SYLLABUS JAVA PROGRAMMING

Paper Code: ETCS-307	**L**	**T/P**	**C**
Paper: Java Programming	**3**	**1**	**4**

INSTRUCTIONS TO PAPER SETTERS: **MAXIMUM MARKS: 75**

Objective: To learn object oriented concepts and enhancing programming skills.

UNIT I

Overview and characteristics of Java, Java program Compilation and Execution Process Organization of the Java Virtual Machine, JVM as an interpreter and emulator, Instruction Set, class File Format, Verification, Class Area, Java Stack, Heap, Garbage Collection. Security Promises of the JVM, Security Architecture and Security Policy. Class loaders and security aspects, sandbox model [T1,R2][No. of Hrs.: 11]

UNIT II

Java Fundamentals, Data Types & Literals Variables, Wrapper Classes, Arrays, Arithmetic Operators, Logical Operators, Control of Flow, Classes and Instances, Class Member Modifiers Anonymous Inner Class Interfaces and Abstract Classes, inheritance, throw and throws clauses, user defined Exceptions, The String Buffer Class, tokenizer, applets, Life cycle of applet and Security concerns. [T1,T2][No. of Hrs.: 12]

UNIT III

Threads: Creating Threads, Thread Priority, Blocked States, Extending Thread Class, Runnable Interface, Starting Threads, Thread Synchronization, Synchronize Threads, Sync Code Block, Overriding Synced Methods, Thread Communication, wait, notify and notify all.AWT Components, Component Class, Container Class, Layout Manager Interface Default Layouts, Insets and Dimensions, Border Layout, Flow Layout, Grid Layout, Card Layout Grid Bag Layout AWT Events, Event Models, Listeners, Class Listener, Adapters, Action Event Methods Focus Event Key Event, Mouse Events, Window Event [T2][No. of Hrs.: 11]

UNIT IV

Input/Output Stream, Stream Filters, Buffered Streams, Data input and Output Stream, Print Stream Random Access File, JDBC (Database connectivity with MS-Access, Oracle, MS-SQL Server), Object serialization, Sockets, development of client Server applications, design of multithreaded server. Remote Method invocation, Java Native interfaces, Development of a JNI based application. Collection API Interfaces, Vector, stack, Hashtable classes, enumerations, set, List, Map, Iterators. [T1][R1][No. of Hrs.: 10]

JAVA UNIT -2

Types of Java programs

There are many types of Java programs which run differently:

- **Java Applet - small program written in Java and that is downloaded from a website and executed within a web browser on a client computer.**
- **Application - executes on a client computer. If online, it has to be downloaded before being run.**
- **JAR file (*Java archive*) - used to package Java files together into a single file (almost exactly like a .zip file).**
- **Servlet - runs on a web server and helps to generate web pages.**

BASIC JAVA PROGRAM:

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

In the first line defines a class called Main.

```
public class Main {
```

In Java, every line of code that can actually run needs to be inside a class. This line declares a class named **`Main`**, which is **`public`**, that means that any other class can access it. For now, we'll just write our code in a class called **Main.** Its either Public or nothing.

Notice that when we declare a public class, we must declare it inside a file with the same name (Main.java), otherwise we'll get an error when compiling.

The next line is:

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

This is the entry point of our Java program. the main method has to have this exact signature in order to be able to run our program.

- `public` again means that anyone can access it.
- `static` means that you can run this method without creating an instance of `Main`.
- `void` means that this method doesn't return any value.
- `main` is the name of the method.

The arguments we get inside the method are the arguments that we will get when running the program with parameters. **It's an array of strings.**

```
System.out.println("This will be printed");
```

- `System` is a pre-defined class that Java provides us and it holds some useful methods and variables.
- `out` is a static variable within System that represents the output of your program (stdout).or it is a reference variable pointing to a object i.e out. (:. Reference variables are used to refer to an object. They are declared with a specific type that cannot be changed.)
- `println` is a method of **out** that can be used to print a line.

- **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating. An object is an instance of a class.
- **Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.

- **Methods** − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

Java Identifiers

All Java components require names. Names used for classes, variables, and methods are called identifiers.

In Java, there are several points to remember about identifiers. They are as follows −

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: age, $salary, _value, __1_value.
- Examples of illegal identifiers: 123abc, -salary.

Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers −

- **Access Modifiers** − default, public , protected, private
- **Non-access Modifiers** − final, abstract, strictfp

Java Variables

Following are the types of variables in Java −

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non-static Variables)

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

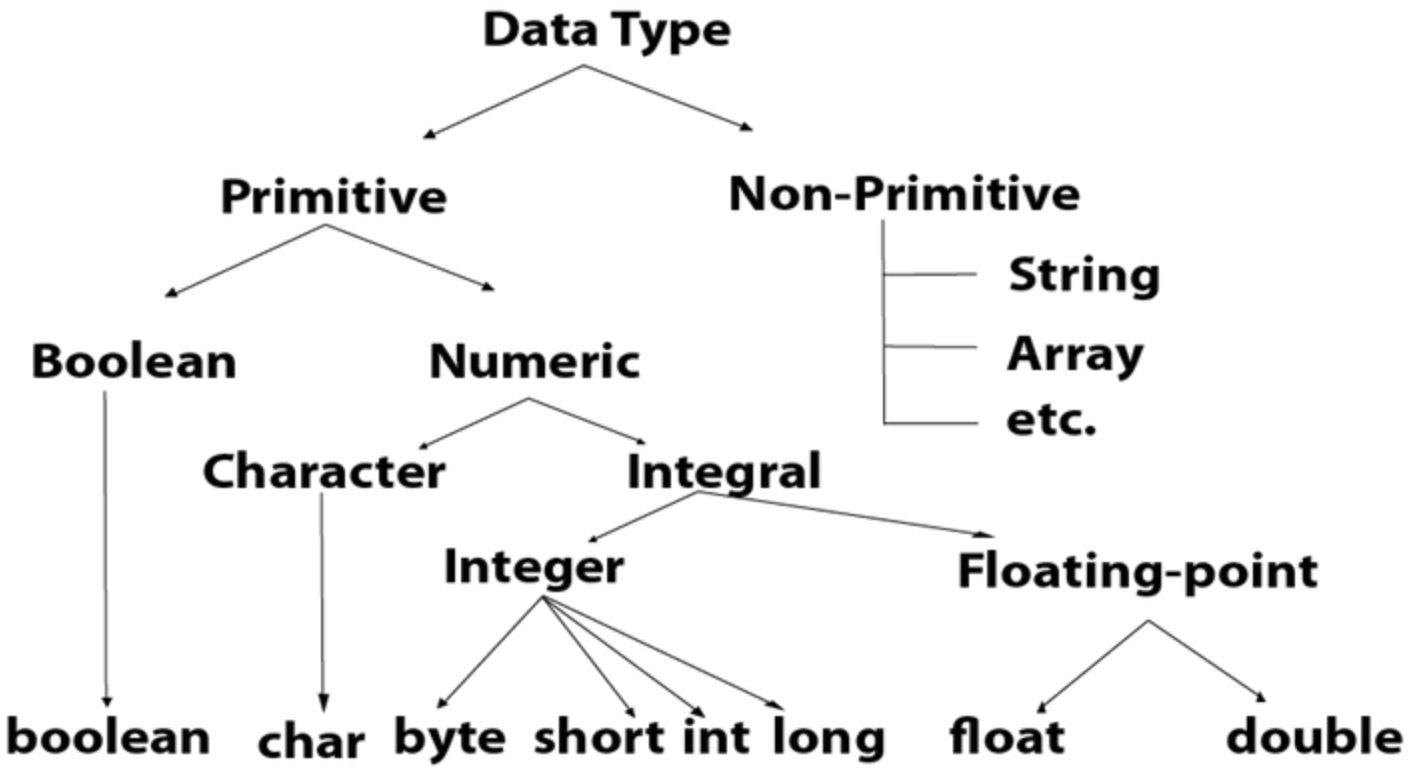
Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

> Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Literals in Java

Literal : Any constant value which can be assigned to the variable is called as literal/constant.

```
// Here 100 is a constant/literal.
int x = 100;
```

1. **Integral literals :** `int x = 101;`
2. **Floating Point Literals** `double d = 123.456;`
3. **Char literal** `char ch = 'a';`
4. **String literal** `String s = "Hello";`
5. **boolean literals** `boolean b = true;`

Java Wrapper Classes

JAVA is not 100 percent OOP language, because of the presence of primitive data types (int, char, boolean, double). As JAVA should consists of objects in every aspect to make it 100 percent OOP language. So, we have a concept of wrapper classes.

Wrapper classes provide a way to use primitive data types (`int`, `boolean`, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

Primitive Data Type	Wrapper Class
byte	Byte

short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

We have classes for all data types and we call them as wrapper classes.

Eg. int data = 20;
(this data is called as variable, not objects)

To make it as object,

Integer i = new Integer (data);
Or
Integer i = new Integer (20);

Now, this i is an object referring to the data 20.
This way of converting primitive data types into object is called **Autoboxing.**

How to take out value from Integer Object

Integer i = new Integer(20);

Int j = i.intValue(); // now, j=20 , right
Getting the primitive value from object is called as **Unboxing**

Use of instanceof

The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

```
1. public class Simple1{
2.  public static void main(String args[]){
3.  Simple1 s=new Simple1();
4.  System.out.println(s instanceof Simple1);//true
5.  }
6. }
```

Output:true

Example 1: Primitive Types to Wrapper Objects

```
public class Main {

  public static void main(String[] args) {


    // create primitive types
    int a = 5;
    double b = 5.65;
```

```
    //converts into wrapper objects
    Integer aObj = Integer.valueOf(a);
    Double bObj = Double.valueOf(b);

    if(aObj instanceof Integer) {
      System.out.println("An object of Integer is created.");
    }

    if(bObj instanceof Double) {
      System.out.println("An object of Double is created.");
    }
  }
}
```

Output

```
An object of Integer is created.
An object of Double is created.
```

In the above example, we have used the `valueOf()` method to convert the primitive types into objects. Here, we have used the `instanceof` operator to check whether the generated objects are of `Integer` or `Double` type or not. However, the Java compiler can directly convert the primitive types into corresponding **objects. For example,**

```
int a = 5;
// converts into object
Integer aObj = a;

double b = 5.6;
// converts into object
Double bObj = b;
```

Program;

public class Main {

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

    int a = 5;
// converts into object
Integer aObj = a;

double b = 5.6;
// converts into object
Double bObj = b;


    if(aObj instanceof Integer) {
      System.out.println("An object of Integer is created.");
    }

    if(bObj instanceof Double) {
      System.out.println("An object of Double is created.");
    }
  }
}
```

Output: An object of Integer is created.
An object of Double is created.

Example 2: Wrapper Objects into Primitive Types

```
class Main {

  public static void main(String[] args) {


    // creates objects of wrapper class
    Integer aObj = Integer.valueOf(23);
    Double bObj = Double.valueOf(5.55);


    // converts into primitive types
    int a = aObj.intValue();
    double b = bObj.doubleValue();


    System.out.println("The value of a: " + a);
```

```
    System.out.println("The value of b: " + b);
  }
}
```

Output

```
The value of a: 23
The value of b: 5.55
```

In the above example, we have used the `intValue()` and `doubleValue()` method to convert the `Integer` and `Double` objects into corresponding primitive types.

However, the Java compiler can automatically convert objects into corresponding primitive types. For example,

```
Integer aObj = Integer.valueOf(2);
// converts into int type
int a = aObj;

Double bObj = Double.valueOf(5.55);
// converts into double type
double b = bObj;
```

Java Arrays

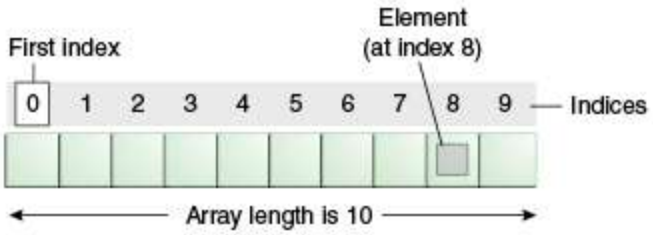
Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

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

Disadvantages

- **Size Limit:** We can store only the 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 which grows automatically.Note: Once the length of the array is defined, it cannot be changed in the program.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)

2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. class Testarray{
4. public static void main(String args[]){
5. int a[]=new int[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. for(int i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}
```

Test it Now

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

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

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

```
1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. public class Testarray1{
4. public static void main(String args[]){
5. int a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
7. for(int i=0;i<a.length;i++)//length is the property of array
8. System.out.println("Array of element " +i+ " is   " +a[i]);
9. }}
10.
11.
```

Output:

```
33
3
4
5
```

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
1. for(data_type variable:array){
2. //body of the loop
3. }
```

Let us see the example of print the elements of Java array using the for-each loop.

```
1. //Java Program to print the array elements using for-each loop
2. class Testarray1{
3. public static void main(String args[]){
4. int arr[]={33,3,4,5};
5. //printing array using for-each loop
6. for(int i:arr)
7. System.out.println(i);
8. }}
```

Output:

```
33
3
4
5
```

Java Array using Scanner

```
1. import java.util.Scanner;
2. public class Array_Sum
3. {
4.    public static void main(String[] args)
5.    {
6.        int n, sum = 0;
7.        Scanner s = new Scanner(System.in);
8.        System.out.print("Enter no. of elements you want in array:");
9.        n = s.nextInt();
10.           int a[] = new int[n];
11.           System.out.println("Enter all the elements:");
12.           for(int i = 0; i < n; i++)
13.           {
14.               a[i] = s.nextInt();
15.               sum = sum + a[i];
16.           }
17.           System.out.println("Sum:"+sum);
```

```
18.       }
19.    }
```

Passing Array to a 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 the minimum number of an array using a method.

```
1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. class Testarray2{
4. //creating a method which receives an array as a parameter
5. static void min(int arr[]){
6. int min=arr[0];
7. for(int i=1;i<arr.length;i++)
8.  if(min>arr[i])
9.   min=arr[i];
10.
11. System.out.println(min);
12. }
13.
14. public static void main(String args[]){
15. int a[]={33,3,4,5};//declaring and initializing an array
16. min(a);//passing array to method
17. }}
```

Output:

3

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
1. //Java Program to demonstrate the way of passing an anonymous array
2. //to method.
```

```
3. public class TestAnonymousArray{
4. //creating a method which receives an array as a parameter
5. static void printArray(int arr[]){
6. for(int i=0;i<arr.length;i++)
7. System.out.println(arr[i]);
8. }
9.
10. public static void main(String args[]){
11. printArray(new int[]{10,22,44,66});//passing anonymous array to method
12. }}
```

Output:

```
10
22
44
66
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

```
1. //Java Program to demonstrate the case of
2. //ArrayIndexOutOfBoundsException in a Java Array.
3. public class TestArrayException{
4. public static void main(String args[]){
5. int arr[]={50,60,70,80};
6. for(int i=0;i<=arr.length;i++){
7. System.out.println(arr[i]);
8. }
9. }}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
```

```
        at TestArrayException.main(TestArrayException.java:5)
50

60

70

80
```

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

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

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

Example to initialize Multidimensional Array in Java

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

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

Example of Multidimensional Java Array

```
1. //Java Program to illustrate the use of multidimensional array
2. class Testarray3{
3. public static void main(String args[]){
4. //declaring and initializing 2D array
5. int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6. //printing 2D array
7. for(int i=0;i<3;i++){
8.  for(int j=0;j<3;j++){
9.    System.out.print(arr[i][j]+" ");
10. }
11. System.out.println();
12.}
13.}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

Copying a Java Array

1.Manual Copy

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

```
         int intArray[] = {12,15,17};

        //print original intArray
        System.out.println("Contents of intArray[] before assignment:");
        for (int i=0; i<intArray.length; i++)
             System.out.print(intArray[i] + " ");

        // Create an array b[] of same size as a[]
        int copyArray[] = new int[intArray.length];

        // intArray is assigned to copyArray; so references point to same
location
        copyArray = intArray;

        // change element of copyArray
        //copyArray[1]++;

        //print both arrays
        System.out.println("\nContents of intArray[]:");
        for (int i=0; i<intArray.length; i++)
              System.out.print(intArray[i] + " ");

        System.out.println("\nContents of copyArray[]:");
        for (int i=0; i<copyArray.length; i++)
              System.out.print(copyArray[i] + " ");
    }
}
```

Output:

```
Contents of intArray[] before assignment:
12 15 17
Contents of intArray[]:
12 15 17
Contents of copyArray[]:
 12 15 17
```

We can copy an array to another by the arraycopy() method of System class.

Syntax of arraycopy method

```
1. public static void arraycopy(
2. Object src, int srcPos,Object dest, int destPos, int length
3. )
```

Example of Copying an Array in Java

```
1. //Java Program to copy a source array into a destination array in Java
2. class TestArrayCopyDemo {
3.     public static void main(String[] args) {
4.         //declaring a source array
5.         char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
6.                 'i', 'n', 'a', 't', 'e', 'd' };
7.         //declaring a destination array
8.         char[] copyTo = new char[7];
9.         //copying array using System.arraycopy() method
10.        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
11.        //printing the destination array
12.        System.out.println(String.valueOf(copyTo));
13.    }
14. }
```

Test it Now

Output:

caffein

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

```
1. //Java Program to clone the array
2. class Testarray1{
3. public static void main(String args[]){
4. int arr[]={33,3,4,5};
5. System.out.println("Printing original array:");
6. for(int i:arr)
7. System.out.println(i);
```

```
8.
9.  System.out.println("Printing clone of the array:");
10. int carr[]=arr.clone();
11. for(int i:carr)
12. System.out.println(i);
13.
14. }}
```

Output:

```
Printing original array:
33
3
4
5
Printing clone of the array:
33
3
4
5
```

Addition of 2 Matrices in Java

```
1.  public class Testarray5{
2.  public static void main(String args[]){
3.  //creating two matrices
4.  int a[][]={{1,3,4},{3,4,5}};
5.  int b[][]={{1,3,4},{3,4,5}};
6.  System.out.println("Array a is:  ");
7.  for(int i=0;i<2;i++){
8.  for(int j=0;j<3;j++){
9.  System.out.print(a[i][j]+" ");
10. }
```

```
11. System.out.println();//new line
12. }
13.
14. System.out.println();
15. System.out.println("Array b is:  ");
16. for(int i=0;i<2;i++){
17. for(int j=0;j<3;j++){
18. System.out.print(b[i][j]+" ");
19. }
20. System.out.println();//new line
21. }
22. System.out.println();
23. //creating another matrix to store the sum of two matrices
24. int c[][]=new int[2][3];
25.
26.  System.out.println("Sum Array c is:  ");
27. //adding and printing addition of 2 matrices
28. for(int i=0;i<2;i++){
29. for(int j=0;j<3;j++){
30. c[i][j]=a[i][j]+b[i][j];
31. System.out.print(c[i][j]+" ");
32. }
33. System.out.println();//new line
34. }
35.
36. }}
37.
```

Output:

Array a is:

1 3 4

3 4 5

Array b is:

1 3 4

3 4 5

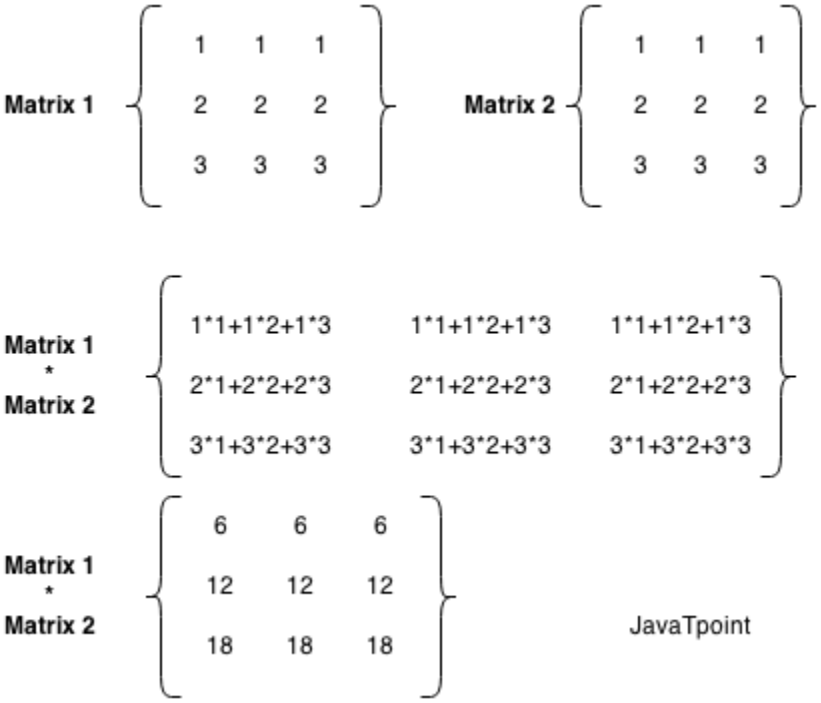
Sum Array c is:

2 6 8

6 8 10

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

```
1. //Java Program to multiply two matrices
2. public class MatrixMultiplicationExample{
3. public static void main(String args[]){
4. //creating two matrices
5. int a[][]={{1,1,1},{2,2,2},{3,3,3}};
6. int b[][]={{1,1,1},{2,2,2},{3,3,3}};
```

```
7.
8.  //creating another matrix to store the multiplication of two matrices
9.  int c[][]=new int[3][3];  //3 rows and 3 columns
10.
11. //multiplying and printing multiplication of 2 matrices
12. for(int i=0;i<3;i++){
13. for(int j=0;j<3;j++){
14. c[i][j]=0;
15. for(int k=0;k<3;k++)
16. {
17. c[i][j]+=a[i][k]*b[k][j];
18. }//end of k loop
19. System.out.print(c[i][j]+" ");  //printing matrix element
20. }//end of j loop
21. System.out.println();//new line
22. }
23. }}
```

Test it Now

Output:

```
6 6 6
12 12 12
18 18 18
```

Operators in Java

Operator in Java is a symbol which is used to perform operations. For example: +, -, *, / etc.

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,

- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
	comparison	< > <= >= instanceof
	equality	== !=
	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
	logical AND	&&
	logical OR	||

Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int x=10;
4. System.out.println(x++);
5. System.out.println(++x);
6. System.out.println(x--);
7. System.out.println(--x);
8. }}
```

Java Unary Operator Example: ++ and --

```
9. class OperatorExample{
10. public static void main(String args[]){
11. int x=10;
12. System.out.println(x++);          //10 (11)
13. System.out.println(++x);          //12
14. System.out.println(x--);          //12 (11)
15. System.out.println(--x);          //10
16. }}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=10;
5. System.out.println(a++ + ++a);                    //10+12=22
6. System.out.println(b++ + b++);                    //10+11=21
7.
8. }}
```

Output:

```
22
21
```

Java Arithmetic Operators

Java arithmatic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. System.out.println(a+b);//15
```

```
6. System.out.println(a-b);//5
7. System.out.println(a*b);//50
8. System.out.println(a/b);//2
9. System.out.println(a%b);//0
10. }}
```

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

```
1. class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10*10/5+3-1*4/2);
4. }}
```

Output:

```
21
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Right Shift Operator

The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

Java Ternary Operator

Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in Java programming. it is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=2;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

2

Another Example:

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Output:

5

Java Assignment Operator

Java assignment operator is one of the most common operator. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=20;
5. a+=4;//a=a+4 (a=10+4)
6. b-=4;//b=b-4 (b=20-4)
7. System.out.println(a);
8. System.out.println(b);
9. }}
```

Output:

```
14
16
```

Java Assignment Operator Example

```
1. class OperatorExample{
2. public static void main(String[] args){
3. int a=10;
4. a+=3;//10+3
5. System.out.println(a);
6. a-=4;//13-4
7. System.out.println(a);
8. a*=2;//9*2
9. System.out.println(a);
10. a/=2;//18/2
11. System.out.println(a);
12. }}
```

Output:

13

9

18

9

Java Keywords

Java keywords are also known as **reserved words**. Keywords are particular words which acts as a key to a code. These are predefined words by Java so it cannot be used as a variable or object name.

List of Java Keywords

A list of Java keywords or reserved words are given below:

1. **abstract:** Java abstract keyword is used to declare abstract class. Abstract class can provide the implementation of interface. It can have abstract and non-abstract methods.
2. **boolean:** Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break:** Java break keyword is used to break loop or switch statement. It breaks the current flow of the program at specified condition.
4. **byte:** Java byte keyword is used to declare a variable that can hold an 8-bit data values.
5. **case:** Java case keyword is used to with the switch statements to mark blocks of text.
6. **catch:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. **class:** Java class keyword is used to declare a class.
9. **continue:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. **default:** Java default keyword is used to specify the default block of code in a switch statement.
11. **do:** Java do keyword is used in control statement to declare a loop. It can iterate a part of the program several times.
12. **double:** Java double keyword is used to declare a variable that can hold a 64-bit floating-point numbers.

13. **else:** Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is applied with a variable. It is used to restrict the user.
17. **finally:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether exception is handled or not.
18. **float:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. **for:** Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some conditions become true. If the number of iteration is fixed, it is recommended to use for loop.
20. **if:** Java if keyword tests the condition. It executes the if block if condition is true.
21. **implements:** Java implements keyword is used to implement an interface.
22. **import:** Java import keyword makes classes and interfaces available and accessible to the current source code.
23. **instanceof:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. **int:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. **interface:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. **long:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. **new:** Java new keyword is used to create new objects.
29. **null:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. **package:** Java package keyword is used to declare a Java package that includes the classes.
31. **private:** Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.

32. **protected:** Java protected keyword is an access modifier. It can be accessible within package and outside the package but through inheritance only. It can't be applied on the class.
33. **public:** Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. **return:** Java return keyword is used to return from a method when its execution is complete.
35. **short:** Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. **static:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is used for memory management mainly.
37. **strictfp:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. **super:** Java super keyword is a reference variable that is used to refer parent class object. It can be used to invoke immediate parent class method.
39. **switch:** The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. **synchronized:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. **this:** Java this keyword can be used to refer the current object in a method or constructor.
42. **throw:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exception. It is followed by an instance.
43. **throws:** The Java throws keyword is used to declare an exception. Checked exception can be propagated with throws.
44. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. **try:** Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. **void:** Java void keyword is used to specify that a method does not have a return value.
47. **volatile:** Java volatile keyword is used to indicate that a variable may change asynchronously.
48. **while:** Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Control statements

Java If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

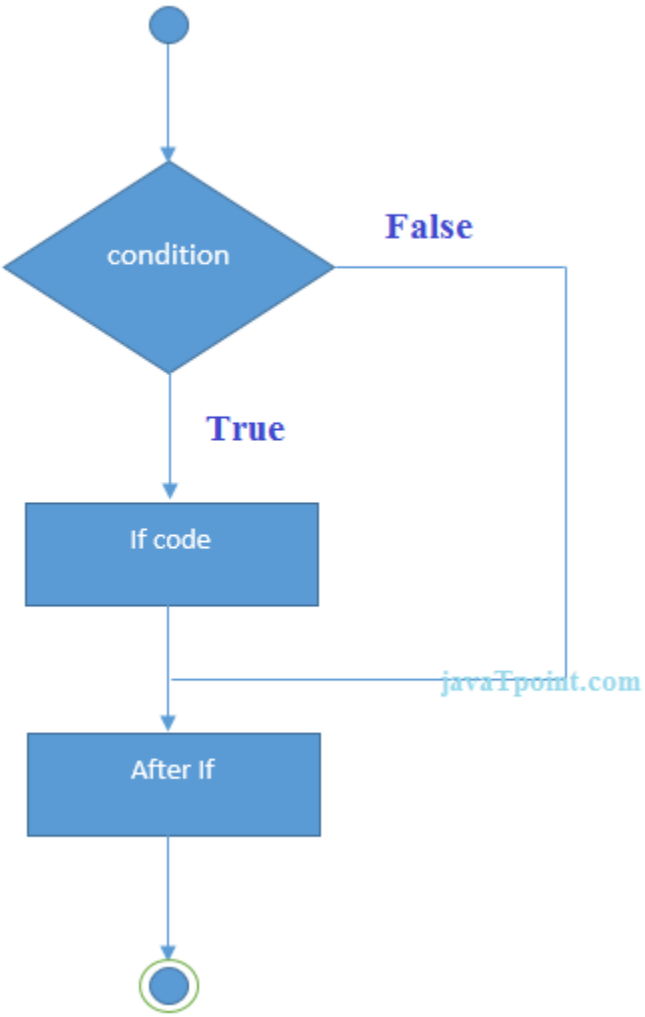
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```
1. if(condition){
2. //code to be executed
3. }
```



Example:

```
1. //Java Program to demonstate the use of if statement.
2. public class IfExample {
3. public static void main(String[] args) {
4.     //defining an 'age' variable
5.     int age=20;
6.     //checking the age
7.     if(age>18){
8.         System.out.print("Age is greater than 18");
9.     }
10. }
11. }
12.
```

Test it Now

Output:

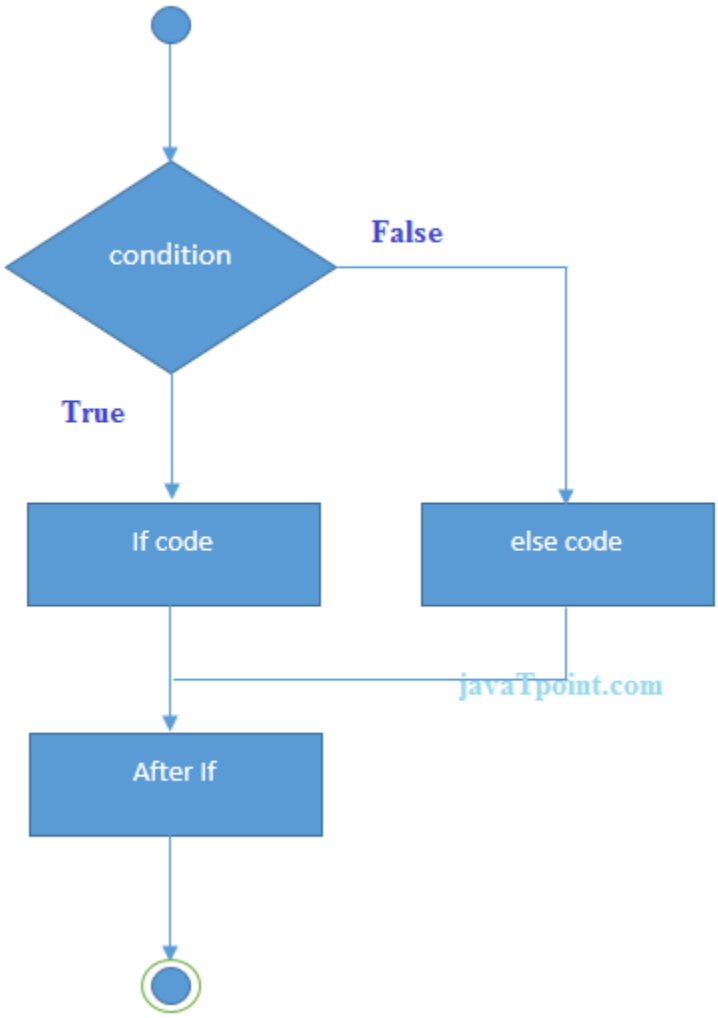
```
Age is greater than 18
```

Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



Example:

```
1. //A Java Program to demonstrate the use of if-else statement.
2. //It is a program of odd and even number.
3. public class IfElseExample {
4. public static void main(String[] args) {
5.     //defining a variable
6.     int number=13;
7.     //Check if the number is divisible by 2 or not
8.     if(number%2==0){
9.         System.out.println("even number");
10.    }else{
11.        System.out.println("odd number");
12.    }
```

```
13. }
14. }
```

Output:

```
odd number
```

Leap Year Example:

A year is leap, if it is divisible by 4 and 400. But, not by 100.

```
1.  public class LeapYearExample {
2.  public static void main(String[] args) {
3.      int year=2020;
4.      if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
5.          System.out.println("LEAP YEAR");
6.      }
7.      else{
8.          System.out.println("COMMON YEAR");
9.      }
10. }
11. }
```

Output:

```
LEAP YEAR
```

Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

Example:

```
1.  public class IfElseTernaryExample {
```

```
2.  public static void main(String[] args) {
3.      int number=13;
4.      //Using ternary operator
5.      String output=(number%2==0)?"even number":"odd number";
6.      System.out.println(output);
7.  }
8.  }
```

Output:

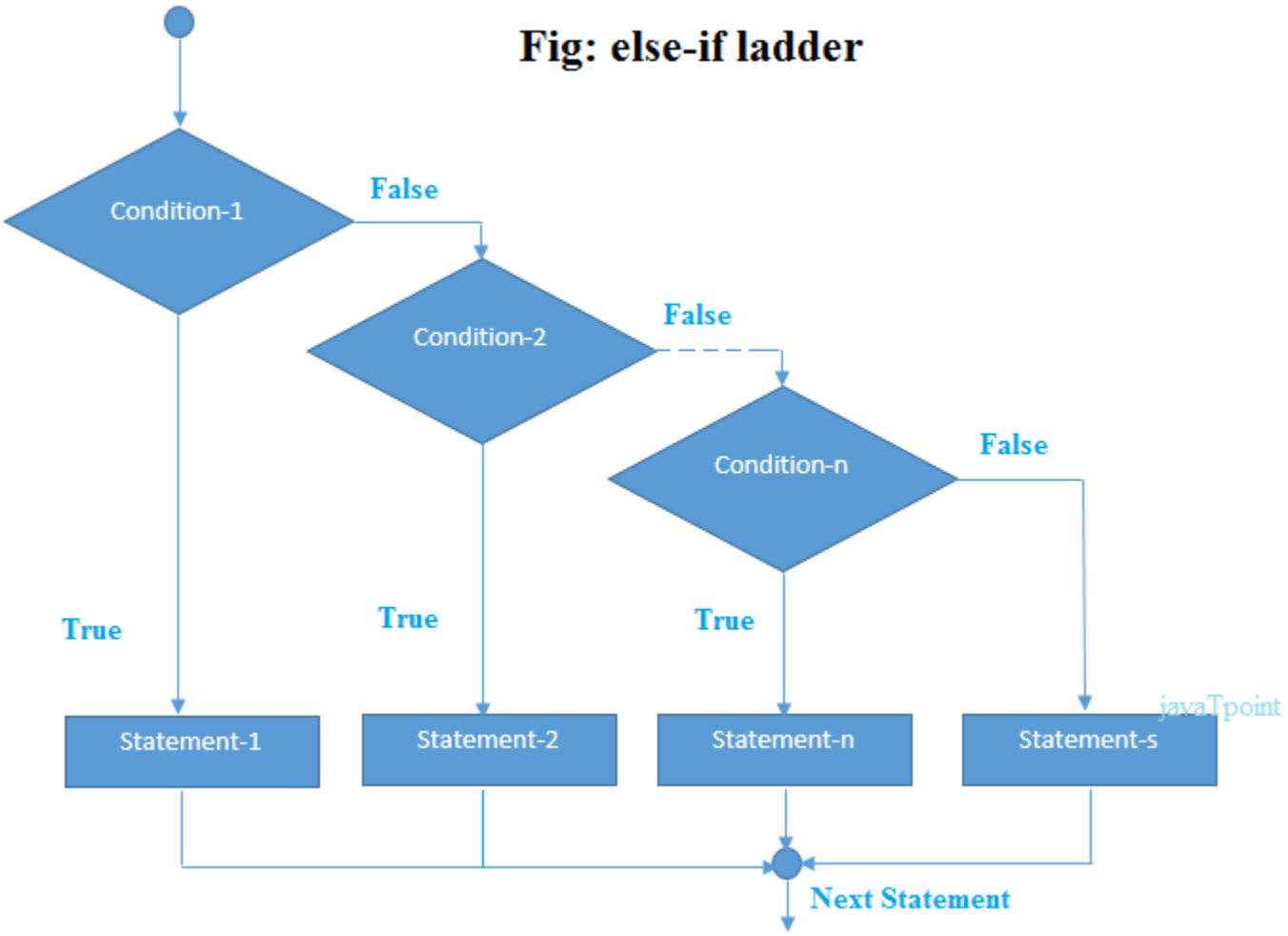
```
odd number
```

Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
1.  if(condition1){
2.  //code to be executed if condition1 is true
3.  }else if(condition2){
4.  //code to be executed if condition2 is true
5.  }
6.  else if(condition3){
7.  //code to be executed if condition3 is true
8.  }
9.  ...
10. else{
11. //code to be executed if all the conditions are false
12. }
```



Example:

```
1.  //Java Program to demonstrate the use of If else-if ladder.
2.  //It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
3.  public class IfElseIfExample {
4.  public static void main(String[] args) {
5.      int marks=65;
6.
7.      if(marks<50){
8.          System.out.println("fail");
9.      }
10.     else if(marks>=50 && marks<60){
11.         System.out.println("D grade");
12.     }
13.     else if(marks>=60 && marks<70){
```

```
14.        System.out.println("C grade");
15.    }
16.    else if(marks>=70 && marks<80){
17.        System.out.println("B grade");
18.    }
19.    else if(marks>=80 && marks<90){
20.        System.out.println("A grade");
21.    }else if(marks>=90 && marks<100){
22.        System.out.println("A+ grade");
23.    }else{
24.        System.out.println("Invalid!");
25.    }
26. }
27. }
```

Output:

```
C grade
```

Program to check POSITIVE, NEGATIVE or ZERO:

```
1.  public class PositiveNegativeExample {
2.  public static void main(String[] args) {
3.     int number=-13;
4.     if(number>0){
5.     System.out.println("POSITIVE");
6.     }else if(number<0){
7.     System.out.println("NEGATIVE");
8.     }else{
9.     System.out.println("ZERO");
10.   }
11. }
12. }
```

Output:

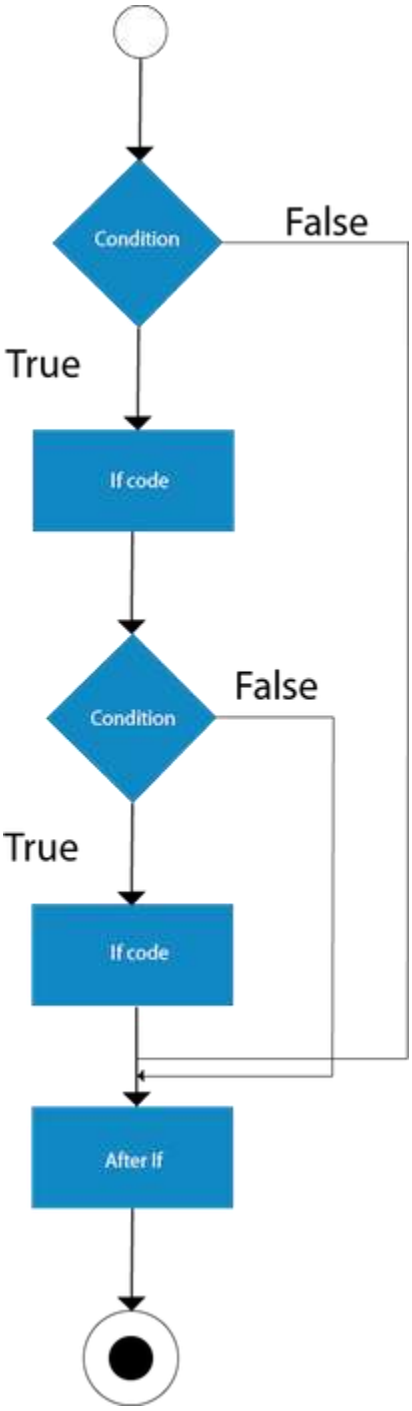
NEGATIVE

Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
1. if(condition){
2.       //code to be executed
3.           if(condition){
4.             //code to be executed
5.      }
6.  }
```



Example:

```
1. //Java Program to demonstrate the use of Nested If Statement.
2. public class JavaNestedIfExample {
3. public static void main(String[] args) {
4.     //Creating two variables for age and weight
5.     int age=20;
6.     int weight=80;
```

```
7.      //applying condition on age and weight
8.      if(age>=18){
9.          if(weight>50){
10.             System.out.println("You are eligible to donate blood");
11.         }
12.     }
13. }}
```

Test it Now

Output:

```
You are eligible to donate blood
```

Example 2:

```
1.  //Java Program to demonstrate the use of Nested If Statement.
2.  public class JavaNestedIfExample2 {
3.  public static void main(String[] args) {
4.      //Creating two variables for age and weight
5.      int age=25;
6.      int weight=48;
7.      //applying condition on age and weight
8.      if(age>=18){
9.          if(weight>50){
10.             System.out.println("You are eligible to donate blood");
11.         } else{
12.             System.out.println("You are not eligible to donate blood");
13.         }
14.     } else{
15.       System.out.println("Age must be greater than 18");
16.     }
17. } }
18.
```

Java Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java *break* statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
1. jump-statement;
2. break;
```

Java Break Statement with Loop

Example:

```
1. //Java Program to demonstrate the use of break statement
2. //inside the for loop.
3. public class BreakExample {
4. public static void main(String[] args) {
5.     //using for loop
6.     for(int i=1;i<=10;i++){
7.         if(i==5){
8.             //breaking the loop
9.             break;
10.        }
11.        System.out.println(i);
12.    }
13. }
14. }
```

Output:

```
1
2
3
```

4

Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

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

We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
1. jump-statement;
2. continue;
```

Java Continue Statement Example

Example:

```
1. //Java Program to demonstrate the use of continue statement
2. //inside the for loop.
3. public class ContinueExample {
4. public static void main(String[] args) {
5.     //for loop
6.     for(int i=1;i<=10;i++){
7.         if(i==5){
8.           //using continue statement
9.           continue;//it will skip the rest statement
10.       }
11.       System.out.println(i);
12.    }
13. }
14. }
```

Test it Now

Output:

```
1
```

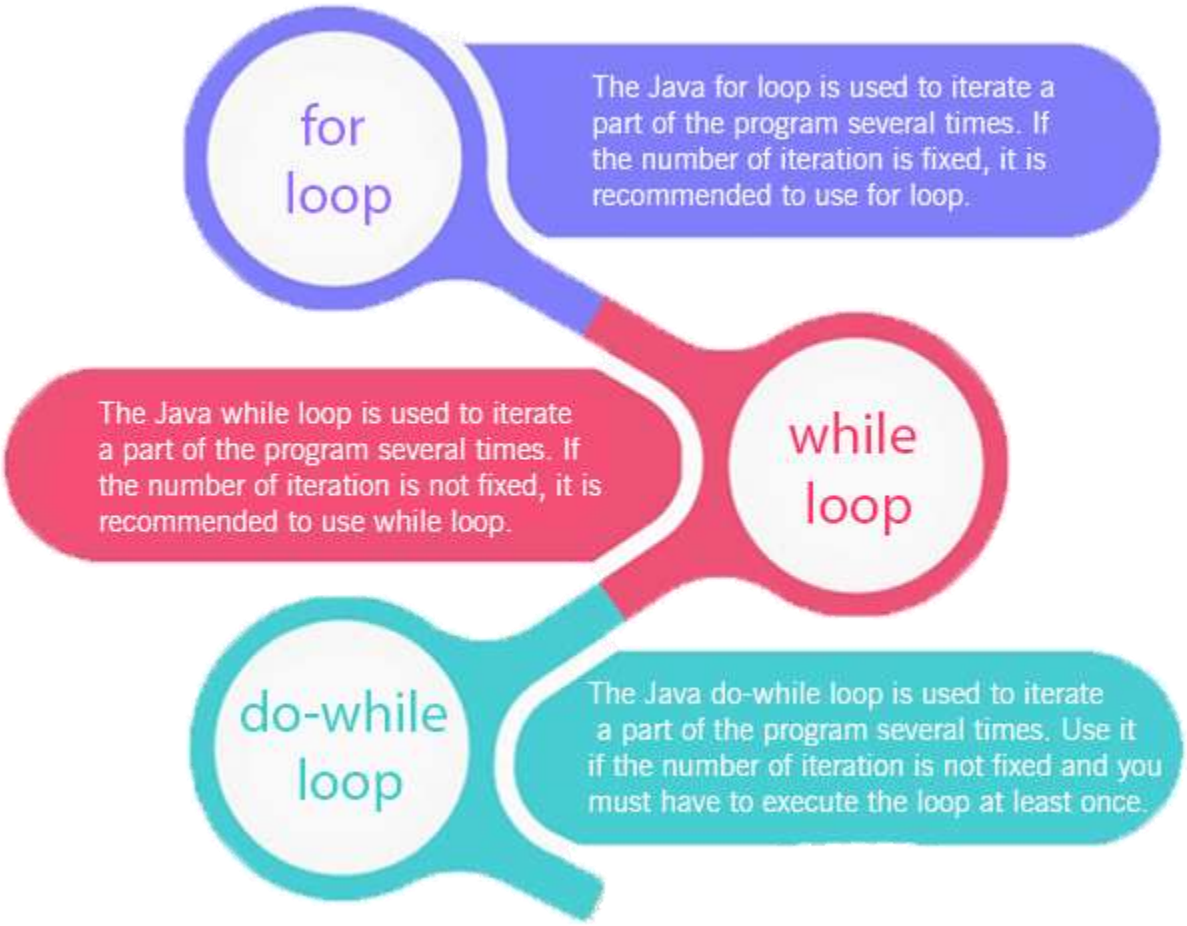
```
2
3
4
6
7
8
9
10
```

As you can see in the above output, 5 is not printed on the console. It is because the loop is continued when it reaches to 5.

Loops in Java

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.

- for loop
- while loop
- do-while loop



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 loops in java.

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

Java Simple For Loop

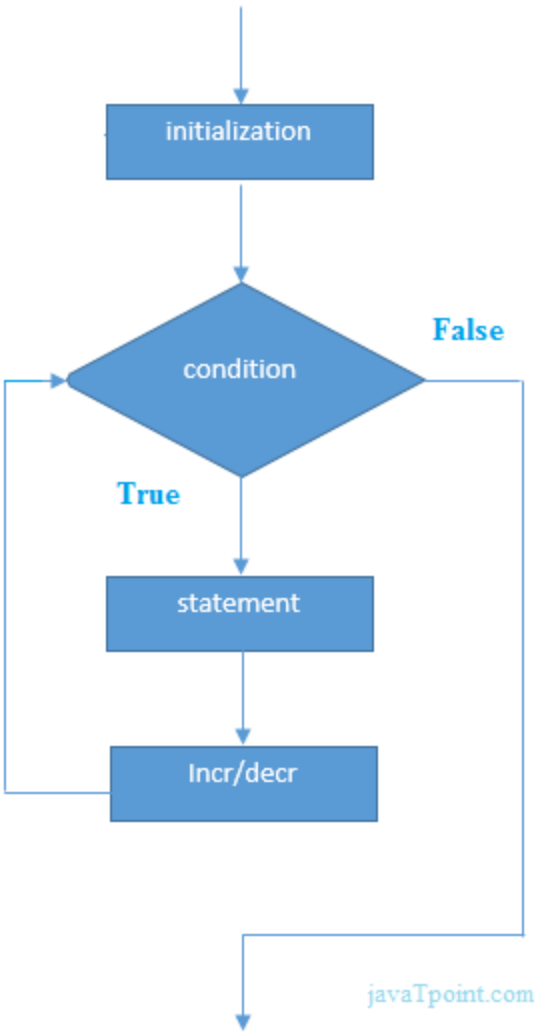
A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. **Statement**: The statement of the loop is executed each time until the second condition is false.
4. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

Syntax:

```
for(initialization;condition;incr/decr){
//statement or code to be executed
}
```

Flowchart:



Example:

```
1. //Java Program to demonstrate the example of for loop
2. //which prints table of 1
3. public class ForExample {
4. public static void main(String[] args) {
5.     //Code of Java for loop
6.     for(int i=1;i<=10;i++){
7.         System.out.println(i);
8.     }
9. }
10. }
```

Test it Now

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:

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[] args) {
    //Declaring an array
```

```
6.      int arr[]={12,23,44,56,78};
7.      //Printing array using for-each loop
8.      for(int i:arr){
9.          System.out.println(i);
10.     }
11. }
12. }
```

Test it Now

Output:

```
12
23
44
56
78
```

Java Labeled For Loop

We can have a name of each Java 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.

Usually, break and continue keywords breaks/continues the innermost for loop only.

Syntax:

```
1. labelname:
2. for(initialization;condition;incr/decr){
3. //code to be executed
4. }
```

Example:

```
1. //A Java program to demonstrate the use of labeled for loop
2. public class LabeledForExample {
3. public static void main(String[] args) {
4.     //Using Label for outer and for loop
5.     aa:
6.         for(int i=1;i<=3;i++){
7.             bb:
8.                 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.

```
public class LabeledForExample2 {
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:

```
//Java program to demonstrate the use of infinite for loop
//which prints an statement
public class ForExample {
public static void main(String[] args) {
    //Using no condition in for loop
    for(;;){
        System.out.println("infinitive loop");
    }
}
}
```

Output:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
Ctrl+c
```

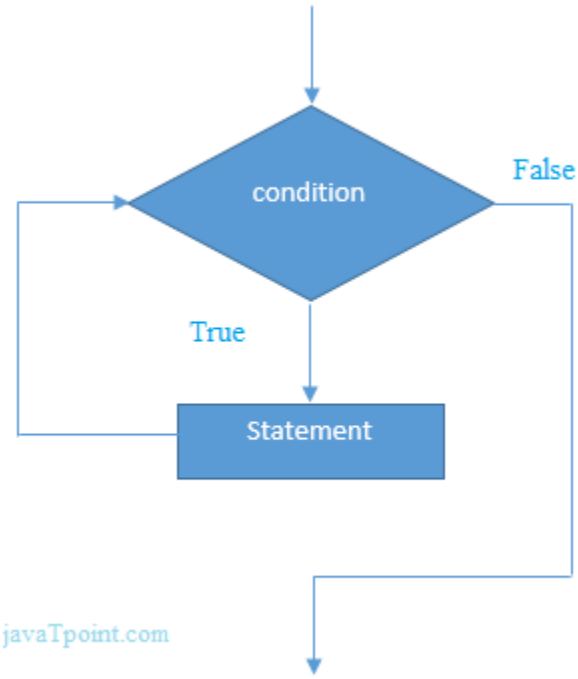
Java While Loop

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){
```

```
2. //code to be executed
3. }
```



Example:

```
1. public class WhileExample {
2. public static void main(String[] args) {
3.     int i=1;
4.     while(i<=10){
5.         System.out.println(i);
6.     i++;
7.     }
8. }
9. }
```

Test it Now

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:

```
1. while(true){
2. //code to be executed
3. }
```

Example:

```
1. public class WhileExample2 {
2. public static void main(String[] args) {
3.     while(true){
4.         System.out.println("infinitive while loop");
5.     }
6. }
7. }
```

Output:

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

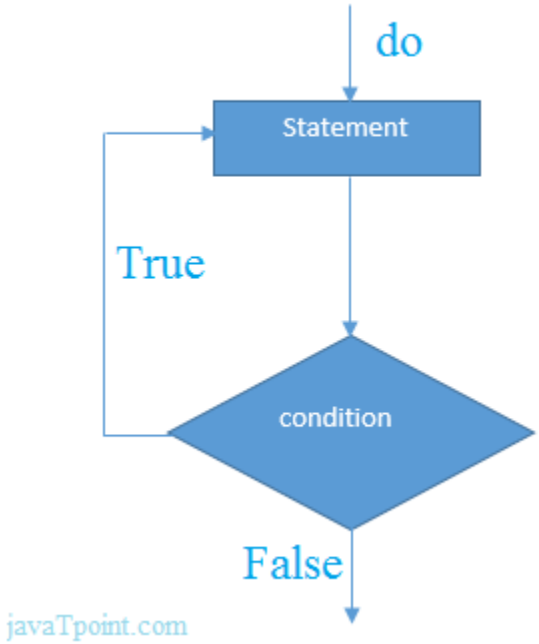
Java do-while Loop

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);
}
}
```

Test it Now

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
```

Switch:

```
public class MyClass {
  public static void main(String[] args) {
    int day = 1;
    switch (day) {
      case 1:
        System.out.println("Monday");
        break;
      case 2:
        System.out.println("Tuesday");
        break;
      case 3:
        System.out.println("Wednesday");
        break;
      case 4:
        System.out.println("Thursday");
        break;
      case 5:
        System.out.println("Friday");
       break;
      case 6:
        System.out.println("Saturday");
       break;
      case 7:
        System.out.println("Sunday");
       break;
        default:
        System.out.println("wrong choice");
       break;
    }
}
}

}
```

Output:
Thursday

For Strings:

```
public class StringInSwitchStatementExample {
    public static void main(String[] args) {
        String game = "Football";
        switch(game){
        case "Hockey":
            System.out.println("Let's play Hockey");
            break;
        case "Cricket":
            System.out.println("Let's play Cricket");
            break;
        case "Football":
            System.out.println("Let's play Football");
            break;
        default:
         System.out.println("No other games");
         break;
        }
    }
}
```

Output: Let's play Football

Vowel Checking:

```
public class VowelConsonant {

    public static void main(String[] args) {

        char ch = 'z';

        switch (ch) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                System.out.println(ch + " is vowel");
```

```
            break;
         default:
            System.out.println(ch + " is consonant");
      }
   }
}
```

Or

```
import java.util.Scanner;
public class VowelSwitch {
   public static void main(String args[]) {
      boolean bool = false;
      System.out.println("Enter a character :");
      Scanner sc = new Scanner(System.in);
      char ch = sc.next().charAt(0);
      switch(ch) {
         case 'A':
         case 'E' :
         case 'I' :
         case 'O' :
         case 'U' :
         case 'a' :
         case 'e' :
         case 'i' :
         case 'o' :
         case 'u' : System.out.println("Given character is an vowel ");
                  break;
         default:
          System.out.println("Given character is a consonant");
      }
   }
}
```

Classes and Objects

Java is an object-oriented programming language.

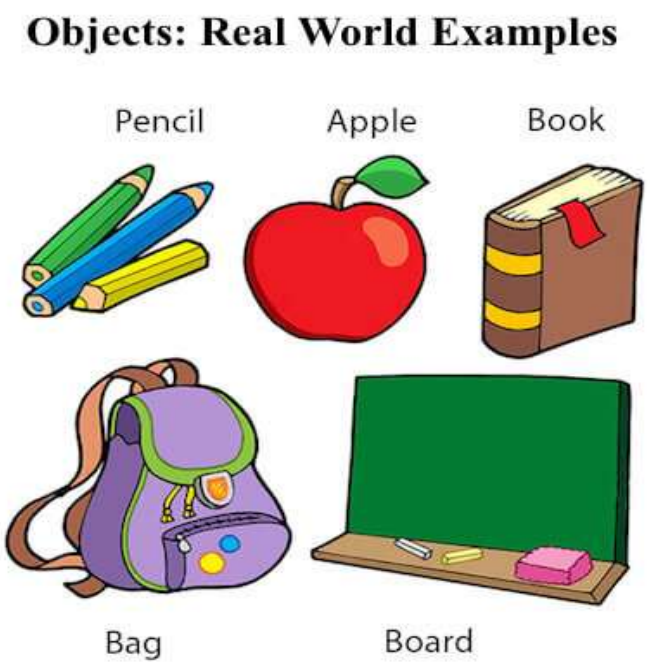
Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

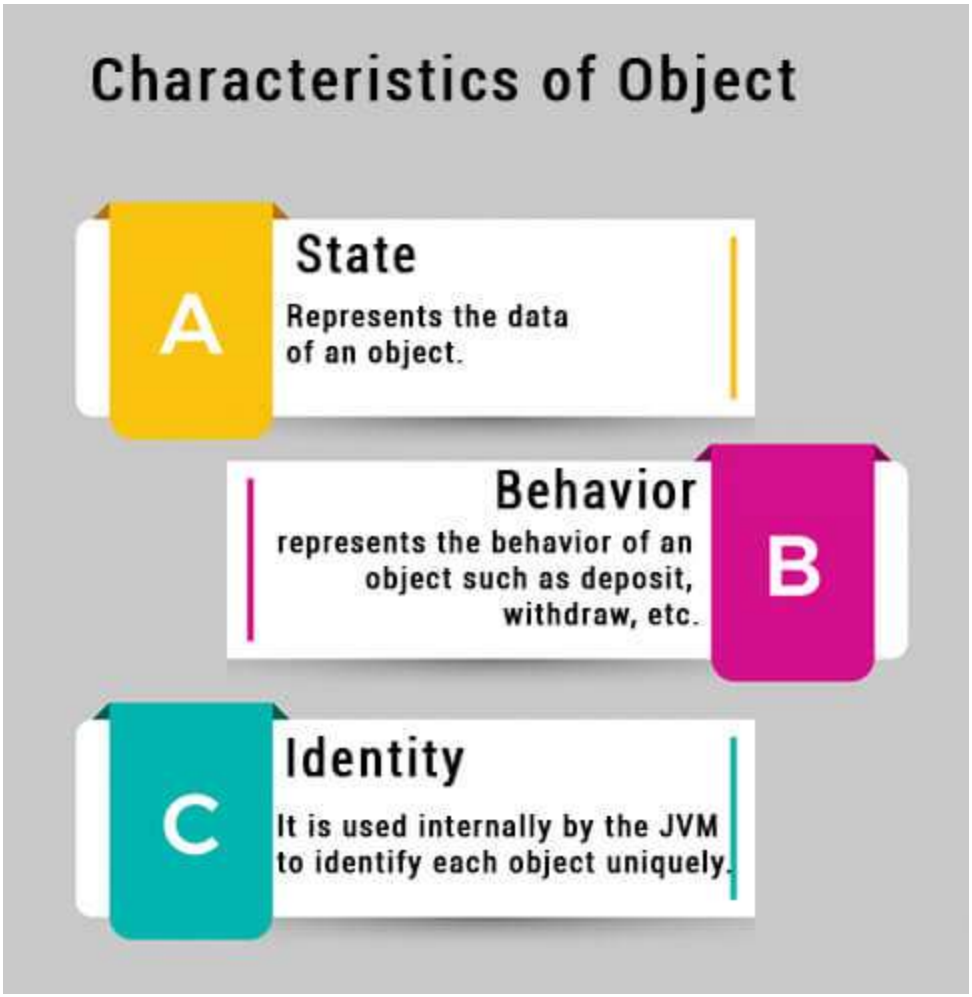
What is an object in Java



An entity that has **state and behavior** is known as an **object** e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

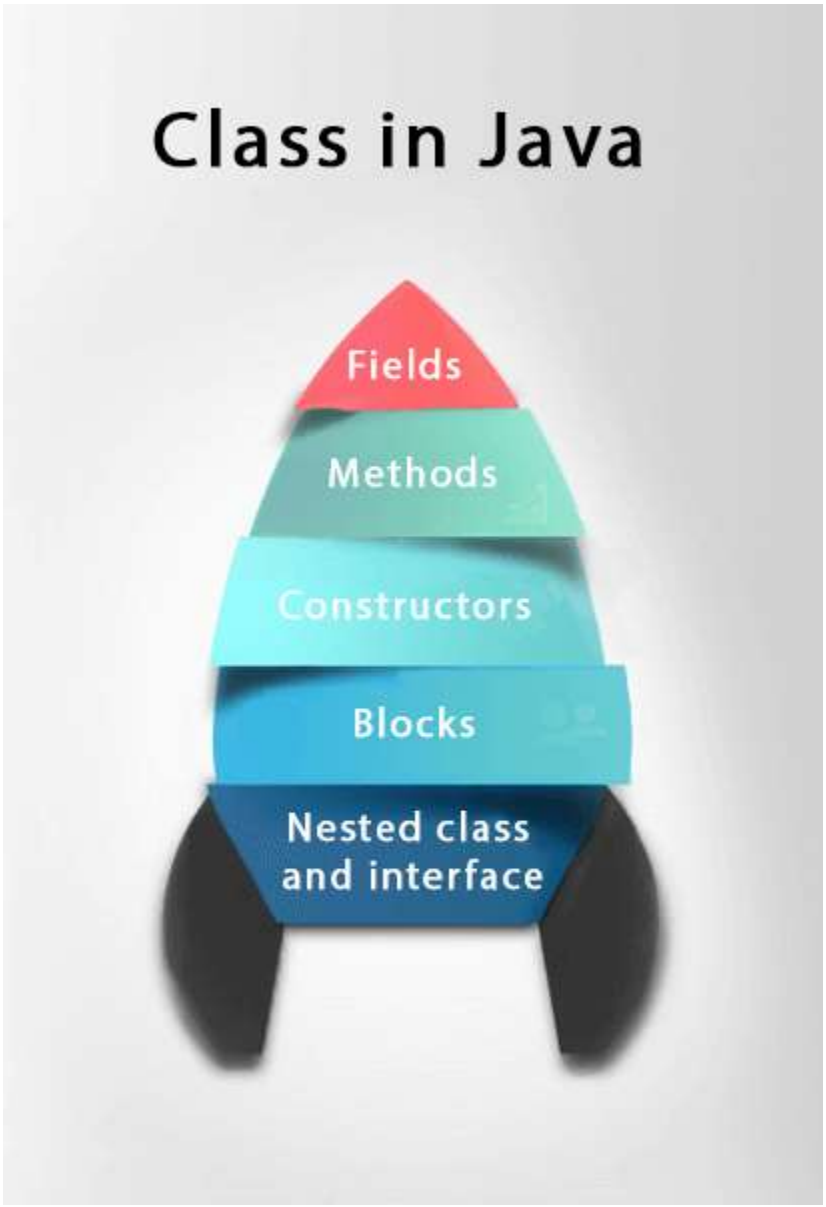
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**



Syntax to declare a class:

```
1. class <class_name>{
2.     field;
```

3. method;
4. }

Create a Class

To create a class, use the keyword `class`:

```
public class MyClass {

  int x = 5;

}
```

Create an Object

In Java, an object is created from a class. We have already created the class named `MyClass`, so now we can use this to create objects.

To create an object of `MyClass`, specify the class name, followed by the object name, and use the keyword `new`:

```
public class MyClass {

  int x = 5;


  public static void main(String[] args) {

    MyClass myObj = new MyClass();

    System.out.println(myObj.x);

  }
```

```
}
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

---

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
1. //Java Program to illustrate how to define a class and fields
2. //Defining a Student class.
3. class Student{
4.  //defining fields
5.  int id;//field or data member or instance variable
```

```
6.  String name;
7.  //creating main method inside the Student class
8.  public static void main(String args[]){
9.   //Creating an object or instance
10.  Student s1=new Student();//creating an object of Student
11.  //Printing values of the object
12.  System.out.println(s1.id);//accessing member through reference variable
13.  System.out.println(s1.name);
14. }
15.}
```

Test it Now

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```
1.  //Java Program to demonstrate having the main method in
2.  //another class
3.  //Creating Student class.
4.  class Student{
5.   int id;
6.   String name;
7.  }
8.  //Creating another class TestStudent1 which contains the main method
9.  class TestStudent1{
10. public static void main(String args[]){
11.  Student s1=new Student();
12.  System.out.println(s1.id);
```

```
13.   System.out.println(s1.name);
14.  }
15. }
```

Test it Now

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
1.  class Student{
2.   int id;
3.   String name;
4.  }
5.  class TestStudent2{
6.   public static void main(String args[]){
7.    Student s1=new Student();
8.    s1.id=101;
9.    s1.name="Sonoo";
10.   System.out.println(s1.id+" "+s1.name);//printing members with a white space
11.  }
12. }
```

Test it Now

Output:

101 Sonoo

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent3{
6.  public static void main(String args[]){
7.   //Creating objects
8.   Student s1=new Student();
9.   Student s2=new Student();
10.  //Initializing objects
11.  s1.id=101;
12.  s1.name="Sonoo";
13.  s2.id=102;
14.  s2.name="Amit";
15.  //Printing data
16.  System.out.println(s1.id+" "+s1.name);
17.  System.out.println(s2.id+" "+s2.name);
18. }
19.}
```

Test it Now

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

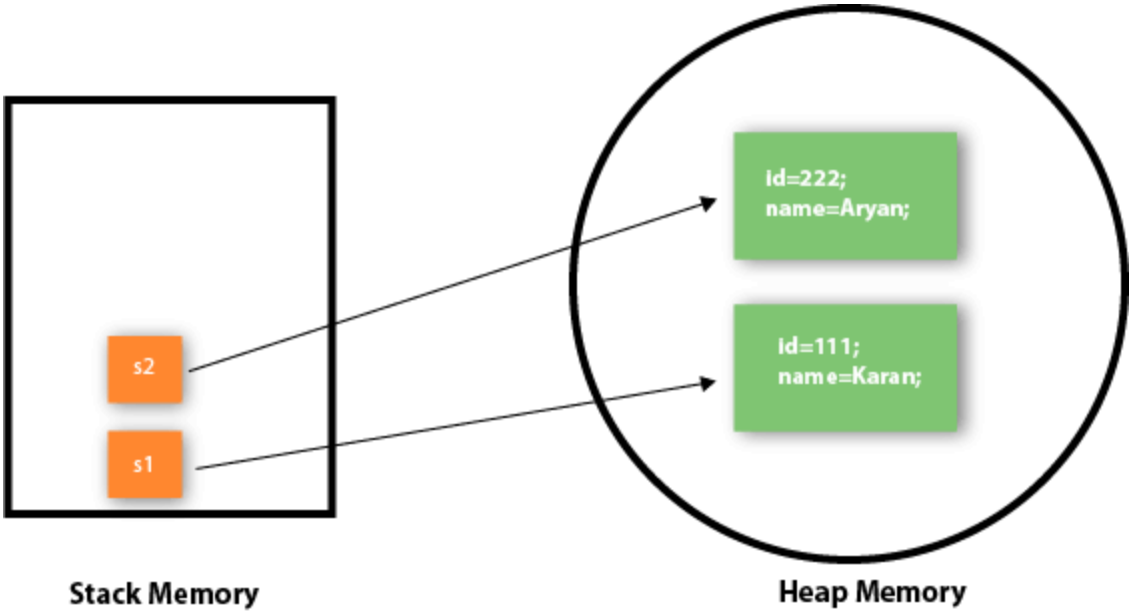
File: TestStudent4.java

```
1. class Student{
2.  int rollno;
3.  String name;
4.  void insertRecord(int r, String n){
5.   rollno=r;
6.   name=n;
7.  }
8.  void displayInformation(){System.out.println(rollno+" "+name);}
9. }
10. class TestStudent4{
11. public static void main(String args[]){
12.  Student s1=new Student();
13.  Student s2=new Student();
14.  s1.insertRecord(111,"Karan");
15.  s2.insertRecord(222,"Aryan");
16.  s1.displayInformation();
17.  s2.displayInformation();
18. }
19. }
```

Test it Now

Output:

```
111 Karan
222 Aryan
```



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

---

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
1.  class Employee{
2.      int id;
3.      String name;
4.      float salary;
5.      void insert(int i, String n, float s) {
6.          id=i;
7.          name=n;
8.          salary=s;
9.      }
10.     void display(){System.out.println(id+" "+name+" "+salary);}
```

```
11. }
12. public class TestEmployee {
13. public static void main(String[] args) {
14.     Employee e1=new Employee();
15.     Employee e2=new Employee();
16.     Employee e3=new Employee();
17.     e1.insert(101,"ajeet",45000);
18.     e2.insert(102,"irfan",25000);
19.     e3.insert(103,"nakul",55000);
20.     e1.display();
21.     e2.display();
22.     e3.display();
23. }
24. }
```

Test it Now

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```
1.  class Rectangle{
2.   int length;
3.   int width;
4.   void insert(int l, int w){
5.    length=l;
6.    width=w;
7.   }
8.   void calculateArea(){System.out.println(length*width);}
9.  }
10. class TestRectangle1{
11.  public static void main(String args[]){
```

```
12.  Rectangle r1=new Rectangle();
13.  Rectangle r2=new Rectangle();
14.  r1.insert(11,5);
15.  r2.insert(3,15);
16.  r1.calculateArea();
17.  r2.calculateArea();
18. }
19. }
```

Test it Now

Output:

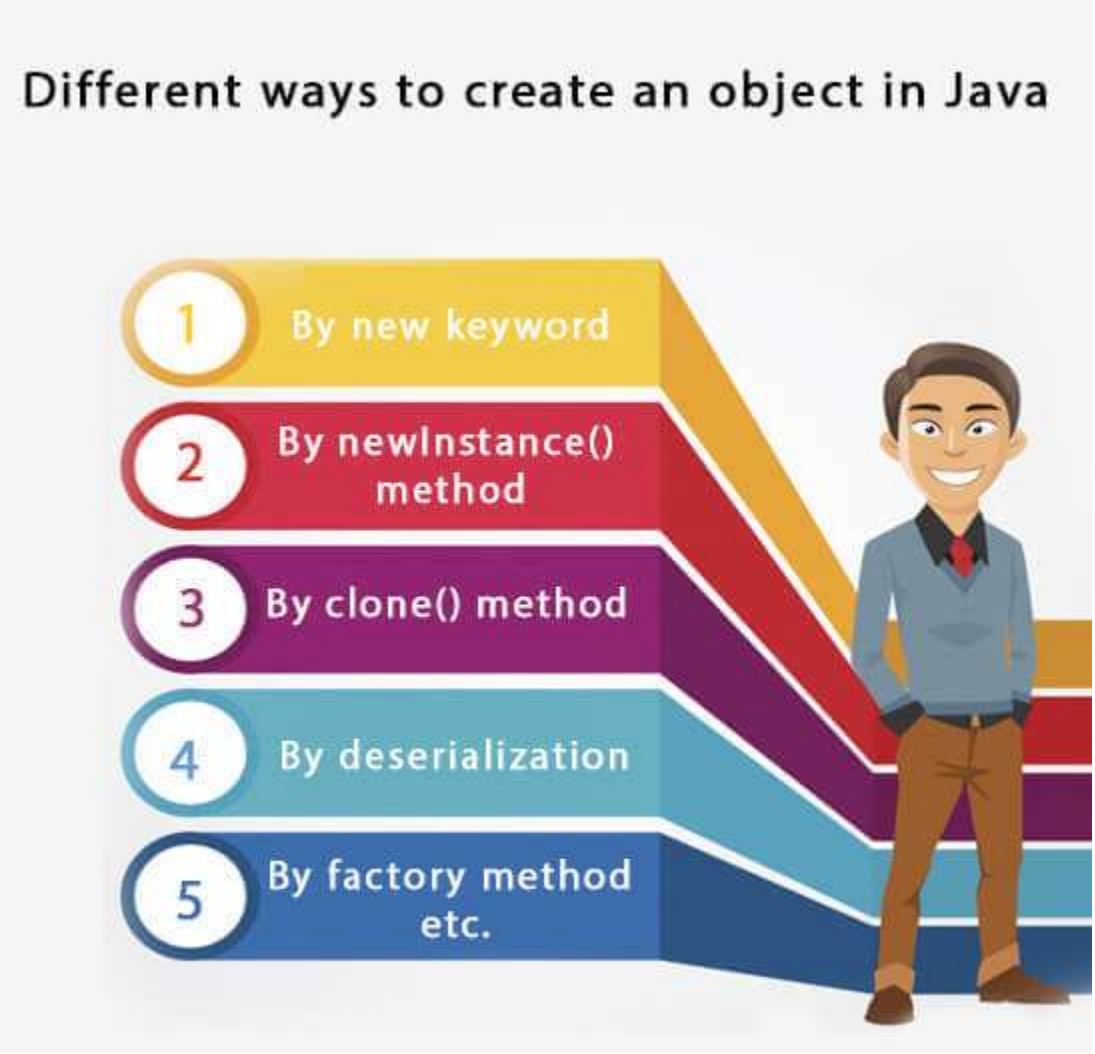
```
55
45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.



Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
1. new Calculation();//anonymous object
```

Calling method through a reference:

```
1. Calculation c=new Calculation();
2. c.fact(5);
```

Calling method through an anonymous object

```
1. new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
1. class Calculation{
2.  void fact(int  n){
3.   int fact=1;
4.   for(int i=1;i<=n;i++){
5.    fact=fact*i;
6.   }
7.  System.out.println("factorial is "+fact);
8. }
9. public static void main(String args[]){
10.  new Calculation().fact(5);//calling method with anonymous object
11. }
12. }
```

Output:

Factorial is 120

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
1. int a=10, b=20;
```

Initialization of refernce variables:

```
1. Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Let's see the example:

```
1. //Java Program to illustrate the use of Rectangle class which
2. //has length and width data members
3. class Rectangle{
4.  int length;
5.  int width;
6.  void insert(int l,int w){
7.   length=l;
8.   width=w;
9.  }
10. void calculateArea(){System.out.println(length*width);}
11. }
12. class TestRectangle2{
13. public static void main(String args[]){
14.  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
15.  r1.insert(11,5);
16.  r2.insert(3,15);
17.  r1.calculateArea();
18.  r2.calculateArea();
19. }
20. }
```

Test it Now

Output:

```
55
45
```

Real World Example: Account

File: TestAccount.java

```
1. //Java Program to demonstrate the working of a banking-system
```

```
2. //where we deposit and withdraw amount from our account.
3. //Creating an Account class which has deposit() and withdraw() methods
4. class Account{
5. int acc_no;
6. String name;
7. float amount;
8. //Method to initialize object
9. void insert(int a,String n,float amt){
10. acc_no=a;
11. name=n;
12. amount=amt;
13. }
14. //deposit method
15. void deposit(float amt){
16. amount=amount+amt;
17. System.out.println(amt+" deposited");
18. }
19. //withdraw method
20. void withdraw(float amt){
21. if(amount<amt){
22. System.out.println("Insufficient Balance");
23. }else{
24. amount=amount-amt;
25. System.out.println(amt+" withdrawn");
26. }
27. }
28. //method to check the balance of the account
29. void checkBalance(){System.out.println("Balance is: "+amount);}
30. //method to display the values of an object
31. void display(){System.out.println(acc_no+" "+name+" "+amount);}
32. }
33. //Creating a test class to deposit and withdraw amount
34. class TestAccount{
35. public static void main(String[] args){
36. Account a1=new Account();
37. a1.insert(832345,"Ankit",1000);
```

```
38. a1.display();
39. a1.checkBalance();
40. a1.deposit(40000);
41. a1.checkBalance();
42. a1.withdraw(15000);
43. a1.checkBalance();
44. }}
```

Test it Now

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Constructors in Java

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

```
public class Box

{

int l,b,h;

public Box()

{l=10; b=20; h=30;

   System.out.println(l+" "+b+" "+h);

}
```

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

{

Box b1= new Box();

}

}
```

```
// Create a MyClass class
public class MyClass {
  int x;  // Create a class attribute

  // Create a class constructor for the MyClass class
  public MyClass() {
    x = 5;  // Set the initial value for the class attribute x
  }

  public static void main(String[] args) {
    MyClass myObj = new MyClass(); // Create an object of class MyClass (This
will call the constructor)
    System.out.println(myObj.x); // Print the value of x
  }
}
```

```
// Outputs 5
```

Note that the constructor name must match the class name, and it cannot have a return type (like `void`).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Types of Java constructors

There are two types of constructors in Java:

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

1. Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

1. **<class_name>(){}**

Example of default constructor

//Java Program to create and call a default constructor

1. **class Bike1{**
2. **//creating a default constructor**

```
3. Bike1()
4. {System.out.println("Bike is created");
5. }
6. //main method
7. public static void main(String args[]){
8. //calling a default constructor
9. Bike1 b=new Bike1();
10.}
11.}
```

Output:

Bike is created

> Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

2. Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.The following example adds an `int y` parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

Example 1.

```
public class Box
{
int l,b,h;
public Box()
{int l=0, b=0, h=0;
   System.out.println(l+" "+b+" "+h);
}
public Box(int L,int B, int H)
{  l=L;
   b=B;
   h=H;
   System.out.println(l+" "+b+" "+h);
}
public static void main(String []args)
{
Box b1= new Box();
Box b2= new Box(10,20,30);
}
}
```

Ex-2:

```
public class MyClass {

  int x;

  public MyClass(int y) {

    x = y;

  }

 public static void main(String[] args) {

    MyClass myObj = new MyClass(5);

    System.out.println(myObj.x);

  }

}// Outputs 5
```

Example 2:

```
public class Car {
  int modelYear;
  String modelName;

  public Car(int year, String name) {
    modelYear = year;
    modelName = name;
  }

  public static void main(String[] args) {
    Car myCar = new Car(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}
// Outputs 1969 Mustang
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor **overloading in Java** is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

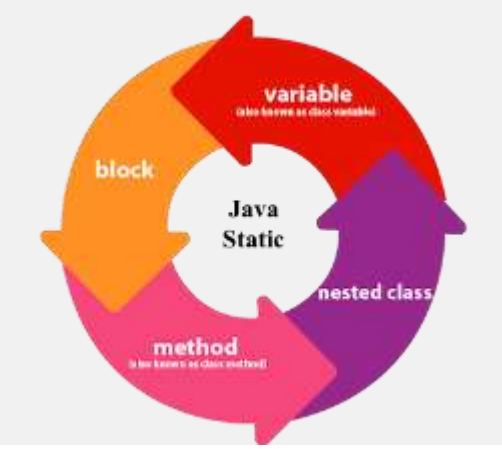
```
1. //Java program to overload constructors
2. class Student5{
3.    int id;
4.    String name;
5.    int age;
6.    //creating two arg constructor
7.    Student5(int i,String n){
8.    id = i;
9.    name = n;
10.   }
11.   //creating three arg constructor
12.   Student5(int i,String n,int a){
13.   id = i;
14.   name = n;
15.   age=a;
16.   }
17.   void display(){System.out.println(id+" "+name+" "+age);}
18.
19.   public static void main(String args[]){
20.   Student5 s1 = new Student5(111,"Karan");
21.   Student5 s2 = new Student5(222,"Aryan",25);
22.   s1.display();
23.   s2.display();
24.  }
25.}
```

Java static keyword

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

The static can be:

1. **Variable (also known as a class variable)**
2. **Method (also known as a class method)**
3. **Block**
4. **Nested class**



1) Java static variable

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

- **The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.**
- **The static variable gets memory only once in the class area at the time of class loading.**

Advantages of static variable

It makes your program memory efficient (i.e., it saves memory).

Understanding the problem without static variable

```
class Student{
    int rollno;  // instance variable
    String name;
    String college="NIEC";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

> Java static property is shared to all objects.

Example of static variable

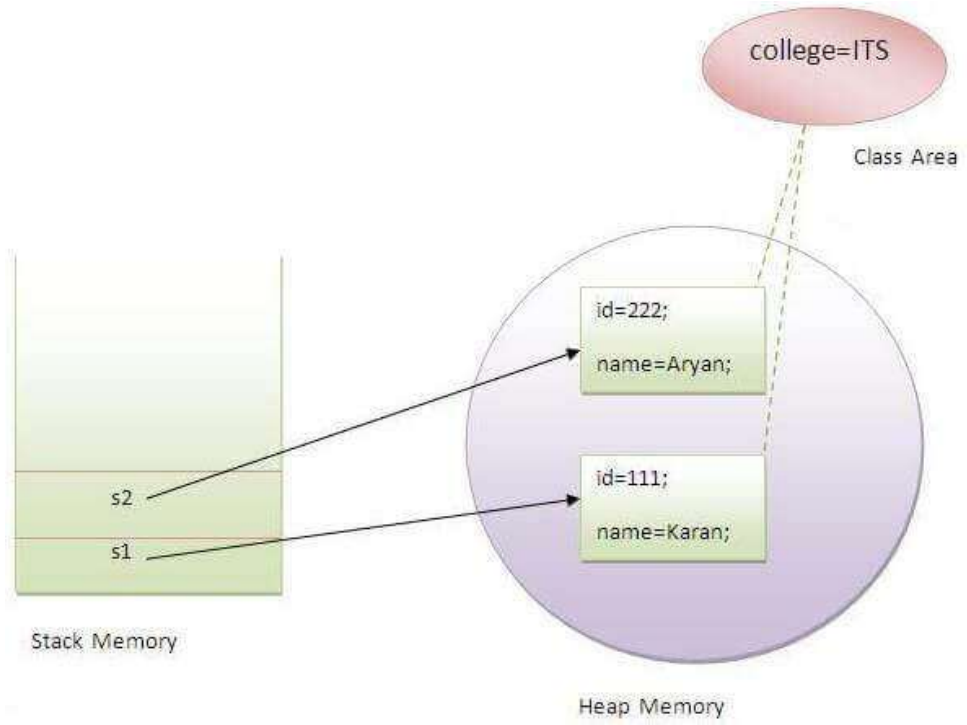
```
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
```

```
22. s2.display();
23. }
24.}
```

Output:

111 Karan ITS

222 Aryan ITS



Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
1. //Java Program to demonstrate the use of an instance variable
```

```
2. //which get memory each time when we create an object of the class.
3. class Counter{
4. int count=0;//will get memory each time when the instance is created
5. Counter(){
6. count++;//incrementing value
7. System.out.println(count);
8. }
9.
10.public static void main(String args[]){
11.//Creating objects
12.Counter c1=new Counter();
13.Counter c2=new Counter();
14.Counter c3=new Counter();
15.}
16.}
```

Output:

```
1

1

1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
1. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.
3. class Counter2{
4. static int count=0;//will get memory only once and retain its value
5. Counter2(){
6. count++;//incrementing the value of static variable
7. System.out.println(count);
8. }
```

```
9.  public static void main(String args[]){
10.//creating objects
11.Counter2 c1=new Counter2();
12.Counter2 c2=new Counter2();
13.Counter2 c3=new Counter2();
14.}
15.}
```

Output:

```
1

2

3
```

---

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

Example of static method

```
class MathOperation
{
   static float mul(float x, float y)
   {
      return x*y;
   }

   static float divide(float x, float y)
   {
      return x/y;
   }
}
```

```
public class HelloWorld{

    public static void main(String []args){
       float a= MathOperation.mul(4.0f, 5.0f);
       float b= MathOperation.divide(a, 2.0f);


      System.out.println("b=  "+b);
    }
}
```

```
1.  //Java Program to demonstrate the use of a static method.
2.  class Student{
3.      int rollno;
4.      String name;
5.      static String college = "NIEC";
6.      //static method to change the value of static variable
7.      static void change(){
8.      college = "ADGITM";
9.      }
10.     //constructor to initialize the variable
11.     Student(int r, String n){
12.     rollno = r;
13.     name = n;
14.     }
15.     //method to display values
16.     void display(){System.out.println(rollno+" "+name+" "+college);}
17.}
18.//Test class to create and display the values of object
19.public class TestStaticMethod{
20.    public static void main(String args[]){
21.    Student.change();//calling change method
22.    //creating objects
23.    Student s1 = new Student(111,"A");
24.    Student s2 = new Student(222,"B");
25.    Student s3 = new Student(333,"C");
26.    //calling display method
```

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

Test it Now

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number using the static method

class Calculate{
 static int cube(int x){
 return x*x*x;
 }

 public static void main(String args[]){
 int result=Calculate.cube(5);  // or int result=cube(5)
 System.out.println(result);
 }
}
```

Output:125

Restrictions for the static method

1. **The static method can not use non static data member or call non-static method directly.**

2. this and super cannot be used in static context.

```
1. class A{
2.  int a=40;//non static
3.
4.  public static void main(String args[]){
5.   System.out.println(a);
6.  }
7. }
```

Output:Compile Time Error

---

3) Java static block

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

Example of static block

```
//creating the static block
static {
p = 18;
System.out.println("This is the static block!");
}
// end of static block
}
public class Main {
public static void main(String args[]) {
// Accessing p without creating an object
System.out.println(Static.p);
}
}
```

```
1. public class A2{
2.  static
3.  {
4.     System.out.println("static block is invoked");
```

```
5.
6.  }
7.  public static void main(String args[]){
8.   System.out.println("Hello main");
9.  }
10.}
11.
12.
```

Output:static block is invoked

Hello main

Instance Initialization Block

```
public class Test{
public  int x;
{
  System.out.println("initialization block: x= " +x);
x=5;
}

public Test()
{
    System.out.println("Constructor: x ="+x);

}

public static void main(String [] args)
{
    Test t1= new Test();
    Test t2= new Test();
}
}
```

Output:

```
initialization block: x= 0
Constructor: x =5
initialization block: x= 0
Constructor: x =5
```

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the **main method**.

```
1. class A3{
2.  static{
3.  System.out.println("static block is invoked");
4.  System.exit(0);
5.  }
6. }
```

Output:

```
static block is invoked
```

Access Modifiers in Java

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor , variable , method or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

1. **Default:** When no access modifier is specified for a class , method or data member – It is said to be having the default access modifier by default.

- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible only within the same package.

2. In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of second package.

```
//Java program to illustrate default modifier

package p1;

//Class Geeks is having Default access modifier

class Example

{

    void display()

        {

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

        }

}
```

```
//Java program to illustrate error while using class fr
different package with
//default modifier
package p2;
import p1.*;

//This class is having default access modifier
class ExampleNew
{
    public static void main(String args[])
        {
            //accessing class Example from package p1
            Example obj = new Example();

            obj.display();
        }
}
```

Output: `Compile time error`

3. **Private**: The private access modifier is specified using the keyword private.
   - The methods or data members declared as private are accessible only within the class in which they are declared.
   - Any other class of same package will not be able to access these members.
   - Top level Classes or interface can not be declared as private because
     1. **private means "only visible within the enclosing class".**
     2. **protected means "only visible within the enclosing class and any subclasses"**
   - Hence these modifiers in terms of application to classes, they apply only to nested classes and not on top level classes

In this example, we will create two classes A and B within same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

```
//Java program to illustrate error while using class fro
with
//private modifier
package p1;
  class A
{
   private void display()
     {
         System.out.println("GeeksforGeeks");
     }
}

class B
{
   public static void main(String args[])
       {
           A obj = new A();
           //trying to access private method of another
class
           obj.display();
       }
}
```

Output:

```
error: display() has private access in A
        obj.display();
```

4.
5. **protected: The protected access modifier is specified using the keyword protected.**

- **The methods or data members declared as protected are accessible within same package or sub classes in different package.**

6. **In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.**

```
//Java program to illustrate
//protected modifier
package p1;

//Class A
public class A
{
   protected void display()
    {
        System.out.println("GeeksforGeeks");
    }
}
```

```
//Java program to illustrate

//protected modifier

package p2;

import p1.*; //importing all classes in package p1


//Class B is subclass of A

class B extends A

{

   public static void main(String args[])

    {

        B obj = new B();

        obj.display();

    }


}
```

Output:

```
GeeksforGeeks
```

7. **public: The public access modifier is specified using the keyword public.**
   - **The public access modifier has the widest scope among all other access modifiers.**
   - **Classes, methods or data members which are declared as public are accessible from every where in the program. There is no restriction on the scope of a public data members**

```
//Java program to illustrate public modifier

package p1;

public class A

{

   public void display()

       {

           System.out.println("GeeksforGeeks");

       }

}

package p2;

import p1.*;

class B

{

    public static void main(String args[])

       {

           A obj = new A;

           obj.display();

       }

}
```

Output:

```
GeeksforGeeks
```

8. **Important Points:**

- **If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.**
- **Avoid public fields except for constants.**

Modifiers

By now, you are quite familiar with the `public` keyword that appears in almost all of our examples:

```
public class MyClass
```

The `public` keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- Access Modifiers - controls the access level
- Non-Access Modifiers - do not control access level, but provides other functionality

---

Access Modifiers

For classes, you can use either `public` or *default*:

Modifier	Description

`public`	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

For attributes, methods and constructors, you can use the one of the following:

Modifier	**Description**
`public`	The code is accessible for all classes
`private`	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you don't specify a modifier.
`protected`	The code is accessible in the same package and subclasses.

Non-Access Modifiers

For classes, you can use either `final` **or** `abstract`**:**

Modifier	**Description**
`final`	The class cannot be inherited by other classes
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.

For attributes and methods, you can use the one of the following:

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified
`static`	Attributes and methods belongs to the class, rather than an object
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example abstract void run();. The body is provided by the subclass (inherited from).

Final

If you don't want the ability to override existing attribute values, declare attributes as `final`:

Example

```
public class MyClass {
  final int x = 10;
  final double PI = 3.14;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 50; // will generate an error: cannot assign a value to a final
variable
```

```
    myObj.PI = 25; // will generate an error: cannot assign a value to a final
variable
    System.out.println(myObj.x);
  }
}
```

Static

A `static` method means that it can be accessed without creating an object of the class, unlike `public`:

Example

An example to demonstrate the differences between `static` and `public` methods:

```
public class MyClass {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating
objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[ ] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would output an error

    MyClass myObj = new MyClass(); // Create an object of MyClass
    myObj.myPublicMethod(); // Call the public method
  }
}
```

Or

class MyClass {

```
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }
}

public class Example{


  // Main method
  public static void main(String[ ] args) {
    MyClass.myStaticMethod(); // Call the static method
    // myPublicMethod(); This would output an error

    MyClass myObj = new MyClass(); // Create an object of MyClass
    myObj.myPublicMethod(); // Call the public method
  }
}
```

Abstract

The `abstract` keyword is a non-access modifier, used for classes and methods.

Class: An abstract class is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

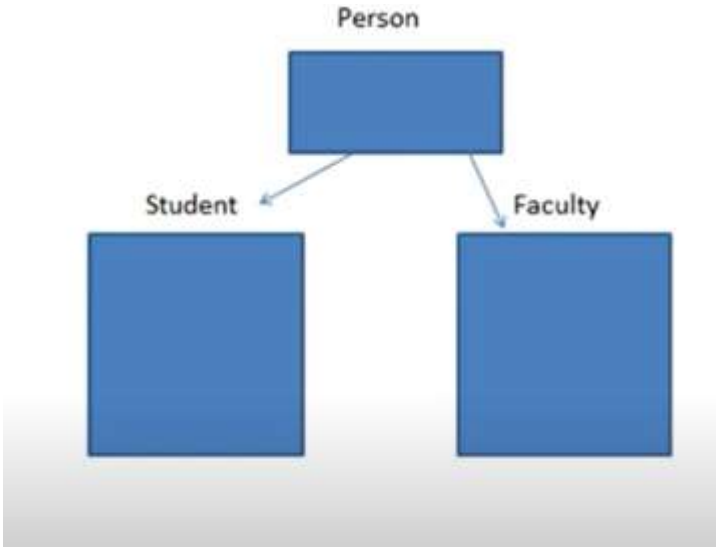
Method: An abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An `abstract` method belongs to an `abstract` class, and it does not have a body. The body is provided by the subclass.

Abstract class in Java is similar to interface except that it can contain default method implementation. An abstract class can have an abstract method without body and it can have methods with implementation also.

abstract keyword is used to create a abstract class and method. Abstract class in java can't be instantiated. **An abstract class is mostly used to provide a base for subclasses to extend and implement the abstract methods and override or use the implemented methods in abstract class**

```
abstract class Person{
    private String name;
    private int age;
    public void setName(String n)  { name=n; }
    public void setAge(int a)  { age=a; }
}

class AbstractExample1{
    public static void main(String[] args)
    { Person p=new Person(); } //can't instantiated
}
```



Example `// abstract class`

o

Inner Classes in Java Programming

If one class is existing within another class is known as inner class or nested class. Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class **private, it cannot be accessed from an object outside the class.**

Syntax

```
class  Outerclass_name
{
.....
.....
```

```
   class  Innerclass_name1
   {
  .....
   .....
     }
    class  Innerclass_name1
   {
   .....
   .....
     }
.....
}
```

The main purpose of using inner class

- To provide more security by making those inner class properties specific to only outer class but not for external classes.
- To make more than one property of classes private properties.

Private is a keyword in java language, it is preceded by any variable that property can be access only within the class but not outside of it (provides more security).

Java Inner Classes

1. **Non Static Inner Class**

Any non-static nested class is known as inner class in java. Java inner class is associated with the object of the class and **they can access all the variables and methods of the outer class.** Since inner classes are associated with the instance, we **can't have any static variables** in them.

The object of java inner class are part of the outer class object and to create an instance of the inner class, we first need to create an instance of outer class.

Java inner class can be instantiated like this;

```
OuterClass outerObject = new OuterClass(); //object of outer
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

Example

```
class OuterClass {
  int x = 10;
class InnerClass {
    int y = 5;
  }
}
public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
// Outputs 15 (5 + 10)
```

EXAMPLE 2:

```
class Outer
 {
     class Inner{
          int in=10;
     void infunc()
     {
          System.out.println("inside Inner claass");
          System.out.println("in=  " +in);
     }}
 }
   public class Example
 {
     public static void main(String []args)
     {
          Outer out= new Outer();
          Outer.Inner o1= out.new Inner();
          o1.infunc();
          System.out.println("inside main, inn=  " + o1.in);
     }
 }
```

```
Output:

inside Inner claass

in=  10

inside main, inn=  10
```

Private data member

```
class Outer
 {
     class Inner{
    private int in=10;
     void infunc()
     {
          System.out.println("inside Inner claass");
          System.out.println("in=  " +in);
     }}
 }
   public class Example
 {
     public static void main(String []args)
     {
          Outer out= new Outer();
          Outer.Inner o1= out.new Inner();
```

```
        o1.infunc();
        //System.out.println("inside main, inn=  " + o1.in);
    }
}
```

Static Inner class

If the nested class is static, then it's called a static nested class. **Static nested classes can access only static members of the outer class**. A static nested class is the same as any other top-level class and is nested for only packaging convenience.

A static class object can be created with the following statement.

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

An inner class can also be `static`, which means that you can access it without creating an object of the outer class:

Example

```
class OuterClass {
  int x = 10;
static class InnerClass {
    int y = 5;
  }
}
```

```
public class MyMainClass {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}
// Outputs 5
```

Note: just like `static` attributes and methods, a `static` inner class does not have access to members of the outer class.

Private Inner Class

Unlike a "regular" class, an inner class can be `private` or `protected`. If you don't want outside objects to access the inner class, declare the class as `private`:

Example

```
class OuterClass {
  int x = 10;
private class InnerClass {
    int y = 5;
  }
```

```
}

public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

If you try to access a private inner class from an outside class (MyMainClass), an error occurs:

```
MyMainClass.java:12: error: OuterClass.InnerClass has private access in OuterClass
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
```

```
class Outer_Demo {
  int num;

  // inner class
  private class Inner_Demo {
    public void print() {
      System.out.println("This is an inner class");
```

```
    }
  }

  // Accessing he inner class from the method within
  void display_Inner() {
    Inner_Demo inner = new Inner_Demo();
    inner.print();
  }
}

public class My_class {

  public static void main(String args[]) {
    // Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();

    // Accessing the display_Inner() method.
    outer.display_Inner();
  }
```

```
}
```

Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

Example

```
class OuterClass {
  int x = 10;


  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
```

```
        System.out.println(myInner.myInnerMethod());

    }

}
```

```
// Outputs 10
```

Java Anonymous Class

In Java, a class can contain another class known as nested class. It's possible to create a nested class without giving any name.

A nested class that doesn't have any name is known as an **anonymous class.**

An anonymous class must be defined inside another class. Hence, it is also known as an **anonymous inner class**. Its syntax is:

```
class outerClass {

    // defining anonymous class
    object1 = new Type(parameterList) { //body };
}
```

Anonymous classes usually extend subclasses or implement interfaces.

Here, Type can be

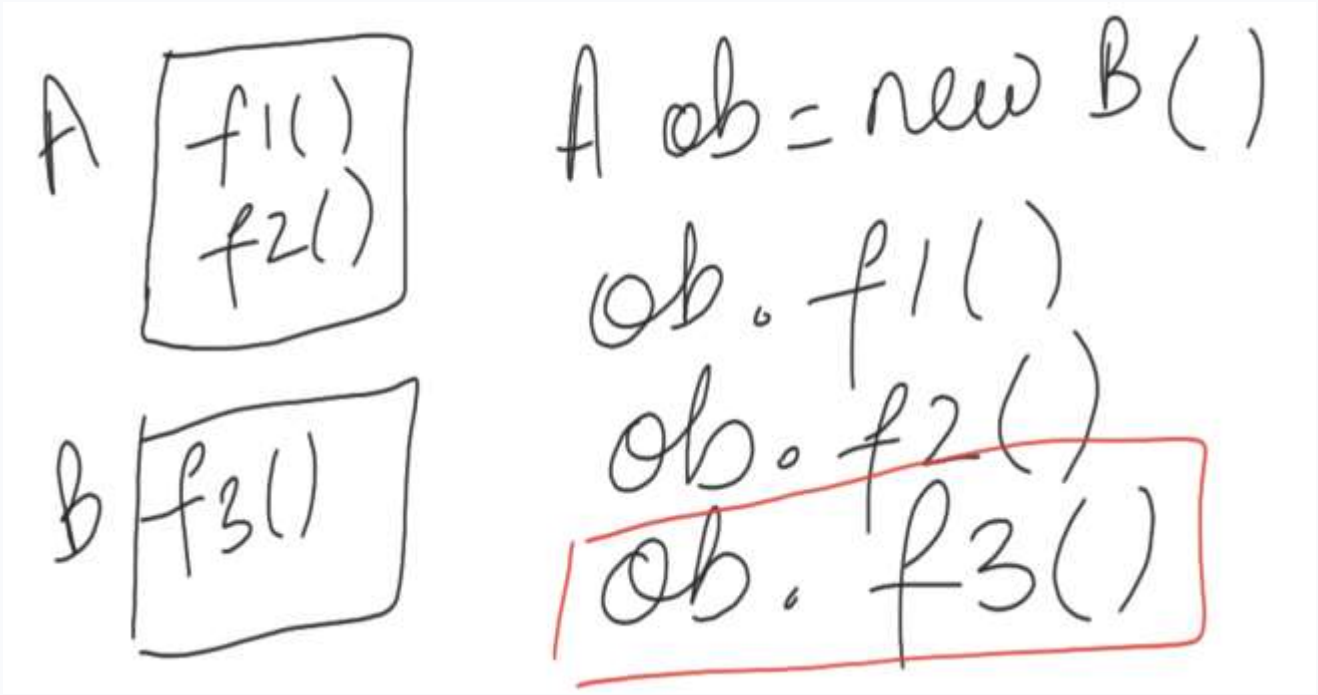
1. a superclass that an anonymous class extends
2. an interface that an anonymous class implements

The above code creates an object, `object1`, of an anonymous class at runtime.

Note: Anonymous classes are defined inside an expression. So, the semicolon is used at the end of anonymous classes to indicate the end of the expression.

2 Important points:

1. **Anonymous class is always inner class.**
2. **Anonymous class is sub class or child class of some other class.**



Have created the object of B, but address is stored in A. ie reference variable.

Reference variable is ob, of type A. So, ob can only access the functions from class A. ob.f3() will give error coz if there is reference variable of parent (A) and it is referring object of child (ie B). So, they can't call the methods of original methods made in child class. (Concept of Early Binding)

When will child methods call?

When we override the version of parent class. Always, there is late binding occurs for overridden functions. So, in this case, it will not make basis for reference type (A), but instead what that type is pointing to.

```
class A
{
   public void show()
   {
      System.out.println("A show()");
   }
}



public class AnonymousExample{


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

```
}
```

To print child class function;

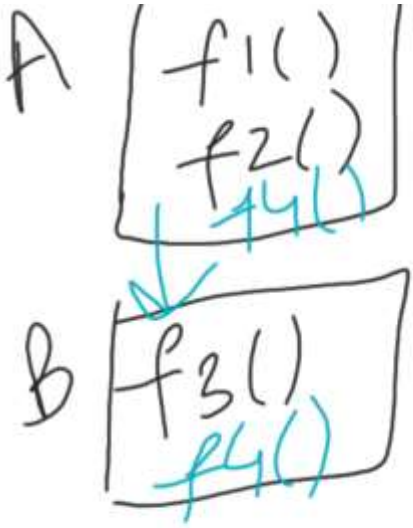
```
class A
{
   public void show()
   {
      System.out.println("A show()");
   }
}

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

```
public class AnonymousExample{


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

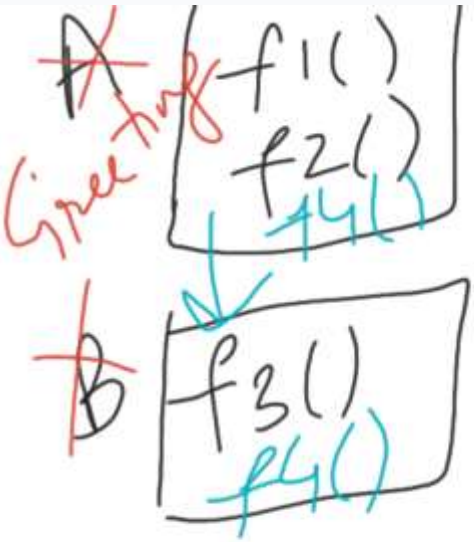
Anonymous Class Example:

```
class A
{
   public void show()
   {
       System.out.println("A show()");
   }
}
```

```
public class AnonymousExample{



    public static void main(String []args){

       A obj=new A(){

          public void show()

         {

             System.out.println("B show()");

          }

       };

       obj.show();

    }

}
```




```
A ob = new B()
ob.f1()
ob.f2()
ob.f3()
ob.f4()
```




```
A ob = new B()
ob.f1()
ob.f2()
ob.f3()
ob.f4()
```

```
class Greeting
{
    public void sayHello()
    {
        System.out.println("hello");
    }
}

class India extends Greeting
{
    Greeting g= new Greeting() //anonymous  class
    {
        public void sayHello()
        {
            System.out.println("Namasty");
        }
    };
}


public class Example
{
     public static void main(String []args){
       Greeting i=new Greeting(); //this will print sayhello of Greeting
       i.sayHello();//this will print sayhello of Greeting
       India in=new India(); //this will print sayhello of Greeting
       in.g.sayHello(); //this will print sayhello of anonymous cls

     }
}
```

Output:

Namasty

Early and Late Binding:

```
// Java program to illustrate concept of early and late binding
```

```
class Vehicle
{
	void start() {
		System.out.println("Vehicle Started");
	}

	static void stop() {
		System.out.println("Vehicle Stopped");
	}
}

class Car extends Vehicle {

	@Override
	void start() {
		System.out.println("Car Started");
	}

	static void stop() {
		System.out.println("Car Stopped");
	}

	public static void main(String args[]) {

		// Car extends Vehicle
		Vehicle vehicle = new Car();

		vehicle.start();
		vehicle.stop();
	}
}
```

Output:

```
Car Started
 Vehicle Stopped
```

Static method cannot be overridden. Ie y stop function of parent class is called (Early binding). But, in non static method, it has overridden and child class of start function is called.(Late Binding)

Java Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An `interface` is a completely "abstract class" that is used to group related methods with empty bodies:

Example

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the `implements` keyword (instead of `extends`). The body of the interface method is provided by the "implement" class:

Example

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
```

```
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

public class MyMainClass {
  public static void main(String[] args) {
     Animal a = new Pig();  // Create a interface ref var
a.animalSound();
    a.sleep();


    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Notes on Interfaces:

- Like abstract classes, **interfaces cannot be used to create objects** (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, **you must override all of its methods**
- Interface methods are by default `abstract` **and** `public`
- Interface attributes are by default `public`, `static` **and** `final`
- An interface **cannot contain a constructor** (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma (see example below).

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }


}
```

Abstract class vs Interface

1. **Type of methods**: Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.

2. **Final Variables**: Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
3. **Type of variables**: Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
4. **Implementation**: Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.
5. **Inheritance vs Abstraction:** A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".
6. **Multiple implementation**: An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.

3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.
9)**Example:** public abstract class Shape{ public abstract void draw();}	**Example:** public interface Drawable{ void draw();}

Java ArrayList

The `ArrayList` class is a resizable array, which can be found in the `java.util` package.ArrayList is a part of **collection framework** and is present in java.util package.

The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an `ArrayList` whenever you want. The syntax is also slightly different:

Example: Create an `ArrayList` object called cars that will store strings:

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>();// Create an ArrayList object
ArrayList<Integer> cars = new ArrayList<Integer>();
Or

// Java program to demonstrate the working of ArrayList in Java

import java.io.*;
import java.util.*;

public class ArrayListExample {
      public static void main(String[] args)
      {
            // Size of the ArrayList
          ArrayList fruit=new ArrayList(10);
          fruit.add("Mango");
          fruit.add(10);
          fruit.add('c');
          fruit.add("Apple");
          System.out.println(fruit);
          System.out.println(fruit.get(0));
          fruit.remove(1);
          fruit.remove("Apple");
          fruit.set(1,"Apple");
          System.out.println(fruit);
          System.out.println(fruit.size());
      }
}
```

Add Items

The `ArrayList` class has many useful methods. For example, to add elements to the `ArrayList`, **use the** `add()` method:

Example

```
import java.util.ArrayList;

public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Access an Item

To access an element in the `ArrayList`, use the `get()` method and refer to the index number:

Example

```
cars.get(0);
```

Remember: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Item

To modify an element, use the `set()` method and refer to the index number:

Example

```
cars.set(0, "Opel");
```

Remove an Item

To remove an element, use the `remove()` method and refer to the index number:

Example

```
cars.remove(0);
```

To remove **all the** elements in the `ArrayList`**, use the** `clear()` method:

Example

```
cars.clear();
```

ArrayList Size

To find out how many elements an ArrayList have, use the `size` method:

Example

```
cars.size();
```

Loop Through an ArrayList

Loop through the elements of an `ArrayList` with a `for` **l**oop, and use the `size()` method to specify how many times the loop should run:

Example

```
public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

You can also loop through an `ArrayList` **w**ith the for-each loop:

Example

```
public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: `Integer`. For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc:

Example

Create an `ArrayList` to store numbers (add elements of type `Integer`):

```
import java.util.ArrayList;

public class MyClass {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

Sort an ArrayList

Another useful class in the `java.util` package is the `Collections` class, which include the `sort()` method for sorting lists alphabetically or numerically:

Example

Sort an ArrayList of Strings:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
```

```
    Collections.sort(cars);  // Sort cars
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Example

Sort an ArrayList of Integers:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class MyClass {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

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

}
```

How ArrayList work internally?

Since ArrayList is a dynamic array and we do not have to specify the size while creating it, the size of the array automatically increases when we dynamically add and remove items. Though the actual library implementation may be more complex, the following is a very basic idea explaining the working of the array when the array becomes full and if we try to add an item:

- Creates a bigger sized memory on heap memory (for example memory of double size).
- Copies the current memory elements to the new memory.
- New item is added now as there is bigger memory available now.
- Delete the old memory.

Inheritance in Java

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

Important terminology:

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is **extends**.

Syntax :

```
class derived-class extends base-class

{

   //methods and fields

}
```

Example: In below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class which extends Bicycle class and class Test is a driver class to run program.

```
//Java program to illustrate the
// concept of inheritance

// base class
class Bicycle
{
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
```

```
        return("No of gears are "+gear
                +"\n"
                + "speed of bicycle is "+speed);
    }
}

// derived class
class MountainBike extends Bicycle
{

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear,int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    // overriding toString() method
    // of Bicycle to print more info
    @Override
```

```
    public String toString()
    {
        return (super.toString()+
                "\nseat height is "+seatHeight);
    }

}

// driver class
public class Test
{
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());

     }
}
```

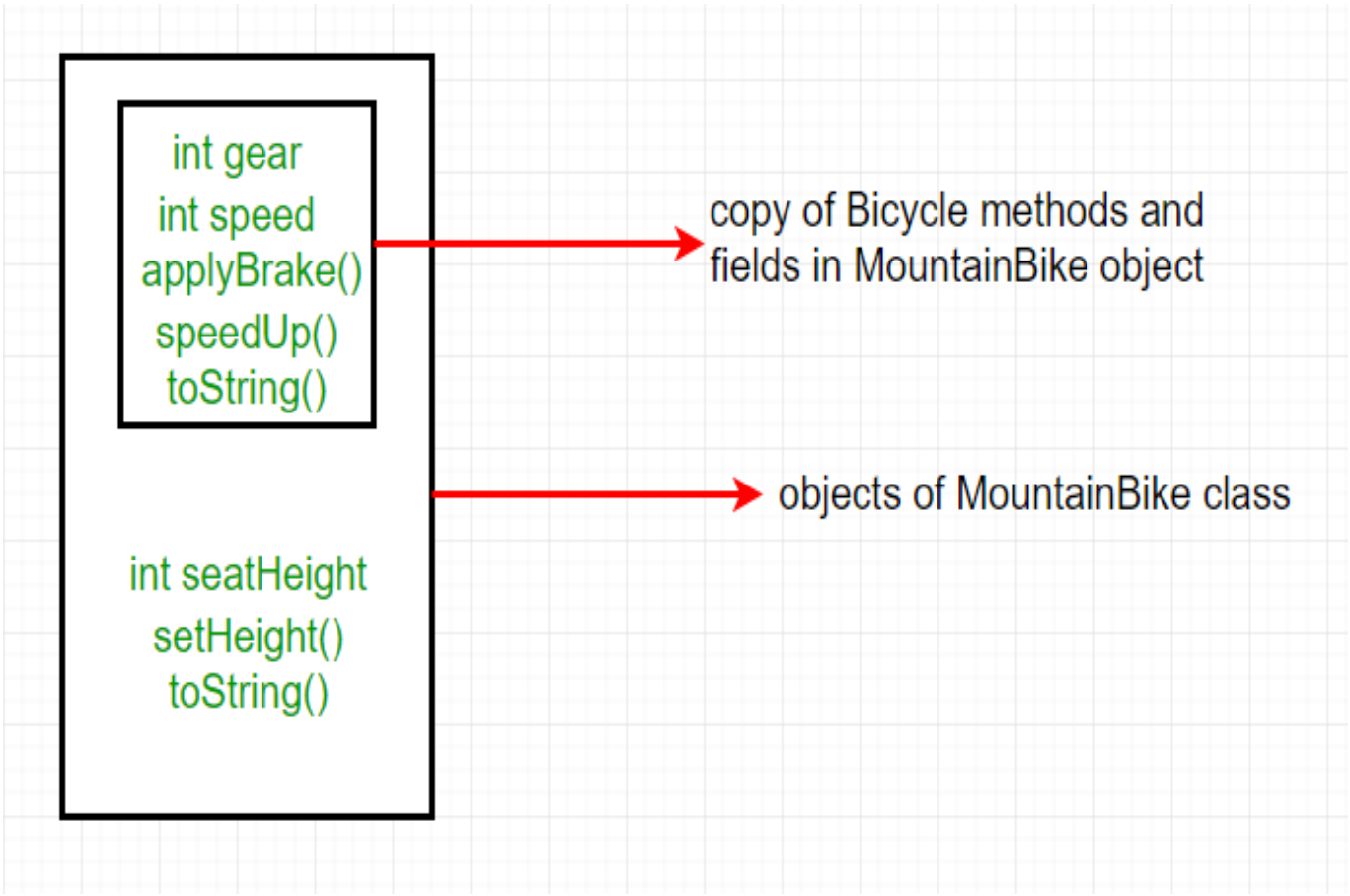
Output:

```
No of gears are 3


speed of bicycle is 100


seat height is 25
```

In above program, when an object of MountainBike class is created, a copy of the all methods and fields of the superclass acquire memory in this object. That is why, by using the object of the subclass we can also access the members of a superclass.

Please note that during inheritance only object of subclass is created, not the superclass.

Illustrative image of the program:



In practice, inheritance and polymorphism are used together in java to achieve fast performance and readability of code.

INHERITANCE

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class.

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Why use inheritance in java

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Syntax:

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Important terminology:

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Simple Example:

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Output:

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Access Modifiers

Access modifiers are simply a keyword in Java that provides accessibility of a class and its member. They set the access level to methods, variable, classes and constructors.

Types of access modifier

There are 4 types of access modifiers available in Java.

- public
- default
- protected
- private

public

The member with public modifiers can be accessed by any classes. The public methods, variables or class have the widest scope.

Example: Sample program for public access modifier

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

{

   // code

}
```

default

When we do not mention any access modifier, it is treated as default. It is accessible only within same package.

Example: Sample program for default access modifier

```
int a = 25;

String str = "Java";

boolean m1()

{

   return true;

}
```

protected

The protected modifier is used within same package. It lies between public and default access modifier. It can be accessed outside the package but through inheritance only.

A class cannot be protected.

Example: Sample program for protected access modifier

```
class Employee

{
```

```
    protected int id = 101;

    protected String name = "Jack";

}

public class ProtectedDemo extends Employee

{

    private String dept = "Networking";

    public void display()

    {

        System.out.println("Employee Id : "+id);

        System.out.println("Employee name : "+name);

        System.out.println("Employee Department : "+dept);

    }

    public static void main(String args[])

    {
```

```
        ProtectedDemo pd = new ProtectedDemo();

        pd.display();

    }

}
```

Output:

```
Employee Id : 101

Employee name : Jack

Employee Department : Networking
```

private

The private methods, variables and constructor are not accessible to any other class. It is the most restrictive access modifier. A class except a nested class cannot be private.

Example: Sample program for private access modifier

```
public class PrivateDemo

{

    private int a = 101;

    private String s = "TutorialRide";

    public void show()

    {

        System.out.println("Private int a = "+a+"\nString s = "+s);

    }

    public static void main(String args[])

    {

        PrivateDemo pd = new PrivateDemo();

        pd.show();

        System.out.println(pd.a+" "+pd.s);

    }
```

```
}
```

Output:

Private int a = 101

String s = TutorialRide

101 TutorialRide

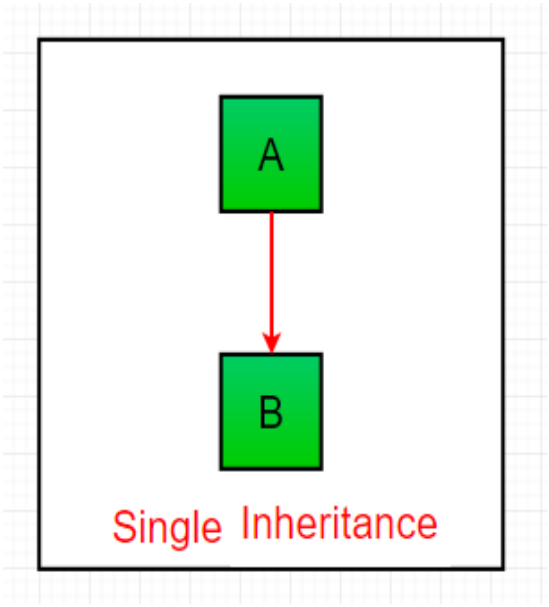
Table for Access Modifier

Access modifier	In class	In package	Outside package by subclass	Outside package
public	Yes	Yes	Yes	No
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No
private	Yes	No	No	No

Types of Inheritance in Java

Below are the different types of inheritance which is supported by Java.

1. **Single Inheritance :** In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

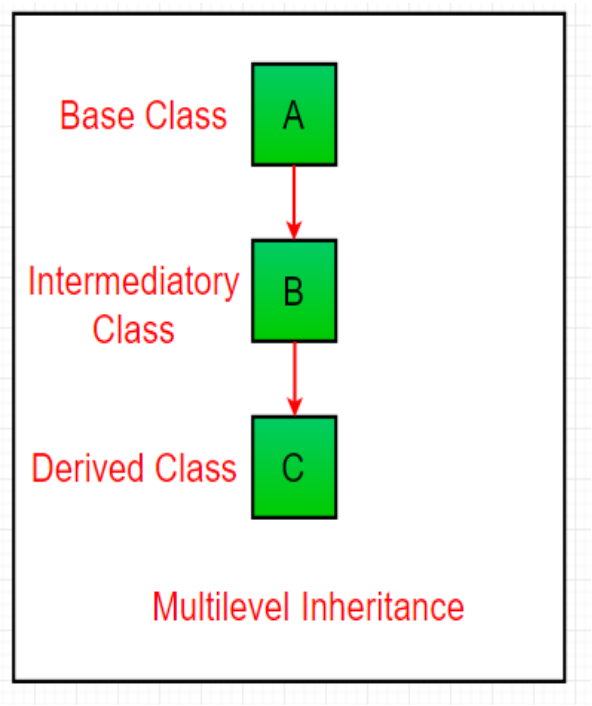



```
//JAVA program to demonstrate Single Inheritance
class Calc    //Super (Java), Parent, Base
{
   public int add(int a, int b)
   {
      return a+b;
   }
}

class CalcAdv extends Calc  //Sub(Java), Child, Derived
{
   public int sub(int a, int b)
   {
      return a-b;
```

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

    CalcAdv c1=new CalcAdv();
    int res_add= c1.add(10,5);
    int res_sub= c1.sub(10,5);
    System.out.println(res_add);
     System.out.println(res_sub);
   }
}
```

Multilevel Inheritance : In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In

Java, a class cannot directly access the grandparent's members.

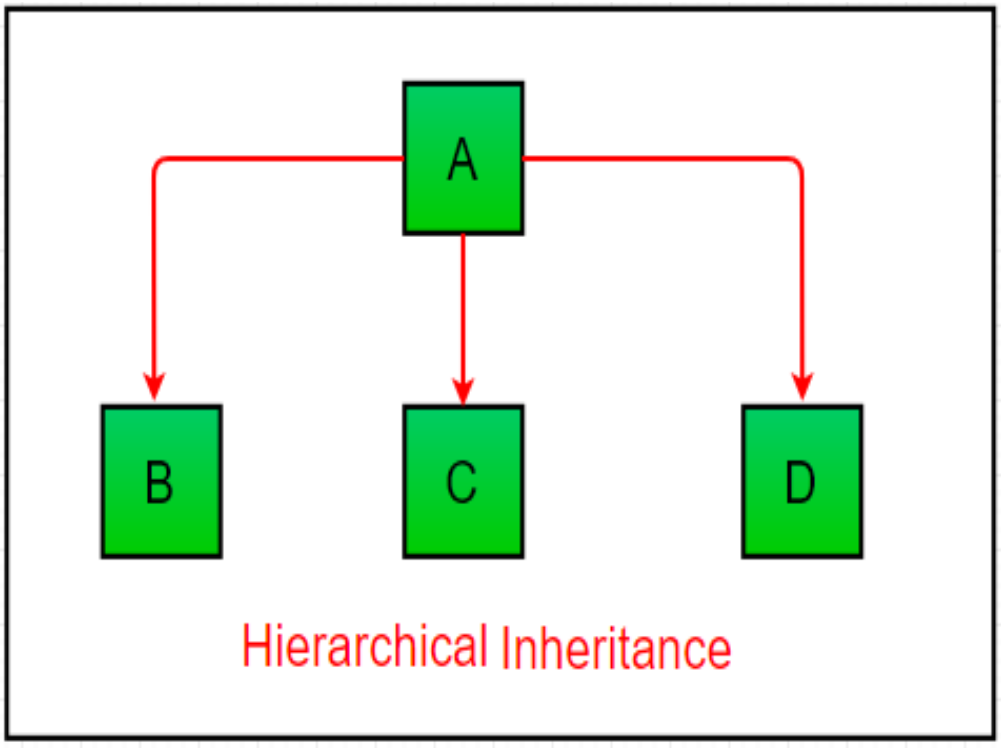



```
//JAVA program to demonstrate Multi-level Inheritance
class Calc    //Super (Java), Parent, Base
{
    public int add(int a, int b)
    {
        return a+b;
    }
}

class CalcAdv extends Calc
```

```
{
    public int sub(int a, int b)
    {
        return a-b;
    }
}

class CalcVeryAdv extends CalcAdv
{
    public int mul(int a, int b)
    {
        return a*b;
    }
}
public class InheritanceDemo
{
   public static void main(String []args)
   {

    CalcVeryAdv c1=new CalcVeryAdv();
```

```
    int res_add= c1.add(10,5);
    int res_sub= c1.sub(10,5);
    int res_mul= c1.mul(10,5);
    System.out.println(res_add);
     System.out.println(res_sub);
     System.out.println(res_mul);
   }
}
```

2. **Hierarchical Inheritance :** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.



Example of Hierarchical Inheritance

We are writing the program where class B, C and D extends class A.

```
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[])
```

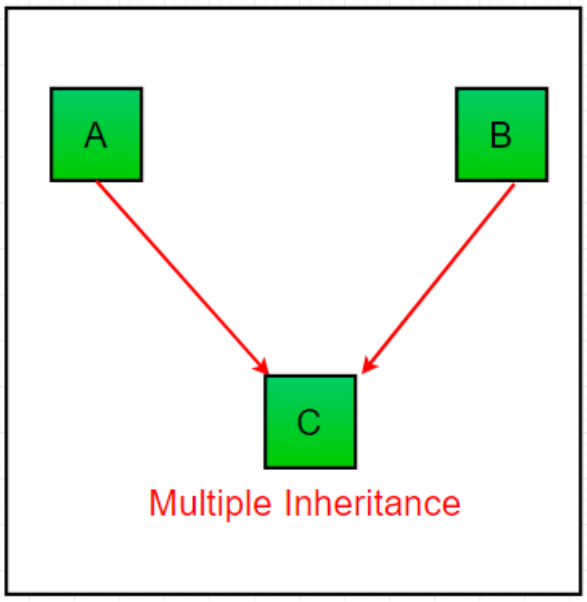
```
  {

      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
```

3. **Multiple Inheritance (Through Interfaces) :** In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



```
// Java program to illustrate the concept of Multiple
inheritance
import java.util.*;
import java.lang.*;
import java.io.*;

interface one
{
    public void print_geek();
}

interface two
{
    public void print_for();
}

interface three extends one,two
{
    public void print_geek();
}
class child implements three
{
    @Override
    public void print_geek() {
        System.out.println("Geeks");
    }

    public void print_for()
    {
        System.out.println("for");
    }
}
```

```
// Drived class

public class Main

{

    public static void main(String[] args)

    {

        child c = new child();

        c.print_geek();

        c.print_for();

        c.print_geek();

    }
```

```
}
```

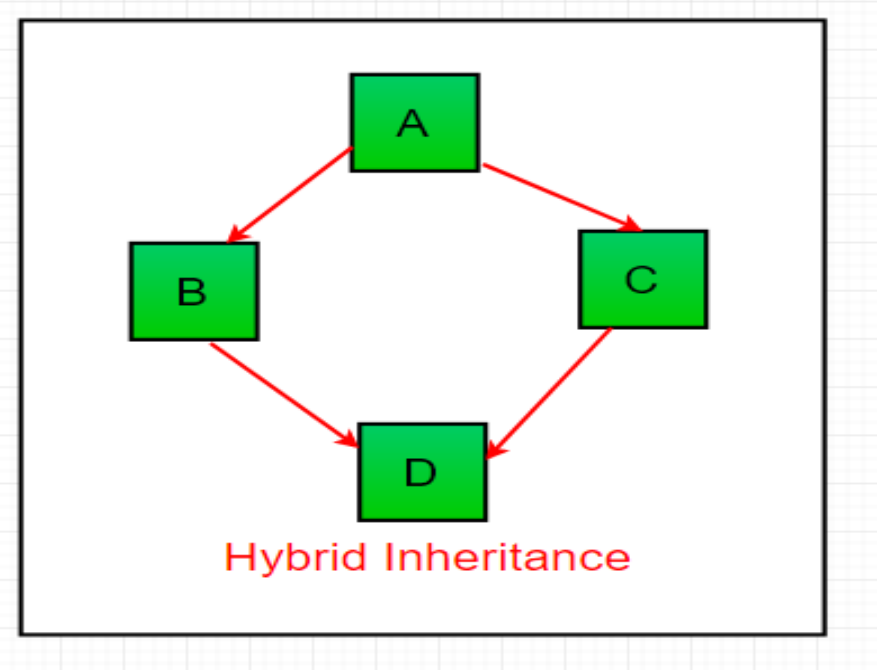
Output:

```
Geeks
```

```
for
```

```
Geeks
```

4. **Hybrid Inheritance(Through Interfaces) :** It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Important facts about inheritance in Java

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.

- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

What all can be done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in example above, *toString()* method is overridden).
- We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Use of This Keyword:

```
class Calc
{
    int num1;
    int num2;
    int result;
```

```
    public Calc(int number1, int number2)
    {
        num1=number1;
        num2=number2;
    }

}

public class ThisDemo{

     public static void main(String []args){

          Calc c=new Calc(4,5);
          System.out.println(c.num1);
           System.out.println(c.num2);
     }
}
```

Output:

4

5

Next Scenario:

```
class Calc
{
    int num1;
    int num2;
    int result;

    public Calc(int num1, int num2)
    {
        num1=num1;
        num2=num2;
    }

}
```

```
public class ThisDemo{

    public static void main(String []args){

        Calc c=new Calc(4,5);
        System.out.println(c.num1);
         System.out.println(c.num2);
    }
}
```

Output:
0
0

Use of this:

```
class Calc
{
   int num1;
   int num2;
   int result;

   public Calc(int num1, int num2)
   {
      this.num1=num1; //current obj, representing instance var
      this.num2=num2; //Current Obj
   }

}

public class ThisDemo{

    public static void main(String []args){

        Calc c=new Calc(4,5);
        System.out.println(c.num1);
         System.out.println(c.num2);
```

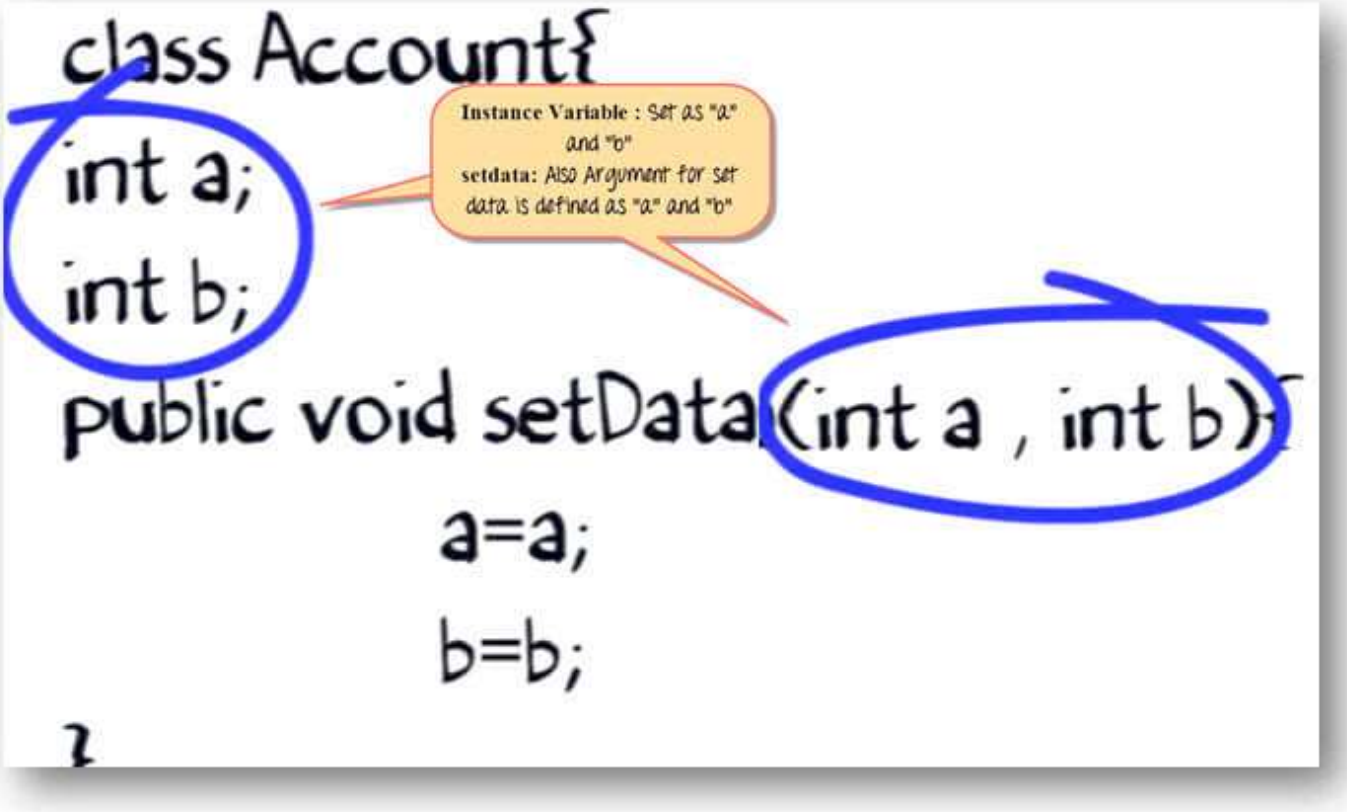
```
    }
}
```

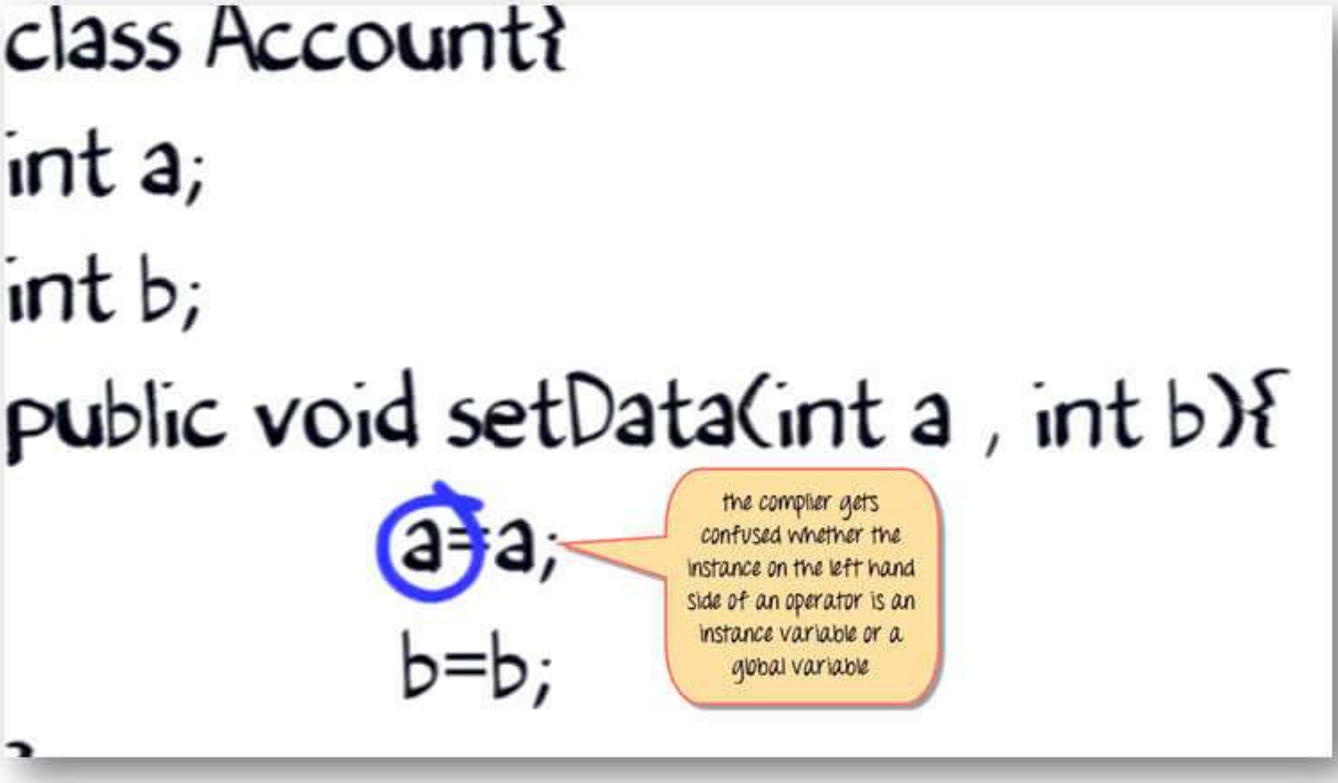
Output:

```
4
5
```

The `this` keyword refers to the current object in a method or constructor.

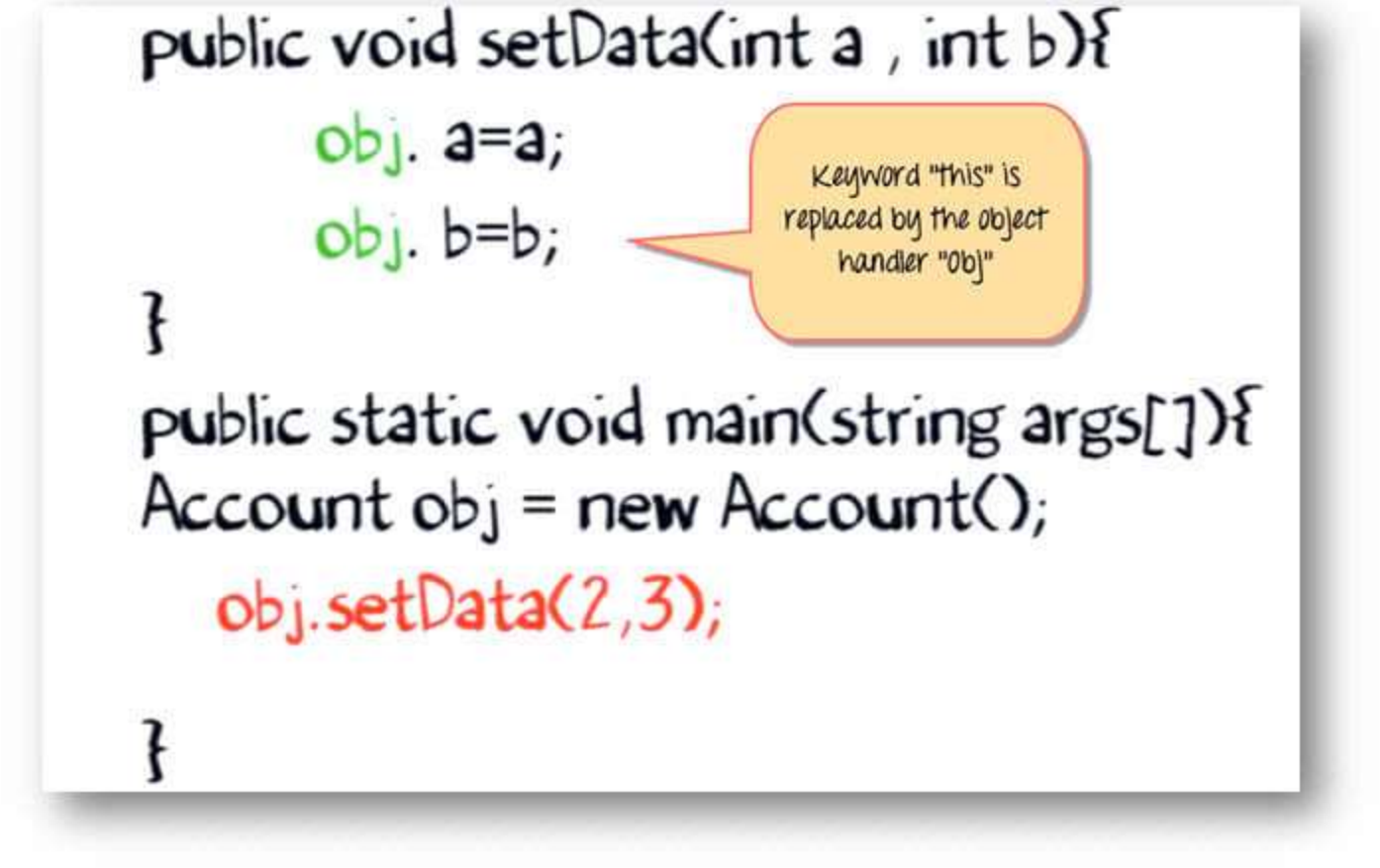
The most common use of `this` keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter). If you omit the keyword in the example above, the output would be "0" instead of "5".




class Account{
int a;
int b;
public void setData(int a , int b){
a=a;
b=b;
the complier gets confused whether the instance on the left hand side of an operator is an instance variable or a global variable

```
class Account{
int a;
int b;
public void setData(int a , int b){
    this. a=a;
    this. b=b;
}
public static void main(string args[]){
Account obj = new Account();


}
```

use keyword "This" to differentiate instance variable from local variable



So now the compiler knows,

- The 'a' on the left-hand side is an Instance variable.
- Whereas the 'a' on right-hand side is a local variable

`this` can also be used to:

- Invoke current class constructor
- Invoke current class method
- Return the current class object
- Pass an argument in the method call
- Pass an argument in the constructor call

Super Keyword in JAVA

The `super` keyword refers to superclass (parent) objects.

It is used to call superclass methods, and to access the superclass constructor.

The most common use of the `super` keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.

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.

Ex-1

```
//WAP to demonstrate Super in inheritance
class Senior
{
   int salary=4000;
}

 class Junior extends Senior
{
   int salary=3000;
      void display()
      {
       System.out.println(salary);
       System.out.println(super.salary);//refering to senior's salary
      }

}

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

```
        Junior j=new Junior();
        j.display();
        System.out.println(j.salary);


    }

}
```

Ex-2

```
//super can be used to invoke immediate parent class method
class Animal{
void eat()
{
    System.out.println("Animal is eating...");}
}
class Dog extends Animal{
void eat()
{
    System.out.println("Dog is eating bread...");

}
void bark()
{
    System.out.println("Dog is barking...");

}
void work()
{
super.eat();
bark();
}
}
public class TestSuper2
{
public static void main(String args[]){
Dog d=new Dog();
```

```
d.work();
}}
```

3. Super in Constructor

Case1:

```
class A
{
    public A()
    {
        System.out.println("in A");
    }
}

class B extends A
{
    public B()
    {
        System.out.println("in B");
    }
}

public class SuperDemo
{
    public static void main(String[] args)
    {
        A obj=new A(); //Object of Parent Constructor
    }
}
```

Output:
In A

Case 2:

```
class A
```

```
{
    public A()
    {
        System.out.println("in A");
    }
}

class B extends A
{
    public B()
    {
        System.out.println("in B");
    }
}

public class SuperDemo
{
    public static void main(String[] args)
    {
        B obj=new B(); //Object of Child Constructor
    }
}
```

Output:

```
in A
 in B
```

Case 3: Parametrized constructor

```
class A
{
    public A()
    {
        System.out.println("in A");
    }
    public A(int i)
    {
```

```
        System.out.println("in A int");
    }
}

class B extends A
{
    //super();
    public B()
    {
        System.out.println("in B");
    }
    public B(int i)
    {
        System.out.println("in B int");
    }
}

public class SuperDemo
{
    public static void main(String[] args)
    {
        B obj=new B(5); //Object of Child Constructor
    }
}
```

Output:

```
in A
 in Bint
```

Case 4:

```
class A
{
    public A()
    {
        System.out.println("in A");
    }
    public A(int i)
    {
```

```
        System.out.println("in A int");
    }
}

class B extends A
{

    public B()
    {
        System.out.println("in B");
    }
    public B(int i)
    {
        super();
        System.out.println("in B int");
    }
}

public class SuperDemo
{
    public static void main(String[] args)
    {
        B obj=new B(5); //Object of Child Constructor
    }
}
```

Output:

```
in A
 in B int
```

Case 5:

```
class A
{
    public A()
    {
        System.out.println("in A");
```

```
    }
    public A(int i)
    {
        System.out.println("in A int");
    }
}

class B extends A
{

    public B()
    {
        System.out.println("in B");
    }
    public B(int i)
    {
        super(i);
        System.out.println("in B int");
    }
}

public class SuperDemo
{
    public static void main(String[] args)
    {
        B obj=new B(5); //Object of Child Constructor
    }
}
```

Output:

```
in A int
 in B int
```

Case 6:

```
class A
{
    public A()
```

```
    {
        System.out.println("in A");
    }
    public A(int i)
    {
        System.out.println("in A int");
    }
}

class B extends A
{

    public B()
    {
        //super();
        System.out.println("in B");
    }
    public B(int i)
    {
        super(i);
        System.out.println("in B int");
    }
}

public class SuperDemo
{
    public static void main(String[] args)
    {
        B obj;
        new B();
        new B(5); //Object of Child Constructor
    }
}
```

Output:

```
in A
in B
in A int
 in B int
```

Java 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:

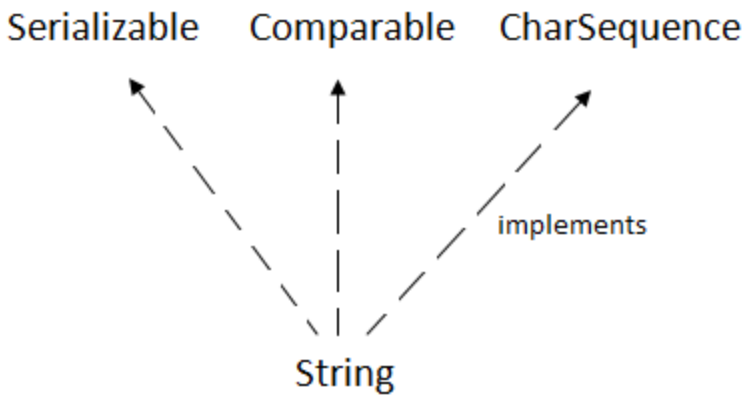
```
1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);
```

is same as:

```
1. String s="java";
```

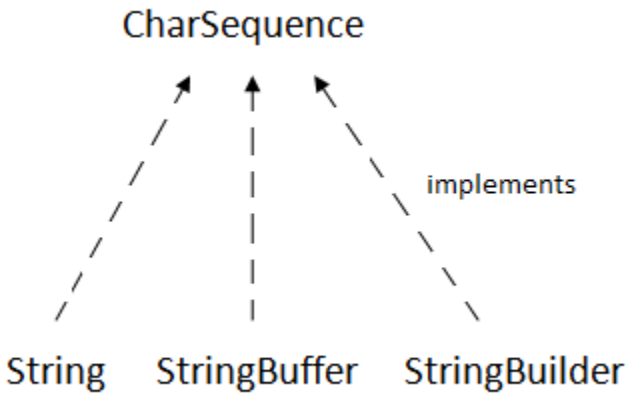
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

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 a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

---

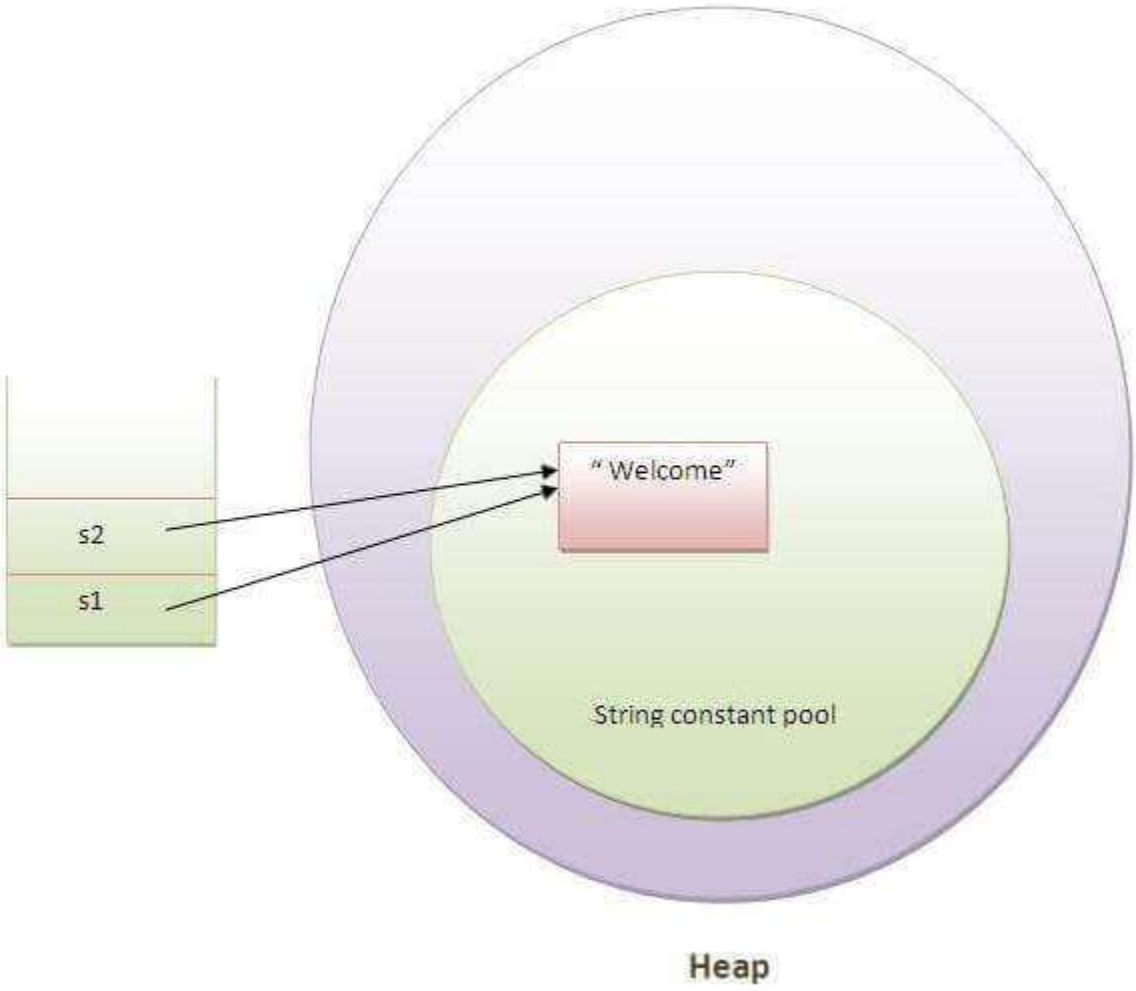
1) String Literal

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

1. 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 the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a 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, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

> Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

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

---

2) By new keyword

```
1. 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 a heap (non-pool).

---

public class StringConcept{

public static void main(String []args){

String str="Hello";

```
        String str1="Java";

        //equals() returns false if both string objects refers to
different memory

        //and true if both string objects refers to same memory

        System.out.println(str.equals(str1));

        String str2="HelloWorld";

        String str3="HelloWorld";

        System.out.println(str2.equals(str3));


    }

}
```

Java String Example

```
1. public class StringExample{
2. public static void main(String args[]){
3. String s1="java";//creating string by java string literal
4. char ch[]={'s','t','r','i','n','g','s'};
5. String s2=new String(ch);//converting char array to string
6. String s3=new String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}
```

Test it Now

java

strings

example

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.
5	String substring(int beginIndex)	returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.

8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string.
10	boolean equals(Object another)	checks the equality of string with the given object.
11	boolean isEmpty()	checks if string is empty.
12	String concat(String str)	concatenates the specified string.
13	String replace(char old, char new)	replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns a split string matching regex.
17	String[] split(String regex, int limit)	returns a split string matching regex and limit.
18	String intern()	returns an interned string.
19	int indexOf(int ch)	returns the specified char value index.

20	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
21	int indexOf(String substring)	returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
23	String toLowerCase()	returns a string in lowercase.
24	String toLowerCase(Locale l)	returns a string in lowercase using specified locale.
25	String toUpperCase()	returns a string in uppercase.
26	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is an overloaded method.

Java StringBuffer class

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.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important Constructors of StringBuffer class

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.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	is used to return the character at the specified position.
public int	length()	is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

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.

```
1. class StringBufferExample{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }
```

Ex-2

```
public class StringMutable
{
   public static void main(String[] args)
   {
      StringBuffer sbf= new StringBuffer("Hello");
      sbf.append("Java");
      System.out.println(sbf);
      sbf.replace(0,4,"Hi");
      System.out.println(sbf);

   }
}
public class StringDemo
{
```

```
    public static void main(String args[])
    {
       String s="Hello";
       System.out.println(s);
       s.concat("world");
       System.out.println(s);
       String s1=s.concat("Hi");
       System.out.println(s1);



    }
}
```

2) StringBuffer insert() method

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

```
1. class StringBufferExample2{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }
```

3) StringBuffer replace() method

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

```
1. class StringBufferExample3{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }
```

4) StringBuffer delete() method

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

```
1. class StringBufferExample4{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }
```

5) StringBuffer reverse() method

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

```
1. class StringBufferExample5{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }
```

6) StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
1. class StringBufferExample6{
```

```
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer();
4.  System.out.println(sb.capacity());//default 16
5.  sb.append("Hello");
6.  System.out.println(sb.capacity());//now 16
7.  sb.append("java is my favourite language");
8.  System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9.  }
10. }
```

Taking input in StringBuffer:

```
import java.util.*;

public class StringBufferScan {

        public static void main(String args[])
   {
      StringBuffer s1 = new StringBuffer();
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter a String: ");
      s1.append(scan.nextLine());
      System.out.println("String is: "+s1); //Hello
      System.out.println("Revrse String is: "+s1.reverse()); //olleH
   }
}
```

Output:

Enter a String:

HelloWorld

Revrse String is: HelloWorld

Revrse String is: dlroWolleH

Memory :

```
public class StringConcept{



    public static void main(String []args){


      // String str="Hello";
      // String str1="Java";
      //equals() returns false if both string objects refers to different memory
      //and true if both string objects refers to same memory
      // System.out.println(str.equals(str1));
      // String str2="HelloWorld";
      // System.out.println(Integer.toHexString(str2.hashCode()));
      // String str3="HelloWorld";
      //  System.out.println(Integer.toHexString(str3.hashCode()));
      // str3=str3+"1";
      // System.out.println(Integer.toHexString(str3.hashCode()));
      //  //System.out.println(str);//A
       //System.out.println(str3); //B
      //System.out.println(str3.equals(str4)); //C
      //System.out.println(str2.equals(str3)); //D
      StringBuffer sb=new StringBuffer("Hello");
       StringBuffer sb1=new StringBuffer("Hello");
```

```
        System.out.println(Integer.toHexString(sb1.hashCode()));

        System.out.println(sb);

       System.out.println(Integer.toHexString(sb.hashCode()));

       sb.append("Java");//now original string is changed

       System.out.println(sb);

       System.out.println(Integer.toHexString(sb.hashCode()));




    }

}
```

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.

3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Java StringBuilder class

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. (Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.Java Synchronization is better option where we want to allow only one thread to access the shared resource). It is available since JDK 1.5.

Important Constructors of StringBuilder class

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.

No.	StringBuffer	StringBuilder
1)	**StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.**	**StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.**
2)	**StringBuffer is *less efficient* than StringBuilder.**	**StringBuilder is *more efficient* than StringBuffer.**

String vs StringBuffer vs StringBuilder

1. String is immutable whereas StringBuffer and StringBuilder are mutable classes.
2. StringBuffer is thread-safe and synchronized whereas StringBuilder is not. That's why StringBuilder is faster than StringBuffer.
3. String concatenation operator (+) internally uses StringBuffer or StringBuilder class.
4. For String manipulations in a non-multi threaded environment, we should use StringBuilder else use StringBuffer class.

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.

What is Exception in Java

Dictionary Meaning: Exception is an abnormal condition.

In Java, an 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 ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

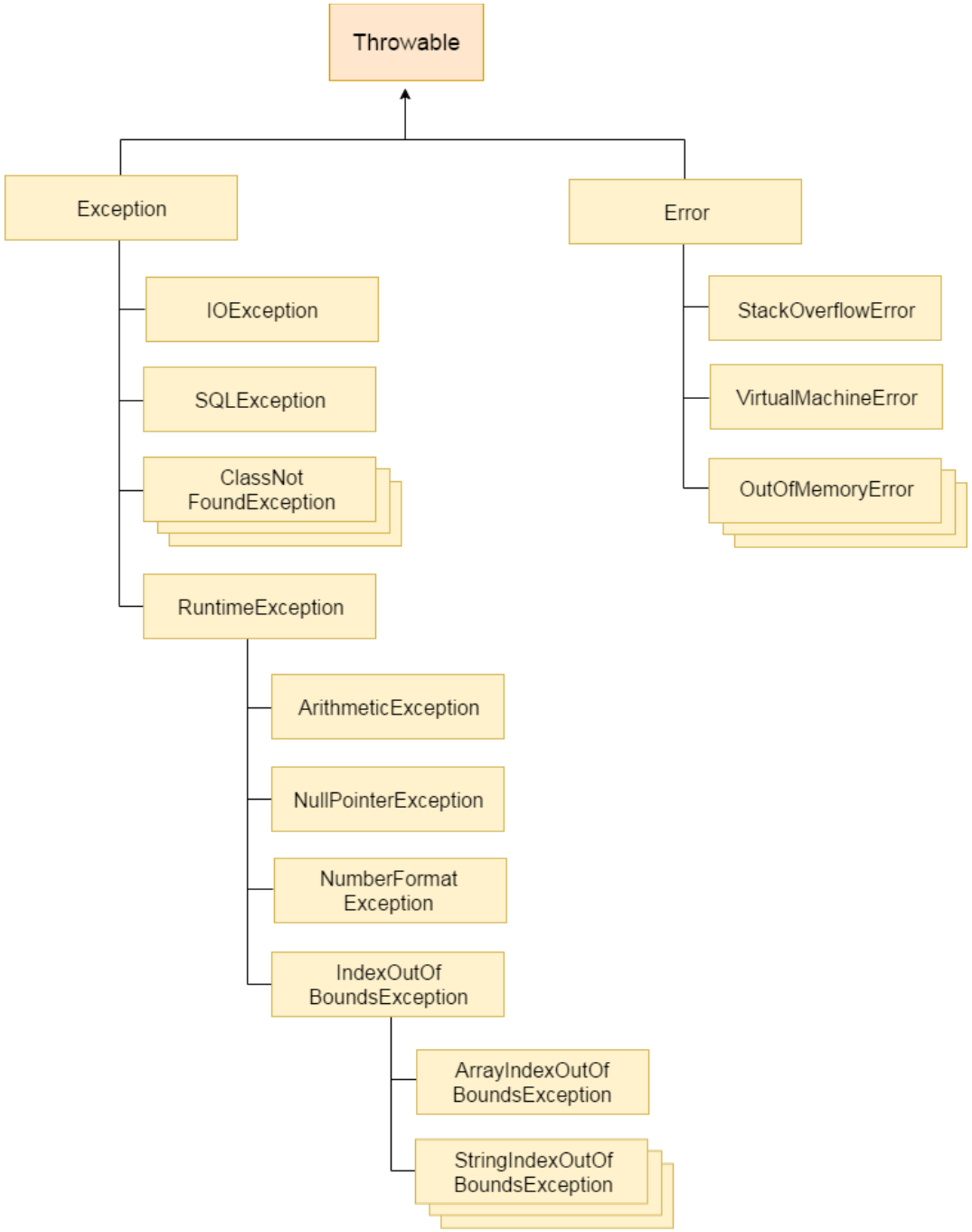
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: **Exception and Error.** A hierarchy of Java Exception classes are given below:



Comparison Chart

BASIS FOR COMPARISON	ERROR	EXCEPTION

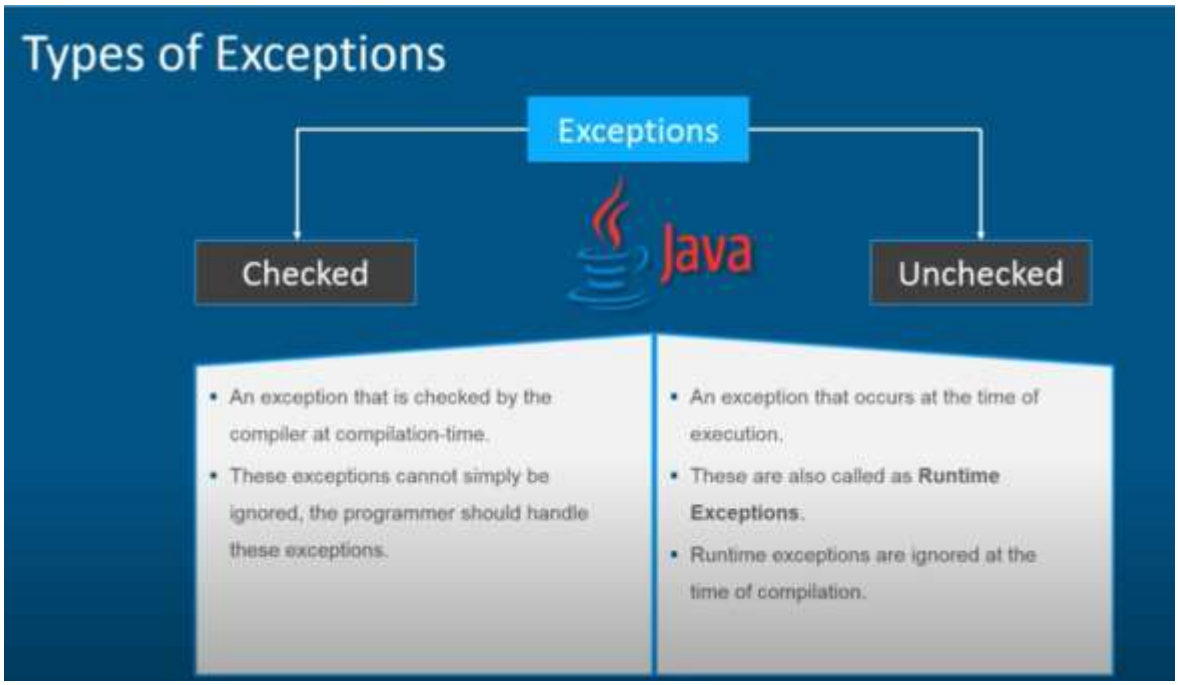
Basic	An error is caused due to lack of system resources	An exception is caused because of the code.
Recovery	An error is irrecoverable.	An exception is recoverable.
Keywords	There is no means to handle an error by the program code.	Exceptions are handled using three keywords "try", "catch", and "throw".
Consequences	As the error is detected the program will terminated abnormally.	As an exception is detected, it is thrown and caught by the "throw" and "catch" keywords correspondingly.
Types	Errors are classified as unchecked type.	Exceptions are classified as checked or unchecked type.
Package	In Java, errors are defined "java.lang.Error" package.	In Java, an exceptions are defined in"java.lang.Exception".

Example	OutOfMemory, StackOverFlow.	Checked Exceptions : NoSuchMethod, ClassNotFound. Unchecked Exceptions : NullPointer, IndexOutOfBounds.

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception (Or Compile Time Exception)
2. Unchecked Exception (Or Runtime Exception)
3. Error



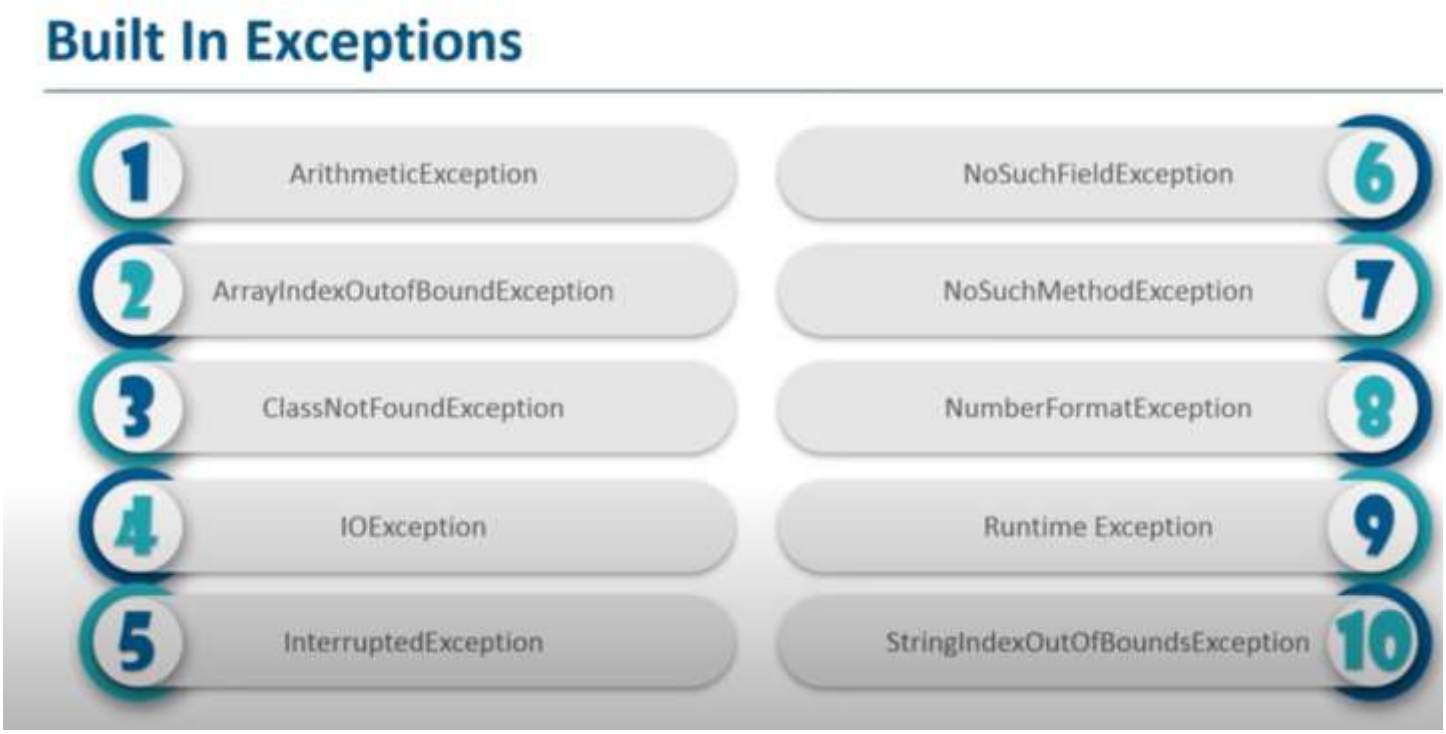
Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit **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 which inherit **RuntimeException** are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at **runtime.**



3) Error

An Error “indicates serious problems that a reasonable application should not try to catch.” Errors are the conditions which cannot get recovered by any handling techniques. It surely cause termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are **Out of memory error or a System crash error.** Error is irrecoverable e.g. **OutOfMemoryError, VirtualMachineError, AssertionError** etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. **It means, we can't use try block alone.**
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't **use catch block alone.** It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is **always executed whether an exception is handled or not.**
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
1. public class JavaExceptionExample{
2.   public static void main(String args[]){
3.    try{
```

```
4.       //code that may raise exception
5.       int data=100/0;
6.     }catch(ArithmeticException e)
7.  {System.out.println(e);}
8.     //rest code of the program
9.     System.out.println("rest of the code...");
10.  }
11. }
```

Output:

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

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

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

1) A scenario where ArithmeticException occurs

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

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

---

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

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

---

3) A scenario where NumberFormatException occurs

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

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

---

4) A scenario where ArrayIndexOutOfBoundsException occurs

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

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exceptions Index

Java try-catch block

Java try block

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

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

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

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

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

Example 1

```
public class TryCatchExample1 {

    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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

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

Example 2

```
1.  public class TryCatchExample2 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.          }
8.              //handling the exception
9.          catch(ArithmeticException e)
10.         {
11.             System.out.println(e);
12.         }
13.         System.out.println("rest of the code");
14.     }
15.
16. }
```

Output:

java.lang.ArithmeticException: / by zero

rest of the code

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

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

```
1.  public class TryCatchExample3 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.                          // if exception occurs, the remaining statement will not exceute
8.          System.out.println("rest of the code");
9.          }
10.             // handling the exception
11.         catch(ArithmeticException e)
12.         {
13.             System.out.println(e);
14.         }
15.
16.     }
17.
18. }
```

Output:

java.lang.ArithmeticException: / by zero

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

Here, we handle the exception using the parent class exception.

```
1.  public class TryCatchExample4 {
```

```
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.         int data=50/0; //may throw exception
7.         }
8.             // handling the exception by using Exception class
9.         catch(Exception e)
10.        {
11.            System.out.println(e);
12.        }
13.        System.out.println("rest of the code");
14.    }
15.
16. }
```

Output:

java.lang.ArithmeticException: / by zero

rest of the code

Example 5

Let's see an example to print a custom message on exception.

```
1.  public class TryCatchExample5 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.          }
8.              // handling the exception
```

```
9.          catch(Exception e)
10.         {
11.                 // displaying the custom message
12.             System.out.println("Can't divided by zero");
13.         }
14.     }
15.
16. }
```

Output:

Can't divided by zero

Example 6

Let's see an example to resolve the exception in a catch block.

```
1.  public class TryCatchExample6 {
2.
3.      public static void main(String[] args) {
4.          int i=50;
5.          int j=0;
6.          int data;
7.          try
8.          {
9.          data=i/j; //may throw exception
10.         }
11.             // handling the exception
12.         catch(Exception e)
13.         {
14.              // resolving the exception in catch block
15.             System.out.println(i/(j+2));
16.         }
17.     }
18. }
```

Output:

Example 7

In this example, along with try block, we also enclose exception code in a catch block.

```
1.  public class TryCatchExample7 {
2.
3.      public static void main(String[] args) {
4.
5.          try
6.          {
7.          int data1=50/0; //may throw exception
8.
9.          }
10.             // handling the exception
11.         catch(Exception e)
12.         {
13.            // generating the exception in catch block
14.         int data2=50/0; //may throw exception
15.
16.         }
17.     System.out.println("rest of the code");
18.     }
19. }
```

Output:

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

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

```
1.  public class TryCatchExample8 {
2.
```

```
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int data=50/0; //may throw exception
7.
8.          }
9.              // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
10.         catch(ArrayIndexOutOfBoundsException e)
11.         {
12.             System.out.println(e);
13.         }
14.         System.out.println("rest of the code");
15.     }
16.
17. }
```

Output:

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

Example 9

Let's see an example to handle another unchecked exception.

```
1.  public class TryCatchExample9 {
2.
3.      public static void main(String[] args) {
4.          try
5.          {
6.          int arr[]= {1,3,5,7};
7.          System.out.println(arr[10]); //may throw exception
8.          }
9.              // handling the array exception
10.         catch(ArrayIndexOutOfBoundsException e)
11.         {
12.             System.out.println(e);
13.         }
```

```
14.         System.out.println("rest of the code");
15.     }
16.
17. }
```

Output:

java.lang.ArrayIndexOutOfBoundsException: 10

rest of t/he code

Example 10

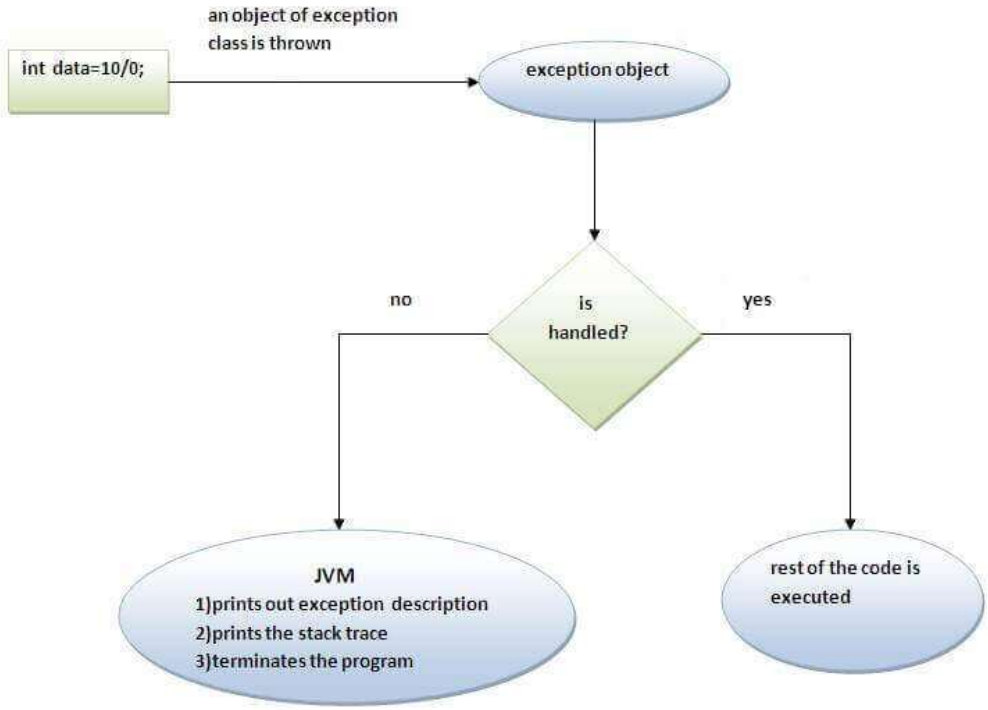
Let's see an example to handle checked exception.

```
1.  import java.io.FileNotFoundException;
2.  import java.io.PrintWriter;
3.
4.  public class TryCatchExample10 {
5.
6.      public static void main(String[] args) {
7.
8.
9.          PrintWriter pw;
10.         try {
11.             pw = new PrintWriter("jtp.txt"); //may throw exception
12.             pw.println("saved");
13.         }
14. // providing the checked exception handler
15.  catch (FileNotFoundException e) {
16.
17.             System.out.println(e);
18.         }
19.     System.out.println("File saved successfully");
20.     }
21. }
```

Output:

File saved successfully

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Difference between throw and throws in Java

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 throw example

```
1. void m(){
2. throw new ArithmeticException("sorry");
3. }
```

Java throws example

```
1. void m()throws ArithmeticException{
2. //method code
3. }
```

```
public class ThrowExample{
  int a=12;
  int b=5;
  void divide()
  {
     if(b==5)
     {
        throw new ArithmeticException();
     }

     else
     {
        int c;
        c=a/b;
        System.out.println("Result is " +c);
     }
  }

    public static void main(String[] args) {
       ThrowExample t=new ThrowExample();
       t.divide();
    }
     }
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException
        at ThrowExample.divide(ThrowExample.java:8)
        at ThrowExample.main(ThrowExample.java:21)
```

Case 2:

```
public class ThrowExample{
  int a=12;
  int b=5;
  void divide()
  {
      if(b==5)
      {
          throw new ArithmeticException("Cannot divide by 5");
      }

      else
      {
          int c;
          c=a/b;
          System.out.println("Result is " +c);
      }
  }

    public static void main(String[] args) {
        ThrowExample t=new ThrowExample();
        try
        {
          t.divide();
        }
        catch(Exception e)
        {
            System.out.println("Arithmetic Exception Caught");
        }
    }

}
```

Output:

Arithmetic Exception Caught

Concept of throws:

In case of checked Exceptions

```
import java.io.*;
public class Example
{
   public static void main(String [] args)
   {
      throw new IOException();
      System.out.println("Last statement");
   }
}
```

Output:

```
Example.java:7: error: unreachable statement
      System.out.println("Last statement");
      ^
Example.java:6: error: unreported exception IOException; must be caught or declared to be thrown
      throw new IOException();
      ^
2 errors
```

Solution:

1. **Either by try catch:**

```
import java.io.*;
public class Example
{
   public static void main(String [] args)
   {
      try{
       throw new IOException();
      }

      catch(Exception e)
      {
         System.out.println("Caught");
      }
      System.out.println("Last statement");
   }
}
```

2. **Or by Throws**

```
import java.io.*;
public class Example
{
   public static void main(String [] args) throws IOException
   {

       throw new IOException();
      //System.out.println("Last statement");
   }
}
```

Compile:
No error
Output:

```
Exception in thread "main" java.io.IOException
at Example.main(Example.java:7)
```

In case of Unchecked Exceptions:

```
import java.io.*;
public class Example
{
   public static void main(String [] args)
   {
      throw new ArithmeticException();
      //System.out.println("Last statement");
   }
}
```

Compile:
No Error

Output:

```
Exception in thread "main" java.lang.ArithmeticException
        at Example.main(Example.java:6)
```

Java throw and throws example

```
1. void m()throws ArithmeticException{
2. throw new ArithmeticException("sorry");
```

Throw keyword:

```
public class JavaTester{
   public void checkAge(int age){
      if(age<18)
         throw new ArithmeticException("Not Eligible for voting");
      else
         System.out.println("Eligible for voting");
   }
   public static void main(String args[]){
      JavaTester obj = new JavaTester();
      obj.checkAge(13);
      System.out.println("End Of Program");
   }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException:
Not Eligible for voting
at JavaTester.checkAge(JavaTester.java:4)
at JavaTester.main(JavaTester.java:10)
```

Throws:

```
public class JavaTester{
   public int division(int a, int b) throws ArithmeticException{
      int t = a/b;
      return t;
```

```
    }
    public static void main(String args[]){
        JavaTester obj = new JavaTester();
        try{
            System.out.println(obj.division(15,0));
        }
        catch(ArithmeticException e){
            System.out.println("You shouldn't divide number by zero");
        }
    }
}
```

```
Output:
You shouldn't divide number by zero
```

StringTokenizer in Java

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

Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

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.**

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

   }
}
```

OUtput:

```
 5
 This
 4
 is
```

3

StringTokenizer

2

-

1

-

Java - Applet Basics

An applet is a special kind of Java program that runs in a Java enabled browser. This is the first Java program that can run over the network using the browser. Applet is typically embedded inside a web page and runs in the browser.

In other words, we can say that Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as apart of a web document.

After a user receives an applet, the applet can produce a graphical user interface. It has limited access to resources so that it can run complex computations without introducing the risk of viruses or breaching data integrity.

To create an applet, a class must class extends java.applet.Applet class.

An Applet class does not have any main() method. It is viewed using JVM. The JVM can use either a plug-in of the Web browser or a separate runtime environment to run an applet application.

JVM creates an instance of the applet class and invokes init() method to initialize an Applet.

Note: Java Applet is deprecated since Java 9. It means Applet API is no longer considered important.

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- **An applet is a Java class that extends the java.applet.Applet class.**
- **A main() method is not invoked on an applet, and an applet class will not define main().**
- **Applets are designed to be embedded within an HTML page.**
- **When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.**
- **A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.**
- **The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.**
- **Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.**
- **Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.**

Life Cycle of an Applet

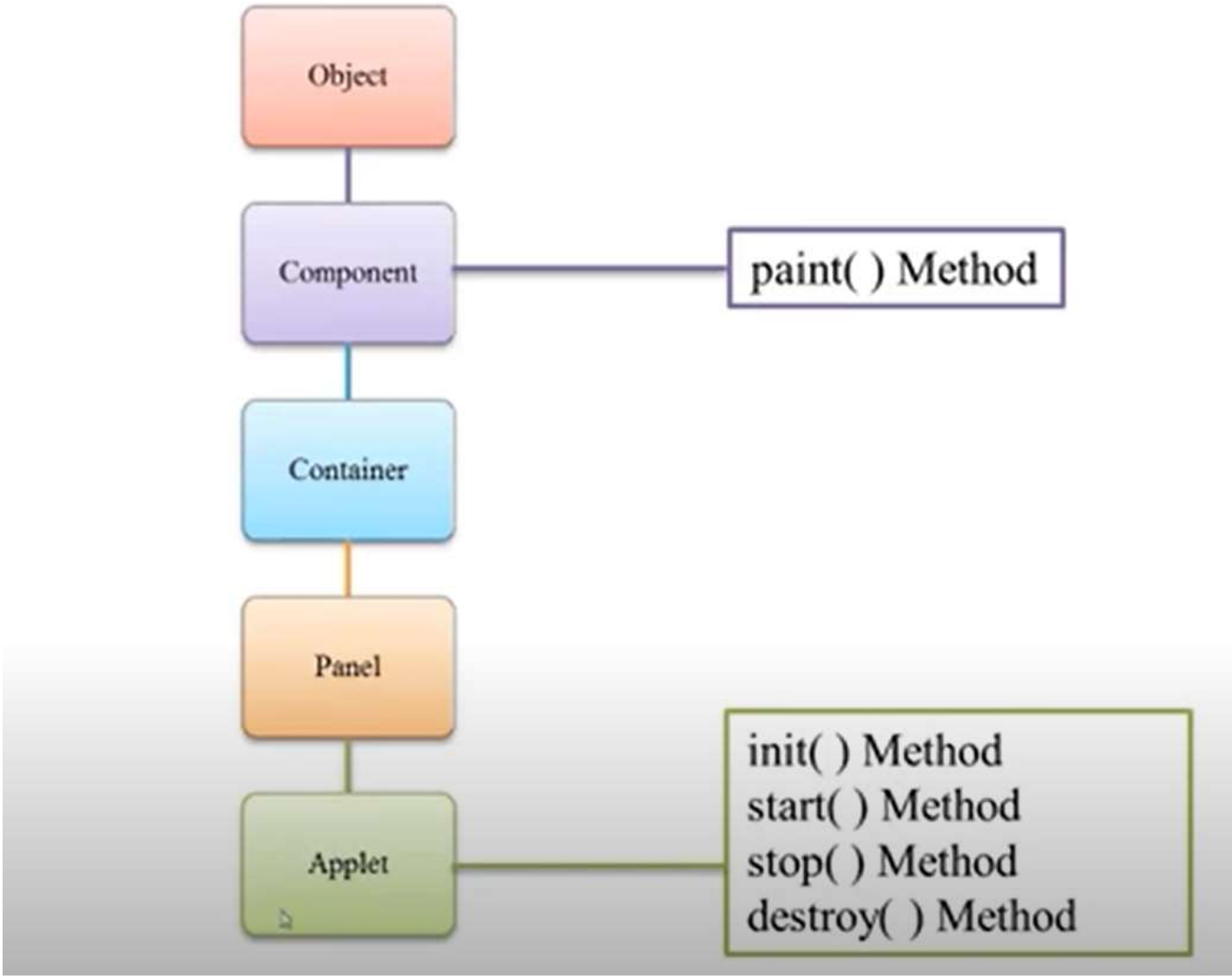
Four methods in the Applet class gives you the framework on which you build any serious applet −

- init − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- start − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- stop − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- destroy − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- paint − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

Lifecycle of Java Applet

Following are the stages in Applet

1. **Applet is initialized.**
2. **Applet is started**
3. **Applet is painted.**
4. **Applet is stopped.**
5. **Applet is destroyed.**



Every Applet application must import two packages - java.awt and java.applet.

java.awt.* imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT. The AWT contains support for a window-based, graphical user interface. java.applet.* imports the applet package, which contains the class Applet. Every applet that you create must be a subclass of Applet class.

The class in the program must be declared as public, because it will be accessed by code that is outside the program.Every Applet application must declare a paint() method. This method is defined by AWT class and must be overridden by the applet. The paint() method is called each time when an applet needs to redisplay its output. Another important thing to notice about applet application is that, execution of an applet does not begin at main() method. In fact an applet application does not have any main() method.

Advantages of Applets

1. **It takes very less response time as it works on the client side.**
2. **It can be run on any browser which has JVM running in it.**

//java.applet.Applet

//java.awt.Graphics

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
   public void paint (Graphics g) {
      g.drawString ("Hello World", 25, 50);//(“String”,int x, int y)
   }
}
```

These import statements bring the classes into the scope of our applet class −

- **java.applet.Applet**
- **java.awt.Graphics**

Applet class

Applet class provides all necessary support for applet execution, such as initializing and destroying of applet. It also provide methods that load and display images and methods that load and play audio clips.

An Applet Skeleton

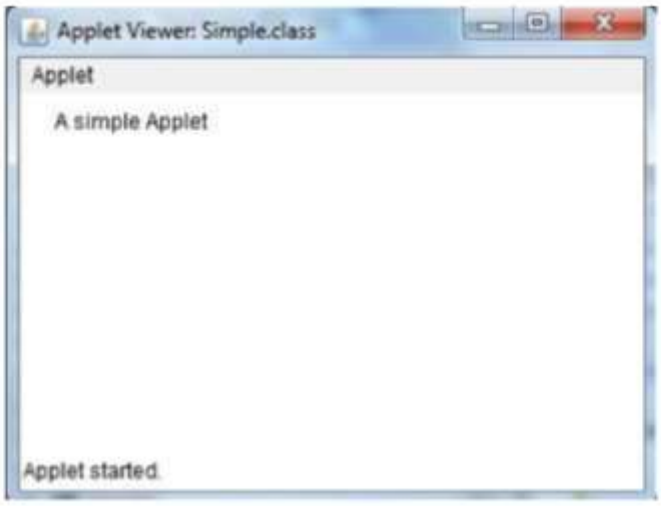
Most applets override these four methods. These four methods forms Applet lifecycle.

- **init() : init() is the first method to be called. This is where variable are initialized. This method is called only once during the runtime of applet.**
- **start() : start() method is called after init(). This method is called to restart an applet after it has been stopped.**
- **stop() : stop() method is called to suspend thread that does not need to run when applet is not visible.**
- **destroy() : destroy() method is called when your applet needs to be removed completely from memory.**

Note: The stop() method is always called before destroy() method.

Simple Applet

```
import java.awt.*;
import java.applet.*;
public class Simple extends Applet
{
public void paint(Graphics g)
 {
  g.drawString("A simple Applet", 20, 20);
 }
}
```



How to run an Applet Program

In the same manner as you compiled your console programs, an Applet program is compiled. There are, however, two methods of running an applet.

- Running the Applet in a web browser compatible with Java.
- Use an applet viewer, like the normal instrument, to view applets. In a window, an applet viewer runs your applet.

Create brief HTML file in the same folder to execute an Applet in a web browser. Include the following code in the file's body tag. (Applet tag loads class Applet).

```
< applet code = "MyApplet" width=400 height=400 >
< /applet >
```

Run the HTML file

Running Applet using Applet Viewer

Write a brief HTML file as mentioned above to run an Applet with an applet viewer. If you name it as run.htm, your applet program will operate the following command.

```
f:/>appletviewer run.htm
```