SYNERGY INSTITUTE OF ENGINEERING & TECHNOLOGY

Department of Computer Science & Engineering

Academic Session 2023-24

LECTURE NOTE

Name of Faculty : Mr. Pratyusabhanu Khuntia
Name of Subject : Object Oriented Programming using Java
Subject Code : ROP3B001
Subject Credit : 3
Semester : III
Year : 2nd
Course : B.Tech
Branch : All
Admission Batch : 2020-24

Introduction to Java

HISTORY OF JAVA

In 1991, a group of Sun Microsystems engineers led by James Gosling decided to develop a language for consumer devices (cable boxes, etc.). Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

They wanted the language to be small and use efficient code since these devices do not have powerful CPUs. They also wanted the language to be hardware independent since different manufacturers would use different CPUs. The project was code-named **Green.**

Firstly, it was called "Greentalk" by James Gosling and file extension was .gt.
After that, it was called Oak and was developed as a part of the Green project.
However, they soon discovered that there was already a programming language called Oak, so they changed the name to Java

Sun Microsystems released the first public implementation as Java 1.0 in 1995.

Currently, Java is used in **internet programming, mobile devices, games, e-business** solutions etc. There are given the major points that describe the history of java.

EDITION OF JAVA

- Java Standard Edition **(J2SE)**
  - J2SE can be used to develop client-side standalone applications or applets.
- Java Enterprise Edition **(J2EE)**
  - J2EE can be used to develop server-side applications such as Java servlets and Java Server Pages.
- Java Micro Edition **(J2ME)**.
  - J2ME can be used to develop applications for mobile devices such as cell phones.

RELEASE VERSION OF JAVA WITH DATE

- JDK 1.0 (January 21, 1996)
- JDK 1.1 (February 19, 1997)
- J2SE 1.2 (December 8, 1998)
- J2SE 1.3 (May 8, 2000)
- J2SE 1.4 (February 6, 2002)
- J2SE 5.0 (September 30, 2004)

- Java SE 6 (December 11, 2006)
- Java SE 7 (July 28, 2011)
- Java SE 8 (March 18, 2014)

Java programming Environment

The **Java Development Kit (JDK)** is an Oracle Corporation product aimed at Java developers. Since the introduction of Java, it has been by far the most widely used Java Software Development Kit (SDK).

JDK = JRE + JVM.

The JDK includes the Java Runtime Environment, the Java compiler and the Java APIs. It's easy for both new and experienced programmers to get started.

A **Java virtual machine (JVM)** is a virtual machine that can execute Java byte code. It is the code execution component of the Java software platform.

Java Runtime Environment, is also referred to as the Java Runtime, Runtime Environment Java Runtime Environment contains JVM, class libraries, and other supporting files. JRE is targeted for execution of Java files. Java Runtime Environment (JRE) The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language

SOFTWARE & TOOLS USED TO EXECUTE THE JAVA

You have to download the JDK from internet and install the software in your machine. For your practice jdk1.5 or higher version is advisable to install.

You can write the program any one of the text editor say example Notepad.

Now a day's many text editors are used like eclipse and net beans to run the program.

STEPS TO EXECUTE THE JAVA PROGRAM

1. Creating a Java source file
2. Compiling a Java source file: java Hello.java
3. Running a Java program: java Hello

BASIC CONCEPTS OF OOPS

The following are the basic concepts applied in object-oriented programming language.

1. **Object**
2. **class**
3. **Abstraction & Encapsulation**
4. **Inheritance**

5. **Polymorphism**
6. **Dynamic binding**
7. **Message passing**

1) Object :

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.

An Object is a collection of data members and associated member functions also known as methods.

For example whenever a class name is created according to the class an object should be created without creating object can't able to use class.

The class of Dog defines all possible dogs by listing the characteristics and behaviors they can have; the object Lassie is one particular dog, with particular versions of the characteristics. A Dog has fur; Lassie has brown-and-white fur.

2) Class:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and is referred to as Methods.

For example consider we have a Class of Cars under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form Properties of the Car class and the associated actions i.e., object functions like Start, Move, Stop form the Methods of Car Class.No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

3) Data abstraction & Encapsulation :

The wrapping up of data and its functions into a single unit is called Encapsulation.

When using **Data Encapsulation**, data is not accessed directly, it is only accessible through the functions present inside the class.

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Abstraction refers to the act of representing essential features without including the background details or explanation between them.

For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components

work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

4) Inheritance:
Inheritance is the process of forming a new class from an existing class or base class.

The base class is also known as parent class or super class, the new class that is formed is called derived class.

Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

It is classifieds into different types, they are

- **Single level inheritance**
- **Multi-level inheritance**
- **Hybrid inheritance**
- **Hierarchial inheritance**

5) Polymorphism:
Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Poly a Greek term ability to take more than one form. Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an exiting operator or function begins to operate on new data type, or class, it is understood to be overloaded.

6. *Dynamic binding:* Binding means linking. It involves linking of function definition to a function call.

1. If linking of function call to function definition, i.e., a place where control has to be transferred is done at compile time, it is known as static binding.

2. When linking is delayed till run time or done during the execution of the program then this type of linking is known as dynamic binding. Which function will be called in response to a function call is find out when program executes.

. *Message passing:* In C++, objects communicate each other by passing messages to each other. A message contains the name of the member function and arguments to pass. The message passing is shown below:

object. method (parameters);

Message passing here means object calling the method and passing parameters. Message passing is nothing but calling the method of the class and sending parameters. The method in turn executes in response to a message.

Features of java

Java has the following characteristics:

- **Object oriented** - Java provides the basic object technology of C++ with some enhancements and some deletions.
- **Architecture neutral/plat form independent** - Java source code is compiled into architecture-independent object code. The object code is interpreted by a Java Virtual Machine (JVM) on the target architecture.
- **Portable** - Java implements additional portability standards. For example, ints are always 32-bit, 2's-complemented integers. User interfaces are built through an abstract window system that is readily implemented in Solaris and other operating environments.
- **Distributed** - Java contains extensive TCP/IP networking facilities. Library routines support protocols such as HyperText Transfer Protocol (HTTP) and file transfer protocol (FTP).
- **Robust** - Both the Java compiler and the Java interpreter provide extensive error checking. Java manages all dynamic memory, checks array bounds, and other exceptions.
- **Secure** - Features of C and C++ that often result in illegal memory accesses are not in the Java language. The interpreter also applies several tests to the compiled code to check for illegal code. After these tests, the compiled code causes no operand stack over- or underflows, performs no illegal data conversions, performs only legal object field accesses, and all opcode parameter types are verified as legal.
- **High performance** - Compilation of programs to an architecture independent machine-like language, results in a small efficient interpreter of Java programs. The Java environment also compiles the Java bytecode into native machine code at runtime.
- **Multithreaded** - Multithreading is built into the Java language. It can improve interactive performance by allowing operations, such as loading an image, to be performed while continuing to process user actions.
- **Dynamic** - Java does not link invoked modules until runtime.
- **Simple** - Java is similar to C++, but with most of the more complex features of C and C++ removed

Data type

Java Primitive Types

Type	Size	Range	Default*
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

Declaring and Assigning Variables

The syntax for declaring a variable is:

Data Type variable Name [= expression];

Examples: float j; int i = 5 + 3;

Conversion between Types (**Typecasting)**

ype Casting

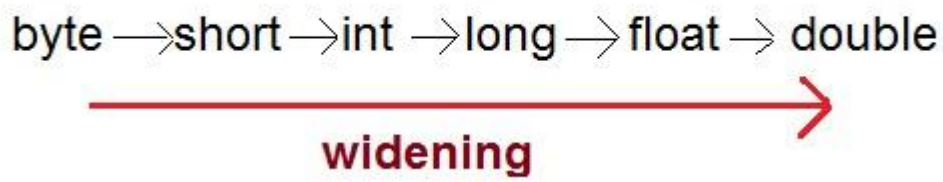
Assigning a value of one type to a variable of another type is known as **Type Casting**.

Example :

```
int x = 10;
byte y = (byte)x;
```

In Java, type casting is classified into two types,

- Widening Casting(Implicit)



- Narrowing Casting(Explicitly done)

Automatic Type casting take place when,

- the two types are compatible
- the target type is larger than the source type

Implicit up-casting: a lower-precision type is automatically converted to a higher precision type if needed:

```
int i = 'A';
```

Explicit down-casting: a manual conversion is required if there is a potential for a loss of precision, using the notation: (lowerPrecType) higherPrecValue

```
int i = (int) 123.456;
```

Declaring Arrays

- An *array* is a simple data structure to hold a series of data elements of the same type.
- Declare an array variable in one of two ways:
  - With [] after the variable type: int[] values;
  - With [] after the variable name: int values[];
- Arrays can be single- or multi-dimensional.
  - A two dimensional array could be declared with: double values[][];
- Array elements are integer indexed.
  - Use *arrayName*.length to get the array length.
  - Elements are indexed from 0 to *arrayName*.length - 1
  - Access individual elements with *arrayName*[*index*]

Operator

Arithmetic Operators

perator	Use	Description
+	x + y	Adds x and y
-	x - y	Subtracts y from x
	-x	Arithmetically negates x
*	x * y	Multiplies x by y
/	x / y	Divides x by y
%	x % y	Computes the remainder of dividing x by y

Shortcut Arithmetic Operators

	x++	y = x++; is the same as y = x; x = x + 1;
	++x	y = ++x; is the same as x = x + 1; y = x;
--	x--	y = x--; is the same as y = x; x = x - 1;
	--x	y = --x; is the same as x = x - 1; y = x;

Relational Operators

Operator	Use	Description
>	x > y	x is greater than y
>=	x >= y	x is greater than or equal to y
<	x < y	x is less than y
<=	x <= y	x is less than or equal to y
==	x == y	x is equal to y
!=	x != y	x is not equal to y

Logical Boolean Operators

Operator	Use	Evaluates to true if
&&	x && y	Both x and y are true
||	x || y	Either x or y are true
!	!x	x is not true

Bitwise Operators

Operator	Use	Evaluates to true if
&	x & y	AND all bits of x and y
|	x | y	OR all bits of x and y

Assignment Operators

=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y

Other Operators

new	d = new Date();	Create a new object
instanceof	o instanceof String	Check for object type, returning boolean
?:	z = b ? x : y	Equivalent to: if (b) { z = x; } else { z = y; }

Operator Precedence

1. [], (), .
2. ++, --, ~, !, (*type*), new, -
3. *, /, %

When evaluation is done it will be from left to write.

First Java Program | Hello World Example

In this page, we will learn how to write the simple program of java. We can write a simple hello java program easily after installing the JDK.

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

The requirement for Java Hello World Example

For executing any java program, you need to

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

Creating Hello World Example

Let's create the hello java program:

```
1.  class Simple{
2.     public static void main(String args[]){
3.      System.out.println("Hello Java");
4.     }
5.  }
```

save this file as Simple.java

To compile:	javac Simple.java
To execute:	java Simple

```
Output:Hello Java
```

Compilation Flow:

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

Parameters used in First Java Program

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

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

To write the simple program, you need to open notepad by **start menu -> All Programs -> Accessories -> notepad** and write a simple program as displayed below:

As displayed in the above diagram, write the simple program of java in notepad and saved it as Simple.java. To compile and run this program, you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt**.

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

To compile:	javac Simple.java
To execute:	java Simple

How many ways can we write a Java program

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

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

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

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

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

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

```
1.  public static void main(String[] args)
2.  public static void main(String []args)
3.  public static void main(String args[])
```

3) You can provide var-args support to the main method by passing 3 ellipses (dots)

Let's see the simple code of using var-args in the main method. We will learn about var-args later in Java New Features chapter.

```
1.  public static void main(String... args)
```

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

Let's see the simple code.

```
1.  class A{
2.  static public void main(String... args){
3.  System.out.println("hello java4");
4.  }
5.  };
```

Valid java main method signature

```
1.  public static void main(String[] args)
2.  public static void main(String []args)
3.  public static void main(String args[])
4.  public static void main(String... args)
5.  static public void main(String[] args)
6.  public static final void main(String[] args)
7.  final public static void main(String[] args)
```

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

Invalid java main method signature

1. **public void** main(String[] args)
2. **static void** main(String[] args)
3. **public void static** main(String[] args)
4. **abstract public static void** main(String[] args)

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

If there occurs a problem like displayed in the below figure, you need to set path. Since DOS doesn't know javac or java, we need to set path. The path is not required in such a case if you save your program inside the JDK/bin directory. However, it is an excellent approach to set the path.

Command line in java

A command-line argument is an information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main( ).

Example

The following program displays all of the command-line arguments that it is called with -

```
public class CommandLine {

   public static void main(String args[]) {

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

         System.out.println("args[" + i + "]: " + args[i]);

      }

   }

}
```

Try executing this program as shown here -

```
$java CommandLine this is a command line 200 -100
```

Output

This will produce the following result -

```
args[0]: this
```

```
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

Selection statements

The if Statement:

An if statement consists of a Boolean expression followed by one or more statements.

Syntax:

```
If (Boolean expression)
{
   //Statements will execute if the Boolean expression is true
}
```

Example:

```
public class Test {public static void main(String args[]){
      int x = 10;
      if( x < 20 ){ System.out.print ("This is if statement");}}}  o/p: This is if statement
```

The if...else Statement:

Syntax:

```
If (Boolean expression){
   //Executes when the Boolean expression is true
} else {
   //Executes when the Boolean expression is false
}
```

Example:

```
public class Test {public static void main(String args[]){int x = 30;
 if( x < 20 ){
        System.out.print ("This is if statement"); }else{System.out.print("This is else statement"); }}}  o/p
:This is else statement
```

The if...else if...else Statement:.Syntax:

The syntax of an if...else is:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
```

```
}else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
}else {
   //Executes when the none of the above condition is true.
}
```

Example:

```
public class Test {
   public static void main(String args[]){
      int x = 30;
      if( x == 10 ){System.out.print("Value of X is 10");
      }else if( x == 20 ){
         System.out.print("Value of X is 20");
      }else if( x == 30 ){
         System.out.print("Value of X is 30");
      }else{
         System.out.print("This is else statement");  } }}  o/p: Value of X is 30
```

Nested if...else Statement:Syntax:

The syntax for a nested if...else is as follows:

```
if(Boolean expression 1){
   //Executes when the Boolean expression 1 is true
   if(Boolean expression 2){
      //Executes when the Boolean expression 2 is true}  }
```

Example:

```
public class Test {
   public static void main(String args[]){
      int x = 30;
      int y = 10;
      if( x == 30 ){
         if( y == 10 ){
            System.out.print ("X = 30 and Y = 10");
          }  }}}
                                                    O/p: X = 30 and Y = 10
```

The switch Statement:A* switch *statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

```
Switch (expression){
    case value:
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
```

```
   //You can have any number of case statements.
   default : //Optional
      //Statements
}
```

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {
  public static void main(String args[]){
           char grade = 'C';

     switch(grade)
     {
        case 'A' :
           System.out.println("Excellent!");
           break;
        case 'B' :
        case 'C' :
           System.out.println("Well done");
           break;
        case 'D' :
           System.out.println("You passed");
        case 'F' :
           System.out.println("Better try again");
           break;
        default :
           System.out.println("Invalid grade");
     }
     System.out.println("Your grade is " + grade);
  }
}
Well done
Your grade is a C
```

Loop in java

There may be a situation when we need to execute a block of code several number of times, and is often referred to as a loop.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

As of Java 5, the *enhanced for loop* was introduced. This is mainly used for Arrays.

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

```
while(Boolean expression)
{     //Statements    }
```

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {

  public static void main(String args[]) {
    int x = 10;
    While(x < 20 ) {
      System.out.print ( x );
      x++;
      }}
```

This would produce the following result: 10 11….. 19

The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

```
do
{
   //Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:

```
public class Test {
   public static void main(String args[]){
      int x = 10;
      do{
         System.out.print( x );
         x++;
              }while( x < 20 ); }}
```

This would produce the following result: 10 11….. 19

The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

```
For (initialization; Boolean_expression; update)
{
   //Statements
}
```

Here is the flow of control in a for loop:

- The initialization step is executed first, and only once. Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test {
  public static void main(String args[]) {
    for(int x = 10; x < 20; x = x+1) {
      System.out.print( x );}}}
```

This would produce the following result: 10 11….. 19

Enhanced for loop in Java:

As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

```
For (declaration : expression)
{
  //Statements
}
```

- **Declaration:** The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression:** This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

```
dataType[] arrayRefVar;   // preferred way.
or
dataType arrayRefVar[];  // works but not preferred way.
```

Note − The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax −

```
double[] myList;   // preferred way.
or
double myList[];   // works but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax −

Syntax

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things −

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows −

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

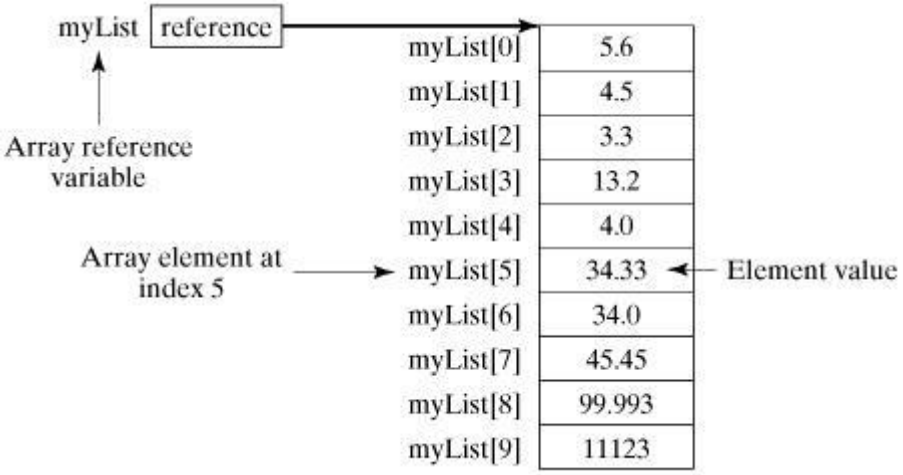
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList −

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays −

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
      for (int i = 0; i < myList.length; i++) {
         System.out.println(myList[i] + " ");
      }

      // Summing all elements
      double total = 0;
      for (int i = 0; i < myList.length; i++) {
         total += myList[i];
      }
      System.out.println("Total is " + total);

      // Finding the largest element
      double max = myList[0];
      for (int i = 1; i < myList.length; i++) {
         if (myList[i] > max) max = myList[i];
      }
      System.out.println("Max is " + max);
   }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Example

The following code displays all the elements in the array myList −

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
      for (double element: myList) {
         System.out.println(element);
      }
   }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
```

Example:

```
public class Test {

  public static void main(String args[]){
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ){
      System.out.print( x );
      System.out.print(",");
    }
    System.out.print("\n");
    String [] names ={"James", "Larry", "Tom", "Lacy"};
    for( String name : names ) {
```

```
        System.out.print( name );
        System.out.print(",");}}}
```

This would produce the following result:

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

The break Keyword:

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a break is a single statement inside any loop:

```
break;
```

Example:

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         if( x == 30 ) {
                  break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

This would produce the following result:

```
10
20
```

The continue Keyword:

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a continue is a single statement inside any loop:

continue;

Example:

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         if( x == 30 ) {
                  continue;
         }
         System.out.print(x);
         System.out.print ("\n"); }}}
```

This would produce the following result:

10 20 40 50

Declaring Class

You've seen classes defined in the following way:

```
Class  cs{
    // field, constructor and method declarations
}
```

The *class body* (the area between the braces) contains all the code like Field, constructor, and method declarations

In general, class declarations can include these components, in order:

1. Modifiers such as *public*, *private*, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. **The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.**
4. **A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.**
5. The class body, surrounded by braces, {}.

Creating Objects

To create a Cs class object, you use the new operator, which returns a reference to a new object. A *reference* is a kind of pointer to an object, which you can use to invoke methods on the object. In other words, to send an object a message (by invoking a method on the object), you have to have a

reference to that object. To keep track of references, you can declare variables in which you can store the references.

Cs obj1 = new Cs(); // here obj1 is object

Creating Method:

Considering the following example to explain the syntax of a method:

```
public static int funcName(int a, int b) {
  // body
}Here,
```

- **public static** : modifier.
- **int**: return type
- **funcName**: function name
- **a, b**: formal parameters
- **int a, int b**: list of parameters

Methods are also known as Procedures or Functions:

- **Procedures:** They don't return any value.
- **Functions:** They return value.

Method definition consists of a method header and a method body. The same is shown below:

```
modifier returnType nameOfMethod (Parameter List) {
 // method body
}
```

How to access the data using object

ClassName object=new ClassName();

Object.methodname();

Method Overloading in Java

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs. So, we perform method overloading to figure out the program quickly.

here are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In java, Methood Overloading is not possible by changing the return type of the method.

Example of Method Overloading by changing the no. of arguments

```
class Calculation{

  void sum(int a,int b){System.out.println(a+b);}

  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){

  Calculation obj=new Calculation();

  obj.sum(10,10,10);    obj.sum(20,20);

} }
```

2)Example of Method Overloading by changing data type of argument

```
1. class Calculation{
2.   void sum(int a,int b){System.out.println(a+b);}
3.   void sum(double a,double b){System.out.println(a+b);}
4.   public static void main(String args[]){
5.   Calculation obj=new Calculation();
6.   obj.sum(10.5,10.5);
7.   obj.sum(20,20);
8.
9.   }
10.}
11.Output:21.0
12.       40
```

Que) Why Method Overloaing is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:because there was problem:

```
1. class Calculation{
2.   int sum(int a,int b){System.out.println(a+b);}
3.   double sum(int a,int b){System.out.println(a+b);}
```

```
4.    public static void main(String args[]){
5.    Calculation obj=new Calculation();
6.    int result=obj.sum(20,20); //Compile Time Error
7.
8.    }
9.  }
10. int result=obj.sum(20,20); //Here how can java determine which sum() method should be called
```

Can we overload main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. Let's see the simple example:

```
1.  class Simple{
2.    public static void main(int a){
3.    System.out.println(a);
4.    }
5.
6.    public static void main(String args[]){
7.    System.out.println("main() method invoked");
8.    main(10);
9.    }
10.}            Output:main() method invoke
```

Constructors in Java

1. Types of constructors
    1. Default Constructor
    2. Parameterized Constructor
2. Constructor Overloading
3. Does constructor return any value?
4. Copying the values of one object into another
5. Does constructor perform other tasks instead of the initialization

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

There are two rules defined for the 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

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

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

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

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

```
1.  //Java Program to create and call a default constructor
2.  class Bike1{
3.  //creating a default constructor
4.  Bike1(){System.out.println("Bike is created");}
5.  //main method
6.  public static void main(String args[]){
7.  //calling a default constructor
8.  Bike1 b=new Bike1();
9.  }
```

```
10.     }
```

Output:

```
Bike is created
```

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

Q) 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.

Example of default constructor that displays the default values

```
1.      //Let us see another example of default constructor
2.      //which displays the default values
3.      class Student3{
4.      int id;
5.      String name;
6.      //method to display the value of id and name
7.      void display(){System.out.println(id+" "+name);}
8.
9.      public static void main(String args[]){
10.     //creating objects
11.     Student3 s1=new Student3();
12.     Student3 s2=new Student3();
13.     //displaying values of the object
14.     s1.display();
15.     s2.display();
16.     }
17.     }
```

Output:

```
0 null
0 null
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

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

```
1.  //Java Program to demonstrate the use of the parameterized constructor.
2.  class Student4{
3.     int id;
4.     String name;
5.     //creating a parameterized constructor
6.     Student4(int i,String n){
7.     id = i;
8.     name = n;
9.     }
10.    //method to display the values
11.    void display(){System.out.println(id+" "+name);}
12.
13.    public static void main(String args[]){
14.    //creating objects and passing values
15.    Student4 s1 = new Student4(111,"Karan");
16.    Student4 s2 = new Student4(222,"Aryan");
17.    //calling method to display the values of object
18.    s1.display();
19.    s2.display();
20.   }
21. }
```

Output:

```
111 Karan
222 Aryan
```

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.     }
```

Output:

```
111 Karan 0
222 Aryan 25
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

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

Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

```
1.  //Java program to initialize the values from one object to another object.
2.  class Student6{
3.     int id;
4.     String name;
5.     //constructor to initialize integer and string
6.     Student6(int i,String n){
7.     id = i;
8.     name = n;
9.     }
10.    //constructor to initialize another object
```

```
11.         Student6(Student6 s){
12.         id = s.id;
13.         name =s.name;
14.         }
15.         void display(){System.out.println(id+" "+name);}
16.
17.         public static void main(String args[]){
18.         Student6 s1 = new Student6(111,"Karan");
19.         Student6 s2 = new Student6(s1);
20.         s1.display();
21.         s2.display();
22.        }
23.     }
```

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1.      class Student7{
2.         int id;
3.         String name;
4.         Student7(int i,String n){
5.         id = i;
6.         name = n;
7.         }
8.         Student7(){}
9.         void display(){System.out.println(id+" "+name);}
10.
11.        public static void main(String args[]){
12.        Student7 s1 = new Student7(111,"Karan");
13.        Student7 s2 = new Student7();
14.        s2.id=s1.id;
15.        s2.name=s1.name;
16.        s1.display();
17.        s2.display();
18.        }
19.     }
```

```
20. 111 Karan
```

21. 111 Karan

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

```
1.  class Student{
2.      int rollno;
3.      String name;
```

```
4.          String college="ITS";
5.      }
```

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

```
1.  //Java Program to demonstrate the use of static variable
2.  class Student{
3.    int rollno;//instance variable
4.    String name;
5.    static String college ="ITS";//static variable
6.    //constructor
7.    Student(int r, String n){
8.    rollno = r;
9.    name = n;
10.   }
11.   //method to display the values
12.   void display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. public class TestStaticVariable1{
16.  public static void main(String args[]){
17.  Student s1 = new Student(111,"Karan");
18.  Student s2 = new Student(222,"Aryan");
19.  //we can change the college of all objects by the single line of code
20.  //Student.college="BBDIT";
21.  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.
6.  Counter(){
7.  count++;//incrementing value
8.  System.out.println(count);
9.  }
10.
11. public static void main(String args[]){
12. //Creating objects
13. Counter c1=new Counter();
14. Counter c2=new Counter();
15. Counter c3=new Counter();
16. }
17. }
```

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.
6.  Counter2(){
7.  count++;//incrementing the value of static variable
8.  System.out.println(count);
```

```
9.          }
10.
11.         public static void main(String args[]){
12.         //creating objects
13.         Counter2 c1=new Counter2();
14.         Counter2 c2=new Counter2();
15.         Counter2 c3=new Counter2();
16.         }
17.         }
```

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

```
1.          //Java Program to demonstrate the use of a static method.
2.          class Student{
3.              int rollno;
4.              String name;
5.              static String college = "ITS";
6.              //static method to change the value of static variable
7.              static void change(){
8.              college = "BBDIT";
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);}
```

```
    }
    //Test class to create and display the values of object
    public class TestStaticMethod{
      public static void main(String args[]){
      Student.change();//calling change method
      //creating objects
      Student s1 = new Student(111,"Karan");
      Student s2 = new Student(222,"Aryan");
      Student s3 = new Student(333,"Sonoo");
      //calling display method
      s1.display();
      s2.display();
      s3.display();
      }
    }
```

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

```
Output:125
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

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

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

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

Static Class

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

1. Nested static class doesn't need reference of Outer class
2. A static class cannot access non-static members of the Outer class

We will see these two points with the help of an example:

Static class Example

```
class JavaExample1{
  private static String str = "BeginnersBook";

  //Static class
  static class MyNestedClass{
          //non-static method
          public void disp() {

            /* If you make the str variable of outer class
             * non-static then you will get compilation error
             * because: a nested static class cannot access non-
             * static members of the outer class.
             */
            System.out.println(str);
          }
```

```
  }
  public static void main(String args[])
  {
     /* To create instance of nested class we didn't need the outer
           * class instance but for a regular nested class you would need
           * to create an instance of outer class first
      */
           JavaExample1.MyNestedClass obj = new JavaExample1.MyNestedClass();
           obj.disp();
  }
}
```

Output:

```
BeginnersBook
```

Constructors and Constructor overloading

A **constructor** looks more like a method but without return type. Moreover, the name of the constructor and the class name should be the same. The advantage of constructors over methods is that they are **called implicitly whenever an object is created. In case of methods**, **they must be called explicitly**. To create an object, the constructor must be called. Constructor gives properties to an object at the time of creation only. Programmer uses constructor to initialize variables, instantiating objects, and setting colors.

Default Constructor – No Argument Constructor

A constructor without parameters is called as "**default constructor**" or "**no-args constructor**". It is called default because if the programmer does not write himself, JVM creates one and supplies. The default constructor supplied by the JVM does not have any functionality (output).

```
1  public class Demo
2  {
3    public Demo()
4    {
5      System.out.println("From default constructor");
6    }
7    public static void main(String args[])
8    {
9      Demo d1 = new Demo();
10     Demo d2 = new Demo();
11   }
12 }
```

public Demo()

"**public**" is the access specifier and "**Demo()**" is the constructor. Notice, it does not have return type and the name is that of the class name.

Demo d1 = new Demo();

In the above statement, **d1** is an object of Demo class. To create the object, the constructor "Demo()" is called. Like this, any number of objects can be created like **d2** and for each object the constructor is called.

Constructor Overloading

Just like method overloading, constructors also can be overloaded. Same constructor declared with different parameters in the same class is known as constructor overloading. Compiler differentiates which constructor is to be called depending upon the number of parameters and their sequence of data types.

```
1  public class Perimeter
2  {
3    public Perimeter()                                  // I
4    {
5      System.out.println("From default");
6    }
7    public Perimeter(int x)                             // II
8    {
9      System.out.println("Circle perimeter: " + 2*Math.PI*x);
10   }
11   public Perimeter(int x, int y)                      // III
12   {
13     System.out.println("Rectangle perimeter: " +2*(x+y));
14   }
15   public static void main(String args[])
16   {
17     Perimeter p1 = new Perimeter();                  // I
18     Perimeter p2 = new Perimeter(10);                // II
19     Perimeter p3 = new Perimeter(10, 20);            // III
20   }
21 }
```

Perimeter constructor is overloaded three times. As per the parameters, an appropriate constructor is called. To call all the three constructors three objects are created.

Static methods

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

The static can be:

1. variable (also known as class variable)

2. method (also known as class method)
3. block

Example of static variable

```
1. //Program of static variable
2.
3. class Student{
4.    int rollno;
5.    String name;
6.    static String college ="ITS";
7.
8.    Student(int r,String n){
9.    rollno = r;
10.   name = n;
11.   }
12. void display (){System.out.println(rollno+" "+name+" "+college);}
13.
14. public static void main(String args[]){
15. Student s1 = new Student (111,"Karan");
16. Student s2 = new Student (222,"Aryan");
17.
18. s1.display();
19. s2.display();
20. }
21.}
```

static method

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

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

Restrictions for static method

There are two main restrictions for the static method.
They are:

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

Q) why main method is static?

Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

Static block

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

Example of static block

```
1.  class A{
2.
3.    static{System.out.println("static block is invoked");}
4.
5.    public static void main(String args[]){
6.     System.out.println("Hello main");
7.    }
8.  }
```

Output: static block is invoked
Hello main

Can we execute a program without main () method?

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

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

Output: static block is invoked (if not JDK7)

Usage of this keyword

Here is given the 6 usage of this keyword.

1. this keyword can be used to refer current class instance variable.

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

Final Keyword

- A java variable can be declared using the keyword final. Then the final variable can be assigned only once.
- Java classes declared as final cannot be extended.
- Methods declared as final cannot be overridden. In methods private is equal to final, but in variables it is not.

Access Modifiers in Java

1. Private access modifier
2. Role of private constructor
3. Default access modifier
4. Protected access modifier
5. Public access modifier
6. Access Modifier with Method Overriding

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

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

```
1.  class A{
2.  private int data=40;
3.  private void msg(){System.out.println("Hello java");}
4.  }
5.
6.  public class Simple{
7.   public static void main(String args[]){
8.     A obj=new A();
9.     System.out.println(obj.data);//Compile Time Error
10.    obj.msg();//Compile Time Error
11.    }
12. }
```

Role of Private Constructor

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

```
1.  class A{
2.  private A(){}//private constructor
3.  void msg(){System.out.println("Hello java");}
4.  }
5.  public class Simple{
6.   public static void main(String args[]){
7.     A obj=new A();//Compile Time Error
8.   }
9.  }
```

Note: A class cannot be private or protected except nested class.

---

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

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

```
1.  //save by A.java
2.  package pack;
3.  class A{
4.    void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.  class B{
5.    public static void main(String args[]){
6.     A obj = new A();//Compile Time Error
7.     obj.msg();//Compile Time Error
8.    }
9.  }
```

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

---

3) Protected

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

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

It provides more accessibility than the default modifer.

Example of protected access modifier

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

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.  protected void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.
5.  class B extends A{
6.    public static void main(String args[]){
7.     B obj = new B();
8.     obj.msg();
9.    }
10. }
```

```
Output:Hello
```

---

4) Public

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

Example of public access modifier

```
1.  //save by A.java
2.
3.  package pack;
4.  public class A{
5.  public void msg(){System.out.println("Hello");}
6.  }
```

```
1.  //save by B.java
2.
3.  package mypack;
4.  import pack.*;
5.
6.  class B{
7.    public static void main(String args[]){
8.     A obj = new A();
9.     obj.msg();
10.   }
```

```
11.     }
Output:Hello
```

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1.      class A{
2.      protected void msg(){System.out.println("Hello java");}
3.      }
4.
5.      public class Simple extends A{
6.      void msg(){System.out.println("Hello java");}//C.T.Error
7.       public static void main(String args[]){
8.         Simple obj=new Simple();
9.         obj.msg();
10.        }
11.     }
```

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

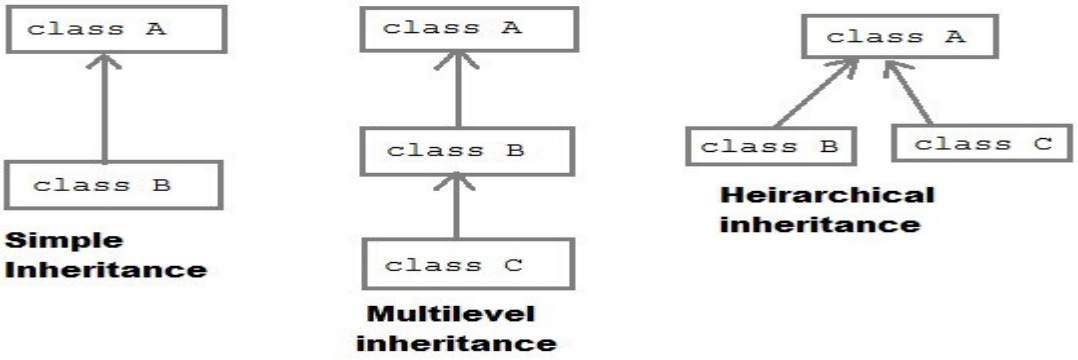
A class that is derived from another class is called *subclass* and inherits all fields and methods of its *superclass*. In Java, only single inheritance is allowed and thus, every class can have at most one direct superclass. A class can be derived from another class that is derived from another class and so on. Finally, we must mention that each class in Java is implicitly a subclass of the Object class.

Suppose we have declared and implemented a class *A*. In order to declare a class *B* that is derived from *A*, Java offers the *extend* keyword that is used as shown below:

```
class A {
        //Members and methods declarations.
}

class B extends A {
        //Members and methods from A are inherited.
        //Members and methods declarations of B.
}

public class Animal {
public void sleep() {System.out.println("An animal sleeps...");}

        public void eat() {System.out.println("An animal eats...");}
}
public class Bird extends Animal {
```

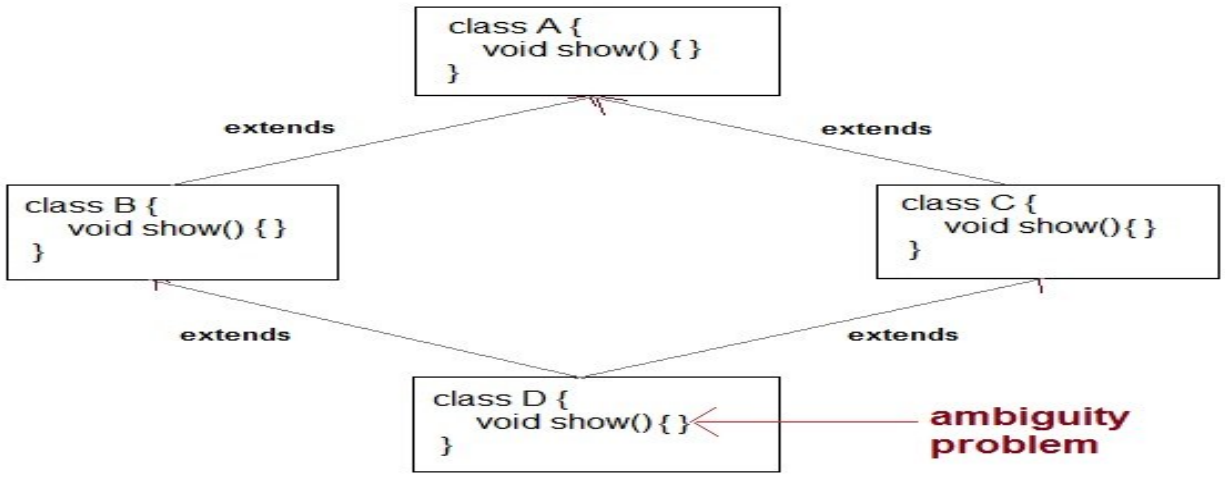
```
public void fly() {
            System.out.println ("A bird flies...");
 }

public class MainClass {
                    public static void main(String[] args)
                {
           Bird  b1= new Bird();
                        b1.sleep ();
          }
```

NOTE : Multiple inheritance is not supported in java



Why multiple inheritance is not supported in Java

- To remove ambiguity.
- To provide more maintainable and clear design.



Methode Overriding:-

when a method in a Subclass has the same name and same return-type and same parameters as that of method in its Superclass, then the method in the Subclass is said to Override the method in the Superclass"

It should be noted that, method overriding occurs only when the names & type signatures fo the two methodes are same. If they are not, then the two methodes are simply overloaded. Following example will explain mehtod overriding in details.

```
public class A
 {
  void show(int i){
   System.out.println("i inside superclass : " + i);
  }
 }

 //Subclass B
 public class B extends A
 {
  //overriedn methode "show(int i)"
  void show(int i){

   System.out.println ("i inside subclass: " + i);

//Access Superclass version of
   //an overriden method show()
   //super.methodname()
     super.show(20);

  }
 }
    public class Main
    {
      public static void main(String[] args)
      {
        B objB= new B();
        objB.show(10);
      }
    }
```

Output:-
i inside subclass : 10
i inside superclass : 20

this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```
1.          class A{
```

```
2.      A(){System.out.println("hello a");}
3.      A(int x){
4.      this();
5.      System.out.println(x);
6.      }
7.      }
8.      class TestThis5{
9.      public static void main(String args[]){
10.     A a=new A(10);
11.     }}
```

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor:

```
1.      class A{
2.      A(){
3.      this(5);
4.      System.out.println("hello a");
5.      }
6.      A(int x){
7.      System.out.println(x);
8.      }
9.      }
10.     class TestThis6{
11.     public static void main(String args[]){
12.     A a=new A();
13.     }}
```

Output:

```
5
hello a
```

Dynamic method dispatch:-

Dynamic method dispatch is one type of mechanism by which a call to an overridden method is resolved at run time, rather than at compile time. Dynamic method dispatch allow Java to implement run-time polymorphism.

When an overridden method is called through the obeject of superclass then Java determines which version of that method to execute, based upon the type of the object being referred to at the time the call occurs, hence determination is made at run time. Go througn following example to clear this concept.

```
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
THE JAVA LANGUAGE
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

Abstract class

If a class contain any abstract method then the class is declared as abstract class. An abstract class is never instantiated. It is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method

The syntax is
abstract class class_name { }

Method that are declared without any body within an abstract class is known as abstract method The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax
Abstract return type function name ();

Example of Abstract class

```
abstract class A
{
 abstract void callme();
}
class B extends A
{
 void callme()
 {
  System.out.println("this is callme.");
 }
 public static void main(String[] args)
 {
  B b=new B();
  b.callme();
 }
}
```

Abstract class with concrete (normal) method

```
abstract class A
{
 abstract void callme();
 public void normal()
 {
  System.out.println("this is concrete method");
 }
}
class B extends A
{
```

```
 void callme()
 {
  System.out.println("this is callme.");
 }
 public static void main(String[] args)
 {
  B b=new B();
  b.callme();
  b.normal();
 }
}
```

When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class

final is used to prevent overriding

You can use final keyword to declare a method, and methods declared with final cannot be overridden. There are certain situations where we need to prevent method overriding and so final is used to achieve that.

```
class Super
{
final void Display()
{
    System.out.println (“Display is final method”);
}
}

/*This is not possible because Display  method is declared final in superclass*/

class SubClass extends Super
{

public void Display()
{System.out.println (“Overriding is not possible for final methods”);}

}
```

3) final is used to prevent Inheritance

You can also use final to prevent inheritance as well, you just need to declare a class as final to do this.

Example:

```
final class Super
{
    statement 1;
    statement 2;
}

/*This is not possible as
final class cannot be inherited*/

public class SubClass extends Super
{
    statement 3;
    statement 4;
}
```

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="javatpoint";
```

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.

We will discuss immutable string later. Let's first understand what is String in Java and how to create the String object.

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).

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.

Do You Know?

- Why are String objects immutable?
- How to create an immutable class?
- What is string constant pool?
- What code is written by the compiler if you concatenate any string by + (string concatenation operator)?
- What is the difference between StringBuffer and StringBuilder class?

What will we learn in String Handling?

- Concept of String
- Immutable String
- String Comparison
- String Concatenation
- Concept of Substring
- String class methods and its usage
- StringBuffer class
- StringBuilder class
- Creating Immutable class
- toString() method
- StringTokenizer class

Immutable String in Java

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

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

Let's try to understand the immutability concept by the example given below:

```
1.  class Testimmutablestring{
2.   public static void main(String args[]){
3.     String s="Sachin";
4.     s.concat(" Tendulkar");//concat() method appends the string at the end
5.     System.out.println(s);//will print Sachin because strings are immutable objects
6.   }
7.  }
```

Test it Now

```
Output:Sachin
```

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

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

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

```
1.  class Testimmutablestring1{
2.   public static void main(String args[]){
3.     String s="Sachin";
4.     s=s.concat(" Tendulkar");
5.     System.out.println(s);
6.   }
7.  }
```

Test it Now

```
Output:Sachin Tendulkar
```

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

Why string objects are immutable in java?

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

OBJECT CLASS

The **java.lang.Object** class is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Methods in object class

- **clone**()
  Creates a new object of the same class as this object.

equals(Object)
  Compares two Objects for equality.

- **finalize**()
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

- **getClass**()
  Returns the runtime class of an object.
- **hashCode**()
  Returns a hash code value for the object.
- **notify**()
  Wakes up a single thread that is waiting on this object's monitor.
- **notifyAll**()
  Wakes up all threads that are waiting on this object's monitor.
- **toString**()
  Returns a string representation of the object.
- **wait**()
  Waits to be notified by another thread of a change in this object.
- **wait**(long)
  Waits to be notified by another thread of a change in this object.
- **wait**(long, int)
  Waits to be notified by another thread of a change in this object.

Java String compare

We can compare string in java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare string in java:

1. By equals() method
2. By = = operator
3. By compareTo() method

1) String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

```
1.  class Teststringcomparison1{
2.   public static void main(String args[]){
3.     String s1="Sachin";
4.     String s2="Sachin";
5.     String s3=new String("Sachin");
```

```
6.          String s4="Saurav";
7.          System.out.println(s1.equals(s2));//true
8.          System.out.println(s1.equals(s3));//true
9.          System.out.println(s1.equals(s4));//false
10.       }
11.      }
```

Test it Now

```
Output:true
       true
       false
```

```
1.       class Teststringcomparison2{
2.        public static void main(String args[]){
3.          String s1="Sachin";
4.          String s2="SACHIN";
5.
6.          System.out.println(s1.equals(s2));//false
7.          System.out.println(s1.equalsIgnoreCase(s2));//true
8.        }
9.       }
```

Test it Now

Output:

```
false
true
```

Click here for more about equals() method

---

2) String compare by == operator

The = = operator compares references not values.

```
1.       class Teststringcomparison3{
2.        public static void main(String args[]){
3.          String s1="Sachin";
4.          String s2="Sachin";
5.          String s3=new String("Sachin");
6.          System.out.println(s1==s2);//true (because both refer to same instance)
7.          System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8.        }
9.       }
```

Test it Now

```
Output:true
       false
```

---

3) String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

```
1.  class Teststringcomparison4{
2.   public static void main(String args[]){
3.     String s1="Sachin";
4.     String s2="Sachin";
5.     String s3="Ratan";
6.     System.out.println(s1.compareTo(s2));//0
7.     System.out.println(s1.compareTo(s3));//1(because s1>s3)
8.     System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
9.    }
10.  }
```

Test it Now

```
Output:0
       1
       -1
```

String Concatenation in Java

In java, string concatenation forms a new string *that is* the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

1) String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings. For Example:

```
1.  class TestStringConcatenation1{
2.   public static void main(String args[]){
3.     String s="Sachin"+" Tendulkar";
4.     System.out.println(s);//Sachin Tendulkar
5.    }
6.  }
```

Test it Now

```
Output:Sachin Tendulkar
```

The **Java compiler transforms** above code to this:

```
1.      String s=(new StringBuilder()).append("Sachin").append(" Tendulkar).toString();
```

In java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

```
1.      class TestStringConcatenation2{
2.       public static void main(String args[]){
3.         String s=50+30+"Sachin"+40+40;
4.         System.out.println(s);//80Sachin4040
5.       }
6.      }
```

Test it Now

```
80Sachin4040
```

Note: After a string literal, all the + will be treated as string concatenation operator.

2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

```
1.      public String concat(String another)
```

Let's see the example of String concat() method.

```
1.      class TestStringConcatenation3{
2.       public static void main(String args[]){
3.         String s1="Sachin ";
4.         String s2="Tendulkar";
5.         String s3=s1.concat(s2);
6.         System.out.println(s3);//Sachin Tendulkar
7.       }
8.      }
```

Test it Now

```
Sachin Tendulkar
```

Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

Note: Index starts from 0.

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex, int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

- **startIndex:** inclusive
- **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

```
1.  String s="hello";
2.  System.out.println(s.substring(0,2));//he
```

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

Example of java substring

```
1.  public class TestSubstring{
2.   public static void main(String args[]){
3.     String s="SachinTendulkar";
4.     System.out.println(s.substring(6));//Tendulkar
5.     System.out.println(s.substring(0,6));//Sachin
6.   }
7.  }
```

Test it Now

```
Tendulkar
Sachin
```

Interface is a pure abstract class. They are syntactically similar to classes, but you cannot create instance of an interface. Interface is used to achieve complete abstraction in Java

interface interface_name { }

```
interface Moveable
{
 int AVG-SPEED = 40;
 void move();
}
```

```
class Vehicle implements Moveable
{
 public void move()
 {
  System .out. print in ("Average speed is"+AVG-SPEED");
 }
 public static void main (String[] arg)
 {
  Vehicle vc = new Vehicle();
  vc.move();
 }
}
```

Interfaces supports Multiple Inheritance

```
interface Moveable
{
 boolean isMoveable();
}

interface Rollable
{
 boolean isRollable
}

class Tyre implements Moveable, Rollable
{
 int width;

 boolean isMoveable()
 {
  return true;
 }

 boolean isRollable()
 {
  return true;
 }
 public static void main(String args[])
 {
  Tyre tr=new Tyre();
  System.out.println(tr.isMoveable());
  System.out.println(tr.isRollable());
 }
}
```

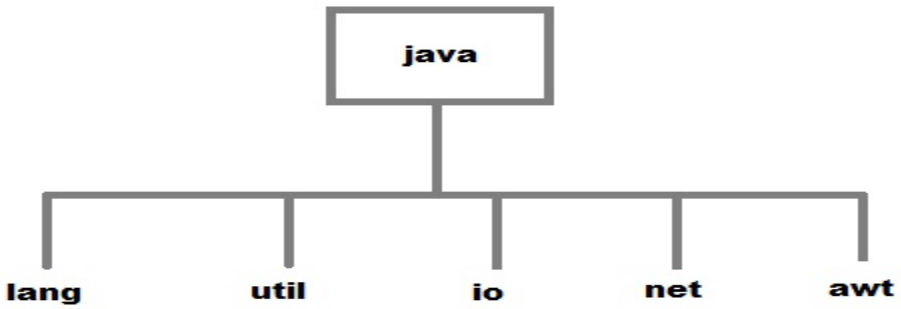
interface extends other interface

```
interface NewsPaper
{
 news();
}

interface Magazine extends NewsPaper
{
 colorful();
}
```

A package can be defined as a group of similar types of classes, interface, enumeration and sub-package. Using package it becomes easier to locate the related classes"/>

Package are categorized into two forms

- Built-in Package:-Existing Java package for example `java.lang`, `java.util` etc.
- User-defined-package:- Java package created by user to categorized classes and interface



Package access protection

Classes within a package can access classes and members declared with *default access* and class members declared with the *protected* access modifier. Default access is enforced when neither the public, protected, nor private access modifier is specified in the declaration. By contrast, classes in other packages cannot access classes and members declared with default access. Class members declared as protected can be accessed from the classes in the same package as well as classes in other packages that are subclasses of the declaring class.

Core packages in Java SE 6

Main article: Java Platform, Standard Edition

java.lang — basic language functionality and fundamental types
java.util — collection data structure classes
java.io — file operations
java.math — multiprecision arithmetics
java.nio — the New I/O framework for Java

java.net — networking operations, sockets, DNS lookups, ...
java.security — key generation, encryption and decryption
java.sql — Java Database Connectivity (JDBC) to access databases
java.awt — basic hierarchy of packages for native GUI components
javax.swing — hierarchy of packages for platform-independent rich GUI components
java.applet — classes for creating an applet

Creating a package

Creating a package in java is quite easy. Simply include a package command followed by name of the package as the first statement in java source file.

```
package mypack;
public class employee
{
 ...statement;
}
```

The above statement create a package called **mypack**.

Java uses file system directory to store package. For example the `.class` for any classes you to define to be part of **mypack** package must be stored in a directory called mypack

To run this program :

- create a directory under your current working development directory(i.e. JDK directory), name it as **mypack**.
- compile the source file
- Put the class file into the directory you have created.
- Execute the program from development directory.

NOTE : Development directory is the directory where your JDK is install.

Uses of java package

Package is a way to organize files in java, it is used when a project consists of multiple modules. It also helps resolve naming conflicts. Package's access level also allows you to protect data from being used by the non-authorized classes.

import keyword

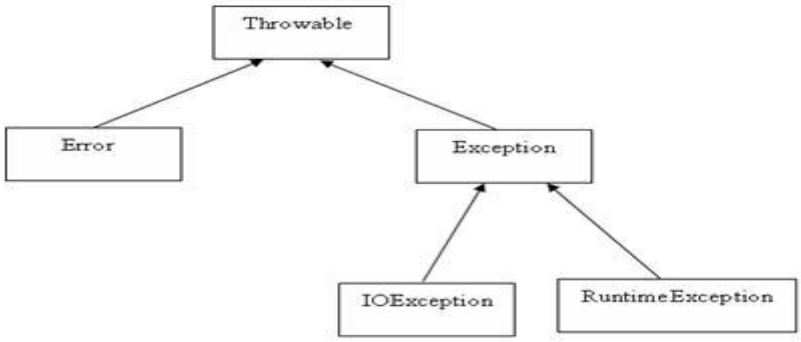
import keyword is used to import built-in and user-defined packages into your java source file. So that your class can refer to a class that is in another package by directly using its name.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



 Checked **exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

 Runtime **exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
   //Protected code
}catch(ExceptionName e1)
{
   //Catch block
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name: ExcepTest.java
import java.io.*;
public class ExcepTest {

   public static void main(String args[]){
      try{
         int a[] = new int[2];
         System.out.println ("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
```

```
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
   //Protected code
}catch (ExceptionType1 e1)
{
   //Catch block

} finally
{
   //The finally block always executes.
}
```

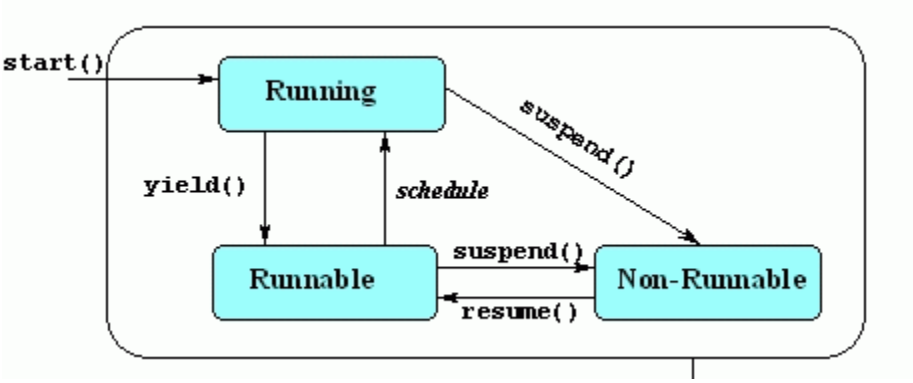
Declaring you own Exception:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

Thread Life cycle



New

When we create a new Thread object using *new* operator, thread state is New Thread. At this point, thread is not alive and it's a state internal to Java programming.

Runnable

When we call start () function on Thread object, it's state is changed to Runnable and the control is given to Thread scheduler to finish it's execution. Whether to run this thread instantly or keep it in runnable thread pool before running it depends on the OS implementation of thread scheduler.

Running

When thread is executing, it's state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it's state to Running and CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

Blocked/Waiting

A thread can be waiting for other thread to finish using thread join or it can be waiting for some resources to available, for example producer consumer problem or waiter notifier implementation or IO resources, then it's state is changed to Waiting. Once the thread wait state is over, its state is changed to Runnable and it's moved back to runnable thread pool.

Dead

Once the thread finished executing, it's state is changed to Dead and it's considered to be not alive.

How to create thread:

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread..
3. **public int getPriority():** returns the priority of the thread.
4. **public int setPriority(int priority):** changes the priority of the thread.
5. **public String getName():** returns the name of the thread.
6. **public void setName(String name):** changes the name of the thread.
7. **public Thread currentThread():** returns the reference of currently executing thread.
8. **public int getId():** returns the id of the thread.
9. **public Thread.State getState():** returns the state of the thread.
10. **public boolean isAlive():** tests if the thread is alive.
11. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
12. **public void suspend():** is used to suspend the thread(depricated).
13. **public void resume():** is used to resume the suspended thread(depricated).
14. **public void stop():** is used to stop the thread(depricated).
15. **public boolean isDaemon():** tests if the thread is a daemon thread.
16. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
17. **public void interrupt():** interrupts the thread.
18. **public boolean isInterrupted():** tests if the thread has been interrupted.
19. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1)By extending Thread class:

```
1. class Multi extends Thread{
2. public void run(){
3. System.out.println ("thread is running...");
4. }
5. public static void main(String args[]){
6. Multi t1=new Multi();
7. t1.start();
8.  }
9. }
```

2)By implementing the Runnable interface:

```
1. class Multi3 implements Runnable{
2. public void run(){
3. System.out.println ("thread is running...");
4. }
5.
6. public static void main(String args[]){
7. Multi3 m1=new Multi3();
8. Thread t1 =new Thread(m1);
9. t1.start();
10.  }
11. }
```

Can we start a thread twice?

No. After staring a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. For Example:

```
1. class Multi extends Thread{
2.  public void run(){
```

```
3.     System.out.println("running...");
4.    }
5.    public static void main(String args[]){
6.     Multi t1=new Multi();
7.     t1.start();
8.     t1.start();
9.    }
10. }
```

Creating multiple threads

Program Example

Thread Priorities

In Java, thread scheduler can use the thread **priorities** in the form of **integer value** to each of its thread to determine the execution schedule of threads . Thread gets the **ready-to-run** state according to their priorities. The **thread scheduler** provides the CPU time to thread of highest priority during ready-to-run state.
Priorities are integer values from 1 (lowest priority given by the constant **Thread.MIN_PRIORITY**) to 10 (highest priority given by the constant **Thread.MAX_PRIORITY**). The default priority is 5(**Thread.NORM_PRIORITY**).

Constant	Description
Thread.MIN_PRIORITY	The maximum priority of any thread (an int value of 1)
Thread.MAX_PRIORITY	The minimum priority of any thread (an int value of 10)
Thread.NORM_PRIORITY	The normal priority of any thread (an int value of 5)

The methods that are used to set the priority of thread shown as:

Method	Description
setPriority()	This is method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.

Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overrite data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement:

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

Multithreading example with Synchronization:

Here is the same example which prints counter value in sequence and every time we run it, it produces same result.

```
class PrintDemo {
   public void printCount(){
    try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
     } catch (Exception e) {
         System.out.println("Thread  interrupted.");
     }
   }

}

class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;
```

```
   ThreadDemo( String name,  PrintDemo pd){
      threadName = name;
      PD = pd;
   }
   public void run() {
     synchronized(PD) {
       PD.printCount();
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

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

      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

This produces same result every time you run this program:

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
```

```
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 1  exiting.
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

wo ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

final boolean isAlive( )

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise. While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate. Here is an improved version of the preceding example that uses **join( )** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive( )** method.

Notify and wait example 1

```
public class ThreadA {
   public static void main(String[] args){
      ThreadB b = new ThreadB();
      b.start();

      synchronized(b){
         try{
            System.out.println("Waiting for b to complete...");
            b.wait();
         }catch(InterruptedException e){
            e.printStackTrace();
         }

         System.out.println("Total is: " + b.total);
      }  }  }

class ThreadB extends Thread{
   int total;
   @Override
```

```
public void run(){
   synchronized(this){
      for(int i=0; i<100 ; i++){
         total += i;
      }
      notify(); } } }
```

Creating Strings:

The most direct way to create a string is to write:

String greeting = "Hello world!";

String Length:
int len = palindrome.length();

Concatenating Strings:

The String class includes a method for concatenating two strings:

string1.concat (string2)

String substring (int beginIndex)
Returns a new string that is a substring of this string.

String substring (int beginIndex, int endIndex)
Returns a new string that is a substring of this string.

String compare

string1.equals (string2) is the way.

It returns **true**, if both string1 is equal to string2. Else, **false**.

How to modify the string

The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.

Unlike Strings objects of type StringBuffer and Stringbuilder can be modified over and over again with out leaving behind a lot of new unused objects.

The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilders methods are not thread safe(not Synchronised).

It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However if thread safety is necessary the best option is StringBuffer objects.

Example:

```
public class Test{

   public static void main(String args[]){
     StringBuffer sBuffer = new StringBuffer(" test");
     sBuffer.append(" String Buffer");
     System.ou.println(sBuffer);
  }
}
```

This would produce the following result:

test String Buffer

StringBuffer Methods:

SN	Methods with Description
1	public StringBuffer append(String s) Updates the value of the object that invoked the method. The method takes boolean, char, int, long, Strings etc.
2	public StringBuffer reverse() The method reverses the value of the StringBuffer object that invoked the method.
3	public delete(int start, int end) Deletes the string starting from start index until end index.
4	public insert(int offset, int i) This method inserts an string s at the position mentioned by offset.
5	replace(int start, int end, String str) This method replaces the characters in a substring of this StringBuffer with characters in the specified String.

The **java.io.Interfaces** provides for system input and output through data streams, serialization and the file system.

Interface Summary

S.N.	Interface & Description
1	**Closeable** This is a source or destination of data that can be closed.
2	**DataInput** This provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.
3	**DataOutput** This provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream.
4	**Externalizable**

This provides only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances.

5 **FileFilter**
This is a filter for abstract pathnames.

6 **FilenameFilter**
This is instances of classes that implement this interface are used to filter filenames.

7 **Flushable**
This is a destination of data that can be flushed.

8 **ObjectInput**
This extends the DataInput interface to include the reading of objects.

9 **ObjectInputValidation**
This is the callback interface to allow validation of objects within a graph.

10 **ObjectOutput**
This is the objectOutput extends the DataOutput interface to include writing of objects.

11 **ObjectStreamConstants**
The constants written into the Object Serialization Stream.

12 **Serializable**
This is enabled by the class implementing the java.io.Serializable interface.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to Files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are , **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file:

```
import java.io.*;

public class CopyFile {
  public static void main(String args[]) throws IOException
  {
    FileInputStream in = null;
    FileOutputStream out = null;

    try {
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");

      int c;
      while ((c = in.read()) != -1) {
```

```
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, where as Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are , **FileReader** and **FileWriter.**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write above example which makes use of these two classes to copy an input file (having unicode characters) into an output file:

```
import java.io.*;

public class CopyFile {
  public static void main(String args[]) throws IOException
  {
    FileReader in = null;
    FileWriter out = null;

    try {
      in = new FileReader("input.txt");
      out = new FileWriter("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
```

```
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

$javac CopyFile.java
$java CopyFile

Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware if C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similar way Java provides following three standard streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**.

serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

public final Object readObject() throws IOException,
ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface:

```
public class Employee implements java.io.Serializable
{
   public String name;
   public String address;
   public transient int SSN;
   public int number;
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + name
                         + " " + address);
   }
}
```

Notice that for a class to be serialized successfully, two conditions must be met:

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object:

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

```
import java.io.*;

public class SerializeDemo
{
  public static void main(String [] args)
  {
    Employee e = new Employee();
    e.name = "Reyan Ali";
    e.address = "Phokka Kuan, Ambehta Peer";
    e.SSN = 11122333;
    e.number = 101;
    try
    {
      FileOutputStream fileOut =
      new FileOutputStream("/tmp/employee.ser");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      out.writeObject(e);
      out.close();
      fileOut.close();
      System.out.printf("Serialized data is saved in /tmp/employee.ser");
    }catch(IOException i)
    {
       i.printStackTrace();
    }
  }
}
```

Deserializing an Object:

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output:

```
import java.io.*;
public class DeserializeDemo
{
  public static void main(String [] args)
  {
    Employee e = null;
    try
    {
      FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
```

```
      e = (Employee) in.readObject();
      in.close();
      fileIn.close();
    }catch(IOException i)
    {
      i.printStackTrace();
      return;
    }catch(ClassNotFoundException c)
    {
      System.out.println("Employee class not found");
      c.printStackTrace();
      return;
    }
    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
    System.out.println("SSN: " + e.SSN);
    System.out.println("Number: " + e.number);
  }
}
```

This would produce the following result:

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

Here are following important points to be noted:

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

APPLET

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 give 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.

Passing parameter to applet

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

public class ParameterExample extends Applet
{
    // We'll save the first HTM parameter as a String
    String parameter1;
    // the second one we will use as an integer
    int parameter2;
    // third one too
    int parameter3;
    // we'll add param2 to param2
    int result;

    public void init()
    {
        // This method will get the specified parameter's value
```

```
        // out of the HTML code that is calling the applet.
        parameter1 = getParameter("param1");
        // Since those are read as text we need to transform them
        // to integers to be able to count with them.
        parameter2 = Integer.parseInt(getParameter("param2"));
        parameter3 = Integer.parseInt(getParameter("param3"));
        result = parameter2 + parameter3;
    }

    public void paint(Graphics g)
    {
        // Shows what was in the HTML param code.
        g.drawString("Parameter 1 is: " + parameter1,20,20);
        g.drawString("Parameter 2 is: " + parameter2,20,40);
        g.drawString("Parameter 3 is: " + parameter3,20,60);
        g.drawString("Parameter 2 + parameter 3 is: " + result,20,80);
     }
}

/* This only works when those paramters are actually in the HTML code.
   That code for this example is :
 <APPLET CODE="ParameterExample" WIDTH=200 HEIGHT=100>
  <param name="param1" value="Hello">
  <param name="param2" value="14">
  <param name="param3" value="2">
 </APPLET>
  If you make applets for others make sure to use parameters, many
  will appreciate it.
```

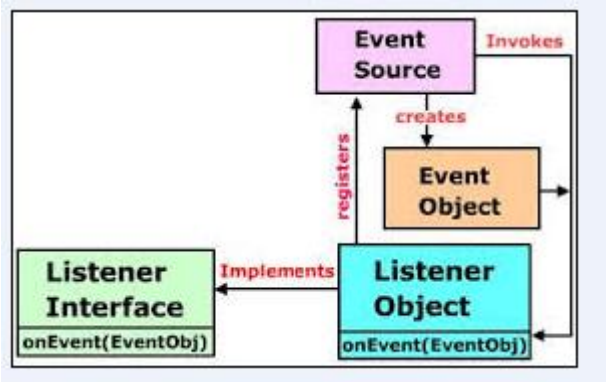
Delegation event model

The event model is based on the Event Source and Event Listeners. Event Listener is an object that receives the messages / events. The Event Source is any object which creates the message / event. The Event Delegation model is based on – The Event Classes, The Event Listeners, Event Objects.

There are three participants in event delegation model in Java;

- Event Source – the class which broadcasts the events
- Event Listeners – the classes which receive notifications of events
- Event Object – the class object which describes the event.

An event occurs (like mouse click, key press, etc) which is followed by the event is broadcasted by the event source by invoking an agreed method on all event listeners. The event object is passed as argument to the agreed-upon method. Later the event listeners respond as they fit, like submit a form, displaying a message / alert etc.



Following is the list of commonly used event classes.

Sr. No.	Control & Description
1	AWTEvent It is the root event class for all AWT events. This class and its subclasses supercede the original java.awt.Event class.
2	ActionEvent The ActionEvent is generated when button is clicked or the item of a list is double clicked.
3	InputEvent The InputEvent class is root event class for all component-level input events.
4	KeyEvent On entering the character the Key event is generated.
5	MouseEvent This event indicates a mouse action occurred in a component.
6	TextEvent The object of this class represents the text events.
7	WindowEvent The object of this class represents the change in state of a window.
8	AdjustmentEvent The object of this class represents the adjustment event emitted by Adjustable objects.
9	ComponentEvent The object of this class represents the change in state of a window.
10	ContainerEvent The object of this class represents the change in state of a window.
11	MouseMotionEvent The object of this class represents the change in state of a window.

EventListener Interfaces

An event listener registers with an event source to receive notifications about the events of a particular type. Various event listener interfaces defined in the `java.awt.event` package are given below :

Interface	Description
ActionListener	Defines the actionPerformed() method to receive and process action events.
MouseListener	Defines five methods to receive mouse events, such as when a mouse is clicked, pressed, released, enters, or exits a component
MouseMotionListener	Defines two methods to receive events, such as when a mouse is dragged or moved.
AdjustmentListner	Defines the adjustmentValueChanged() method to receive and process the adjustment events.
TextListener	Defines the textValueChanged() method to receive and process an event when the text value changes.
WindowListener	Defines seven window methods to receive events.
ItemListener	Defines the itemStateChanged() method when an item has been selected or deselected by the user.

The Java programming language class library provides a user interface toolkit called the Abstract Windowing Toolkit, or the AWT.

What is a user interface

The user interface is that part of a program that interacts with the user of the program.

Components and containers

A graphical user interface is built of graphical elements called components.

Containers contain and control the layout of components
These nine classes are class Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, and TextField.

Types of containers

The AWT provides four container classes. They are class Window and its two subtypes -- class Frame and class Dialog -- as well as the Panel class. In addition to the containers provided by the AWT, the Applet class is a

container -- it is a subtype of the Panel class and can therefore hold components. Brief descriptions of each container class provided by the AWT are provided below.

Window	A top-level display surface (a window). An instance of the Window class is not attached to nor embedded within another container. An instance of the Window class has no border and no title.
Frame	A top-level display surface (a window) with a border and title. An instance of the Frame class may have a menu bar. It is otherwise very much like an instance of the Window class.
Dialog	A top-level display surface (a window) with a border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.
Panel	A generic container for holding components. An instance of the Panel class provides a container to which to add components.

Creating a container

Before adding the components that make up a user interface, the programmer must create a container. When building an application, the programmer must first create an instance of class Window or class Frame. When building an applet, a frame (the browser window) already exists. Since the Applet class is a subtype of the Panel class, the programmer can add the components to the instance of the Applet class itself.

The code in Listing 1 creates an empty frame. The title of the frame ("Example 1") is set in the call to the constructor. A frame is initially invisible and must be made visible by invoking its `show()` method.

```
import java.awt.*;

public class Example1

{

   public static void main(String [] args)

   {

      Frame f = new Frame("Example 1");

      f.show();

   }

}
```

Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.

Class declaration

Following is the declaration for **java.awt.Canvas** class:

```
public class Canvas
   extends Component
      implements Accessible
```

Class constructors

S.N.	Constructor & Description
1	**Canvas()** Constructs a new Canvas.
2	**Canvas(GraphicsConfiguration config)** Constructs a new Canvas given a GraphicsConfiguration object.

Class methods

S.N.	Method & Description
1	**void addNotify()** Creates the peer of the canvas.
2	**void createBufferStrategy(int numBuffers)** Creates a new strategy for multi-buffering on this component.
3	**void createBufferStrategy(int numBuffers, BufferCapabilities caps)** Creates a new strategy for multi-buffering on this component with the required buffer capabilities.
4	**AccessibleContext getAccessibleContext()** Gets the AccessibleContext associated with this Canvas.
5	**BufferStrategy getBufferStrategy()** Returns the BufferStrategy used by this component.
6	**void paint(Graphics g)** Paints this canvas.
7	**void pdate(Graphics g)** Updates this canvas.

Methods inherited

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

How to create a canvas

Canvas C1 = new Canvas (); C1.setSize (120,120); C1.setBackground (Color.white); Frame F1 = new Frame (); F1.add (C1); F1.setLayout (new FlowLayout ()); F1.setSize (250,250); F1.setVisible (true);

What is an Event?

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. These mechanisms have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener process the event an then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

EVENT HANDLING IN JAVA/Event-driven programming

ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform EventHandling:

Following steps are required to perform event handling :

1. Implement the Listener interface and overrides its methods
2. Register the component with the Listener

Example of event handling within class:

```
1. import java.awt.*;
2. import java.awt.event.*;
3.
4. class AEvent extends Frame implements ActionListener{
5. TextField tf;
6. AEvent(){
7.
8. tf=new TextField();
9. tf.setBounds(60,50,170,20);
10.
11. Button b=new Button("click me");
12. b.setBounds(100,120,80,30);
13.
14. b.addActionListener(this);
15.
16. add(b);add(tf);
17.
18. setSize(300,300);
19. setLayout(null);
20. setVisible(true);
21.
22. }
23.
24. public void actionPerformed(ActionEvent e){
25. tf.setText("Welcome");
26. }
27.
28. public static void main(String args[]){
29. new AEvent();
30. }
31. }
```

Layout

Layout means the arrangement of components within the container. In other way we can say that placing the components at a particular position within the container. The task of layouting the controls is done automatically by the Layout Manager.

Layout Manager

The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager. It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

- It is very tedious to handle a large number of controls within the container.
- Oftenly the width and height information of a component is not given when we need to arrange them.

Java provide us with various layout manager to position the controls. The properties like size,shape and arrangement varies from one layout manager to other layout manager. When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the LayoutManager interface.

Following are the interfaces defining functionalities of Layout Managers.

Sr. No.	Interface & Description
1	LayoutManager The LayoutManager interface declares those methods which need to be implemented by the class whose object will act as a layout manager.
2	LayoutManager2 The LayoutManager2 is the sub-interface of the LayoutManager.This interface is for those classes that know how to layout containers based on layout constraint object.

AWT Layout Manager Classes:

Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	LayoutManager & Description
1	BorderLayout The borderlayout arranges the components to fit in the five regions: east, west, north, south and center.
2	CardLayout The CardLayout object treats each component in the container as a card. Only one card is visible at a time.
3	FlowLayout The FlowLayout is the default layout.It layouts the components in a directional flow.
4	GridLayout

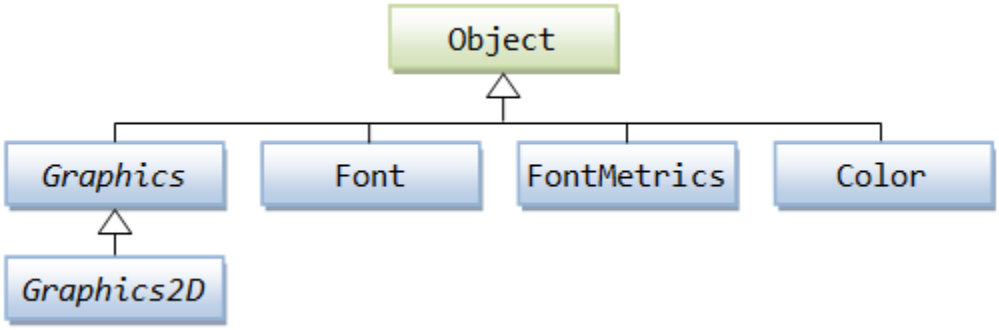
The GridLayout manages the components in form of a rectangular grid.

5 GridBagLayout
This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically,horizontally or along their baseline without requiring the components of same size.

6 GroupLayout
The GroupLayout hierarchically groups components in order to position them in a Container.

7 SpringLayout
A SpringLayout positions the children of its associated container according to a set of constraints.

The `java.awt.Graphics` is an `abstract` class, as the actual act of drawing is system-dependent and device-dependent. Each operating platform will provide a subclass of `Graphics` to perform the actual drawing under the platform, but conform to the specification defined in `Graphics`.

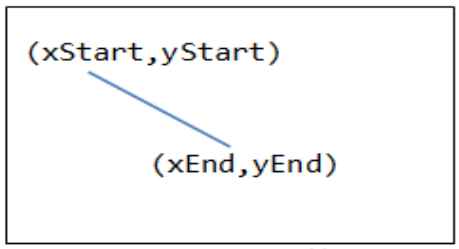


The `Graphics` class provides methods for drawing three types of graphical objects:

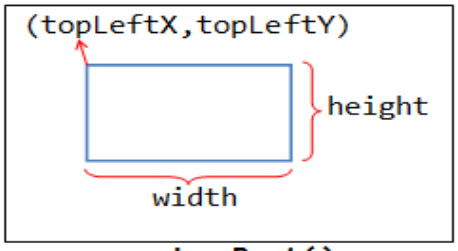
1. Text strings: via the `drawString()` method. Take note that `System.out.println()` prints to the system console, not to the graphics screen.
2. Vector-graphic primitives and shapes: via methods `drawXxx()` and `fillXxx()`, where `Xxx` could be `Line`, `Rect`, `Oval`, `Arc`, `PolyLine`, `RoundRect`, or `3DRect`.
3. Bitmap images: via the `drawImage()` method.

```
// Drawing (or printing) texts on the graphics screen:
drawString(String str, int xBaselineLeft, int yBaselineLeft);

// Drawing lines:
drawLine(int x1, int y1, int x2, int y2);
drawPolyline(int[] xPoints, int[] yPoints, int numPoint);

// Drawing primitive shapes:
drawRect(int xTopLeft, int yTopLeft, int width, int height);
drawOval(int xTopLeft, int yTopLeft, int width, int height);
drawArc(int xTopLeft, int yTopLeft, int width, int height, int startAngle, int arcAngle);
draw3DRect(int xTopLeft, int, yTopLeft, int width, int height, boolean raised);
drawRoundRect(int xTopLeft, int yTopLeft, int width, int height, int arcWidth, int
arcHeight)
drawPolygon(int[] xPoints, int[] yPoints, int numPoint);

// Filling primitive shapes:
fillRect(int xTopLeft, int yTopLeft, int width, int height);
fillOval(int xTopLeft, int yTopLeft, int width, int height);
fillArc(int xTopLeft, int yTopLeft, int width, int height, int startAngle, int arcAngle);
fill3DRect(int xTopLeft, int, yTopLeft, int width, int height, boolean raised);
```

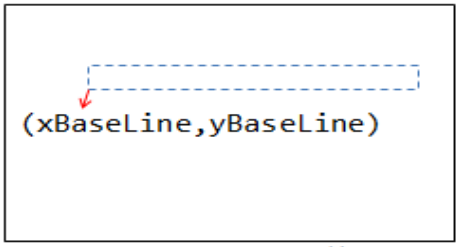
```
fillRoundRect(int xTopLeft, int yTopLeft, int width, int height, int arcWidth, int
arcHeight)
fillPolygon(int[] xPoints, int[] yPoints, int numPoint);

// Drawing (or Displaying) images:
drawImage(Image img, int xTopLeft, int yTopLeft, ImageObserver obs);  // draw image with
its size
drawImage(Image img, int xTopLeft, int yTopLeft, int width, int height, ImageObserver o);
// resize image on screen
```



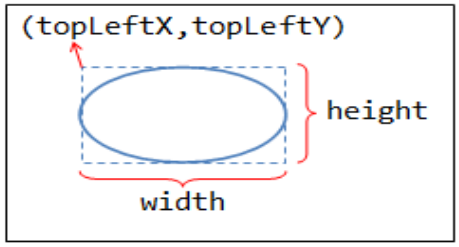
drawLine()



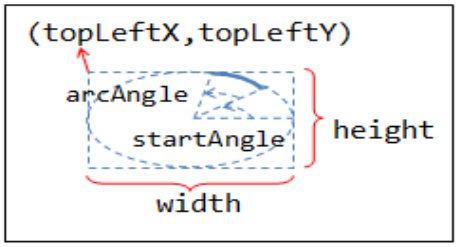
drawRect()



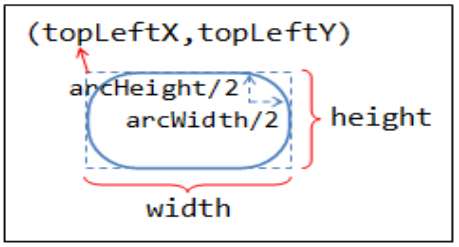
drawString()



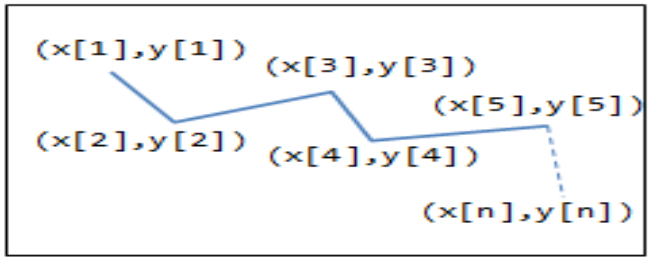
drawOval()



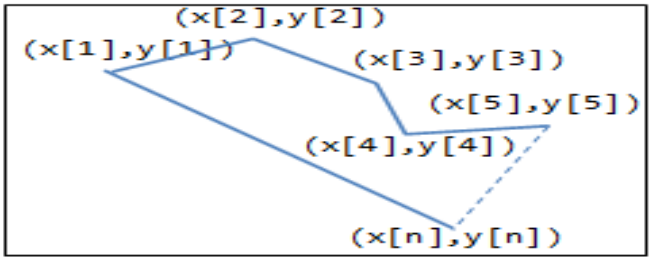
drawArc()



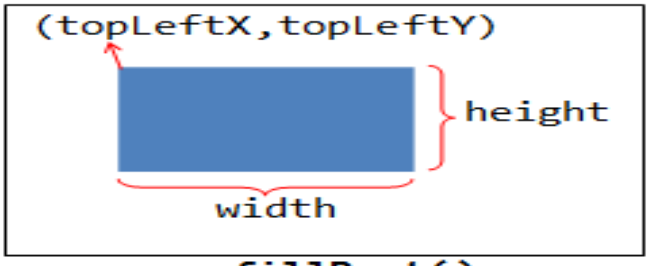
drawRoundRect()



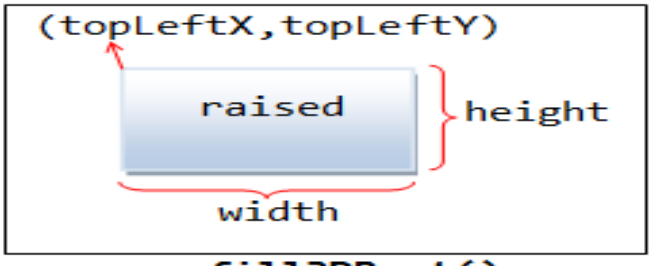
drawPolyline()



drawPolygon()



fillRect()



fill3DRect()

Remote Method Invocation (RMI)

The **Remote Method Invocation** (RMI) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

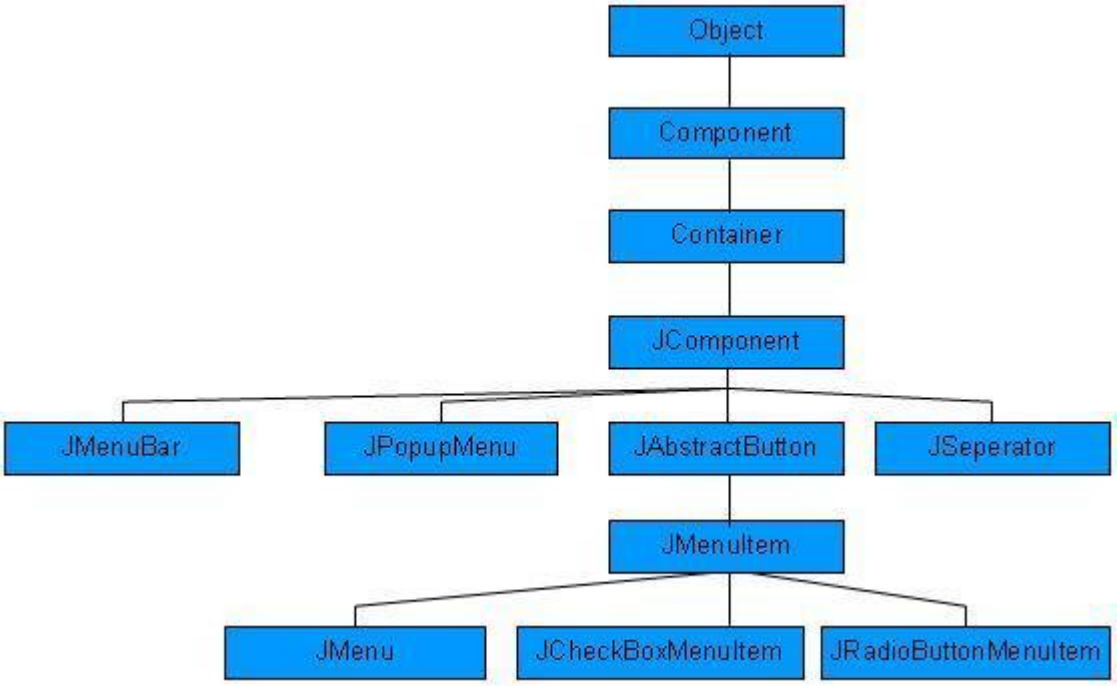
skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

s we know that every top-level window has a menu bar associated with it. This menu bar consist of various menu choices available to the end user. Further each choice contains list of options which is called drop down menus. Menu and MenuItem controls are subclass of MenuComponent class.

Menu Hiearchy



Class declaration

Following is the declaration for **javax.swing.JLabel** class:

```
public class JLabel
   extends JComponent
      implements SwingConstants, Accessible
```

Field

Following are the fields for **javax.swing.JLabel** class:

- **protected Component labelFor**

Class constructors

S.N.	Constructor & Description
1	**JLabel()** Creates a JLabel instance with no image and with an empty string for the title.
2	**JLabel(Icon image)** Creates a JLabel instance with the specified image.
3	**JLabel(Icon image, int horizontalAlignment)** Creates a JLabel instance with the specified image and horizontal alignment.
4	**JLabel(String text)** Creates a JLabel instance with the specified text.
5	**JLabel(String text, Icon icon, int horizontalAlignment)** Creates a JLabel instance with the specified text, image, and horizontal alignment.
6	**JLabel(String text, int horizontalAlignment)**

Creates a JLabel instance with the specified text and horizontal alignment.

The class **JTextField** is a component which allows the editing of a single line of text.

Class declaration

Following is the declaration for **javax.swing.JTextField** class:

```
public class JTextField
   extends JTextComponent
      implements SwingConstants
```

Field

Following are the fields for **javax.swing.JList** class:

- **static String notifyAction** -- Name of the action to send notification that the contents of the field have been accepted.

Class constructors

S.N.	Constructor & Description
1	**JTextField()** Constructs a new TextField.
2	**JTextField(Document doc, String text, int columns)** Constructs a new JTextField that uses the given text storage model and the given number of columns.
3	**JTextField(int columns)** Constructs a new empty TextField with the specified number of columns.
4	**JTextField(String text)** Constructs a new TextField initialized with the specified text.
5	**JTextField(String text, int columns)** Constructs a new TextField initialized with the specified text and columns

Introduction

The class **JButton** is an implementation of a push button. This component has a label and generates an event when pressed. It can have Image also.

Class declaration

Following is the declaration for **javax.swing.JButton** class:

```
public class JButton
   extends AbstractButton
      implements Accessible
```

Class constructors

S.N.	Constructor & Description
1	**JButton()** Creates a button with no set text or icon.
2	**JButton(Action a)** Creates a button where properties are taken from the Action supplied.
3	**JButton(Icon icon)** Creates a button with an icon.
4	**JButton(String text)** Creates a button with text.
5	**JButton(String text, Icon icon)** Creates a button with initial text and an icon.

Introduction

The class **JComboBox** is a component which combines a button or editable field and a drop-down list.

Class declaration

Following is the declaration for **javax.swing.JComboBox** class:

```
public class JComboBox
   extends JComponent
      implements ItemSelectable, ListDataListener,
         ActionListener, Accessible
```

Class constructors

S.N.	Constructor & Description
1	**JComboBox()** Creates a JComboBox with a default data model.
2	**JComboBox(ComboBoxModel aModel)** Creates a JComboBox that takes its items from an existing ComboBoxModel.
3	**JComboBox(Object[] items)** Creates a JComboBox that contains the elements in the specified array.
4	**JComboBox(Vector<?> items)** Creates a JComboBox that contains the elements in the specified Vector.

Jtable

`public JTable()` -- creates an instance of the JTable class that is initialized with a default data model, column model, and selection model.

`public JTable(TableModel dm)` -- creates an instance of the JTable class that is initialized with a default column model and selection model, and with the specified data model.

Jtree:

JTree()
JTree(Object[] value)

Memory Management in Java

IN Java, **Memory Management** means **Garbage Collection** and **memory allocation**. Memory allocation is very small process as compared to Garbage Collection. Indeed, a well playing Garbage collection makes everything easy for memory allocation. Only major issue before memory allocation is Weather sufficient Memory available?. And its Garbage Collector responsibility to ensure enough memory is available all the time, otherwise ready to a face biggest obstacle "**OutOfMemory**" in running application. Writing a efficient Garbage Collection algorithm is very tedious task. Thanks to JVM, that they come up with several algorithms, and by default apply the best one. In day to day programming one might never feel a need to understand those algorithms, unless you have passion for knowing whats going on under the hood. OR some day after years of programming you face "OutOfMemory" Exception, and then you check that there are plenty of options which can be used to avoid such situation, for example JVM comes with several algorithms, and you start googling which one to use. After so many years of working over Java/J2EE i really need a feel to understand those algorithms, JVM memory management, memory sizing, and defaults.

Memory Management is all about recognizing which objects are no longer needed, freeing the memory used by such objects, and making it available. In many modern Object oriented languages this process is automatic, and this automatic process is called **Garbage Collection**. There are several Algorithms which are used by Garbage Collector.

Garbage Collection Algorithms **-**

Reference Counting - Most straight forward GC(Garbage Collection) algorithm is reference counting. Its most simple algorithm, where we count the number of references to each object. If count is zero for any object, consider the object to be garbage. Compiler has responsibility of increasing the count, after executing any assignment statement for any particular object. The major issue with this algorithm is that it can never claim unreachable cyclic references. for example in the following figure –