JAVA FOR TESTING

INTRODUCTION JAVA

JAVA was developed by Sun Microsystems Inc in 1991, later acquired by Oracle Corporation. It was developed by James Gosling and Patrick Naughton. It is a simple programming language and a platform**.** Writing, compiling and debugging a program is easy in java. It helps to create modular programs and reusable code.

Java is a high level, robust, secured and object-oriented programming language.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API. It is called platform

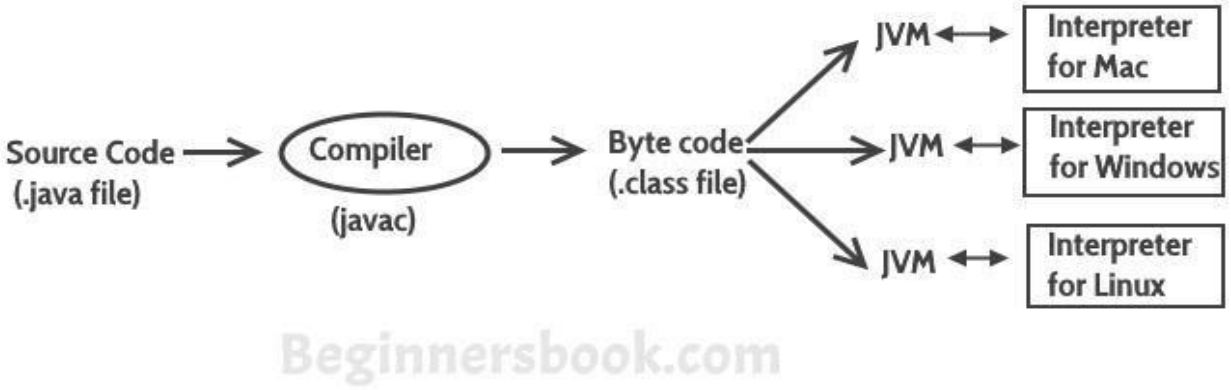
JAVA DEVELOPMENT KIT(JDK) :. As the name suggests this is complete java development kit that includes JRE (Java Runtime Environment), compilers and various tools like JavaDoc, Java debugger etc. In order to create, compile and run Java program you would need JDK installed on your computer

JAVA RUNTIME ENVIRONMENT (JRE) : JRE is a part of JDK which means that JDK includes JRE. When you have JRE installed on your system, you can run a java program however you won't be able to compile it. JRE includes JVM, browser plugins and applets support. When you only need to run a java program on your computer, you would only need JRE.

JVM (JAVA VIRTUAL MACHINE) : Java is a high level programming language. A program written in high level language cannot be run on any machine directly. First, it needs to be translated into that particular machine language. The **javac compiler** does this thing, it takes java program (.java file containing source code) and translates it into machine code (referred as byte code or .class file).

The Java Virtual machine (JVM) is the virtual machine that runs on actual machine (your computer) and executes Java byte code. The JVM doesn't understand Java source code, that's why we need to have javac compiler that compiles *.java files to obtain *.class files that contain the byte codes understood by the JVM. JVM makes java portable (write once, run anywhere). Each operating

system has different JVM, however the output they produce after execution of byte code is same across all operating systems.



Heap : Heap is a part of JVM memory where objects are allocated. JVM creates a Class object for each .class file.

Stack : Stack is a also a part of JVM memory but unlike Heap, it is used for storing temporary variables.

Garbage collection : A class instance is explicitly created by the java code and after use it is automatically destroyed by garbage collection for memory management.

FEATURES OF JAVA

Java is a platform independent language : A program that is compiled on windows can run on Linux and vice-versa. Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems. That is why we call java as platform independent language.

Java is an Object Oriented language : Object oriented programming is a way of organizing programs as collection of objects, each of which represents an instance of a class.

Simple : Java is considered as one of simple language because it does not have complex features like Operator overloading, Multiple inheritance, pointers and Explicit memory allocation.

Robust Language : Robust means reliable. Java programming language is developed in a way that puts a lot of emphasis on early checking for possible errors, that's why java compiler is able to detect errors that are not easy to detect in other programming languages. The main features of java that makes it robust are garbage collection, Exception Handling and memory allocation.

Secure : We don't have pointers and we cannot access out of bound arrays (you get ArrayIndexOutOfBoundsException if you try to do so) in java. That's why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

Java is distributed : Using java programming language we can create distributed applications. RMI(Remote Method Invocation) and EJB(Enterprise Java Beans) are used for creating distributed applications in java. In simple words: The java programs can be distributed on more than one systems that are connected to each other using internet connection. Objects on one JVM (java virtual machine) can execute procedures on a remote JVM.

Multithreading : Java supports multithreading. Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilisation of CPU**.**

Portable : java code that is written on one machine can run on another machine. The platform independent byte code can be carried to any platform for execution that makes java code portable.

CLASSPATH

Classpath is a parameter in the Java Virtual Machine or the Java compiler that specifies the location of user-defined classes and packages. The parameter may be set either on the command-line, or through an environment variable.

Install JDK →C:\Program Files\Java\jdk1.8.0_71\bin (Copy this path) → Rightclick My computer → Advanced system settings → Environment variables → Paste the copied path there.

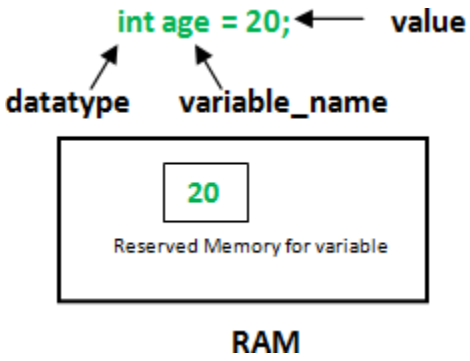
VARIABLES

A variable is the name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before they can be used.

How to declare variables?

We can declare variables in java as follows:



datatype : Type of data that can be stored in this variable.

variable_name : Name given to the variable.

value : It is the initial value stored in the variable.

***Examples**:*

```
float simpleInterest; //Declaring float variable

int time = 10, speed = 20; //Declaring and Initializing integer variable

char var = 'h'; // Declaring and Initializing character variable
```

- Variables naming cannot contain white spaces, for example: int num ber = 100; is invalid because the variable name has space in it.
- Variable name can begin with special characters such as $ and _

- As per the java coding standards the variable name should begin with a lower case letter,for example int number; For lengthy variables names that has more than one words do it like this: int smallNumber; int bigNumber; (start the second word with capital letter).

- Variable names are case sensitive in Java.

There are three types of variables in Java.

1) Static (or class) variable

2) Instance variable

3) Local variable

STATIC (OR CLASS) VARIABLE : Static variables are also known as Class variables.

- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at start of program execution and destroyed automatically when execution ends.

To access static variables, we need not to create any object of that class, we can simply access the variable as:

```
class_name.variable_name;
```

Sample Program:

```
import java.io.*;
class Emp {

  // static variable salary
  public static double salary;
```

```
  public static String name = "Harsh";
}

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

     //accessing static variable without object
     Emp.salary = 1000;
     System.out.println(Emp.name + "'s average salary:" + Emp.salary);
  }

}
```

Output:

```
Harsh's average salary:1000.0
```

Instance variable : Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.

Sample Program:

```
import java.io.*;
class Marks
{
   //These variables are instance variables.
   //These variables are in a class and are not inside any function
   int engMarks;
   int mathsMarks;
   int phyMarks;
}

class MarksDemo
{
   public static void main(String args[])
   { //first object
      Marks obj1 = new Marks();
```

```
        obj1.engMarks = 50;
        obj1.mathsMarks = 80;
        obj1.phyMarks = 90;

        //second object
        Marks obj2 = new Marks();
        obj2.engMarks = 80;
        obj2.mathsMarks = 60;
        obj2.phyMarks = 85;

        //displaying marks for first object
        System.out.println("Marks for first object:");
        System.out.println(obj1.engMarks);
        System.out.println(obj1.mathsMarks);
        System.out.println(obj1.phyMarks);

        //displaying marks for second object
        System.out.println("Marks for second object:");
        System.out.println(obj2.engMarks);
        System.out.println(obj2.mathsMarks);
        System.out.println(obj2.phyMarks);
    }
}
```

Output:

```
Marks for first object:
50
80
90
Marks for second object:
80
60
85
```

Local Variable : A variable defined within a block or method or constructor is called local variable.

- These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.

EXAMPLE:

```
public class StudentDetails
{
   public void StudentAge()
   { //local variable age
      int age = 0;
      age = age + 5;
      System.out.println("Student age is : " + age);
   }

   public static void main(String args[])
   {
      StudentDetails obj = new StudentDetails();
      obj.StudentAge();
   }
}
```

Output: Student age is : 5

In the above program the variable age is local variable to the function StudentAge(). If we use the variable age outside StudentAge() function, the compiler will produce an error as shown in below program.

Sample Program 2:

```
public class StudentDetails
{
   public void StudentAge()
   { //local variable age
      int age = 0;
      age = age + 5;
   }

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

```
    //using local variable age outside it's scope
    System.out.println("Student age is : " + age);
  }
}
```

Output: error: cannot find symbol " + age);

DATA TYPES

- **Integers:** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers:** This group includes float and double, which represent numbers with fractional precision.
- **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes boolean, which is a special type for representing true/false values.

OPERATORS

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>

relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

CONTROL STRUCTURES

When we need to execute a set of statements based on a condition then we need to use **control flow**

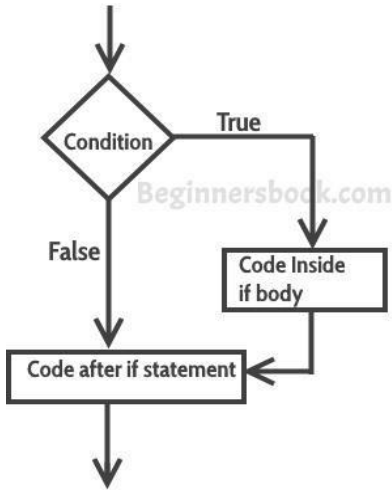
IF STATEMENT :

If statement consists a condition, followed by statement or a set of statements as shown below:

if(condition){

```
 Statement(s);
}
```

The statements gets executed only when the given condition is true. If the condition is false then the statements inside if statement body are completely ignored.



Example of if statement

```
public class IfStatementExample {

  public static void main(String args[]){
    int num=70;
    if( num < 100 ){
            /* This println statement will only execute,
             * if the above condition is true
             */
            System.out.println("number is less than 100");
    }
  }
}
```

Output: *number is less than 100*

Nested if statement in Java :

When there is an if statement inside another if statement then it is called the **nested if statement**.

The structure of nested if looks like this:

```
if(condition_1) {
  Statement1(s);
if(condition_2){
  Statement2(s);
  }
}
```

Statement1 would execute if the condition_1 is true. Statement2 would only execute if both the conditions(condition_1 and condition_2) are true.

Example of Nested if statement

```
public class NestedIfExample {

  public static void main(String args[]){
     int num=70;
          if( num < 100 ){
       System.out.println("number is less than 100");
       if(num > 50){
             System.out.println("number is greater than 50");
           }
          } } }
```

Output: *number is less than 100*

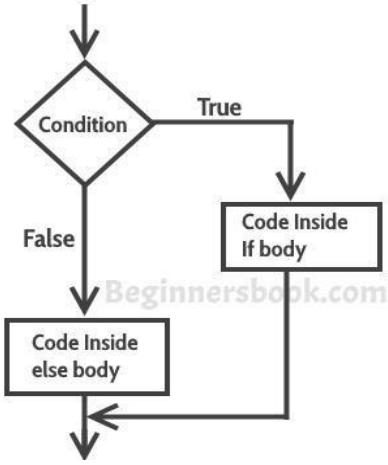
number is greater than 50

If else statement in Java

This is how an if-else statement looks:

```
if(condition) {
  Statement(s);
}
else {
  Statement(s);
}
```

The statements inside “if” would execute if the condition is true, and the statements inside “else” would execute if the condition is false.



Example of if-else statement

```
public class IfElseExample {

  public static void main(String args[]){
   int num=120;
   if( num < 50 ){
            System.out.println("num is less than 50");
   }
   else {
            System.out.println("num is greater than or equal 50");
   }
  }}
```

Output: *num is greater than or equal 50*

if-else-if Statement

f-else-if statement is used when we need to check multiple conditions. In this statement we have only one “if” and one “else”, however we can have multiple “else if”. It is also known as if else if ladder. This is how it looks:

```
if(condition_1) {
  /*if condition_1 is true execute this*/
  statement(s);
```

```
}
else if(condition_2) {
  /* execute this if condition_1 is not met and
   * condition_2 is met
   */
  statement(s);
}
else if(condition_3) {
  /* execute this if condition_1 & condition_2 are
   * not met and condition_3 is met
   */
  statement(s);
}
……
else {
  /* if none of the condition is true
   * then these statements gets executed
   */
  statement(s);
 }
```

Note: The most important point to note here is that in if-else-if statement, as soon as the condition is met, the corresponding set of statements get executed, rest gets ignored. If none of the condition is met then the statements inside "else" gets executed.

Example of if-else-if

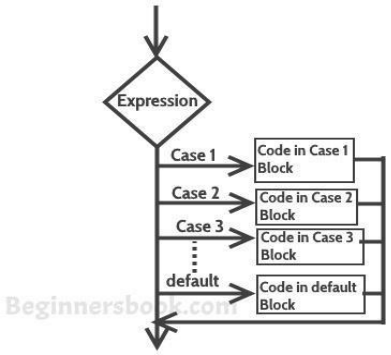
```
public class IfElseIfExample {

  public static void main(String args[]){
        int num=1234;
        if(num <100 && num>=1) {
          System.out.println("Its a two digit number");
```

```
}
else if(num <1000 && num>=100) {
  System.out.println("Its a three digit number");
}
else if(num <10000 && num>=1000) {
  System.out.println("Its a four digit number");
}
else if(num <100000 && num>=10000) {
  System.out.println("Its a five digit number");
}
else {
  System.out.println("number is not between 1 & 99999");
} }}
```

Output : *Its a four digit number*

Switch case statement :

It is used when we have number of options (or choices) and we may need to perform a different task for each choice.



The syntax of Switch case statement looks like this –

```
switch (variable or an integer expression)
{
   case constant:
   //Java code
   ;
   case constant:
   //Java code
   ;
   default:
```

```
//Java code ; }
```

Switch Case statement is mostly used with break statement even though it is optional. We will first see an example without break statement and then we will discuss switch case with break

A Simple Switch Case Example

```
public class SwitchCaseExample1 {

  public static void main(String args[]){
    int num=2;
    switch(num+2)
    {
      case 1:
              System.out.println("Case1: Value is: "+num);
            case 2:
              System.out.println("Case2: Value is: "+num);
            case 3:
              System.out.println("Case3: Value is: "+num);
      default:
              System.out.println("Default: Value is: "+num);
    }
  }
}
```

Output :

Default: Value is: 2

Break statement in Switch Case : Break statement is optional in switch case but you would use it almost every time you deal with switch case. Before we discuss about break statement, Let's have a look at the example below where we are not using the break statement:

```
public   class   SwitchCaseExample2   {

  public static void main(String args[]){
    int i=2;
    switch(i)
    {
            case 1:
            System.out.println("Case1 ");
            case 2:
```

```
        System.out.println("Case2 ");
      case 3:
      System.out.println("Case3 ");
      case 4:
   System.out.println("Case4 ");
      default:
        System.out.println("Default ");    } } }
```

Output:

Case2
Case3
Case4
Default

In the above program, we have passed integer value 2 to the switch, so the control switched to the case 2, however we don't have break statement after the case 2 that caused the flow to pass to the subsequent cases till the end. The solution to this problem is break statement.

Break statements are used when you want your program-flow to come out of the switch body. Whenever a break statement is encountered in the switch body, the execution flow would directly come out of the switch, ignoring rest of the cases

Example with break statement

```
public class SwitchCaseExample2 {

  public static void main(String args[]){
    int i=2;
    switch(i)
    {
          case 1:
            System.out.println("Case1 ");
            break;
          case 2:
            System.out.println("Case2 ");
            break;
          case 3:
            System.out.println("Case3 ");
            break;
          case 4:
       System.out.println("Case4 ");
       break;
          default:
```

```
System.out.println("Default ") } }}
```

Output : *Case2 (Now you can see that only case 2 had been executed, rest of the cases were ignored.)*

Java Continue Statement: The Java *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Syntax: jump-statement;

```
continue;
```

Example:

```
public class ContinueExample {
    public static void main(String[] args) {
    for(int i=1;i<=10;i++){
    if(i==5){
    continue;
     }
    System.out.println(i);     } } }
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Java Continue Statement with Inner Loop

It continues inner loop only if you use continue statement inside the inner loop.

Example:

```
public class ContinueExample2 {

    public static void main(String[] args) {
for(int i=1;i<=3;i++){
    for(int j=1;j<=3;j++){
```

```
        if(i==2&&j==2){
           continue;
        }
        System.out.println(i+" "+j);
    }  }   } }
```

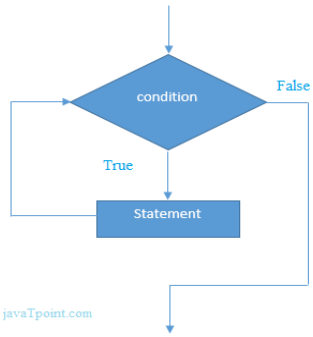
Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

Java While Loop: The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax: while*(condition){*

//code to be executed }



Example: public class *WhileExample {*

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

int *i=1;*

while*(i<=10){*

System.out.println(i);

```
i++; } }  }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Java Infinitive While Loop: If you pass **true** in the while loop, it will be infinitive while loop.

Syntax: while(true){

```
//code to be executed }
```

Example: public class *WhileExample2 {*

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

while(true){
System.out.println("infinitive while loop");
} }  }
```

Output:

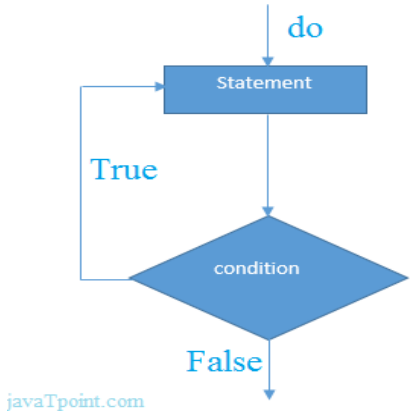
```
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
ctrl+c
```

Now, you need to press ctrl+c to exit from the program.

Java do-while Loop: The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax: do*{ //code to be executed*
*}**while**(condition);*



Example: public class *DoWhileExample {*

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

int *i=1;*

do*{*
System.out.println(i);
i++;
*}**while**(i<=10); } }*

Output:

1
2
3
4
5
6
7
8
9
10

- **Java Infinitive do-while Loop:** If you pass **true** in the do-while loop, it will be infinitive do-while loop.

Syntax: do*{*

//code to be executed

*}**while(true);***

Example: public class *DoWhileExample2 {*

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

do*{*

System.out.println("infinitive do while loop");

*}**while(true);***

} }

Output:

infinitive do while loop

infinitive do while loop

infinitive do while loop

ctrl+c

Now, you need to press ctrl+c to exit from the program

Java For Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

There are three types of for loop in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

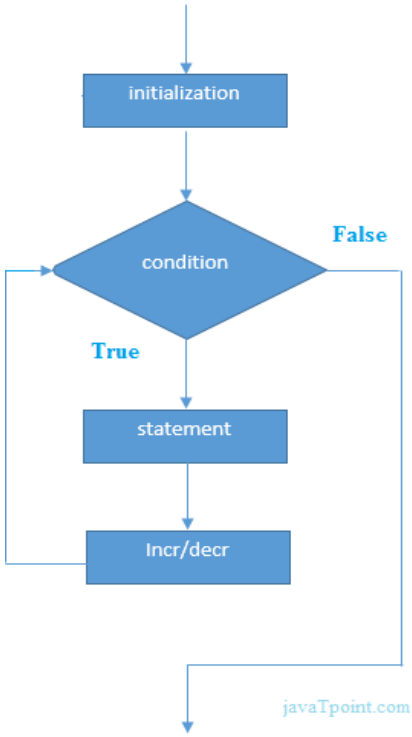
Java Simple For Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax: for*(initialization;condition;incr/decr){*

//code to be executed

}



Example:

```
public class ForExample {
public static void main(String[] args) {
   for(int i=1;i<=10;i++){
      System.out.println(i);
   } }
 }
```

Output:

1
2
3
4
5
6

7
8
9
10

Java For-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on elements basis not index. It returns element one by one in the defined variable.

Syntax: for*(Type var:array){*
//code to be executed
}

Example:

```
public class ForEachExample {
public static void main(String[] args) {
   int arr[]={12,23,44,56,78};
   for(int i:arr){
      System.out.println(i);
   } } }
```

Output:

12
23
44
56
78

Java Labeled For Loop

We can have name of each for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.

Normally, break and continue keywords breaks/continues the inner most for loop only.

Syntax: *labelname:*
for*(initialization;condition;incr/decr){*
//code to be executed
}

Example:

```
public class LabeledForExample {
public static void main(String[] args) {
   aa:
      for(int i=1;i<=3;i++){
         bb:
            for(int j=1;j<=3;j++){
               if(i==2&&j==2){
                  break aa;
               }
               System.out.println(i+" "+j);
           } } }  }
```

Output:

```
1 1
1 2
1 3
2 1
```

If you use **break bb;**, it will break inner loop only which is the default behavior of any loop.

Example:

```
public class LabeledForExample {
public static void main(String[] args) {
   aa:
      for(int i=1;i<=3;i++){
         bb:
            for(int j=1;j<=3;j++){
               if(i==2&&j==2){
                  break bb; }
               System.out.println(i+" "+j);
            }  }  }  }
```

Output:

```
1 1
1 2
1 3
```

2 1
3 1
3 2
3 3

Java Infinitive For Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

Syntax: for*(;;){*
//code to be executed
}

Example:

```
public class ForExample {
public static void main(String[] args) {
  for(;;){
    System.out.println("infinitive loop");
  } } }
```

Output: *infinitive loop*
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c

Now, you need to press ctrl+c to exit from the program.

Types of Java Comments

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1) Java Single Line Comment

The single line comment is used to comment only one line.

Syntax: *//This is single line comment*\

2) Java Multi Line Comment

The multiline comment is used to comment multiple lines of code.

Syntax: *`/* This`*
`Is`
`multi line`
`comment`
`*/`

3) Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax: *`/** This is documentation`*
`comment`
`*/`

Java Scanner class

There are various ways to read input from the keyboard, the **java.util.Scanner** class is one of them. The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values. Java Scanner class is widely used to parse text for string and primitive types using regular expression. Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

Java Scanner Example to get input from console

```
import java.util.Scanner;
class ScannerTest{
 public static void main(String args[]){
  Scanner sc=new Scanner(System.in);
  System.out.println("Enter your rollno");
  int rollno=sc.nextInt();
  System.out.println("Enter your name");
  String name=sc.next();
  System.out.println("Enter your fee");
  double fee=sc.nextDouble();
```

```
  System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
  sc.close(); }
 }
```

Output: *Enter your rollno*
111
Enter your name
Ratan
Enter
450000
Rollno:111 name:Ratan fee:450000

Java Scanner Example with delimiter

Let's see the example of Scanner class with delimiter. The \s represents whitespace.

```
import java.util.*;

public class ScannerTest2{
public static void main(String args[]){
    String input = "10 tea 20 coffee 30 tea buiscuits";
    Scanner s = new Scanner(input).useDelimiter("\\s");
    System.out.println(s.nextInt());
    System.out.println(s.next());
    System.out.println(s.nextInt());
    System.out.println(s.next());
    s.close();
}
}
```

Output:

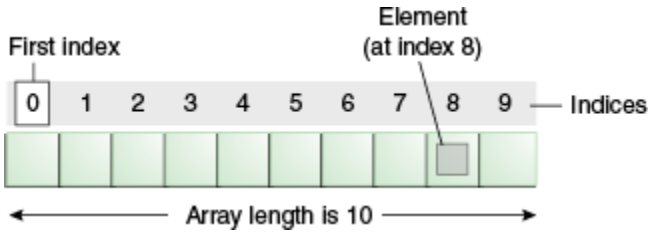
```
10
tea
20
coffee
```

JAVA ARRAY

Normally, array is a collection of similar type of elements that have contiguous memory location.

Java array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

Single Dimensional Array

Multidimensional Array

SYNTAX TO DECLARE AN ARRAY IN JAVA

dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];

INSTANTIATION OF AN ARRAY IN JAVA

arrayRefVar=new datatype[size];

EXAMPLE OF SINGLE DIMENSIONAL JAVA ARRAY

simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

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

```
{
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
  //printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
  }
}
```

Output: *10*
20
70
40
50

Declaration, Instantiation And Initialization Of Java Array

int a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

```
class Testarray1{
public static void main(String args[]){

int a[]={33,3,4,5};//declaration, instantiation and initialization

//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]); }}
```

Output*: 33*
3
4
5

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];

System.out.println(min); }
public static void main(String args[]){
int a[]={33,3,4,5};
min(a);//passing array to method }}
```

Output: *3*

MULTIDIMENSIONAL ARRAY IN JAVA

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
```

```
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional java array

Example to declare, instantiate, initialize and print the 2Dimensional array.

```
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//printing 2D array
 for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}
}
```

Output: 1 2 3
2 4 5
4 4 5

JAVA SORTING

```
public class BubbleSortExample {

  static void bubbleSort(int[] arr) {
```

```
    int n = arr.length;
    int temp = 0;
     for(int i=0; i < n; i++){
          for(int j=1; j < (n-i); j++){
               if(arr[j-1] > arr[j]){
                    //swap elements
                    temp = arr[j-1];
                    arr[j-1] = arr[j];
                    arr[j] = temp;
               }
                 }    }  }
  public static void main(String[] args) {
          int arr[] ={3,60,35,2,45,320,5};

          System.out.println("Array Before Bubble Sort");
          for(int i=0; i < arr.length; i++){
               System.out.print(arr[i] + " ");
          }
          System.out.println();

          bubbleSort(arr);//sorting array elements using bubble sort

          System.out.println("Array After Bubble Sort");
          for(int i=0; i < arr.length; i++){
               System.out.print(arr[i] + " ");
          }
}
 }
```

Output:

Array Before Bubble Sort

3 60 35 2 45 320 5

Array After Bubble Sort

2 3 5 35 45 60 320

STRING

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

char*[] ch={'j','a','v','a','t','p','o','i','n','t'};*
*String s=**new** String(ch);*

is same as: *String s="javatpoint";*

Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc

What is String in java

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

How to create String object?

- String Literal
- String new keyword

1) String Literal

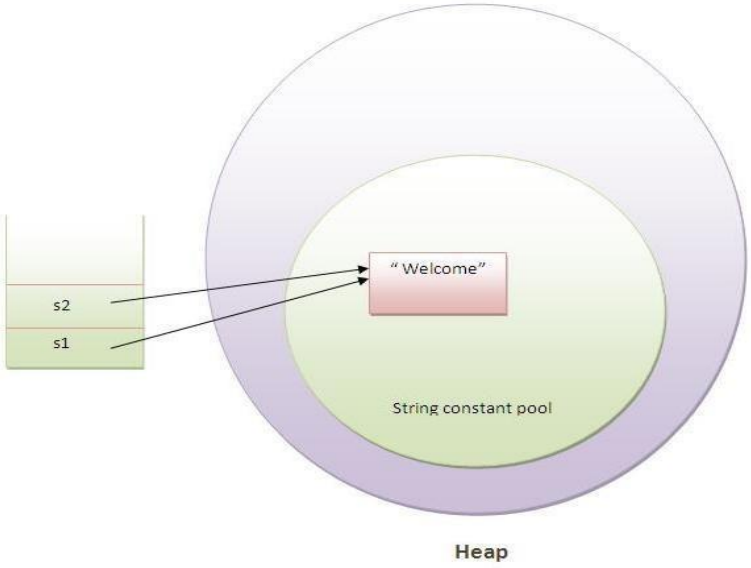
Java String literal is created by using double quotes. For Example:

String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example: String s1="Welcome";

String s2="Welcome";//will not create new instance



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

Why java uses concept of string literal?

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

2) By new keyword

*String s=**new** String("Welcome");//*creates two objects and one reference variable

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

Java String Example

***public class** StringExample{*

```
public static void main(String args[]){
String s1="java";//creating string by java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3); } }
```

Output: *java*

Strings

example

STRING METHODS :

charAt() :

```
public class CharAtExample{
public static void main(String args[]){
String name="javatpoint";
char ch=name.charAt(4);//returns the char value at the 4th index
System.out.println(ch);
}

}
```

Output: t

compareTo() :

```
public class CompareToExample{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="meklo";
String s4="hemlo";
String s5="flag";
System.out.println(s1.compareTo(s2));
//0 because both are equal
```

```
System.out.println(s1.compareTo(s3));
//-5 because "h" is 5 times lower than "m"
System.out.println(s1.compareTo(s4));
//-1 because "l" is 1 times lower than "m"
System.out.println(s1.compareTo(s5));
//2 because "h" is 2 times greater than "f"
}
}
```

Output: 0
-5
-1
2

equals() :

```
public class EqualsExample{
public static void main(String args[]){
String s1="javatpoint"; String s2="javatpoint"; String s3="JAVATPOINT"; String s4=
"python"; System.out.println(s1.equals(s2));//true because content and case is same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
}
}
```

Output : true
false
false

trim() :

```
public static void main(String args[]){
String s1=" hello string ";
System.out.println(s1+"javatpoint");//without trim()
System.out.println(s1.trim()+"javatpoint");//with trim()
}
}
```

Output : hello string javatpoint
hello stringjavatpoint

substring() :

```
public class SubstringExample{
public static void main(String args[]){
String s1="javatpoint";
System.out.println(s1.substring(2,4));//returns va
System.out.println(s1.substring(2));//returns vatpoint
}
}
```

Output :
```
va
vatpoint
```

concat() :

```
public class ConcatExample{
public static void main(String args[]){
String s1="java string";
s1.concat("is immutable");
System.out.println(s1);
s1=s1.concat(" is immutable so assign it explicitly"); System.out.println(s1);  }}
```

Output : java string
java string is immutable so assign it explicitly

replace():

```
public class ReplaceExample1{
public static void main(String args[]){
String s1="javatpoint is a very good website";
String replaceString=s1.replace('a','e');//replaces all occurrences of 'a' to 'e'
System.out.println(replaceString);
}
}
```

Output : jevetpoint is e very good website

MUTABLE AND IMMUTABLE STRING :

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

EXAMPLES : ***class*** *Testimmutablestring{*

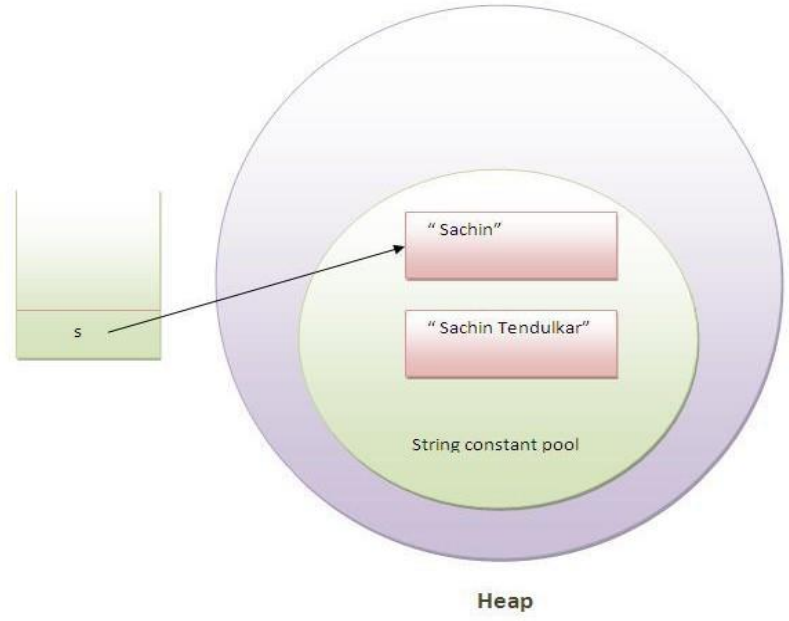
public static void *main(String args[]){*

```
String s="Sachin";
s.concat(" Tendulkar");//concat() method appends the string at the end
System.out.println(s);//will print Sachin because strings are immutable objects
} }
```

Output : Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.

As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".



But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

```
Class Testimmutablestring1{
public static void main(String args[]){
String s="Sachin";
s=s.concat(" Tendulkar");
```

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

Output: Sachin Tendulkar

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

Why string objects are immutable in java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

WHAT IS MUTABLE STRING?

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
```

```
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
} }
```

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
} }
```

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
} }
```

5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
} }
```

STRING BUFFER : Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Constructor	Description
StringBuffer()	*-- creates an empty string buffer with the initial capacity of 16.*
StringBuffer(String str)	*-- creates a string buffer with the specified string*
StringBuffer(int capacity)	*-- creates an empty string buffer with the specified capacity as length.*

STRING BUILDER : Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Constructor	Description
StringBuilder()	-- creates an empty string Builder with the initial capacity of 16.
StringBuilder(String str)	-- creates a string Builder with the specified string.
StringBuilder(int length)	-- creates an empty string Builder with the specified capacity as Length.

STRINGTOKENIZER IN JAVA

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	-- creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	-- creates StringTokenizer with specified string and delimeter.

METHODS OF STRINGTOKENIZER CLASS

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimeter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

SIMPLE EXAMPLE OF STRINGTOKENIZER CLASS

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
  StringTokenizer st = new StringTokenizer("my name is khan"," ");       while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
   }    } }
```

Output:*my*

name

is

khan

EXAMPLE OF NEXTTOKEN(STRING DELIM) METHOD OF STRINGTOKENIZER CLASS

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

```
public class Test {
  public static void main(String[] args) {
     StringTokenizer st = new StringTokenizer("my,name,is,khan");

    // printing next token       System.out.println("Next token is : " + st.nextToken(","));
  }
}
```

Output:*Next token is : my*

OBJECT ORIENTED PROGRAMMING (OOP)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

OBJECT: Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

CLASS : Collection of objects is called class. It is a logical entity.

- **INHERITANCE :** *When one object acquires all the properties and behaviours of parent object* ***i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.***

Syntax: Inheritance in Java

To inherit a class we use extends keyword. Here class A is child class and class B is parent class.

class A extends B{ ……. }

Inheritance Example : In this example, we have a parent class Teacher and a child class MathTeacher. In the MathTeacherclass we need not to write the same code which is already present in the present class. Here we have college name, designation and does() method that is common for all the teachers, thus MathTeacher class does not need to write this code, the common data members and methods can inherited from the Teacher class.

```
class Teacher {
  String designation = "Teacher";
  String college = "Book";

  void does(){
          System.out.println("Teaching");
  } }
public class MathTeacher extends Teacher{
  String mainSubject = "Maths";
  public static void main(String args[]){
    MathTeacher obj = new MathTeacher();
    System.out.println(obj.college);
    System.out.println(obj.designation);
    System.out.println(obj.mainSubject);
    obj.does();
  } }
```

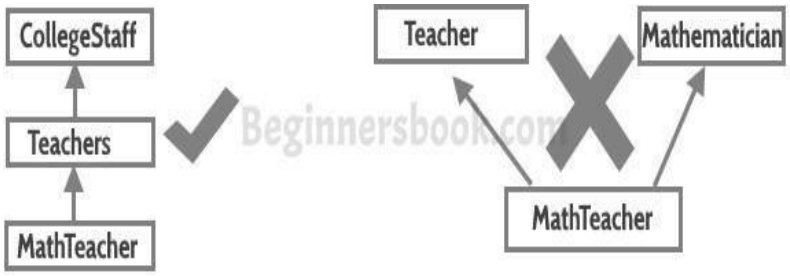
Output : *Book*

Teacher

Maths

Teaching

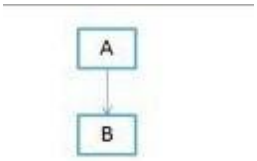
Note: Multi-level inheritance is allowed in Java but **not multiple inheritance**



TYPES OF INHERITANCE

1) Single Inheritance

Single inheritance is damn easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



(a) Single Inheritance

Single Inheritance example program in Java

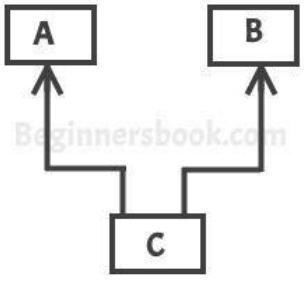
```
Class A
{
  public void methodA()
  {
```

```
    System.out.println("Base class method"); } }
Class B extends A
{
  public void methodB()
  {
    System.out.println("Child class method");
  }
  public static void main(String args[])
  {
    B obj = new B();
    obj.methodA(); //calling super class method
    obj.methodB(); //calling local method
  }
}
```

2) Multiple Inheritance

"**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.

When one class extends more than one classes then this is called **multiple inheritance**. For example: Class C extends class A and B then this type of inheritance is known as multiple inheritance. Java doesn't allow multiple inheritance. why java doesn't allow multiple inheritance and how we can use interfaces instead of classes to achieve the same purpose.



Multiple Inheritance

Can We Implement More Than One Interfaces In A Class

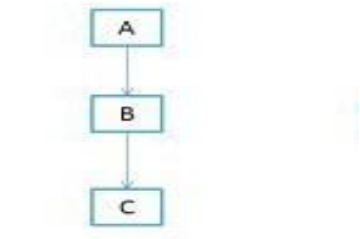
Yes, we can implement more than one interfaces in our program because that doesn't cause any ambiguity(see the explanation below).

```
interface X
{
  public void myMethod();
}
interface Y
{
  public void myMethod();
}
class JavaExample implements X, Y
{
  public void myMethod()
  {
     System.out.println("Implementing more than one interfaces"); }
  public static void main(String args[]){
              JavaExample obj = new JavaExample();
              obj.myMethod();
  }
}
```

Output : *Implementing more than one interfaces*

3) Multilevel Inheritance

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A. F



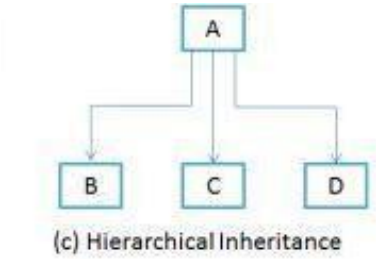
(d) Multilevel Inheritance

Multilevel Inheritance example program in Java

```
Class X{

public void methodX()

  {
    System.out.println("Class X method");
  }
}
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
} }
Class Z extends Y
{
  public void methodZ()
  {
    System.out.println("class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX(); //calling grand parent class method
    obj.methodY(); //calling parent class method
    obj.methodZ(); //calling local method
 }
 }
```

4) Hierarchical Inheritance

In such kind of inheritance one class is inherited by many **sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.



(c) Hierarchical Inheritance

Example of Hierarchical Inheritance

```
class A
{
  public void methodA()
  {
    System.out.println("method of Class A");
  }
}
class B extends A
{
  public void methodB()
  {
    System.out.println("method of Class B");
  }
}
class C extends A
{
 public void methodC()
 {
   System.out.println("method of Class C");
 }
}
class D extends A
{
```

```
 public void methodD()
 {
   System.out.println("method of Class D");
 }
}
class JavaExample
{
 public static void main(String args[])
 {
   B obj1 = new B();
   C obj2 = new C();
   D obj3 = new D();
   //All classes can access the method of class A
   obj1.methodA();
   obj2.methodA();
   obj3.methodA();
 }
}
```

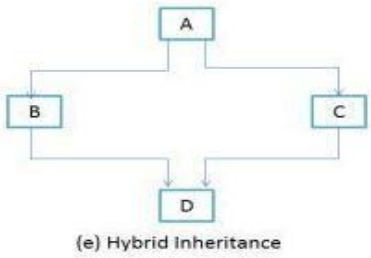
Output: *method of Class A*

method of Class A

method of Class A

5) Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple** inheritance. A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



(e) Hybrid Inheritance

Program: This example is just to demonstrate the hybrid inheritance in Java. Although this example is meaningless, you would be able to see that how we have implemented two types of inheritance(single and hierarchical) together to form hybrid inheritance

Class A and B extends class C → Hierarchical inheritance
Class D extends class A → Single inheritance

```
class C
{
  public void disp() {
        System.out.println("C"); } }
class A extends C
{
  public void disp()
  {
        System.out.println("A");
  } }

class B extends C
{
  public void disp()
  {
        System.out.println("B");
  } }

class D extends A
{
  public void disp()
  {
        System.out.println("D");
  }
  public static void main(String args[]){

        D obj = new D();
        obj.disp(); } }
```

Output : D

- **POLYMORPHISM:** When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism. Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Types of Polymorphism

1) Static Polymorphism

2) Dynamic Polymorphism

Static Polymorphism:

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading can be considered as static polymorphism example.

Method Overloading: This allows us to have more than one methods with same name in a class that differs in signature.

```
class DisplayOverloading
{
   public void disp(char c)
   {
      System.out.println(c);
   }
   public void disp(char c, int num)
   {
      System.out.println(c + " "+num);
   } }
public class ExampleOverloading
{
  public static void main(String args[])
  {
```

```
    DisplayOverloading obj = new DisplayOverloading();
    obj.disp('a');
    obj.disp('a',10);
  }
}
```

Output : *a*
a 10

Dynamic Polymorphism

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime rather, thats why it is called runtime polymorphism.

Example : *class Animal{*

```
public void animalSound(){

        System.out.println("Default Sound");
  }
}
public class Dog extends Animal{

  public void animalSound(){
        System.out.println("Woof");
  }
  public static void main(String args[]){
        Animal obj = new Dog();
        obj.animalSound(); }
}
```

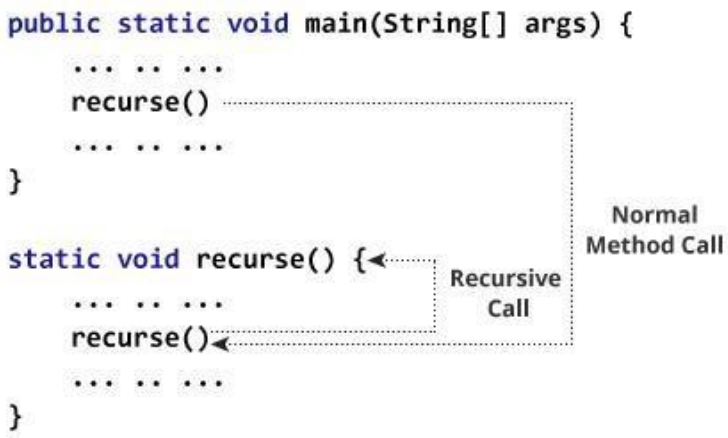
Output : *Woof*

Since both the classes, child class and parent class have the same method animalSound. Which of the method will be called is determined at runtime by JVM.

RECURSION

A method that calls itself is known as a recursive method. And, this technique is known as recursion. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

HOW RECURSION WORKS?



In the above program, recurse() method is called from inside the main method at first (normal method call).

Also, recurse() method is called from inside the same method, recurse(). This is a recursive call. The recursion continues until some condition is met to prevent it from execution. If not, infinite recursion occurs. Hence, to prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't

ADVANTAGES AND DISADVANTAGES OF RECURSION

When a recursive call is made, new storage location for variables are allocated on the stack.

As, each recursive call returns, the old variables and parameters are removed from the stack. Hence,

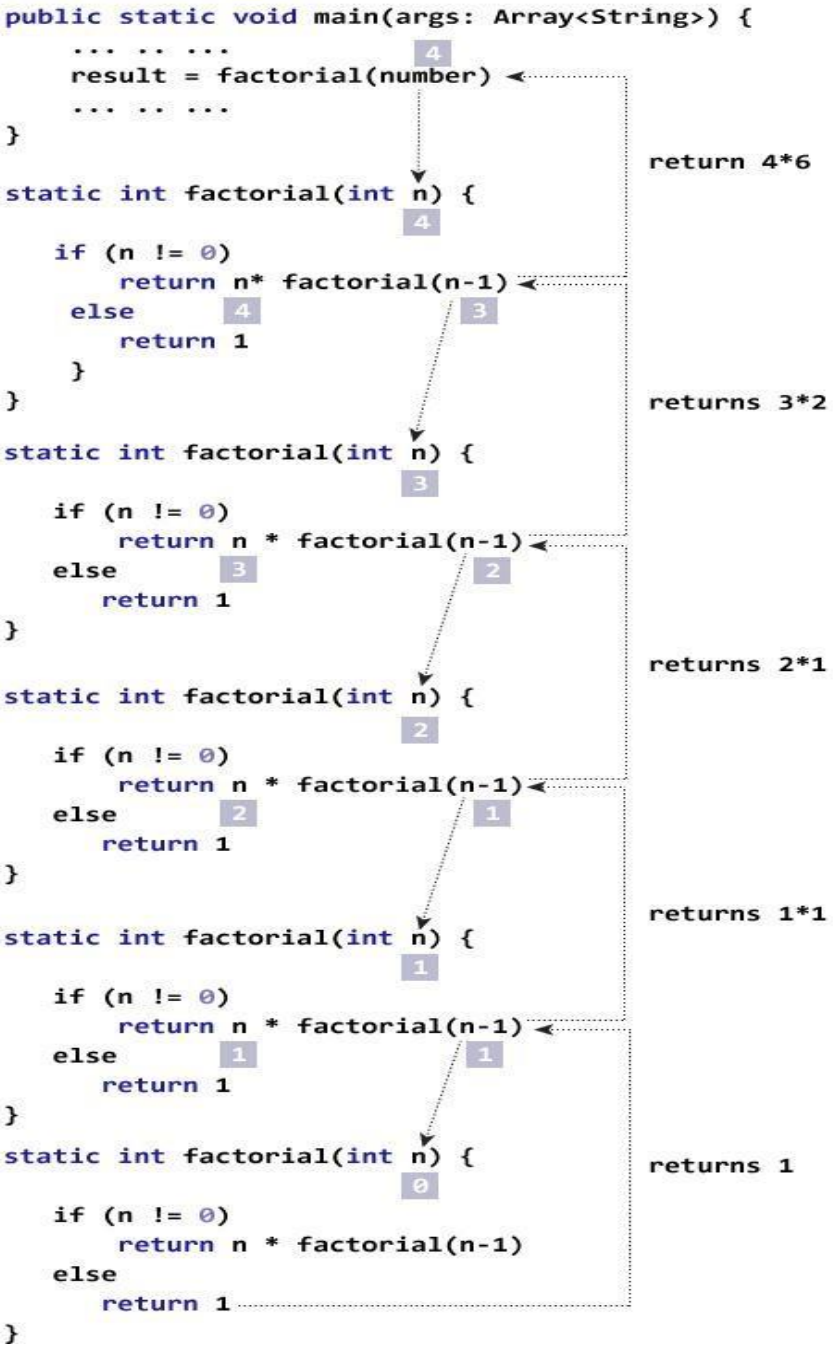
recursion generally use more memory and are generally slow. On the other hand, recursive solution is much simpler and takes less time to write, debug and maintain.

Example: Factorial of a Number Using Recursion

```
class Factorial {

   static int factorial( int n ) {

      if (n != 0)

         return n * factorial(n-1); // recursive call

      else

         return 1;

   }

   public static void main(String[] args) {

      int number = 4, result;

      result = factorial(number);

      System.out.println(number + " factorial = " + result); } }
```

When you run above program, the output will be: 4 factorial = 24 Initially, factorial() is called from the main() method with number passed as an argument. Inside factorial() method, the value of n is 4 initially. During the next recursive call, 3 is passed to the factorial() method. This process

continues until n is equal to 0. When n is equal to 0, if condition fails and the else part is executed which returns 1, and accumulated result is passed to the main() method. This figure will give you a better idea on how the above program works.



- **ABSTRACTION :**

  Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

  In java, we use abstract class and interface to achieve abstraction.

Abstract Class and methods in OOPs Concepts

Abstract method:

1) A method that is declared but not defined. Only method signature no body.
2) Declared using the abstract keyword
3) Example : abstract public void playInstrument();

4) Used to put some kind of compulsion on the class who inherits the class has abstract methods. The class that inherits must provide the implementation of all the abstract methods of parent class else declare the subclass as abstract.
5) These cannot be abstract

- Constructors
- Static methods
- Private methods
- Methods that are declared “final”

Abstract Class

An abstract class outlines the methods but not necessarily implements all the methods.

```
abstract class A{
  abstract void myMethod();
  void anotherMethod(){
    //Does something
  } }
```

Example of Abstract class and Methods

```
//abstract class
abstract class Animal{
  //abstract method
  public abstract void animalSound();
}
public class Dog extends Animal{

  public void animalSound(){
        System.out.println("Woof");
  }
  public static void main(String args[]){
        Animal obj = new Dog();
        obj.animalSound();
  }
}
```

Output: Woof

INTERFACES IN JAVA

An interface is a blueprint of a class, which can be declared by using **interface** keyword. Interfaces can contain only constants and abstract methods (methods with only signatures no body).Like abstract classes, Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces. Interface is a common way to achieve full abstraction in Java.

Note:

1. Java does not support Multiple Inheritance, however a class can implement more than one interfaces
2. Interface is similar to an abstract class but it contains only abstract methods.
3. Interfaces are created by using interface keyword instead of the keyword class
4. We use implements keyword while implementing an interface(similar to extending a class with extends keyword)

Interface: Syntax

class ClassName extends Superclass implements Interface1, Interface2, ...

EXAMPLE OF AN INTERFACE :

```
interface MyInterface
{
  /* compiler will treat them as:
   * public abstract void method1();
   * public abstract void method2();
   */
  public void method1();
  public void method2();
}
class Demo implements MyInterface
{
  /* This class must have to implement both the abstract methods
   * else you will get compilation error
   */
  public void method1()
  {
        System.out.println("implementation of method1");
  }
  public void method2()
  {
        System.out.println("implementation of method2");
  }
  public static void main(String arg[])
  {
        MyInterface obj = new Demo();
        obj.method1();
  }
}
```

Output : *implementation of method1*

	Abstract Class	Interface
1	An abstract class can extend only one class or one abstract class at a time	An interface can extend any number of interfaces at a time
2	An abstract class can extend another concrete (regular) class or abstract class	An interface can only extend another interface
3	An abstract class can have both abstract and concrete methods	An interface can have only abstract methods
4	In abstract class keyword "abstract" is mandatory to declare a method as an abstract	In an interface keyword "abstract" is optional to declare a method as an abstract
5	An abstract class can have protected and public abstract methods	An interface can have only have public abstract methods
6	An abstract class can have static, final or static final variable with any access specifier	interface can only have public static final (constant) variable

Difference No.1: Abstract class can extend only one class or one abstract class at a time

```
class Example1{
  public void display1(){
    System.out.println("display1 method"); } }
abstract class Example2{
  public void display2(){
    System.out.println("display2 method");
  }
}
abstract class Example3 extends Example1{
  abstract void display3();
}
class Example4 extends Example3{
  public void display3(){
    System.out.println("display3 method");
```

```
  }
}
class Demo{
  public static void main(String args[]){
     Example4 obj=new Example4();
     obj.display3();
 }
}
```

Output: display3 method

Interface can extend any number of interfaces at a time

```
//first interface

interface Example1{

   public void display1();
}
//second interface

interface Example2 {
   public void display2();
}
//This interface is extending both the above interfaces
interface Example3 extends Example1,Example2{
}
class Example4 implements Example3{
   public void display1(){
      System.out.println("display2 method");
   }
   public void display2(){
      System.out.println("display3 method");
   } }
class Demo{
   public static void main(String args[]){
      Example4 obj=new Example4();
      obj.display1();
 }
}
```

Output : display2 method

Difference No.2: Abstract class can be extended(inherited) by a class or an abstract class

```
class Example1{
  public void display1(){
    System.out.println("display1 method");
  }
}
abstract class Example2{
  public void display2(){
     System.out.println("display2 method");
  }
}
abstract class Example3 extends Example2{
  abstract void display3();
}
class Example4 extends Example3{
  public void display2(){
     System.out.println("Example4-display2 method");
  }
  public void display3(){
     System.out.println("display3 method");
  }
}
class Demo{
  public static void main(String args[]){
     Example4 obj=new Example4();
     obj.display2();
  }
}
```

Output : *Example4-display2 method*

Interfaces can be extended only by interfaces. Classes has to implement them instead of extend

```
interface Example1{
   public void display1(); }
interface Example2 extends Example1{ }
class Example3 implements Example2{
  public void display1(){
    System.out.println("display1 method");
  }
}
class Demo{
  public static void main(String args[]){
```

```
    Example3 obj=new Example3();
    obj.display1();
  }
}
```

Output : display1 method

Difference No.3: Abstract class can have both abstract and concrete methods

```
abstract class Example1 {
  abstract void display1();
  public void display2(){
   System.out.println("display2 method");
  }
}
class Example2 extends Example1{
  public void display1(){
    System.out.println("display1 method");
  }
}
class Demo{
  public static void main(String args[]){
   Example2 obj=new Example2();
   obj.display1();
  }
}
```

Interface can only have abstract methods, they cannot have concrete methods

```
interface Example1{
  public abstract void display1();
}
class Example2 implements Example1{
  public void display1(){
    System.out.println("display1 method");
  }
}
class Demo{
  public static void main(String args[]){
    Example2 obj=new Example2();
    obj.display1();
  }
}
```

***Output** : display1 method*

Difference No.4: In abstract class, the keyword 'abstract' is mandatory to declare a method as an abstract

```
abstract class Example1{
  public abstract void display1();
}

class Example2 extends Example1{
  public void display1(){
    System.out.println("display1 method");
  }
  public void display2(){
    System.out.println("display2 method");
  }
}
class Demo{
  public static void main(String args[]){
     Example2 obj=new Example2();
     obj.display1();
  }
}
```

In interfaces, the keyword 'abstract' is optional to declare a method as an abstract because all the methods are abstract by default

```
interface Example1{
   public void display1();
}
class Example2 implements Example1{
   public void display1(){
      System.out.println("display1 method");
   }
   public void display2(){
      System.out.println("display2 method");
   }
}

class Demo{
  public static void main(String args[]){
     Example2 obj=new Example2();
     obj.display1();
  }
}
```

difference No.5: Abstract class can have protected and public abstract methods

```
abstract class Example1{
  protected abstract void display1();
  public abstract void display2();
  public abstract void display3();
}
class Example2 extends Example1{
  public void display1(){
     System.out.println("display1 method");
  }
  public void display2(){
    System.out.println("display2 method");
  }
  public void display3(){
    System.out.println("display3 method");
  }
}
class Demo{
  public static void main(String args[]){
    Example2 obj=new Example2();
    obj.display1();
  }
}
```

Interface can have only public abstract methods

```
interface Example1{
  void display1();
}
class Example2 implements Example1{
  public void display1(){
    System.out.println("display1 method");
  }
  public void display2(){
    System.out.println("display2 method");
  }
}
class Demo{
  public static void main(String args[]){
     Example2 obj=new Example2();
     obj.display1();
}
}
```

Difference No.6: Abstract class can have static, final or static final variables with any access specifier

```
abstract class Example1{
  private int numOne=10;
  protected final int numTwo=20;
  public static final int numThree=500;
  public void display1(){
    System.out.println("Num1="+numOne);
  }
}
class Example2 extends Example1{
  public void display2(){
    System.out.println("Num2="+numTwo);
    System.out.println("Num2="+numThree);
  }
}
class Demo{
  public static void main(String args[]){
    Example2 obj=new Example2();
    obj.display1();
    obj.display2();
  }
}
```

Interface can have only public static final (constant) variable

```
interface Example1{
  int numOne=10;
}
class Example2 implements Example1{
  public void display1(){
    System.out.println("Num1="+numOne);
  }
}
class Demo{
  public static void main(String args[]){
    Example2 obj=new Example2();
    obj.display1();
  }
 }
```

- ***ENCAPSULATION :*** *Binding (or wrapping) code and data together into a single unit is known as encapsulation****. For example: capsule, it is wrapped with different medicines.***

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Simple example of encapsulation in java

```
//save as Student.java
package com.javatpoint;
public class Student{
private String name;

public String getName(){
return name;
}
public void setName(String name){
this.name=name
} }
//save as Test.java
package com.javatpoint;
class Test{
public static void main(String[] args){
Student s=new Student();
s.setName("vijay");
System.out.println(s.getName());
} }
```

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: *vijay*

ACCESSORS AND MUTATORS

Introduction

In Java **Accessors** are used to get the value of a private field and **Mutators** are used to set the value of a private field. Accessors are also known as getters and mutators are also known as setters. If we have declared the variables as private then they would not be accessible by all so we need to use getter and setter methods.

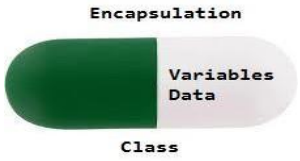
Accessors

An Accessor method is commonly known as a get method or simply a getter. A property of the object is returned by the accessor method. They are declared as public. A naming scheme is followed by accessors, in other words they add a word to get in the start of the method name. They are used to return the value of a private field. The same data type is returned by these methods depending on their private field.

Mutators

A Mutator method is commonly known as a set method or simply a setter. A Mutator method mutates things, in other words change things. It shows us the principle of encapsulation. They are also known as modifiers. They are easily spotted because they started with the word set. They are declared as public. Mutator methods do not have any return type and they also accept a parameter of the same data type depending on their private field. After that it is used to set the value of the private field.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
- Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.




```
// Java program to demonstrate encapsulation
public class Encapsulate
{
   // private variables declared these can only be accessed by public methods of class
   private String geekName;
   private int geekRoll;
   private int geekAge;

   // get method for age to access
   // private variable geekAge
   public int getAge()
   {
     return geekAge;
   }
   // get method for name to access
   // private variable geekName
   public String getName()
   {
     return geekName;
   }
   // get method for roll to access
   // private variable geekRoll
   public int getRoll()
   {
      return geekRoll;
```

```
}
// set method for age to access
// private variable geekage
public void setAge( int newAge)
{
  geekAge = newAge;
}
// set method for name to access
// private variable geekName
public void setName(String newName)
{
  geekName = newName;
}
// set method for roll to access
// private variable geekRoll
public void setRoll( int newRoll)
{
  geekRoll = newRoll;
} }
```

In the above program the class EncapsulateDemo is encapsulated as the variables are declared as private. The get methods like getAge() , getName() , getRoll() are set as public, these methods are used to access these variables. The setter methods like setName(), setAge(), setRoll() are also declared as public and are used to set the values of the variables.

The program to access variables of the class EncapsulateDemo is shown below:

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

```
    Encapsulate obj = new Encapsulate();

    // setting values of the variables
    obj.setName("Harsh");
    obj.setAge(19);
    obj.setRoll(51);

    // Displaying values of the variables
    System.out.println("Geek's name: " + obj.getName());
    System.out.println("Geek's age: " + obj.getAge());
    System.out.println("Geek's roll: " + obj.getRoll());

    // Direct access of geekRoll is not possible
    // due to encapsulation
    // System.out.println("Geek's roll: " + obj.geekName);
  }
}
```

OUTPUT:

Geek's name: Harsh

Geek's age: 19

Geek's roll: 51

Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as

read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program

- **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

CONSTRUCTOR IN JAVA

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

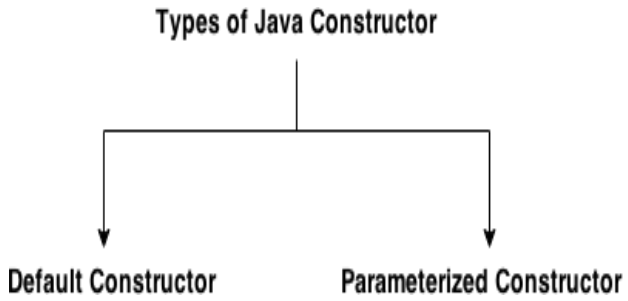
Rules for creating java constructor

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor : <class_name>(){}

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
 }
```

Output : Bike is created

Java parameterized constructor

A constructor that have parameters is known as parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
class Student4{
   int id;
   String name;

   Student4(int i,String n){
   id = i;
   name = n;
```

```
  }
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  Student4 s1 = new Student4(111,"Karan");
  Student4 s2 = new Student4(222,"Aryan");
  s1.display();
  s2.display();
 }
}
```

Output: 111 Karan
222 Aryan

CONSTRUCTOR OVERLOADING IN JAVA

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
class Student5{
  int id;
  String name;
  int age;
  Student5(int i,String n){
  id = i;
  name = n;
  }
  Student5(int i,String n,int a){
  id = i;
  name = n;
  age=a;
  }
  void display(){System.out.println(id+" "+name+" "+age);}

  public static void main(String args[]){
  Student5 s1 = new Student5(111,"Karan");
```

```
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

OUTPUT: 111 Karan 0

222 Aryan 25

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

ACCESS CONTROL MODIFIERS

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

1) PRIVATE ACCESS MODIFIER

The private access modifier is accessible only within class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}
 public class Simple{
 public static void main(String args[]){
  A obj=new A();
  System.out.println(obj.data);//Compile Time Error
  obj.msg();//Compile Time Error
  }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
  A obj=new A();//Compile Time Error
 }
}
```

If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello"); } }
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error } }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) PROTECTED ACCESS MODIFIER

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
```

```
package pack;
public class A{
protected void msg(){System.out.println("Hello");} }
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
  B obj = new B();
   obj.msg();
  }
  }
```

Output:Hello

4) PUBLIC ACCESS MODIFIER

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg(); } }
```

***Output**:Hello*

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

GENERICS IN JAVA

Generics in Java is similar to templates in C++. The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

1. Code Reuse: We can write a method/class/interface once and use for any type we want.
2. Type Safety : Generics make errors to appear compile time than at run time .
    - Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()

Note: In Parameter type we can not use primitives like
    'int','char' or 'double'.
```

// A Simple Java program to show working of user defined
// Generic classes

```
// We use < > to specify Parameter type
class Test<T>
{
   // An object of type T is declared
   T obj;
```

```
    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        // instance of Integer type
        Test <Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test <String> sObj = new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
    }
}
```

OUTPUT: *15*

```
GeeksForGeeks
```

```
        // A Simple Java program to demonstrate that NOT using
        // generics can cause run time exceptions
        import java.util.*;
        class Test
        {
            public static void main(String[] args)
            {
                // Creatinga an ArrayList without any type specified
                ArrayList al = new ArrayList();
                al.add("Sachin");
                al.add("Rahul");
                al.add(10); // Compiler allows this

                String s1 = (String)al.get(0);
                String s2 = (String)al.get(1);
                // Causes Runtime Exception
                String s3 = (String)al.get(2);
        }
        }
```

OUTPUT:

```
Exception in thread "main" java.lang.ClassCastException:
  java.lang.Integer cannot be cast to java.lang.String
        at Test.main(Test.java:19)
```

We can also pass multiple Type parameters in Generic classes.

```
// A Simple Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
   T obj1; // An object of type T
   U obj2; // An object of type U

   // constructor
   Test(T obj1, U obj2)
   {
      this.obj1 = obj1;
      this.obj2 = obj2;
   }

   // To print objects of T and U
   public void print()
   {
      System.out.println(obj1);
      System.out.println(obj2);
   }
}

// Driver class to test above
class Main
{
   public static void main (String[] args)
   {
      Test <String, Integer> obj =
         new Test<String, Integer>("GfG", 15);

      obj.print();
   } }
```

Output: *GfG*

15

JAVA INNER CLASSES

Java inner class or nested class is a class which is declared inside the class or interface.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
    1. Member inner class
    2. Anonymous inner class
    3. Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

- **JAVA MEMBER INNER CLASS**

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax: ***class*** *Outer{*
//code
class *Inner{*
//code
} }

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 } }
```

Output: *data is 30*

- **JAVA ANONYMOUS INNER CLASS**

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Syntax: The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

```
// Test can be interface,abstract/concrete class
```

```
Test t = new Test()
{
  // data members and methods
  public void test_method()
  {
    ........
    ........
   }
};
```

To understand anonymous inner class, let us take a simple program

```
//Java program to demonstrate need for Anonymous Inner class
interface Age
{
   int x = 21;
   void getAge();
}
class AnonymousDemo
{
   public static void main(String[] args)
   {
      // Myclass is implementation class of Age interface
      MyClass obj=new MyClass();

      // calling getage() method implemented at Myclass
      obj.getAge();
   }
}

// Myclass implement the methods of Age Interface
class MyClass implements Age
{
   @Override
   public void getAge()
   {
      // printing the age
      System.out.print("Age is "+x);
   }
}
```

In the program, interface Age is created with getAge() method and x=21. Myclass is written as implementation class of Age interface. As done in Program, there is no need to write a separate class

Myclass. Instead, directly copy the code of Myclass into this parameter, as shown here:

```
Age oj1 = new Age() {
        @Override
        public void getAge() {
           System.out.print("Age is "+x);
        } };
```

Here, an object to Age is not created but an object of Myclass is created and copied in the entire class code as shown above. This is possible only with anonymous inner class. Such a class is called 'anonymous inner class', so here we call 'Myclass' as anonymous inner class.

Anonymous inner class version of the above Program

```
//Java program to demonstrate Anonymous inner class
interface Age
{
   int x = 21;
   void getAge();
}
class AnonymousDemo
{
   public static void main(String[] args) {

      // Myclass is hidden inner class of Age interface
      // whose name is not written but an object to it
      // is created.
      Age oj1 = new Age() {
         @Override
         public void getAge() {
            // printing age
            System.out.print("Age is "+x);
         }
      };
      oj1.getAge();
   }
}
```

Difference between Normal/Regular class and Anonymous Inner class:

- A normal class can implement any number of interfaces but anonymous inner class can implement only one interface at a time.
- A regular class can extend a class and implement any number of interface simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time.
- For regular/normal class, we can write any number of constructors but we cant write any constructor for anonymous Inner class because anonymous class does not have any name and while defining constructor class name and constructor name must be same.

JAVA LOCAL INNER CLASS

A class is created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class example

```
  public class localInner1{
 private int data=30;//instance variable
void display(){
class Local{
 void msg(){System.out.println(data);}
}
Local l=new Local();
l.msg();
}
public static void main(String args[]){
localInner1 obj=new localInner1();
obj.display();
} }
```

Output: *30*

JAVA NESTED INTERFACE

An interface i.e. declared within another interface or class is known as nested interface. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.

Syntax of nested interface which is declared within the interface

```
interface interface_name{
...
interface nested_interface_name{
...
} }
```

Syntax of nested interface which is declared within the class

```
class class_name{
...
interface nested_interface_name{
...
} }
```

Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
interface Showable{
 void show();
 interface Message{
 void msg();
 } }
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface"); }
 public static void main(String args[]){
 Showable.Message message=new TestNestedInterface1();//upcasting here
 message.msg();
 } }
```

Output: *hello nested interface*

STATIC NESTED CLASSES

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference. They are accessed using the enclosing class name.

OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

```
// Java program to demonstrate accessing
// a static nested class

// outer class
class OuterClass
{
   // static member
   static int outer_x = 10;

   // instance(non-static) member
   int outer_y = 20;

   // private member
   private static int outer_private = 30;

   // static nested class
   static class StaticNestedClass
   {
      void display()
      {
         // can access static member of outer class
         System.out.println("outer_x = " + outer_x);

         // can access display private static member of outer class
         System.out.println("outer_private = " + outer_private);

         // The following statement will give compilation error
         // as static nested class cannot directly access non-static membera
         // System.out.println("outer_y = " + outer_y);
```

```
    } } }
// Driver class
public class StaticNestedClassDemo
{
   public static void main(String[] args)
   {
      // accessing a static nested class
      OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

      nestedObject.display();
   }
 }
```

Output:

outer_x = 10

outer_private = 30

- **JAVA STATIC KEYWORD**

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.

- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (ie. it saves memory).

Example of static variable

```
//Program of static variable
 class Student8{
   int rollno;
   String name;
   static String college ="ITS";

   Student8(int r,String n){
   rollno = r;
   name = n;
   }
 void display (){System.out.println(rollno+" "+name+" "+college);}

 public static void main(String args[]){
 Student8 s1 = new Student8(111,"Karan");
 Student8 s2 = new Student8(222,"Aryan");

 s1.display();
 s2.display();
 } }
```

Output: 111 Karan ITS
222 Aryan ITS

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created. All students have its unique roll no and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

//Program of changing the common property of all objects(static field).

```
class Student9{
    int rollno;
    String name;
    static String college = "ITS";

   static void change(){
   college = "BBDIT";
    }

    Student9(int r, String n){
    rollno = r;
    name = n;
    }
    void display (){ System.out.println(rollno+" "+name+" "+college); }

   public static void main(String args[]){
   Student9.change();
   Student9 s1 = new Student9 (111,"Karan");
   Student9 s2 = new Student9 (222,"Aryan");
   Student9 s3 = new Student9 (333,"Sonoo");

   s1.display();
   s2.display();
  s3.display();
   }
 }
```

Output: 111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT

3) Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[])
{
   System.out.println("Hello main");
 }
}
```

Output: static block is invoked
Hello main

this KEYWORD IN JAVA

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}
 }
```

Output: *0 null 0.0*
0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee); } }
```

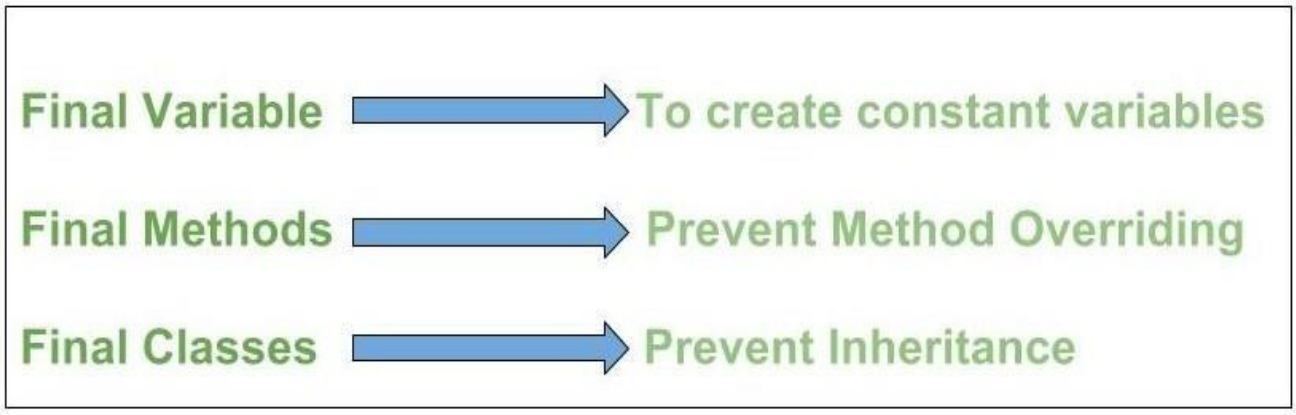
```
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}
}
```

Output: *111 ankit 5000*
112 sumit 6000

FINAL KEYWORD IN JAVA

final keyword is used in different contexts.

First of all, *final* is a non-access modifier applicable **only to a variable, a method or a class**. Following are different contexts where final is used.



FINAL VARIABLES

When a variable is declared with *final* keyword, it's value can't be modified, essentially, a constant. This also mean that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from final array or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

Examples :

```
// a final variable
final int THRESHOLD = 5;
// a blank final variable
final int THRESHOLD;
// a final static variable PI
static final double PI = 3.141592653589793;
// a blank final static variable
static final double PI;
```

Initializing a final variable :

We must initialize a final variable, otherwise compiler will throw compile-time error. A final variable can only be initialized once, either via an initializer or an assignment statement.

There are three ways to initialize a final variable :

1. You can initialize a final variable when it is declared. This approach is the most common. A final variable is called **blank final variable**, if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
2. A blank final variable can be initialized inside instance-initializer block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
3. A blank final static variable can be initialized inside static block.

Let us see above different ways of initializing a final variable through an example.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new Bike9();
```

```
 obj.run();
 }
} //end of class
```

Output: Compile Time Error

When to use a final variable :

The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned. Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

Reference final variable:

When a final variable is a reference to an object, then this final variable is called reference final variable. For example, a final StringBuffer variable looks like

```
final StringBuffer sb;
```

As you know that a final variable cannot be re-assign. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of *final* is called *non-transitivity.*

FINAL CLASSES

When a class is declared with *final* keyword, it is called a final class. A final class cannot be extended(inherited). There are two uses of a final class :

One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float etc. are final classes. We cannot extend them.

```
final class A{
  // methods and fields}
// The following class is illegal.
class B extends A {
```

```
  // COMPILE-ERROR! Can't subclass A
}
```

The other use of final with classes is to create an immutable class like the predefined Stringclass.You cannot make a class immutable without making it final.

FINAL METHODS

When a method is declared with *final* keyword, it is called a final method. A final method cannot be overridden. The Object class does this—a number of its methods are final. We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. The following fragment illustrates final keyword with a method:

```
class A {
  final void m1() {
    System.out.println("This is a final method.");
  }
}
class B extends A {
  void m1() {
    // COMPILE-ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

protected void *finalize() { …… }*

JAVA GARBAGE COLLECTION

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

public static void *gc(){ }*

Simple Example of garbage collection in java

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

Output: *object is garbage collected*

object is garbage collected

SUPER KEYWORD IN JAVA

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){ System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
} }
```

Output: *black*

white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

***2)* super can be used to invoke parent class method**

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
} }
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}
}
```

Output: *eating...*
barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

***3)* super is used to invoke parent class constructor.**

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}
}
```

Output: *animal is created*

dog is created

JAVA PACKAGE

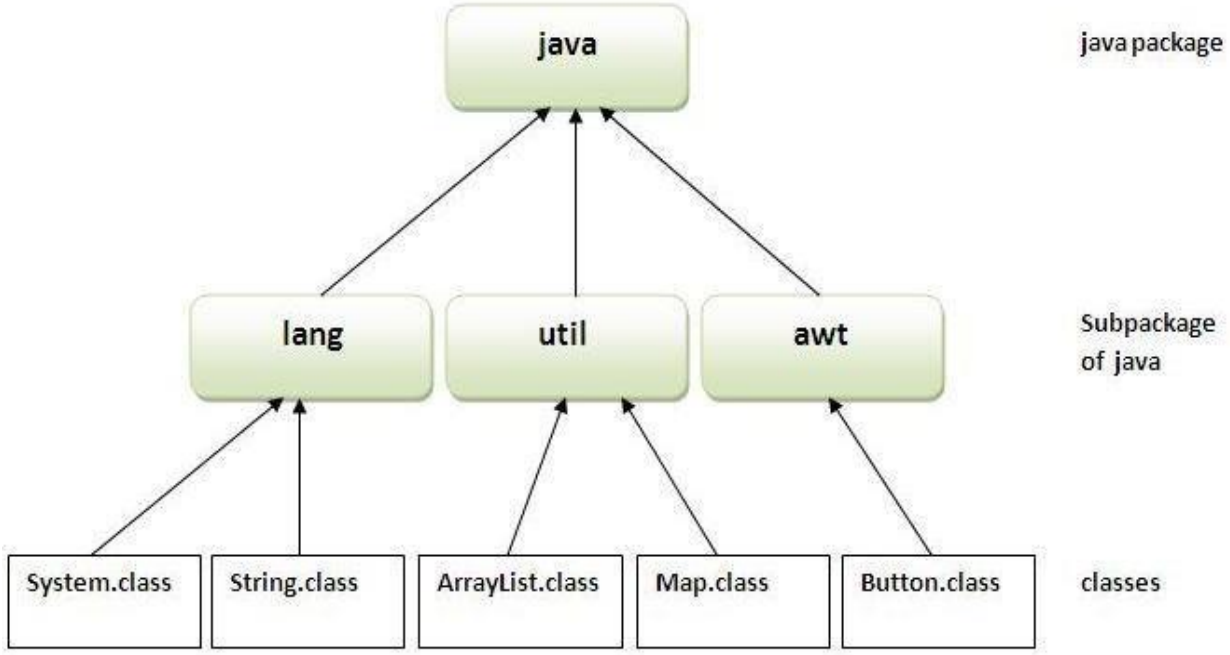
- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages. The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
```

```
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
  }
```

Output: *Hello*

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java
package pack;
public class A{ }
  public void msg(){System.out.println("Hello");}
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: *Hello*

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output: *Hello*

JAVA STATIC IMPORT

The static import features of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage of static import:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

Simple Example of static import

```
import static java.lang.System.*;
class StaticImportExample{
  public static void main(String args[]){

   out.println("Hello");//Now no need of System.out
```

```
  out.println("Java");
  }
}
```

Output: *Hello*

Java

EXCEPTION HANDLING IN JAVA

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

What is exception : Dictionary Meaning: **Exception is an abnormal condition.**

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime

WHAT IS EXCEPTION HANDLING

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

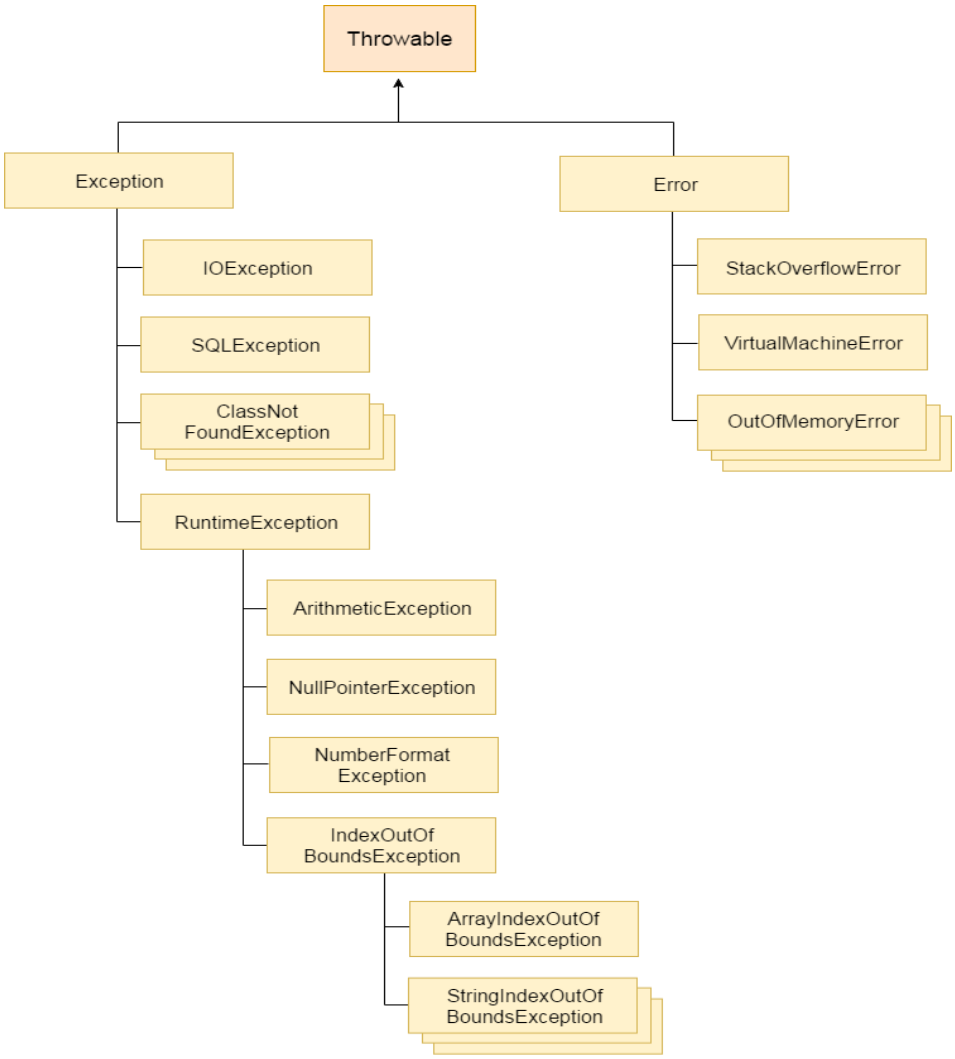
TYPES OF EXCEPTION

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

HIERARCHY OF JAVA EXCEPTION CLASSES



1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

***int** a[]=**new int**[5];*

a[10]=50; //ArrayIndexOutOfBoundsException

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try-catch

- **Java try block**

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch
***try**{*
//code that may throw exception
*}**catch**(Exception_class_Name ref){}*

Syntax of try-finally block
***try**{*
//code that may throw exception
*}**finally**{}*

- **Java catch block**

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
  public static void main(String args[]){
     int data=50/0;//may throw exception
     System.out.println("rest of the code...");
} }
```

Output: *Exception in thread main java.lang.ArithmeticException:/ by zero*

Solution by exception handling

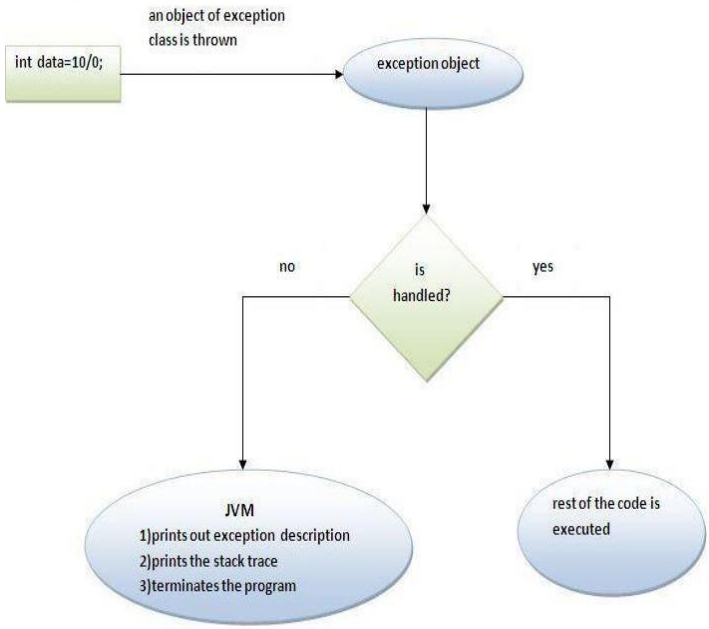
Let's see the solution of above problem by java try-catch block.

```
public class Testtrycatch2{
  public static void main(String args[]){
   try{
     int data=50/0;
   }catch(ArithmeticException e){System.out.println(e);}
   System.out.println("rest of the code...");
}
 }
```

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

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

INTERNAL WORKING OF JAVA TRY-CATCH BLOCK



- **Java catch multiple exceptions**

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
public class TestMultipleCatchBlock{
 public static void main(String args[]){
 try{
  int a[]=new int[5];
  a[5]=30/0;
 }
 catch(ArithmeticException e){System.out.println("task1 is completed");}
 catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
 catch(Exception e){System.out.println("common task completed");
 }


 System.out.println("rest of the code...");
}
}
```

Output: *task1 completed*
rest of the code...

- **Java Nested try block**

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax: ***....***

```
 try
{
  statement 1;
  statement 2;
  try
```

```
   {
     statement 1;
      statement 2;    }
   catch(Exception e)
   {     }  }
catch(Exception e) {
}
....
```

Java nested try example

Let's see a simple example of java nested try block.

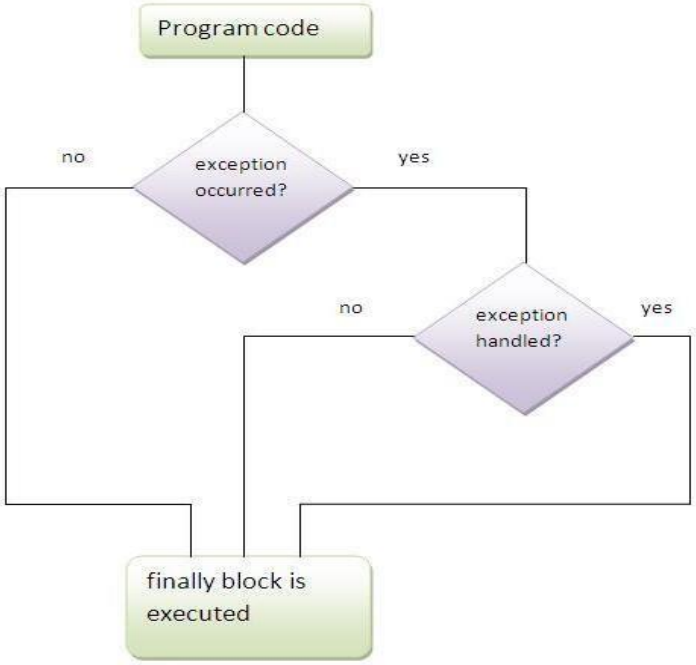
```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");

int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

   System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}

   System.out.println("normal flow..");
  } }
```

- **JAVA FINALLY BLOCK**

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.


Program code
exception occurred?
no
yes
exception handled?
no
yes
finally block is executed

- **USAGE OF JAVA FINALLY**

Let's see the different cases where java finally block can be used.

- **Case 1**

Let's see the java finally example where **exception doesn't occur**.

```
class TestFinallyBlock{
 public static void main(String args[]){
 try{
  int data=25/5;
  System.out.println(data);
 }
 catch(NullPointerException e){System.out.println(e);}
 finally{System.out.println("finally block is always executed");}
 System.out.println("rest of the code...");    }  }
```

Output: *5*

finally block is always executed
rest of the code...

- **Case 2**

Let's see the java finally example where **exception occurs and not handled**.

```
class TestFinallyBlock1{
 public static void main(String args[]){
 try{
  int data=25/0;
  System.out.println(data);
 }
 catch(NullPointerException e){System.out.println(e);}
 finally{System.out.println("finally block is always executed");}
 System.out.println("rest of the code...");
 }
}
```

Output: *finally block is always executed*

Exception in thread main java.lang.ArithmeticException:/ by zero

- **Case 3**

Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2{
  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...

- **JAVA THROW EXCEPTION**

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
throw exception;
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error);
```

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{
  static void validate(int age){
  if(age<18)
    throw new ArithmeticException("not valid");
   else
    System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    validate(13);
    System.out.println("rest of the code...");
 }
}
```

OUTPUT:

Exception in thread main java.lang.ArithmeticException:not valid

Example of throw keyword

Lets say we have a requirement where we we need to only register the students when their age is less than 12 and weight is less than 40, if any of the condition is not met then the user should get an ArithmeticException with the warning message “Student is not eligible for registration”. We have implemented the logic by placing the code in the method that checks student eligibility if the entered student age and weight doesn’t met the criteria then we throw the exception using throw keyword.

```
/* In this program we are checking the Student age
 * if the student age<12 and weight <40 then our program
 * should return that the student is not eligible for registration.
 */
public class ThrowExample {
  static void checkEligibilty(int stuage, int stuweight){
    if(stuage<12 && stuweight<40) {
      throw new ArithmeticException("Student is not eligible for registration");
```

```
    }
    else {
      System.out.println("Student Entry is Valid!!");
    }
  }
  public static void main(String args[]){
    System.out.println("Welcome to the Registration process!!");
    checkEligibilty(10, 39);
    System.out.println("Have a nice day..");
 }
}
```

OUTPUT:

Welcome to the Registration process!!Exception in thread "main"
java.lang.ArithmeticException: Student is not eligible for registration
at beginnersbook.com.ThrowExample.checkEligibilty(ThrowExample.java:9)
at beginnersbook.com.ThrowExample.main(ThrowExample.java:18)

In the above example we have throw an unchecked exception, same way we can throw uncheckedand user-defined exception as well.

- **JAVA THROWS KEYWORD**

The **Java throws keyword** 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.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws : *return_type method_name()* ***throws*** *exception_class_name{*

//method code }

Example of throws Keyword

In this example the method myMethod() is throwing two **checked exceptions** so we have declared these exceptions in the method signature using **throws** Keyword. If we do not declare these exceptions then the program will throw a compilation error.

```
import java.io.*;
class ThrowExample {
  void myMethod(int num)throws IOException, ClassNotFoundException{
    if(num==1)
      throw new IOException("IOException Occurred");
    else
      throw new ClassNotFoundException("ClassNotFoundException");
  }
}
public class Example1{
  public static void main(String args[]){
   try{
    ThrowExample obj=new ThrowExample();
    obj.myMethod(1);
   }catch(Exception ex){
    System.out.println(ex);
   }
    }
}
```

OUTPUT:

```
java.io.IOException: IOException Occurred
```

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

- **Java Custom Exception**

If you are creating your own Exception that is known as custom exception or user-defined exception. Java 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.

example of java custom exception.

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
  }
class TestCustomException1{

  static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }

  public static void main(String args[]){
    try{
    validate(13);
    }catch(Exception m){System.out.println("Exception occured: "+m);}
```

```
    System.out.println("rest of the code...");
 } }
```

Output: *Exception occured: InvalidAgeException:not valid*
rest of the code...

- **CHECKED EXCEPTIONS**

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.:

Checked Exception Example

In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where a checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

```
import java.io.*;
class Example {
  public static void main(String args[])
  {
            FileInputStream fis = null;
            /*This constructor FileInputStream(File filename)
             * throws FileNotFoundException which is a checked
             * exception
      */
     fis = new FileInputStream("B:/myfile.txt");
            int k;

            /* Method read() of FileInputStream class also throws
             * a checked exception: IOException
      */
```

```
        while(( k = fis.read() ) != -1)
        {
                System.out.print((char)k);
        }

        /*The method close() closes the file input stream
         * It throws IOException*/
        fis.close();
  }
  }
```

Output: *Exception in thread "main" java.lang.Error: Unresolved compilation problems:*

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Unhandled exception type IOException

Why this compilation error? As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn't handled/declared the exceptions, our program gave the compilation error.

How to resolve the error? There are two ways to avoid this error. We will see both the ways one by one.

Method 1: Declare the exception using throws keyword.

As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. The reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare them like this `public static void main(String args[]) throws IOException, FileNotFoundException`.

```
import java.io.*;
class Example {
  public static void main(String args[]) throws IOException
  {
    FileInputStream fis = null;
    fis = new FileInputStream("B:/myfile.txt");
```

```
    int k;

    while(( k = fis.read() ) != -1)
    {
            System.out.print((char)k);
    }
    fis.close();
  }
}
```

Output: *File content is displayed on the screen.*

Method 2: Handle them using try-catch blocks.

The approach we have used above is not good at all. It is not the best exception handling practice. You should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

```
import java.io.*;
class Example {
  public static void main(String args[])
  {
          FileInputStream fis = null;
          try{
             fis = new FileInputStream("B:/myfile.txt");
          }catch(FileNotFoundException fnfe){
        System.out.println("The specified file is not " +
                                  "present at the given path"); }
          int k;
          try{
             while(( k = fis.read() ) != -1)
             {
                     System.out.print((char)k);
             }
             fis.close();
          }catch(IOException ioe){
             System.out.println("I/O error occurred: "+ioe);
           }
  }
 }
```

This code will run fine and will display the file content.

- **UNCHECKED EXCEPTIONS**

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error.

Unchecked Exception Example

```
class Example {
  public static void main(String args[])
  {
            int num1=10;
            int num2=0;
            /*Since I'm dividing an integer with 0
             * it should throw ArithmeticException
       */
            int res=num1/num2;
            System.out.println(res);
  }
}
```

If you compile this code, it would compile successfully however when you will run it, it would throw ArithmeticException. That clearly shows that unchecked exceptions are not checked at compile-time, they occurs at runtime. Lets see another example.

```
class Example {
  public static void main(String args[])
  {
            int arr[] ={1,2,3,4,5};
            /* My array has only 5 elements but we are trying to
       * display the value of 8th element. It should throw
             * ArrayIndexOutOfBoundsException
       */
            System.out.println(arr[7]);
  }
  }
```

This code would also compile successfully since ArrayIndexOutOfBoundsException is also an uncheckedexception.

Note: It **doesn't mean** that compiler is not checking these exceptions so we shouldn't handle them. In fact we should handle them more carefully. For e.g. In the above example there should be a exception message to user that they are trying to display a value which doesn't exist in array so that user would be able to correct the issue.

```
class Example {
  public static void main(String args[]) {
            try{
              int arr[] ={1,2,3,4,5};
              System.out.println(arr[7]);
            }
      catch(ArrayIndexOutOfBoundsException e){
              System.out.println("The specified index does not exist " +
                        "in array. Please correct the error.");
            }
  }
}
```

Output: The specified index does not exist in array. Please correct the error.

Here are the few unchecked exception classes:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- IllegalArgumentException
- NumberFormatException