }
}
```

```

        c1.display();// Audi : 9999999.99
        c2.display();//null :0.0
        c3.display();//"Rolls Royce", 999999999999999999.99, 2018
    }
}

```

#### Output :

*Audi : 9999999.99 :0*

*null : 0.0 :0*

*Rolls Royce : 999.0 :2018*

## Inheritance :

### Question : What is inheritance ?

Ans:

- Inheritance is the process of acquiring all the properties and behaviours of one class in to another class.
- Inheritance is also known as IS-A relationship
- We can achieve inheritance by using extends keyword.

### Question : What is a parent class ?

- Class with all generic or common properties or behaviours is called parent class.
- Parent class is also known as Super class or Base class or Generic class.

### Question : What is a child class ?

- A class that extends a parent class is called child class.
- Child class is also known as Sub class or Derived class or Extended class.

### Question : Why inheritance ?

- To achieve code reusability.
- to achieve generalisation.
- to achieve runtime polymorphism.

### What are the different types of inheritance ?

There are 5 types of inheritance.

- *Single inheritance [Simple inheritance]*
- *Multi Level Inheritance*
- *Hierarchical inheritance*
- *Multiple Inheritance*

→ Hybrid Inheritance

### → Single inheritance [Simple inheritance]

These is a type of inheritance where in one class inherits all the properties of another class.

Eg :

1. Class System extends Object
2. Class String extends Object

### → MultiLevel Inheritance

This is a type of inheritance where in one class acts as both subclass and superclass.

Eg :

Object

| extends

Number

| extends

Integer

Here, ( Number class is sub class of Object class and at the same time, it is a super class to Integer class)

### → Hierarchical inheritance

This is a type of inheritance where in one super class is inherited by multiple subclasses.

Eg :

Object

| extends | extends

Number String

OR

Number

| extends

Integer Double Float

### → Multiple Inheritance

This s a type of inheritance where in one sub class inherits more than one super class.

[Multiple inheritance is not possible in case of class]

Question : Why multiple inheritance in case of class is not possible in java ?

Ans:

1. Because of ambiguity in method invocation between 2 super classes.
2. And, it also creates an ambiguity as to which super class constructor to be invoked.

### → Hybrid Inheritance

Hybrid inheritance is a combination of 2 or more types of inheritance.

[eg : Single inheritance + multilevel inheritance + hierarchical inheritance etc]

---

### What is Run time polymorphism ?

Ans:

The process of JVM taking a decision as to which method implementation to be executed at run time based on the object created is called run time polymorphism.

Program :

#### Pen.java [Parent class]

```
public class Pen {  
    void write(){  
        System.out.println("Enjoy writing with Pen...");  
    }  
}
```

---

-

#### InkPen.java [Child Class]

```
public class InkPen extends Pen{  
    @Override  
    void write() {  
        System.out.println("writing with Ink Pen..");  
    }  
}
```

---

-

#### MarkerPen.java[Child Class]

```
public class MarkerPen extends Pen{  
    @Override  
    void write() {  
        System.out.println("writing with Marker Pen..");  
    }  
}
```

---

-

#### SketchPen.java[Child Class]

```

public class SketchPen extends Pen{
    @Override
    void write() {
        System.out.println("writing with Sketch Pen..");
    }
}

```

---

### PenFactory.java

```

public class PenFactory {
    public static Pen getPen(String penType){
        Pen p = null;
        if (penType.equalsIgnoreCase("ink")) {
            p = new InkPen();
            System.out.println("ink pen block");
        } else if(penType.equalsIgnoreCase("sketch")){
            p = new SketchPen();
            System.out.println("sketch pen block");
        } else if(penType.equalsIgnoreCase("marker")){
            p = new MarkerPen();
            System.out.println("marker pen block");
        }
        return p;
    }
}

```

---

### Customer.java

```

public class Customer {
    public static void main(String[] args) {
        System.out.println("Enter the type of Pen which you want {'ink','sketch',
        'marker'}");
        Scanner sc = new Scanner(System.in);
        String penType = sc.next();
        System.out.println("Customer want this type of pen :--> "+penType);
        Pen penObj = PenFactory.getPen(penType);
        if (penObj != null) {
            penObj.write();
        } else {
            System.out.println("We don't sell this type of pen..");
        }
    }
}

```

```
}
```

### Output :

```
Enter the type of Pen which you want {'ink','sketch', 'marker'}
```

```
ink
```

```
Customer want this type of pen :--> ink
```

```
ink pen block
```

```
Enjoy writting...
```

---

### What is Upcasting ?

Ans :

It is a mechanism where in a superclass reference refers to a sub class object to achieve generalisation.

OR

Creating an object of sub class and referring it by super class reference variable is called upcasting.

### Program example :

#### Create a super class - Fruit.java

```
public class Fruit {  
    void eat(){  
        System.out.println("Eat fruit - and stay healthy..");  
    }  
}
```

#### Create a sub class - Apple.java

```
public class Apple extends Fruit{  
    @Override  
    void eat() {  
        System.out.println("One apple a day - keep the doctor away..");  
    }  
}
```

#### Create another sub class - Banana.java

```
public class Banana extends Fruit{  
    @Override  
    void eat() {  
        System.out.println("One banana a day - keep your muscles healthy");  
    }  
}
```

#### Customer.java

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



```

        Fruit f = new Apple();// Super class [Fruit] reference pointing to sub class
object[Apple]
        f.eat();//One apple a day - keeps the doctor away..
        f = new Banana();// Super class [Fruit] reference pointing to sub class
object[Banana]
        f.eat();//One banana a day - keeps the muscles healthy
    }
}

```

### Output :

*One apple a day - keep the doctor away..*

*One banana a day - keep your muscles healthy*

---

### What is method overriding ?

Ans :

Method overriding is the process of providing specific implementation to sub class method.

### Rules of method overriding ?

- Method overriding is possible only in case of inheritance.
- Method name must be same in both super and sub class.
- The number of parameters must be same.
- The data type of parameters must be same
- The sequence of parameters must be same.
- Method return type must be same in case of primitive data. But in case of non primitive data type, method return type can be changed.

### What is overridden method and overriding method ?

Super class method is overridden by subclass method, hence we call super class method as overridden method.

Sub class method overrides super class method, hence we call subclass method as overriding method.

### Method Overriding - Program example :

#### Create Base class - Bank.java

```

public class Bank {
    void getRateOfInterest(){
        System.out.println("interest free .. ");
    }
}

```

#### Create a Derived class - Citi.java

```

public class Citi extends Bank{
    @Override // Providing specific implementation to sub class method

```

```

        void getRateOfInterest() {
            System.out.println("Rate of interest is 15.75%");
        }
    }
}

```

#### Create a Derived class - Sbi.java

```

public class SBI extends Bank {
    @Override // Providing specific implementation to sub class method
    void getRateOfInterest() {
        System.out.println("Rate of interest is 8.8%");
    }
}

```

#### Customer goes to a bank and apply for a loan - TakeLoan.java

```

public class TakeLoan {
    public static void main(String[] args) {
        Bank b = new Citi();
        b.getRateOfInterest();
        b = new SBI();
        b.getRateOfInterest();
    }
}

```

#### Output :

*Citi bank - Rate of interest is 15.75%*

*SBI bank - Rate of interest is 8.8%*

#### Question : What is instanceof keyword in java ?

It checks whether an object reference is of the specified class or interface type. If it is of the same type, it returns true and if it is not of the same type, it returns false.

```
Dog d = new Dog();
```

```
Cat c = new Cat();
```

```
SOP(d instanceof Dog); // output is TRUE
```

```
SOP(c instanceof Cat); // output is TRUE
```

```
SOP(d instanceof Cat); // output is FALSE
```

```
SOP(c instanceof Dog); // output is FALSE
```

#### COVARIANT RETURN TYPE:

It is a mechanism where in the return type of sub class method can be same as that of the super class method or it can be changed to any of its subclass type is called co-variant return type.

*Note :*

*in case of primitive data type, the return type of sub class method must be same as that of the super class method.*

### **Program Example :**

#### **Food.java**

```
public class Food {  
}
```

#### **VegFood.java**

```
public class VegFood extends Food{  
}
```

#### **NonVegFood.java**

```
public class NonVegFood extends Food{  
}
```

#### **NorthIndianFood.java**

```
public class NorthIndianFood extends VegFood{  
}
```

#### **SouthIndianFood.java**

```
public class SouthIndianFood extends VegFood{  
}
```

#### **CholeBatura.java**

```
public class CholeBatura extends NorthIndianFood{  
}
```

#### **PuriSabji.java**

```
public class PuriSabji extends NorthIndianFood{  
}
```

#### **Idly.java**

```
public class Idly extends SouthIndianFood{  
}
```

#### **Upma.java**

```
public class Upma extends SouthIndianFood{  
}
```

#### **FatherHotel.java**

```
public class FatherHotel {  
    VegFood serve(int choice){  
        switch(choice){  
            case 1:  
                return new CholeBatura();  
        }  
    }  
}
```

```

        case 2:
            return new PuriSabji();
        case 3:
            return new Idly();
        case 4:
            return new Upma();
        default :
            return null;
    }
}
}

```

### Customer.java [When father is in the hotel]

```

public class Customer {
    public static void main(String[] args) {
        FatherHotel f = new FatherHotel();
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter your choice..\n 1. Chole Batora..\n 2. Poori Sabji"
            + "\n 3. Idly..\n 4. Upma");
        int choice = sc.nextInt();
        VegFood vf = f.serve(choice);
        if (vf == null) {
            System.out.println("Item not available");
        } else {
            if (vf instanceof CholeBatura) {
                System.out.println("Enjoy chole batura..");
            } else if (vf instanceof PuriSabji){
                System.out.println("enjoy puri sabji with chatni..");
            } else if(vf instanceof Idly){
                System.out.println("suffer with Idly..");
            }else{
                System.out.println("Die with Upma..");
            }
        }
    }
}
}

```

### Output :

```

Enter your choice..
1. Chole Batora..
2. Poori Sabji
3. Idly..
4. Upma

```

User enter a choice → 1

Enjoy chole batura..

**Scenario : Now Son inherits Father's business, but he serves only the North Indian food and not south indian food.**

### **SonHotel.java**

```
public class SonHotel extends FatherHotel {
    @Override
    NorthIndianFood serve(int choice) {
        switch(choice){
            case 1:
                return new CholeBatura();
            case 2:
                return new PuriSabji();
            default:
                return null;
        }
    }
}
```

### **Customer.java [When Son is in the hotel]**

```
public class Customer {
    public static void main(String[] args) {
        SonHotel s = new SonHotel();
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter your choice..\n 1. Chole Batura..\n 2. Poori Sabji");
        int choice = sc.nextInt();
        NorthIndianFood vf = s.serve(choice);
        if (vf == null) {
            System.out.println("Item not available");
        } else {
            if (vf instanceof CholeBatura) {
                System.out.println("Enjoy chole batura..");
            } else if (vf instanceof PuriSabji){
                System.out.println("enjoy puri sabji with chatni..");
            }
        }
    }
}
```

### **Output :**

Enter your choice..

1. Chole Batura..

## 2. Poori Sabji

User enter a choice → 2

enjoy puri sabji with chatni..

User enter a choice → 4

Item not available

---

### Question : What is Constructor Chaining :

→ It is a mechanism where in we call one constructor from another constructor.

### Rules of Constructor chaining ?

- Constructor call should be invoked from a constructor only. [We can't call a constructor from any method or a block].
- Constructor calling statement must be the first line of executable code within a constructor.
- Only one constructor can be called from a given constructor.

### Question : What is this() and super() ?

- this() is a constructor calling statement which is used to call the current class constructor
- where as super() constructor calling statement is used to call the immediate super class constructor.

Note :

this() - we read this as

**this of**

OR

**this parenthesis**

OR

**this calling statement.**

### Question : Advantage of Constructor Chaining ?

- it enhances code reusability
  - it is used for code optimisation [it reduces the number of lines of code]
  - it increases the code readability
- 

Program example for this() calling statement :

```
public class Demo {  
    int a = 10;  
    Demo(int a){
```

```

        this();
        System.out.println(" one param constrctor "+a);
    }
    public Demo() {
        //this();
        //this(20);
        System.out.println(" Zero param constrctor ");
    }
    public static void main(String[] args) {
        Demo d1 = new Demo();
        Demo d2 = new Demo(10);
    }
}

```

Output:

```

Zero param constrctor
Zero param constrctor
one param constrctor 10

```

-----

Program example for Super() :

```

public class Parent {
    int a = 10;
    Parent(){
        System.out.println("Parent class zero param constructor..");
    }
    public Parent(int i) {
        System.out.println("Parent class One param constructor.." + i);
    }
    void methodOne(){
        System.out.println("method one in Parent class...");
    }
}

```

Child.java

```

public class Child extends Parent{
    int a = 20;
    int b = 30;
    void methodTwo(){
        System.out.println(this.a);//20
        System.out.println(a);//20
        System.out.println(super.a);//10
        System.out.println(this.b);//30
        System.out.println(b);//30
        System.out.println("method two in Child class...");
    }
}

```

```

    }
    Child(){
        super(10);
        System.out.println("Child class Zero param constructor ");
    }
    Child(int x){
        super();
        System.out.println("Child class parameterised constructor " + x);
    }
    public static void main(String[] args) {
        //Child c1 = new Child(10);
        Child c2 = new Child();
        c2.methodTwo();
    }
}

```

Output :

Parent class One param constructor..10

Child class Zero param constructor

20

20

10

30

30

method two in Child class...

---

### What is this keyword ?

→ this is a keyword which refers to current class object.

→ this keyword can be used to access any instance members. [may be instance variables or instance methods]

→ this keyword must be called from either a constructor body or an instance method.

[Note: This keyword can't be called from a static area.]

---

### What is super keyword ?

→ super is a keyword which refers to immediate super class object.

→ super keyword can be used to access any instance members of super class. [it may be instance variables or instance methods]

→ super keyword must be called from either a constructor body or an instance method.

---



## Difference between method and constructor ?

Method	Constructor
The name of the method need not be same as that of the class name. It can be any name as long as it is meeting up the rules of identifiers.	The name of the constructor must be same as that of the class name.
Methods are used to perform a specific task, that is it exposes the behaviour of an object.	Constructors are used to instantiate an object.
Methods must be invoked explicitly.	Constructors are invoked implicitly at the time of object creation.
Methods can return either primitive or non primitive data.	Constructor does not have any return type - not even void.
Methods can be inherited and hence, can be overridden.	Constructor can't be inherited and hence, can't be overridden.
Methods can be either static or non static.	Constructor can't be static

---

## What is an Anonymous object in java ?

→ An object without any reference is called anonymous object. We also call this object as deferred object. Such objects are eligible for garbage collection.

---

## Question : What is object instantiation ?

→ The process of assigning values to instance variables of an object is called instantiation of an object.

---

## What is a package ?

→ It is a group of similar type of classes or interfaces or sub-packages.

---

-

## How many types of package do we have ?

→ There are 2 types of packages in java.  
→ Inbuilt package ( Predefined package)

→ User defined package (**Custom** package)

---

### What are Inbuilt package in java ?

- Package which is already created by java community are known as inbuilt package.
  - All predefined packages are inbuilt packages.
- 

### Any example of inbuilt packages ?

- *java*
  - *lang* [ *It is the default package in java* ]
  - *io*
  - *util*
  - *awt*
  - *net*
  - *swing*
- 

### Question : Why we call java.lang as the default package in java ?

- Because, we don't have to import this package explicitly in order to access any of its resources.
- 

### Question : What are user defined package ?

- The package which is created by the user is called user defined package or custom package.
- 

### Question : How do you create an user defined package ?

- By using a keyword called ***package***.

Syntax : **Package** <package\_name>

---

### Question : What is the naming convention to define a package ?

- **domain.company.application.module**

Eg : com.google.gmail.inbox  
com.google.gmail.sentitems

---

● **Question : How do you access any resource from another package ?**

→ In order to access any resources from another package, we first have to import.

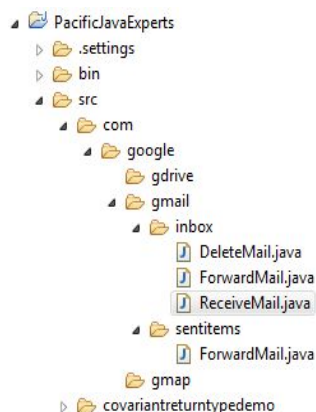
---

**How many ways of importing any resources ?**

→ There are 3 ways of importing any resources.

1. Importing a specific class or interface
  2. Importing all the classes or interfaces by using star ( \*)
  3. By specifying the fully qualified name of the class or interface
- 

Example :



```
1 package com.google.gmail.sentitems;
2 //importing all using *
3 import com.google.gmail.inbox.*;
4 //importing a specific class
5 import com.google.gmail.inbox.ReceiveMail;
6 public class ForwardMail {
7
8     public static void main(String[] args) {
9         ForwardMail fm = new ForwardMail();
10        ReceiveMail rm = new ReceiveMail();
11        //fully qualified class name
12        com.google.gmail.inbox.DeleteMail dm = new DeleteMail();
13    }
14 }
15
16
```

---

**Why do we go for package ?**

- It provides easy data access and manipulation.
- it provides access protection.
- it avoids naming collisions.
- it helps us in achieving modularity in the application.

**Access modifier:**

---

Question : What is accessed modifiers ?

1. Ans : Access modifiers are something that defines the accessibility level of any java gx. Here, java members can be a class, or an interface, or a variable, a method or a constructor.
- 

Question : How many types of access modifiers do we have in java ?

There are 4 types of access modifiers in java. They are

- public
  - protected
  - default
  - private
- 

Question : Describe the accessibility level of each access modifier ?

- public : have the highest visibility
  - protected : have higher visibility than default
  - default : have lower visibility than protected
  - private : have the lowest visibility
- 

Private Access Modifier :

- We can't declare a class as private. Unless it is an inner class [nested class]
- Private variables and methods must be accessed within the same class,

```
class Father {  
    private double accBal = 9999.00;  
    private void smoke(){  
        System.out.println("I am a chain smoker, my son should not know..");  
    }  
    public static void main(String[] args) {  
        Father f = new Father();  
        System.out.println(f.accBal);  
        f.smoke();  
    }  
}
```

- Private variables and methods can't be accessed outside the class.

Example :

```
public class Son extends Father{  
    public static void main(String[] args) {  
        Father f = new Father();  
        //Private members can't be accessed outside the class  
        f.accBal;//CE  
        f.smoke();//CE  
    }  
}
```

- Private constructor can't be accessed outside the class.
- If we want to stop someone to create an object of a class, declare the constructor as private.

*[Note : Object can be created within the same class if it has a private constructor]*

Program Example :

```
class Father {
```

```

        private double accBal = 9999.00;
        private void smoke(){
            System.out.println("I am a chain smoker, my son should not know..");
        }
        private Father() {
            // this is the constructor
        }
    }
}
public class Son {
    public static void main(String[] args) {
        //Private constructor can't be accessed outside
        //We can't create an object of a class if it has a private constructor
        Father f = new Father();//CE
    }
}

```

---

Default access modifier :

If we don't set any java member with either public or protected or private, then it is set to default level. Default access modifier is within the package level.

1. A default java member must be accessed with in the same package only. It can't be accessed outside the package.

Program example :

```

package accessmodifiers1;
class Father {

}
package accessmodifiers2;
import accessmodifiers1.Father;//CE
public class Son {
    public static void main(String[] args) {
        Father f;//CE
    }
}

```

---

-Scenario : Default variable , default method and default constructor can't be accessed outside the package.

Example :

```

package accessmodifiers1;

```

```

public class Father {
    double accBal = 9999.00;
    void smoke(){
        System.out.println("I am a chain smoker, my son should not
know..");
    }
    Father() {
        // this is the constructor
    }
}
package accessmodifiers2;
import accessmodifiers1.Father;
public class Son {
    public static void main(String[] args) {
        Father f = new Father();//CE
        System.out.println(f.accBal);//CE
        f.smoke();//CE
    }
}

```

Important Note : Default members can't be accessed even from any of its sub-packages.

---

Public access modifier.

1. Any java member declared with public keyword is public java member.
2. A public member can be accessed from anywhere within the same project or from outside the project as well.

Program example :

Scenario :

Public class, public variable, public methods and public constructor can be accessed outside the package anywhere with in the project.

```

package accessmodifiers1;
public class Father {
    public double accBal = 9999.00;
    public void smoke(){
        System.out.println("I am a chain smoker, my son should not know..");
    }
    public Father() {
        // this is the constructor
    }
}

```

```

}
package accessmodifiers2;
import accessmodifiers1.Father;
public class Son {
    public static void main(String[] args) {
        Father f = new Father();
        System.out.println(f.accBal);
        f.smoke();
    }
}

```

Output:

9999.0

I am a chain smoker, my son should not know..

---

Question :How do you access a public member from outside the package ?

By using import statements.

---

Protected access modifier.

1. Any java member declared with protected keyword is a protected java member.
2. A protected member can be accessed from within the same package and also from outside the package by using inheritance.

Program example :

Scenario : Protected members can't be accessed outside the package without inheritance

```

package accessmodifiers1;
public class Father {
    protected double accBal = 9999.00;
    protected void smoke(){
        System.out.println("I am a chain smoker, my son should not know..");
    }
    protected Father() {
        // this is the constructor
    }
}

```

Outside the package, not accessible without inheritance

```

package accessmodifiers2;
import accessmodifiers1.Father;
public class Son {

```

```

    public static void main(String[] args) {
        //Class which is set at default level can't be accessed outside the package
        Father f = new Father();//CE
        System.out.println(f.accBal);//CE
        f.smoke();//CE
    }
}

```

But, can be accessed outside the package using inheritance, followed by import and finally by using subclass reference variable.[Note : Not by superclass reference variable]

```

package accessmodifiers2;
import accessmodifiers1.Father;
public class Son extends Father{
    public static void main(String[] args) {
        //Parent class protected members can't be accessed by Parent class reference
        variable
        Father f = new Father();//CE
        System.out.println(f.accBal);//CE
        f.smoke();//CE
        //Parent class protected members must be accessed by Child class reference variable
        Son s = new Son();
        System.out.println(s.accBal);
        s.smoke();
    }
}

```

What are the advantages of access modifiers ?

OR

Why access modifiers ?

- It helps to control the data.
- It provides access protection.
- It enables security in the application

---

-

## Encapsulation

### Question : What is encapsulation ?

→ The process of wrapping the data member and the function members in to single unit is called encapsulation.



**Question : What is an encapsulated class ?**

- Class with at least one private data member is called encapsulated class.
- An encapsulated class is also known as business object class or data transfer object class.

**Question : What is a fully encapsulated class ?**

- Class with all private data members is called fully encapsulated class.

**Question : How do you access private data members ?**

- By using getters and setters method.

**Question : What is getters and setters methods ?**

- Getters method is the one using which we can read private data members.
- Setters method is the one using which we can update the private data members.

**Question : How do you make a class as Read-Only ?**

- By specifying only getters methods in a class.

**Question : How do you make a class as Write-Only ?**

- By specifying only setters methods in a class.

**Question : How do you make a class as both Read-Only and Write-Only ?**

- By specifying both getters and setters methods in a class.

**Question : What are the advantages of encapsulation ?**

- It is used to control the data flow
- It enhances security in the application.
- We can make a class as either Read-Only or Write-Only or both.

Sample Program :

Account.java

```
1 package encapsulationdemo;
2
3 public class Account {
4     private double accBal;
5     private String accHolderName;
6     public double getAccBal() {
7         return accBal;
8     }
9     public void setAccBal(double accBal) {
10         this.accBal = accBal;
11     }
12     public String getAccHolderName() {
13         return accHolderName;
14     }
15     public void setAccHolderName(String accHolderName) {
16         this.accHolderName = accHolderName;
17     }
18 }
```

CreateAccount.java

```
1 package encapsulationdemo;
2 public class CreateAccount {
3     public static void main(String[] args) {
4         Account a = new Account();
5         //Update the private data members
6         a.setAccHolderName("Amit");
7         a.setAccBal(1000);
8         //Read the values of private data members
9         System.out.println(a.getAccHolderName());
10        System.out.println(a.getAccBal());
11    }
12 }
```

Console

<terminated> CreateAccount [Java Application] C:\Program Files\Java\jdk1.8.0\_14

Amit

1000.0

wrapper class

### **Question : What is wrapper class ?**

- Every primitive data type has its corresponding non primitive data type in the form of class. This class is nothing but wrapper class.
- These classes are present in java.lang package.

OR

- Object representation of primitive data type is called wrapper class.
- 

-

### **Question : What are the advantages of Wrapper class ?**

1. Wrapper classes provides predefined method support.
  2. In collection, we can't use primitive data type, but we can use wrapper type.
- 

-

### **Question : What do you know about constructor of each wrapper class ?**

- Every wrapper class has overloaded constructor except Character class.
- Character class has only 1 constructor and Float class has 3 constructors.

Wrapper Class	Available Constructors
Byte	Byte(byte b)
	Byte(String s)
Short	Short(short s)
	Short(String s)
Integer	Integer(int i)
	Integer(String s)
Long	Long(long l)
	Long(String s)
Float	Float(float f)
	Float(String s)
	Float(double d)
Double	Double(double d)
	Double(String s)
Character	Character(char c)
Boolean	Boolean(boolean b)
	Boolean(String s)

---

### What is AutoBoxing ?

The process of automatic conversion of primitive data type in to its corresponding non primitive data type by the compiler is called autoboxing.

Eg :

Integer i = 25;

Boolean status = true;

AutoBoxing was introduced in JDK 1.5

---

### Explicit conversion :

2 ways :

1st approach :

```
Integer age = new Integer(25);
```

2nd approach :

```
Integer age = Integer.valueOf(25);
```

---

### **What is Unboxing ?**

The process of automatic conversion of non primitive data type in to its corresponding primitive type by the compiler is called unboxing.

```
int age = new Integer(25);
```

---

### **Explicit conversion :**

Eg1:

```
Character cnonprimitive = new Character('a');  
char cprimitive = cnonprimitive.charValue();
```

Eg2:

```
Boolean bnonprimitive = new Boolean(true); //object form  
boolean bprimitive = bnonprimitive.booleanValue();
```

Eg:3

```
int age = new Integer(25).intValue();
```

---

### **Program example**

```
public class WrapperClassDemo {  
    public static void main(String[] args) {  
        //Explicit conversion of primitive type to Non primitive type  
        byte b = 10;  
        Byte valueOf = Byte.valueOf(b);  
  
        //Implicit Conversion by the compiler (AutoBoxing)  
        Byte b1 = 10;  
  
        //Explicit conversion of Non primitive type to primitive type  
        Character c1 = new Character('c');  
        char charValue = c1.charValue();  
  
        //Implicit Conversion by the compiler (UnBoxing)  
        char ch = new Character('c');
```

```

//Conversion of int primitive type in to String format
int a = 10;
Integer i = new Integer(a);
String iStr = i.toString();
sysout(iStr instanceof String);//true

//explicit conversion of primitive type to non primitive type
Integer objFormA = Integer.valueOf(10);

//Use toString() method to convert in to String format

String aInStringForm = objFormA.toString();
System.out.println(aInStringForm);//"10"
//Use instanceof keyword to check whether it has converted in to String form

System.out.println(aInStringForm instanceof String);//true

//Conversion of boolean primitive type in to String format

boolean areYouMarried = true;
Boolean maritalStatus = Boolean.valueOf(areYouMarried);
String strFormat = maritalStatus.toString();
System.out.println(strFormat);//true
System.out.println(strFormat instanceof String);//true

//Convert String format in to int primitive type
Integer i = new Integer("20");
int ageInIntForm = i.intValue();
System.out.println(ageInIntForm);// 20 - primitive

//parseXXX() method
String age2 = "50";
int parseInt = Integer.parseInt(age2);

//Convert String format in to boolean primitive type
String maritalStatus1 = "true";
System.out.println("String format : "+maritalStatus1);
Boolean b3 = new Boolean(maritalStatus1);
boolean booleanValue = b3.booleanValue();
System.out.println("boolean type : "+booleanValue);
}
}

```

Summary :

1. valueOf()

→ is a static method present in all wrapper class.

→ it is used to convert a primitive type in to corresponding non primitive type.

Program eg :

```
byte b = 10;  
Byte valueOf = Byte.valueOf(b);
```

2. xxxValue()

→ is a non static method present in all the wrapper class.

→ it is used to convert a non primitive type in to its corresponding primitive type.

Program eg:

```
//Convert String format in to int primitive type  
Integer i = new Integer("20");  
int ageInIntForm = i.intValue();  
System.out.println(ageInIntForm); // 20 - primitive
```

3. toString()

→ is a non static method present in all the wrapper class.

→ it returns the corresponding string representation of a given non primitive type.

Eg program :

```
Integer a = 10;  
System.out.println(a); //10 - numeric  
String a1 = a.toString();  
System.out.println(a1); // "10" - string format  
  
System.out.println(a instanceof String);  
System.out.println(a instanceof Integer); //true  
  
System.out.println(a1 instanceof String); //true  
System.out.println(a1 instanceof Integer);
```

4. parseXXX()

→ is a static method present in all the wrapper class.

→ it is used to convert a string format value in to its corresponding primitive type

Program Eg :

```
String age = "10";  
int parseInt = Integer.parseInt(age);  
System.out.println(parseInt); //10
```

---

## **String**

- is a sequence of characters
- is a predefined class in java present in java.lang package.
- is a direct subclass of Object class.
- String is immutable in nature. That is, once a string object is created, it will not allow any modification. Still, if you want to make any changes, it will create a new String object.
- String class implements Comparable Interface.

### **How many ways you can create String object ?:**

String object can be created by 2 ways.

1. By using string literal
2. By using new Keyword and String class constructor

---

-

How many constructors do you have in String class ?

There are 13 constructors in String class.

Few Important constructors are mentioned below.

1. String()
2. String(String s)
3. String(byte[] b)
4. String(char[] c)
5. String(StringBuffer sb)
6. String(StringBuilder sb)

---

-

String Object creation -- **By using string literal**

Eg :

```
String s1 = "java";
```

→ String object created by literal will be stored in String Constant pool memory [SCP area].

Question : How string objects are created in SCP area ?

OR

Why duplicate objects are not allowed in SCP area ?

→ When an String object is created in SCP area, before it actually creates an object, it first checks whether an object is already created with the same content in the SCP area. If it is not already created, then it creates an object, and if it is already created, it doesn't create a



new string object and it starts referring to the existing object. Hence, duplicate object is not allowed in SCP area.

Question : What do you mean by dereferenced objects ?

→ Objects which no longer have any reference are known as dereferenced objects.

Question : What happens to dereferenced object in SCP area ?

→ In SCP area, dereferenced objects are not eligible for garbage collection because garbage collector doesn't have an access to SCP area. These dereferenced objects get released once the JVM is shut down. JVM will be shut down once we restart the server.

String Object creation -- By using new keyword and String class constructor

Eg :

```
String s = new String("java");
```

1. String object created using new keyword and object will be created in HEAP area.
  2. Duplicate objects allowed in HEAP memory. That is, if we try to create another string object with the same content, it will create a new object altogether. Hence, String are immutable in nature.
  3. Any dereferenced objects in HEAP memory will be garbage collected.
- 

### **String Concatenation :**

String can be concatenated using 2 options.

- " + " operator
- concat() method

#### **Using + operator :**

Eg: String message = "java" + "selenium";

3 objects will be created in total. [2 in SCP and 1 object in HEAP]

1st object content : **java [SCP area]**

2nd object content : **selenium [SCP area]**

1st object content : **javaselenium [HEAP]**

---

## Using concat() method

Eg :

```
String s = "java";  
s.concat("selenium");
```

3 objects will be created in total. [2 in SCP and 1 object in HEAP]

1st object content : **java [SCP area]**

2nd object content : **selenium [SCP area]**

1st object content : **javaselenium [HEAP]**

---

## String comparison :

Eg:

```
String s1 = "java";  
String s2 = "selenium";  
String s3 = "java";
```

```
SOP(s1==s2); // false  
SOP(s1==s3); // true  
SOP(s1.equals(s2)); // false  
SOP(s1.equals(s3)); // true
```

---

## Few Methods of String class :

```
→ package asdgdgsd;  
import java.util.Date;  
public class StringDemo {  
    public static void main(String[] args) {  
        Object o = new Object();  
        System.out.println(o.toString());  
        System.out.println(o);  
        String s = "babu";  
        System.out.println("initial string object content is :"+s.toString());  
        System.out.println("initial string object content is :"+s);  
        char c = s.charAt(1);  
        System.out.println(c);//a  
        //System.out.println(s.charAt(6));
```

```

String s1 = s.concat(" is my baby");
System.out.println(s1);
char[] charArray = s.toCharArray();
for (char c1 : charArray) {
    System.out.println(c1);
}
int length = s.length();
System.out.println("Total characters in a given string is :"+length);
String upperCase = s.toUpperCase();
System.out.println("Capital letter : "+upperCase);
String lowerCase = upperCase.toLowerCase();
System.out.println("Small letters : "+lowerCase);
String replacedString = s.replace('u', 'y');
System.out.println(replacedString);
Date d = new Date();
String d1 = d.toString();
System.out.println(d1);//
//Split function
String[] split = d1.split(":");
for (String name : split) {
    System.out.println(name);
}
String replaceAll = d1.replaceAll(":", "_");
System.out.println(replaceAll);
int firstOccurance = s.indexOf('b');
System.out.println("Index at first Occurance of B-->" + firstOccurance);
int secondOccurance = s.indexOf("b", 1);
System.out.println("Index at Second Occurance of B-->" +
secondOccurance);
String substring = s.substring(1);
System.out.println(substring);
String substring2 = s.substring(1,4);
System.out.println(substring2);//abu
boolean contains = s.contains("bu");//CTRL + 1 + ENTER
System.out.println(contains);
System.out.println(s.contains("BU"));
String subject1 = "java";
String subject2 = "Java";
String subject3 = "java";
//Equality
System.out.println(subject1==subject2);//false
System.out.println(subject1==subject3);//true

```

```

        System.out.println(subject1.equals(subject2));//false
        System.out.println(subject1.equals(subject3));//true
        String name = "malayalam";
        String revStr = "";
        for (int i = name.length()-1; i >= 0; i--) {
            System.out.println(name.charAt(i));
            revStr= revStr + name.charAt(i);
            System.out.println(revStr);
        }
        System.out.println(" Reversed String.." +revStr);
    }
}

```

#### StringBuffer Class :

- it is a predefined class present in java.lang package.
- it is a direct sub class of Object class in java.
- StringBuffer is mutable in nature. That is, once an stringBuffer object is created, it can be modified.

#### StringBuilder Class :

- it is a predefined class present in java.lang package.
- it is a direct subclass of Object class in java.
- StringBuilder is mutable in nature. That is, once an stringBuilder object is created, it can be modified.

**Question : what are the difference between StringBuffer and StringBuilder ?**

StringBuffer	StringBuilder
It is synchronized	It is non synchronized
StringBuffer is single threaded. That is only one single thread can act on a StringBuffer object	StringBuilder is multi threaded. That is multiple threads can simultaneously act on a single StringBuilder object
It is thread safe	It is not thread safe
Performance is relatively slower than StringBuilder	Performance is relatively faster than StringBuffer

**Question : How do you concatenate StringBuffer object ?**

**Program example :**

```

//Concatenation of 2 StringBuffer objects.
StringBuffer sb1 = new StringBuffer("java");
System.out.println(sb1);//java
sb1 = sb1.append("selenium");

```

```
System.out.println(sb1);//javaselenium
```

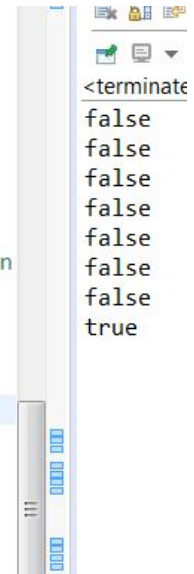
**Question : How do you compare StringBuffer objects ?**

```
//Comparison of 2 StringBuffer objects
StringBuffer sb1 = new StringBuffer("java");
StringBuffer sb2 = new StringBuffer("Java");
StringBuffer sb3 = new StringBuffer("java");

System.out.println(sb1==sb2);//false
System.out.println(sb1==sb3);//false
System.out.println(sb1.equals(sb2));//false
System.out.println(sb1.equals(sb3));//false -- meant for address comparison

String s1 = sb1.toString();
String s2 = sb2.toString();
String s3 = sb3.toString();

System.out.println(s1==s2);//false
System.out.println(s1==s3);//false
System.out.println(s1.equals(s2));//false
System.out.println(s1.equals(s3));//true - meant for content comparison
```



```
<terminate
false
false
false
false
false
false
false
true
```

**Question : when should we go for StringBuffer ?**

*Ans : when we want a string object to be modified with thread safety, then we go for StringBuffer.[Here, performance is not a concern]*

**Question : when should we go for StringBuilder?**

*Ans : when we want a string object to be modified with multiple threads working simultaneously for better performance, then we go for StringBuilder.*

**Question : What is the difference between equals() method of String class and StringBuffer class?**

*In String class - equals() method checks for the content. If the content is same, it returns true. And if the content is different, it returns false.*

*In StringBuffer class - equals() method checks for the address. If the address of two StringBuffer objects is same, it returns true. And if the address is different, it returns false.*

**Question : Equals() method is overridden in which Class ?**

*It is overridden in String class. And not in StringBuffer or StringBuilder class.*

---

## ABSTRACTION

**Question : What is Abstraction ?**

Ans:

1. It is the process of hiding the actual implementation from the end user and providing just the functionality is called abstraction.
2. It talks more of what an object does, rather than how it does.

**Question : How can we achieve abstraction ?**

Ans : Abstraction can be achieved by using abstract class and interface.

Using Abstract class - 0 to 100 % abstraction can be achieved.

Using interface - 100 % abstraction can be achieved.

**Question : Can we create an object of Abstract class ?**

Ans : No, we can't create an object of abstract class.

**Question : Can we instantiate an Abstract class ?**

Ans : No, abstract class can't be instantiated.

**Question : Why we can't create an object of Abstract class ?**

Ans : Because, abstract class can contains abstract methods or unimplemented methods.

If it allows us to create an object, then using that object reference, it will also allow us to invoke that method. And when we invoke a method, it is supposed to perform some or the other action. But, since abstract method does not have any implementation, it does not know what action it has to perform. And Hence, in java, we can't create an object of abstract class.

**Question : Can we have constructor in Abstract class ?**

Ans : Yes, we can have constructor in abstract class.

**Question : When we can't create an object of abstract class, then what is the need of constructor in abstract class ?**

Ans : To initialize the instance variable of abstract class, we have constructor in abstract class.

**Question : When we can inherits an abstract class, sub class constructor is enough to instantiate variables of super abstract class, then still why should we have constructor in abstract class ?**

Ans : To reduce the number of lines of codes in every implementation class of abstract class and thereby achieving code optimization.

**Explanation :**

Instead of initialising instance variable of super abstract class by having individual constructor in every implementation class, we can just have one constructor in super abstract class itself, which will automatically get called from subclass constructor as soon as we create an object of subclass. In this way, we can avoid code redundancy asked well.

**Question : In which scenario, we must declare a class as abstract ?**

Ans : If a class has at least one abstract method, then we must declare the class as abstract class.

**Question : Is it mandatory to provide implementation to all the abstract methods of abstract class ?**

Ans : No, it is not mandatory. It is optional. But, if we don't provide implementation to all the abstract method in the implementation class, then in that case, we must declare the class as abstract.

**Questions : Interface contains all abstract methods, and at the same time, an abstract class also can have all the methods as abstract, in that case, can we replace interface with an abstract class ? What is recommended to opt for? Interface or Abstract ?**

Ans : It is a good practice to go for interface, if we are dealing with all abstract methods.

**Reason 1:**

If we use interface, then we can have a sub class that can implements this interface and at the same time, we will also have an option to extends a class based on our requirement.

But, on the other hand, if we go for abstract class, then once we extends this abstract class, we will lose all the opportunity to extends another class, since multiple inheritance is not possible in case of class.

**Reason 2:**

Creation of an object of a class that implements an interface is not as time consuming as that of a class that extends an abstract class.

Because, in case of abstract class, the sub class constructor will also have to call the super abstract class constructor. It takes little time to do this process. Whereas in case of interface, since there is no concept of constructor, it simply creates an object of subclass. In this case, the process is relatively faster.

**Question : What are the differences between an Abstract class and an interface ?**

Interface	Abstract Class
-----------	----------------

An interface is declared using interface keyword.	An abstract class is declared using abstract keyword.
All the variables in an interface are by default public, static and final	All the variables in an abstract class need not be public , static and final.
We can't declare interface variables as either private or protected or default access modifiers	We can very well declare variables of an abstract class as private or protected or default. There are no restrictions as such.
We can't declare the variables as transient or volatile → transient (Because interface object can't be created and hence variables are not synchronized) → volatile (because these variables are final).	There are no restriction as such.
In interface, variables must be initialized at the time of declaration.	In Abstract class, variables need not be initialized at the time of declaration.
In interface, all the methods are abstract and hence an interface can also be called as a 100 percent abstract class.  Note : From JDK 1.8 onwards, an interface can contain concrete methods as well.	Abstract class can have both abstract and concrete methods.
Methods in interface are by default public and abstract.	There are no such restrictions in abstract class,
Methods in interface can't be private, protected, default, static or final	There are no such restrictions in abstract class.
Interface neither have any default constructor nor we can define one.	Abstract class can have both default constructor or user defined constructor.
We can't have instance block or static block in interface	Abstract class can have both instance and static block.

### Question : When should we go for interface or abstract class ?

Ans : If we just have the requirement specification but no idea about the actual implementation, then we should go for interface.



On the other hand, if we have very little information about the implementation part (partial implementation), then we should go for abstract class

---

## Collection Framework

### Limitation of Array:

- Array is fixed in size.
- Array can't store heterogeneous type of data. It holds only homogenous data type.
- Array does not have any underlying data structure and hence, it does not have any predefined method support.
- Arrays allows both primitive and non primitive datatype.
- Array is not a recommended option in terms of memory.
- Performance wise – Array is a better option.

### Advantages of Collection :

- Collection is growable in size.
- It can hold heterogeneous data type.
- Every collection class has underlying data structure, so we have predefined method support.
- Collection does not allow primitives. It allows only non primitive type.
- Collection is always the recommended option when it comes to memory.
- Performance wise – collection is not a better option.

### What are the Differences between Array and Collection ?

ARRAY	COLLECTION
→ Array is fixed in size.	→ Collection is growable in size.
→ Array can't store heterogeneous type of data. It holds only homogenous data type.	→ It can hold heterogeneous data type.

→ Array does not have any underlying data structure and hence, it does not have any predefined method support.	→ Every collection class has underlying data structure, so we have predefined method support.
→ Array allows both primitive and non primitive data type.	→ Collection does not allow primitives. It allows only non primitive data type.
→ Array is not a recommended option in terms of memory.	→ Collection is always the recommended option when it comes to memory
→ Performance wise – Array is a better option	→ Performance wise – collection is not a better option

#### **Question : What is Collection ?**

Ans : Collection is a representation for group of individual objects as a single entity.

#### **Question : What is Collection Framework in java?**

Ans: Collection framework in java is a collection of readily available classes, interfaces and methods to represent group of entities as a single entity.

#### **Question : What are the 9 key interfaces of Collection framework ?**

1. Collection
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map
8. SortedMap
9. NavigableMap

#### **1. Collection Interface :**

- Collection is the root interface in Collection interface.
- This interface contains those methods which are commonly used across any collection objects.
- There is no concrete class which provides implementation to all the abstract methods of Collection interface.

### **Question : Difference between Collection and Collections ?**

Ans :

→ Collection is an interface which is used to represent a group of individual objects as a single entity.

where as,

→ Collections is a class in java present in java.util package, using which we can sort the elements present in collection object or we can search the elements in collection object.

### **2. List Interface :**

- List is the child interface in Collection interface.
- List preserves insertion order and it also allows duplicate.
- List interface is implemented by 3 classes :
  - o ArrayList,
  - o LinkedList
  - o Vector

### **Question : When do we go for List interface ?**

Ans : If we want to represent a group of individual objects as a single entity by preserving their insertion order and allowing duplicates, then we should go for List interface.

### **3. Set Interface**

- It is the child interface of Collection interface.
- It neither allows duplicates and it does not preserve insertion order.

### **Question : When should we go for Set ?**

Ans : If we want to represent a group of individual objects as a single entity without allowing duplicates and without preserving insertion order then we should go for Set.

### **4. SortedSet Interface**

- It is the child interface of Set interface.

- It neither allows duplicates and it does not preserve insertion order.
- It stores the elements in sorted order.

#### **Question : When should we go for SortedSet ?**

Ans : If we want to store a group of individual objects in some sorting order without allowing duplicates and without preserving insertion order then we should go for SortedSet.

#### **5. NavigableSet Interface**

- It is the child interface of SortedSet interface.
- It neither allows duplicates and it does not preserve insertion order.
- It has various methods using which we can navigate to previous or next elements in the collection objects.

#### **Question : When should we go for NavigableSet ?**

Ans : If we want to navigate within the elements present in a Set type collection object then we should go for Navigable Set.

#### **6. Queue Interface**

- It is the child interface of Collection interface.
- It is used to represent a group of individual objects before processing

#### **Question : When should we go for Queue ?**

Ans : If we want to represent a group of individual objects before we actually process it, then we should go for Queue.

#### **7. Map Interface**

- It is **NOT** the child interface of Collection interface.
- It stores data in the form of key and value pair, where Key can't be duplicated, but values can be duplicate.
- Each key and value pair in a map object is known as an ENTRY.

#### **Question : When do you go for MAP ?**

Ans : When we want to represent a group of individual objects in the form of key and value pair then we should go for map.

## 8. SortedMap Interface

- It is the child interface of Map interface.
- It stores a group of key and value pair in some sorting order based on the key.

### Question : When should you go for SortedMap ?

Ans : If we want to represent a group of key and value pair in some sorting order based on the key, then we should go for SortedMap.

## 9. NavigableMap Interface

- It is the child interface of SortedMap interface.
- It has few utility methods, using which , we can navigate within the entries present in a given map object.

### Question : When should you go for NavigableMap ?

- If we want to navigate back and forth within the entries present in a given map object, then we should go for NavigableMap.

## Collection interface :

Following are the important methods in Collection interface.

- boolean --- add(Object o)
- boolean --- addAll(Collection c)
- boolean --- remove(Object o)
- boolean --- removeAll(Collection c)
- boolean --- retainAll(Collection c)
- void --- clear()
- boolean --- contains(Object o)
- Boolean --- containsAll(Collection c)
- Boolean --- isEmpty()
- int --- size()
- Object[] --- toArray()
- Iterator --- iterator()

## Note :

- Collection interface doesn't have any method to fetch the object.
  - No Concrete class which directly implements Collection interface.
-

### List Interface :

Few methods of List interface :

- void --- add(int index, Object o)
  - boolean --- addAll(int index, Collection c)
  - Object get(int index)
  - Object remove(int index)
  - Object --- set(int index, Object o)
  - int --- indexOf(Object o)
  - int --- lastIndexOf(Object o)
  - ListIterator --- listIterator()
- 

### ArrayList:

- ArrayList is the implementation class of List interface
- The underlying data structure is resizable or growable array.
- ArrayList allows duplicate values.
- It preserves insertion order.
- It allows NULL insertion
- The initial capacity of ArrayList object is 10. Once it reaches the maximum capacity, then it increases the capacity using the below formula.

$$\text{New Capacity} = (\text{Current Capacity} * 3/2) + 1$$

### ArrayList Constructors :

- ArrayList()
- ArrayList(int initialCapacity)
- ArrayList(Collection c)

**Note :** Every collection class implements 2 interfaces – Serializable and Cloneable.

### Question : What is Serializable ?

Ans : The ability of transferring objects across the network is called Serializable

### Question : What is Cloneable ?

Ans : The ability of creating a clone of an object is called cloneable.

### Question : When do you think ArrayList is the best choice ?

Ans : If our frequent operation is retrieval, then ArrayList is the best option.

**Question : When do you think ArrayList is the worst choice ?**

Ans : If the frequent operation is either insertion or deletion of elements, then ArrayList is the worst choice, because it involves lot of shift operation which is really time consuming.

**Question : Why should you go for ArrayList when your frequent operation is Retrieval ?**

Ans : Because ArrayList implements RandomAccess interface, using which, we can access any random element with the same speed.

**Question : What is RandomAccess ?**

- RandomAccess is an interface in java present under java.util package.
- It is one of the Marker interface in java, which is implemented by ArrayList and Vector class.
- It provides a flexibility to access any element in the ArrayList object with the same speed.

<b>ArrayList</b>	<b>Vector</b>
→ Introduced in 1.2v and hence it is a non legacy class	→ Introduced in 1.0v and hence it is a legacy class
→ It is non synchronized.	→ It is synchronized
→ It is multi-threaded and hence, it is not thread safe	→ It is single-thread and hence thread safe.
→ Performance is high	→ Performance is low

**Question : Why ArrayList is performance wise better than Vector ?**

Ans : Because multiple threads can act on single ArrayList object at the same time and hence Performance is relatively higher than Vector.

**Question : Why ArrayList is not synchronized ?**

Ans : Because all the methods of ArrayList are non synchronized.

**Question : Why Vector is synchronized ?**

Ans : Because all the methods of Vector are synchronized.

**Question : How do you get the synchronized version of List ?**

Ans : By using synchronizedList() method of Collections class.

```
List<String> list = new ArrayList<String>();
```

```
List<String> synchronizedList = Collections.synchronizedList(list);
```

### Question : How do you get the synchronized version of Set ?

Ans : By using synchronizedSet() method of Collections class

```
Set<String> hset = new HashSet<String>();
```

```
Set<String> synchronizedSet = Collections.synchronizedSet(hset);
```

### Question : How do you get the synchronized version of Map ?

Ans : By using synchronizedMap() method of Collections class

```
Map<Integer,String> hsMapObj = new HashMap<Integer,String>();
```

```
Map<Integer,String> synchronizedMap =  
Collections.synchronizedMap(hsMapObj);
```

---

### LinkedList Class:

.

- LinkedList is the implementation class of List interface
- The underlying data structure is doubly Linked list.
- It allows duplicate values.
- It preserves insertion order.
- It allows NULL insertion
- It implements Serializable and Cloneable interface but not RandomAccess interface.
- Initial capacity concept is not applicable for LinkedList.

### LinkedList Constructors :

- LinkedList () – it creates an empty linkedList object.
- LinkedList (Collection c) – it creates an equivalent linkedlist object

### Question : When do you think LinkedList is the best choice ?

Ans : If our frequent operation is insertion or deletion, then LinkedList is the best option.

### Question : When do you think LinkedList is the worst choice ?



Ans : If the frequent operation is retrieval, then LinkedList is the worst choice, because here, one node stores the address of another node and communication between multiple nodes is very much time consuming.

#### **Few methods of LinkedList :**

- addFirst()
- addLast()
- getFirst()
- getLast()
- removeFirst()
- removeLast()

#### **Question : What are the difference between ArrayList and LinkedList ?**

<b>ArrayList</b>	<b>LinkedList</b>
→ Underlying data structure is resizable or growable array	→ Underlying data structure is doubly linked list
→ ArrayList implements RandomAccess	→ LinkedList does not implements RandomAccess
→ ArrayList is the best option when the frequent option is retrieval	→ LinkedList is the best option when the frequent operation is insertion or deletion of elements in between.
→ ArrayList is the worst choice when the frequent operation is insertion or deletion of elements in between.	→ LinkedList is the worst choice when the frequent operation is retrieval.

---

#### **Vector Class:**

- It is an implementation class of List interface.
- Underlying data structure is resizable array or growable array.
- The initial capacity of Vector is 10. Once it reaches the maximum capacity, it doubles the capacity.
- It allows duplicate values.
- It preserves insertion order
- It allows NULL insertion.
- It allows heterogeneous data type.

- It implements Serializable, Cloneable and RandomAccess interface.
- It is introduced in 1.0 version and hence it is a legacy class.
- All methods of Vector are synchronized and hence it is thread-safe.
- It works the best when we try for retrieval operations.

### Constructors of Vector:

- Vector()

### Program example :

```

3 public class VectorClassDemo {
4     public static void main(String[] args) {
5         Vector<Integer> v = new Vector<Integer>();
6         for (int i = 1; i <= 10; i++) {
7             v.add(i);
8         }
9         System.out.println(v);
10        System.out.println(v.capacity()); //10
11        v.addElement(11);
12        System.out.println(v.capacity()); //20
13        System.out.println(v);
14    }
15 }

```

Console

```

<terminated> VectorClassDemo [Java Application] C:\Program Files\Java\jre1.8.0_144\bin\javaw.exe (13-
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
10
20
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

```

- Vector(int initialCapacity)
- Vector(int initialCapacity, int incrementalValue)
- Vector(Collection c)

### Methods of Vector :

- addElement(Object o)
- removeElement(Object o)
- removeElementAt(int index)

- removeAllElements()
- elementAt(int index)
- firstElement()
- lastElement()
- int size()
- int capacity()
- Enumeration elements()

### Three Cursors in Java collection :

#### 1. Enumeration :

- It is used to iterate in legacy collection object.

It has methods like –

- Boolean --- hasMoreElements()
- Object --- nextElement()

Program example :

```

1 package collectionsbyme;
2 import java.util.Enumeration;
4 public class EnumerationDemo {
5     public static void main(String[] args) {
6         Vector<Integer> v = new Vector<Integer>();
7         for (int i = 0; i <= 10; i++) {
8             v.addElement(i);
9         }
10        Enumeration<Integer> allElements = v.elements();
11        while(allElements.hasMoreElements()){
12            Integer element = allElements.nextElement();
13            //System.out.println(element);
14            if (element%2==0) {
15                System.out.println(element);
16            }
17        }
18    }
19 }

```

Console output: 0, 2, 4, 6, 8, 10

#### 2. Iterator :

- It is the universal cursor, that is, it works with any collection objects.
- Using Iterator, not only we can read the data but also we can remove the elements in the collection object.

#### Methods of Iterator :

- boolean - hasNext()
- Object -- next()

→ void -- remove()

### Program Example :

```
IteratorDemo.java 25
6 public class IteratorDemo {
7     public static void main(String[] args) {
8         ArrayList<Integer> al = new ArrayList<Integer>();
9         System.out.println("empty ArrayList object --> "+al);
10        for (int i = 0; i <= 10; i++) {
11            al.add(i);
12        }
13        System.out.println("*****elements inside*****");
14        System.out.println(al);
15        Iterator<Integer> iterator = al.iterator();
16        System.out.println("*****If the value is even, then print, else remove*****");
17        while(iterator.hasNext()){
18            Integer element = iterator.next();
19            if (element %2 == 0) {
20                System.out.println(element);
21            } else {
22                iterator.remove();
23            }
24        }
25        System.out.println("*****Print the final updated arraylist object*****");
26        System.out.println(al);
27    }
28 }
29
```

### Output :

```
Console
<terminated> IteratorDemo [Java Application] C:\Program Files\Java\jre1.8.0_144\bin\javaw.exe (05-Mar-2019, 1
empty ArrayList object --> []
*****elements inside*****
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
*****If the value is even, then print, else remove*****
0
2
4
6
8
10
*****Print the final updated arraylist object*****
[0, 2, 4, 6, 8, 10]
```

### 3. ListIterator:

- It is a bidirectional cursor, that is, it is used to traverse in the collection object both in forward and backward direction.
- It can be used to remove an element from the given collection object.
- It can be used to replace an existing element in the collection object
- It can also be used to add an element in to the collection object.

### Methods of ListIterator :

- boolean --- hasNext()
- Object --- next()
- int --- nextIndex()
- boolean --- hasPrevious()
- Object --- previous()
- int --- previousIndex()
- void --- remove()
- void --- set(Object o)
- void --- add(Object o)

### Program example:

```
public class ListIteratorDemo {
    public static void main(String[] args) {
        LinkedList<String> listObj = new LinkedList<String>();
        System.out.println("****Empty LinkedList object****");
        System.out.println(listObj);
        //Add cities in the linkedlist object
        listObj.add("Amritsar");
        listObj.add("Bombay");
        listObj.add("Chennai");
        listObj.add("Delhi");
        System.out.println("****LinkedList object with elements inside****");
        System.out.println(listObj);
        ListIterator<String> lstltr = listObj.listIterator();
        System.out.println("****Print all the names in forward direction****");
        while(lstltr.hasNext()){
            String name = lstltr.next();
            System.out.println(name);
        }
        System.out.println("*****Print all the names in backward direction*****");
        while(lstltr.hasPrevious()){
            String name = lstltr.previous();
            System.out.println(name);
        }
    }
}
```

```

System.out.println("*****Remove Bombay if it exists"
+ "*****Replace Chennai with Calcutta*****"+
"*****Add England next to Delhi*****");
while(lstltr.hasNext()){
String name = lstltr.next();
if (name.equals("Bombay")) {
lstltr.remove();
}else if(name.equals("Chennai")){
lstltr.set("Calcutta");
} else if(name.equals("Delhi")){
lstltr.add("England");
}
}
System.out.println("Final updated linkedlist object values is below");
System.out.println(listObj);
}
}

```

### Output :

\*\*\*\*Empty LinkedList object\*\*\*\*

[]

\*\*\*\*LinkedList object with elements inside\*\*\*\*

[Amritsar, Bombay, Chennai, Delhi]

\*\*\*\*Print all the names in forward direction\*\*\*\*

Amritsar

Bombay

Chennai

Delhi

\*\*\*\*\*Print all the names in backward direction\*\*\*\*\*

Delhi

Chennai

Bombay

Amritsar

\*\*\*\*\*Remove Bombay if it exists\*\*\*\*\*Replace Chennai with Calcutta\*\*\*\*\*Add  
England next to Delhi\*\*\*\*\*

[Amritsar, Calcutta, Delhi, England]

---

**Question : Difference between Enumeration, Iterator and ListIterator ?**

Enumeration	Iterator	ListIterator
-------------	----------	--------------

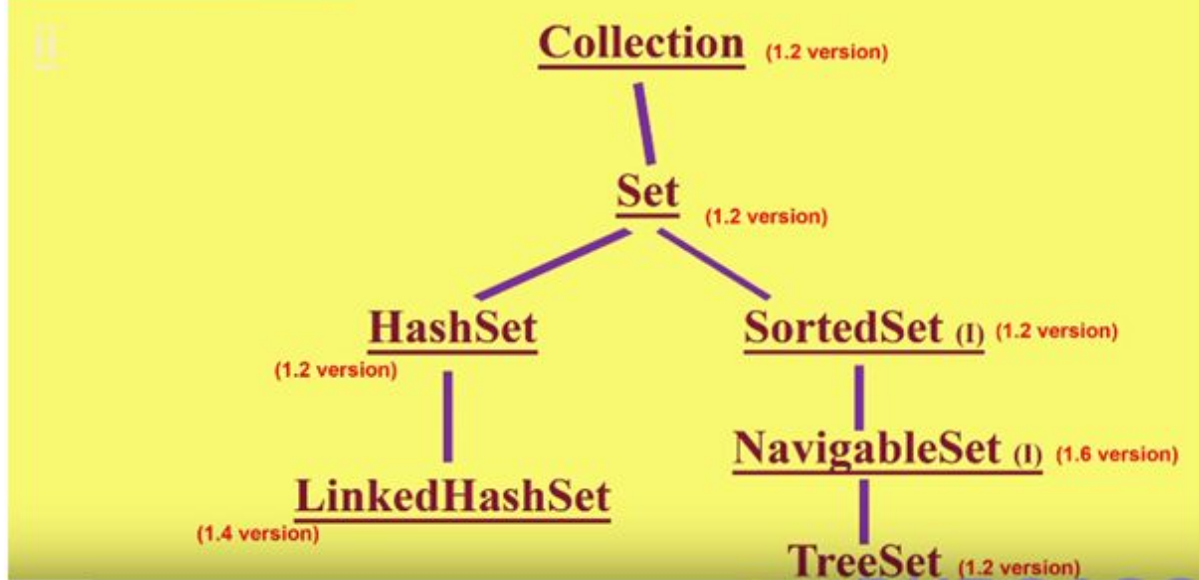
→ It is legacy interface introduced in 1.0v	→ It is not a legacy since it is introduced in 1.2v	→ It is not a legacy since it is introduced in 1.2v
→ Enumeration works only with Legacy class, that is Vector class.	→ Iterator works with any type of collection objects.	→ ListIterator works only with List type of collection objects.
→ We get Enumeration object reference by using elements() method of Vector class	→ We get Iterator object reference by using iterator() method of Collection interface.	→ We get ListIterator object reference by using listIterator() method of List interface.
→ Using Enumeration, we can traverse ONLY in forward direction. Hence, we call Enumeration as single directional.	→ Using Iterator, we can traverse in only forward direction. Hence, we call Iterator as single directional.	→ Using ListIterator, we can traverse in both forward and backward direction. Hence, we call ListIterator as bi-directional.
→ Using Enumeration, we can only READ the elements	→ Using Iterator, apart from reading the elements, we can also remove elements.	→ Using ListIterator, we can READ, REMOVE, ADD and also REPLACE the existing elements.
→ Enumeration has 2 methods. 1. hasNextElement() 2. nextElement()	∅ Iterator has 3 methods. 1. hasNext() 2. next() 3. remove()	∅ ListIterator has 9 methods. 1. hasNext() 2. next() 3. nextIndex() 4. hasPrevious() 5. previous() 6. previousIndex() 7. remove() 8. set(Object o) 9. add(Object o )

---

### SET Interface

Set in collection hierarchy :

## Set Interface :



### Question : When do you go for Set ?

Ans : We go for SET when we want to represent a group of individual objects as a single entity where duplicates are not allowed or insertion order is not preserved.

*Note : Set interface does not have any methods of its own.*

---

### HASHSET Class :

- HashSet is the implementation class of Set interface.
- The underlying data structure is Hashtable
- It does not allow duplicate.
- It does not preserve insertion order.
- It stores the objects based on their hashCode.
- It allows NULL insertion
- It allows heterogeneous objects.
- It implements both Serializable and Cloneable interface but not RandomAccess
- It is a best choice, when our frequent operation is retrieval.

### Constructors of HashSet Class :

#### → HashSet()

- It creates an empty hashset object with default initial capacity as 16 and default fill ratio or load factor as 0.75



→ **HashSet(int initialCapacity)**

→ It creates an empty hashset object with specified initial capacity and default fill ratio or load factor as 0.75

→ **HashSet(int initialCapacity, float loadFactor)**

→ It creates an empty hashset object with the specified initial capacity and specified fill ratio or load factor.

→ **HashSet(Collection c)**

→ It creates an equivalent hashset object from any given collection object.

### **Program Example :**

---

### **LINKEDHASHSET**

- LinkedHashSet is the child class of HashSet class.
- The underlying data structure is Hashtable and LinkedList
- It does not allow duplicate.
- It preserves insertion order.
- It allows NULL insertion
- It allows heterogeneous objects.
- It implements both Serializable and Cloneable interface but not RandomAccess

### **Question : When should we go for LinkedHashSet ?**

→ When we want to represent a group of individual objects as single entity without allowing duplicate and preserving the insertion order, then we should go for LinkedHashSet.

→ This is mainly used to develop cache based application.

### **Program Example :**

```
package collectionDemo;
import java.util.LinkedHashSet;
public class LinkedHashSetDemo {
    public static void main(String[] args) {
        LinkedHashSet lh=new LinkedHashSet();
        lh.add(null);
        lh.add(10);
        lh.add(20);
        lh.add(50);
    }
}
```

```

        lh.add("Java");
        lh.add("selenium");
        System.out.println(lh);
    }
}

```

---

#### **OUTPUT:**

**[null, 10, 20, 50, Java, selenium]**

**\*It preserve insertion order.**

---

#### **TREESET Class :**

- It is the implementation class of SortedSet interface.
- The underlying data structure is Balanced Tree.
- It does not allow duplicate.
- Insertion order is not preserved but elements will be stored based on some sorting order.
- It can store only one NULL
- It does not allow heterogeneous object.

#### **Question : When do you get ClassCastException in TreeSet ?**

Answer : TreeSet does not allow heterogeneous object, if we still try to add heterogeneous object, we get ClassCastException.

#### **Question : Can you add NULL in to a TreeSet object ?**

**OR**

#### **What is NULL acceptance in TreeSet object ?**

- In case of an empty TreeSet object, we can insert only one NULL as the first element.  
After inserting NULL as the first element, if we try to insert any other element, then we get **NullPointerException**.
- Whereas , in case of non empty TreeSet object, we can't insert NULL. If we try to do so, we get NullPointerException.

#### **Constructors of TreeSet :**

- `TreeSet ts = new TreeSet()`  
It is used to create an empty TreeSet object with default natural sorting order.
- `TreeSet ts = new TreeSet(Comparator c)`  
It is used to create an empty TreeSet object with custom sorting order.
- `TreeSet ts = new TreeSet(Collection c)`
- `TreeSet ts = new TreeSet(SortedSet s)`

## Default Sorting in TreeSet :

Program example :

```
4 public static void main(String[] args) throws Exception {
5     TreeSet tsObj = new TreeSet();
6     tsObj.add("ajit");
7     tsObj.add("AJIT");
8     tsObj.add("amit");
9     tsObj.add("AJIT");//Duplicate not allowed
10    //tsObj.add(new Integer(10));/// Heterogeneous objects not allowed - ClassCastException
11    System.out.println(tsObj);
12 }
13 }
```

Console [terminated] TreeSetDemo (1) [Java Application] C:\Program Files\Java\jdk1.8.0\_144\bin\javaw.exe (07-Mar-2019, 2:01:46 PM)

[AJIT, ajit, amit]

## Question : What are the preconditions for default sorting?

Ans : The object should be both comparable and homogeneous.

Note : If these 2 preconditions are not satisfied, then we get ClassCastException.

-----  
-

## Comparable Interface

- It is an interface present in java.lang package
- It has ONLY one non static method –
  - int --- compareTo(Object obj)

## Question : Explain compareTo() method ?

Ans:

- compareTo() is a non static method present in Comparable interface.
- It is used to compare 2 objects and returns an integer value.
  - It returns negative value, if obj1 has to come before obj2.
  - It returns positive value, if obj2 has to come after obj1.
  - If obj1 == obj2 , then it returns zero
- The return type is int.

## Program Example :

## Example of Default sorting in TreeSet :

In default sorting, JVM internally calls compareTo() method of Comparable interface.

Program :

**Question : What are the different types of sorting ?**

- Default natural sorting and Custom sorting
  - Default natural sorting can be achieved by using Comparable interface and Custom sorting by using Comparator interface.
- 

**COMPARATOR Interface :**

- It is an interface present in java.util package.
- It is used for custom sorting.

**Methods of Comparator :**

- int --- Compare(Object obj1, Object obj2)
- It is used for custom sorting.

**Question : Explain compare() method ?**

Ans :

- compare() is a non static method present in Comparator interface.
- Using this method, we can define our custom sort logic.

**Question : How many methods we have in Comparator interface ?**

Ans : Two methods [1. compare() and 2. equals()]

**Question : Which method is mandatory to be implemented in any implementation class of Comparator interface ?**

Ans : compare() method.

**Question : Why equals() method implementation is optional in any Comparator implementation class ?**

Ans : Because equals() method of Object class is already inherited in every implementation class of Comparator interface through inheritance.

**Custom Sort Example using comparator :**

Program :

Display these integers in descending form.

Custom sort using Comparator :

Output

Various possible combination of Compare() method :

Output : Only first element is inserted as shown below.

**Question : When you try to insert StringBuffer objects in to TreeSet, What happens ?**

Ans : We get ClassCastException, because for default natural sorting, the object must be comparable. And by default, StringBuffer class doesn't implements Comparable

Program :

**Question : How do you sort StringBuffer object in reverse alphabetical order ?**

Ans : By defining custom sort logic using Comparator interface.

Program :

Class : MyComparator.java

Class : StringBuffer\_ReverseAlphabetic\_Order.java

NOTE : If we define our own custom sorting using Comparator, then the object need not be Comparable.

**Question : When you define your own custom sorting using Comparator, does the object need be homogeneous and comparable ?**

Ans : No, Objects need not be homogeneous or comparable if we define our own custom sorting using Comparator interface.

That is, it can be heterogeneous and non comparable.

**Scenario 1 :** Sort both String and StringBuffer objects based on their length, If length is same, then sort based on the alphabetical order.

Program :

Comparator sort logic below:

**Scenario 2 :** Sort both String and StringBuffer objects based on their length, If length is same, then sort based on reverse alphabetical order.

Program :

Comparator sort logic below:

---

**MAP Interface :**

→ if we want to represent group of individual objects in the form of key and value pair, then we go for Map interface.

## **HashMap Class :**

- HashMap is an implementation class of Map interface.
- Combination of a key and value pair is called an entry.
- Only one null key is allowed but multiple null values is possible.
- Insertion order is not preserved but elements will be stored based on hashcode of keys.
- Duplicate key not allowed, but values can be duplicated.
- It allows heterogeneous object.
- Underlying data structure is Hashtable.

## **Constructors of HashMap :**

- HashMap()  
It is used to create an empty HashMap object with initial capacity 16 and default fill ratio of 0.75
- HashMap(int initialCapacity)  
It is used to create an empty HashMap object with the specified initial capacity and default fill ratio of 0.75
- HashMap(int initialCapacity, float fillRatio)
- HashMap (Map m)

## **Few methods of HashMap :**

- boolean --- isEmpty()
- int --- size()
- Object --- put()  
It is used to add an entry in to the map object and it returns the old value of the specified key if exists. If old value does not exists, it returns null.
- Object --- get()
- Object --- remove()
- void --- clear()
- boolean --- containsKey()
- boolean --- containsValue()
- Set<Object>--- keySet()
- Collection --- values()
- Set<Entry<Object, Object>>--- entrySet()

## **Program example of HashMap :**

```
public class HashMapDemo {  
    public static void main(String[] args) {  
        HashMap<Integer, String> hsmmap = new HashMap<>();  
        System.out.println(hsmmap.size());//0  
    }  
}
```

```

System.out.println(hsmap.isEmpty()); //true
System.out.println(hsmap.put(101, "ajit")); //Null
System.out.println(hsmap.put(101, "amit")); //ajit
System.out.println(hsmap.size()); //1
System.out.println(hsmap.isEmpty()); //false
hsmap.put(201, "anju");
System.out.println(hsmap); //{101=amit, 201=anju}
System.out.println(hsmap.remove(101)); //amit
System.out.println(hsmap); //{201=anju}
hsmap.put(201, "anju");
hsmap.put(301, "sanju");
hsmap.put(401, "prateek");
System.out.println(hsmap); //{401=prateek, 201=anju, 301=sanju}
System.out.println(hsmap.containsKey(301)); //true
System.out.println(hsmap.containsValue("prateek")); //true

```

#### //Example of keySet() method

```

Set<Integer> allKeys = hsmap.keySet();
Iterator<Integer> itr = allKeys.iterator();
while(itr.hasNext()){
    Integer rollNum = itr.next();
    System.out.println(rollNum + " : " + hsmap.get(rollNum));
}

```

#### //Example of values() method

```

Collection<String> allNames = hsmap.values();
Iterator<String> itr1 = allNames.iterator();
while(itr1.hasNext()){
    String name = itr1.next();
    System.out.println(name);
}

```

#### //Example of entrySet() method

```

Set<Entry<Integer, String>> allEntries = hsmap.entrySet();
Iterator<Entry<Integer, String>> itr2 = allEntries.iterator();
while(itr2.hasNext()){
    Entry<Integer, String> entry = itr2.next();
    System.out.println(entry.getKey() + " --> " + entry.getValue());
    if (entry.getKey().equals(301)) {
        entry.setValue("manju");
    }
}
System.out.println(hsmap); //{401=prateek, 201=anju, 301=manju}
}

```

}

---

**Output:**

0

true

null

ajit

1

false

{101=amit, 201=anju}

amit

{201=anju}

{401=prateek, 201=anju, 301=sanju}

true

true

401 : prateek

201 : anju

301 : sanju

prateek

anju

sanju

401 --> prateek

201 --> anju

301 --> sanju

{401=prateek, 201=anju, 301=manju}

---

### **LinkedHashMap Class :**

- LinkedHashMap is a subclass of HashMap class.
- Only one null key is allowed but multiple null values is possible.
- Insertion order is preserved but elements will be stored based on hashcode of keys.
- Duplicate key not allowed, but values can be duplicated.
- It allows heterogeneous object.
- Underlying data structure is Hashtable and LinkedList.

### **Constructors of LinkedHashMap:**

- LinkedHashMap()



It is used to create an empty HashMap object with initial capacity 16 and default fill ratio of 0.75

→ `LinkedHashMap(int initialCapacity)`

It is used to create an empty HashMap object with the specified initial capacity and default fill ratio of 0.75

→ `LinkedHashMap(int initialCapacity, float fillRatio)`

→ `LinkedHashMap(Map m)`

### **Few methods of LinkedHashMap :**

→ `boolean --- isEmpty()`

→ `int --- size()`

→ `Object --- put()`

It is used to add an entry in to the map object and it returns the old value of the specified key if exists. If old value does not exist, it returns null.

→ `Object --- get()`

→ `Object --- remove()`

→ `void --- clear()`

→ `boolean --- containsKey()`

→ `boolean --- containsValue()`

→ `Set<Object>--- keySet()`

→ `Collection --- values()`

→ `Set<Entry<Object, Object>>--- entrySet()`

### **Program example of LinkedHashMap :**

```

public class LInkedHashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> lhsmapObj = new LinkedHashMap<String, Integer>();
        System.out.println(lhsmapObj.isEmpty()); //true, map object is empty
        System.out.println(lhsmapObj.size()); //Size is zero
        System.out.println(lhsmapObj.put("anu", 23)); //it returns NULL since anu doesnot have any old value
        System.out.println(lhsmapObj.put("anu", 24)); //returns 23 which is the old value of anu
        System.out.println(lhsmapObj.isEmpty()); //False, map object is not empty
        System.out.println(lhsmapObj.size()); //1 - now size is not zero
        System.out.println(lhsmapObj.put("manu", 34)); //it returns NULL since manu doesnot have any old value
        System.out.println(lhsmapObj.put("shanu", 54)); //it returns NULL since shanu doesnot have any old value
        System.out.println(lhsmapObj.put("renu", 54)); //it returns NULL since renu doesnot have any old value
        System.out.println(lhsmapObj);
        System.out.println(lhsmapObj.remove("shanu")); //shanu is removed successfully, and value of shanu is returned i.e 54
        System.out.println(lhsmapObj);
        System.out.println(lhsmapObj.remove("manu", 26)); // returns false, becoz manu with 26 doesnt exists, remove failed
        System.out.println(lhsmapObj.remove("manu", 34)); // returns false, becoz manu with 34 exists, remove success
        //Print all keys and values using entrySet method
        Set<Entry<String, Integer>> allEntries = lhsmapObj.entrySet();
        Iterator<Entry<String, Integer>> itr = allEntries.iterator();
        while(itr.hasNext()){
            Entry<String, Integer> entry = itr.next();
            System.out.println(entry.getKey() + " : " + entry.getValue());
            //If renu key exists, update the value of renu to 100
            if (entry.getKey().equals("renu")) {
                System.out.println(" Old value of Renu -->" + entry.setValue(100)); // setValue() method returns old value of renu i.e 54
            }
        }
        System.out.println(lhsmapObj);
    }
}

```

## Output:

```

<terminated> LinkedHashMapDemo [Java Application] C:\Program Files\J
true
0
null
23
false
1
null
null
null
{shanu=54, manu=34, anu=24, renu=54}
54
{manu=34, anu=24, renu=54}
false
true
anu : 24
renu : 54
Old value of Renu -->54
{anu=24, renu=100}

```

---

## TreeMap Class :

- It is an implementation class of Map interface.
- Only one null key is allowed but multiple null values is possible.
- Insertion order is not preserved but elements will be stored based on default natural sorting order of Keys.
- Duplicate key not allowed, but values can be duplicated.
- It allows heterogeneous object.
- Underlying data structure is Red-Black tree.

## Constructors of TreeMap :

- TreeMap()
- TreeMap(Comparator c)
- TreeMap(SortedMap sm)
- TreeMap(Map m)

## Program example :

```
public class LinkedHashMapDemo {
    public static void main(String[] args) {
        TreeMap<String, Integer> treemapObj = new TreeMap<String, Integer>();
        System.out.println(treemapObj.isEmpty()); //true, map object is empty
        System.out.println(treemapObj.size()); //Size is zero
        System.out.println(treemapObj.put("anu", 23)); //it returns NULL since anu doesnot have any old value
        System.out.println(treemapObj.put("anu", 24)); //returns 23 which is the old value of anu
        System.out.println(treemapObj.isEmpty()); //False, map object is not empty
        System.out.println(treemapObj.size()); //1 - now size is not zero
        System.out.println(treemapObj.put("manu", 34)); //it returns NULL since manu doesnot have any old value
        System.out.println(treemapObj.put("shanu", 54)); //it returns NULL since shanu doesnot have any old value
        System.out.println(treemapObj.put("renu", 54)); //it returns NULL since renu doesnot have any old value
        System.out.println(treemapObj);
        System.out.println(treemapObj.remove("shanu")); //shanu is removed successfully, and value of shanu is returned i.e 54
        System.out.println(treemapObj);
        System.out.println(treemapObj.remove("manu", 26)); // returns false, becoz manu with 26 doesnt exists, remove failed
        System.out.println(treemapObj.remove("manu", 34)); // returns false, becoz manu with 34 exists, remove success
        //Print all keys and values using entrySet method
        Set<Entry<String, Integer>> allEntries = treemapObj.entrySet();
        Iterator<Entry<String, Integer>> itr = allEntries.iterator();
        while(itr.hasNext()){
            Entry<String, Integer> entry = itr.next();
            System.out.println(entry.getKey() + " : " + entry.getValue());
            //If renu key exists, update the value of renu to 100
            if (entry.getKey().equals("renu")) {
                System.out.println(" Old value of Renu -->" + entry.setValue(100)); // setValue() method returns old value of renu i.e 54
            }
        }
    }
}
```

## Output :

```
Console
<terminated> LinkedHashMapDemo [Java Application] C:\Program Files\J
true
0
null
23
false
1
null
null
null
{anu=24, manu=34, renu=54, shanu=54}
54
{anu=24, manu=34, renu=54}
false
true
anu : 24
renu : 54
  Old value of Renu -->54
{anu=24, renu=100}
```

#### Assignment :

Display the entries based on reverse alphabetical order of keys

#### Using comparator :

```
package Collections;
import java.util.Comparator;
public class MyOwnComparator implements Comparator<String>{
    @Override
    public int compare(String s1, String s2) {
        return s2.compareTo(s1);
    }
}
```

```

public class TreeMapDemo {
    public static void main(String[] args) {
        TreeMap<String, Integer> treemapObj = new TreeMap<String, Integer>(new MyOwnComparator());
        System.out.println(treemapObj.isEmpty()); //true, map object is empty
        System.out.println(treemapObj.size()); //Size is zero
        System.out.println(treemapObj.put("anu", 23)); //it returns NULL since anu doesnot have any old value
        System.out.println(treemapObj.put("anu", 24)); //returns 23 which is the old value of anu
        System.out.println(treemapObj.isEmpty()); //False, map object is not empty
        System.out.println(treemapObj.size()); //1 - now size is not zero
        System.out.println(treemapObj.put("manu", 34)); //it returns NULL since manu doesnot have any old value
        System.out.println(treemapObj.put("shanu", 54)); //it returns NULL since shanu doesnot have any old value
        System.out.println(treemapObj.put("renu", 54)); //it returns NULL since renu doesnot have any old value
        System.out.println(treemapObj);
        System.out.println(treemapObj.remove("shanu")); //shanu is removed successfully, and value of shanu is returned i.e 54
        System.out.println(treemapObj);
        System.out.println(treemapObj.remove("manu", 26)); // returns false, becoz manu with 26 doesnt exists, remove failed
        System.out.println(treemapObj.remove("manu", 34)); // returns false, becoz manu with 34 exists, remove success
        //Print all keys and values using entrySet method
        Set<Entry<String, Integer>> allEntries = treemapObj.entrySet();
        Iterator<Entry<String, Integer>> itr = allEntries.iterator();
        while(itr.hasNext()){
            Entry<String, Integer> entry = itr.next();
            System.out.println(entry.getKey() + " : " + entry.getValue());
            //If renu key exists, update the value of renu to 100
            if (entry.getKey().equals("renu")) {
                System.out.println(" Old value of Renu -->" + entry.setValue(100)); // setValue() method returns old value of renu i.e 54
            }
        }
    }
}

```

## Output :

```

Console
<terminated> TreeMapDemo [Java Application] C:\Program Files\Java\jre
true
0
null
23
false
1
null
null
null
{shanu=54, renu=54, manu=34, anu=24}
54
{renu=54, manu=34, anu=24}
false
true
renu : 54
  Old value of Renu -->54
anu : 24
{renu=100, anu=24}

```

## Exception Handling



**Exception** : Exception is an abnormal condition.

**Exception Handling**: exception handling is to maintain the normal flow of the application.

An exception normally disrupts the normal flow of the application that is why we use exception handling.

**Example:**

statement 1;

statement 2;

statement 3;//exception occurs

statement 4;

statement 5;

Suppose there are 5 statements in the program and an exception occurs at statement 3, the rest of the code will not be executed i.e. statement 4 and 5 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling.

### **Hierarchy of Java Exception classes**

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error.

**Error**: It is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc

### **Types of Java Exceptions**

**1.Checked Exception**: exceptions that are checked at compile-time by the compiler.

**2.Unchecked Exception**: exceptions that are checked at runtime by the JVM.

### **Exception Keywords**

**1.Try**: The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.

**2.Catch**:The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

**3.Finally**: The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. A finally block can not exist alone it should followed by try block or try catch block.

**4.Throw**: The "throw" keyword is used to raise an exception.

**5.Throws:**The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method

#### **signature.**

```
Try{  
  // risky code(which may raise exception)  
}catch(Exception e){  
  //handling code  
}finally{  
  //clean up code  
}
```

All 3 blocks executed in this sequence only.

#### **exmp:**

```
public static void main(String args[]){  
  try{  
    //code that may raise exception  
    int a=10/0;  
  }catch(ArithmeticException ae){  
    System.out.println(ae);  
  }  
  System.out.println("rest of the code...");  
}
```

Output: Exception in thread main java.lang.ArithmeticException:/ by zero  
rest of the code...

#### **Points to remember**

At a time only one exception occurs and at a time only one catch block is executed. All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

#### **exmp:**

```
public static void main(String[] args) {  
  
    try{  
        int a[]=new int[5];  
        a[5]=30/0;
```

```

    }
    catch(ArithmeticException e)
    {
        System.out.println("Arithmetic Exception occurs");
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("ArrayIndexOutOfBoundsException occurs");
    }
    catch(Exception e)
    {
        System.out.println("Parent Exception occurs");
    }
    System.out.println("rest of the code");
}
}

```

**Output:** Arithmetic Exception occurs  
rest of the code

### **finally block:**

Java finally block is always executed whether exception is handled or not.  
Finally block can be used to put "cleanup" code such as closing a file, closing connection etc.

```

public static void main(String args[]){
    try{
        int data=25/0;
        System.out.println(data);
    }
    catch(ArithmeticException e){System.out.println(e);}
    finally{System.out.println("finally block is always executed");}
    System.out.println("rest of the code...");
}
}

```

Output:Exception in thread main java.lang.ArithmeticException:/ by zero  
finally block is always executed  
rest of the code...

**Note:** The finally block will not be executed if program exits(either by calling System.exit() or when JVM is shutdown.or An exception arising in finally block itself.

### **Exception Propagation**



```

public class ExceptionProDemo {

    void m1()
    {
        String s=null;
        System.out.println(s.length());
    }
    void m2()
    {
        m1();
    }
    void m3()
    {
        m2();
    }
    public static void main(String[] args) {
        ExceptionProDemo epd=new ExceptionProDemo();
        try {
            epd.m3();
        } catch (NullPointerException np) {
            System.out.println(np);
        }
        System.out.println("continue");
    }
}

```

Output:

```

java.lang.NullPointerException
continue

```

In the above example exception occurs in m1() method where it is not handled,so it is propagated to m2() method where it is not handled, again it is propagated to m3() method where exception it is not handled again it is propagated to main() method where exception is handled.

So unchecked Exception can be propagated, handled in any method in call stack (either in main() method,m1() method,m2() method or m3() method). But checked exception can not be propagated, it must be handled or fixed in the same block where it is occurred.

**throw and throws keywords**

**throw** : keyword is used to explicitly throw an exception or raise an exception.

We can raise either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. It is used inside the method by creating an object of the exception class.

**throws** : It is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained. It is used with method signature by exception class name.

checked exceptions can be propagated by throws keyword.( It provides information to the caller of the method about the exception.

)

**exmp:**

```
class Demo{
    void m1() throws FileNotFoundException //declare an exception(exception class name)
    {
        throw new FileNotFoundException(); //raise an exception(exception class object)
    }
    void m2()throws FileNotFoundException
    {
        m1();
    }
    void m3()throws FileNotFoundException
    {
        m2();
    }
    public static void main(String args[]){
        Demo d=new Demo();
        Try{
            d.m3();
        }catch(FileNotFoundException fe) {
            System.out.println("exception handled");
        }
        System.out.println("continue");
    }
}
```

When main() method also doesn't handle the exception, then caller will be the JVM so the exception will be handled by the default exception handler which may simply print the exception to standard output.

### Q. Which exception should be declared

Ans) checked exception only.

Q. Can we rethrow an exception?

Yes, by throwing same exception in catch block.

### Difference between throw and throws

throw	throws
→ throw keyword is used to explicitly raise an exception.	throws keyword is used to declare an exception.
→ Checked exception cannot be propagated using throw only.	→ Checked exception can be propagated with throws.
→ Throw is followed by an Object.	→ Throws is followed by class.
→ Throw is used within the method.	→ Throws is used with the method signature.
→ You cannot throw multiple exceptions.	→ You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

### ExceptionHandling with MethodOverriding

#### -If the superclass method does not declare an exception

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

#### -If the superclass method declares an exception

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Exception Declared		
Parent	No	Yes

<b>child</b>	Both yes or no	1.should be same exception type
	If yes: any unchecked exception	2.sub class type
	If No: also fine	3. Can't be of parent type

### Inner Try catch Block

```

public class InnerTryCatchDemo {

    public static void main(String[] args) {
        try {
            try{
                String s=null;
                System.out.println(s.length());
            } catch (NullPointerException np) {
                System.out.println(np);
            }
        }
        try {

            } catch (ArithmeticException ae) {
                System.out.println(ae);
            }

        }catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("continue");
    }
}

```

Output:

```

java.lang.NullPointerException
continue

```

### Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.  
All custom Exceptions are Runtime Exception or Unchecked Exception.

Exmp:

```
package ExceptionHandling;
```

```
public class Vote {
```

```
    static void validAgeForVote(int age) throws InvalidAge
    {
        if (age<18) {
            throw new InvalidAge("not eligible for vote");
        } else {
            System.out.println("eligible for vote");
        }
    }
}
```

```
public class InvalidAge extends Exception {
```

```
    InvalidAge(String s)
    {
        System.out.println(s);
    }
}
```

```
public class TestCustomException {
```

```
    public static void main(String[] args) {
        try {
            Vote.validAgeForVote(10);
        } catch (InvalidAge ia) {

        }

    }
}
```

Output:

not eligible for vote

## Difference between final, finally and finalize

<b>final</b>	<b>finally</b>	<b>finalize</b>
Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
Final is a keyword.	Finally is a block.	Finalize is a method.

## Java File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Few Methods of File class

exists():- it checks whether folder is there or not, if there it returns true(boolean).

mkdir():-It creates the directory named by this provided pathname.(boolean)

isDirectory():-It tests whether the file denoted by this abstract pathname is a directory.(boolean)

isFile():-It tests whether the file denoted by this abstract pathname is a normal file.(boolean)

getName():-It returns the name of the file or directory denoted by this abstract pathname.(String)

isAbsolute():-It tests whether this abstract pathname is absolute.(boolean)

## Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use `FileOutputStream` class. You can write byte-oriented as well as character-oriented data through `FileOutputStream` class. But, for character-oriented data, it is preferred to use `FileWriter` than `FileOutputStream`.

### **Java FileInputStream Class**

Java `FileInputStream` class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use `FileReader` class.