Course Title: 403 Java Programming Language

Unit 1. Introduction to Java

1.1 Properties of Java

1.2 Comparison of java with C++

1.3 Java Compiler, Java Interpreter

1.4 Identifier, Literals, Operators, Variables, Keywords, Data Types

1.5 Branching: If – Else, Switch

1.6 Looping: While, Do-while, For

1.7 Type Casting

Java Programming- History of Java

➔ The history of java starts from Green Team. 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.

➔ For the green team members, it was an advance concept at that time. But it was suited for internet programming. Later, Java technology as incorporated by Netscape.

➔ Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc.

➔ There are given the major points that describes the history of java.

1. **James Gosling, Mike Sheridan, and Patrick Naughton** initiated the Java language project in **June 1991**. The small team of sun engineers called Green Team
2. Originally designed for small, embedded systems in electronic appliances like set top boxes.
3. Firstly, it was called **"Green talk"** by James Gosling and file extension was .gt.
4. After that, it was called **Oak** and was developed as a part of the Green project.

Why Oak name for java language?

5. Why Oak? Oak is a symbol of strength and choose as a national tree of many countries like U.S.A., France, Germany, Romania etc.
6. **In 1995**, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why they choose java name for java language?

7. The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred java.
8. Java is an island of Indonesia where first coffee was produced (called java coffee).
9. Notice that Java is just a name not an acronym.
10. **Originally developed by James Gosling at Sun Microsystems** (which is now a subsidiary of **Oracle Corporation**) and **released in 1995** as core component of Sun Microsystems 'Java platform (Java 1.0 [J2SE]).
11. In 1995, Time magazine called Java one of the Ten Best Products of 1995.
12. With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: **J2EE for Enterprise Applications, J2ME for Mobile Applications.**
13. **Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME, respectively**.
14. Java is guaranteed to be **Write Once, Run Anywhere. (WORA)**
15. Java is an **Object Oriented**. In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

1.1 Properties of Java

➔ The primary objective of Java programming language creation was to make it portable, simple, and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as **Java buzzwords.**

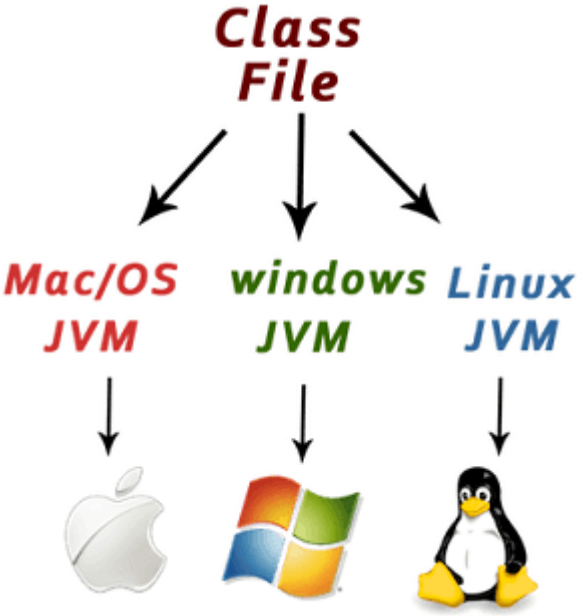
➔ A list of the most important features of the Java language is listed below.

1. **Compiled and Interpreted**

Basically, a computer language is either compiled or interpreted. Java comes together both these approaches thus making Java a two-stage system. Java compiler translates Java code to Bytecode instructions and Java Interpreter generate machine code that can be directly executed by machine that is running the Java program.

2. **Platform Independent and portable**

Java supports the feature portability. Java programs can be easily moved from one computer system to another and anywhere. Changes and upgrades in operating systems, processors and system resources will not force any alteration in Java programs. This is reason why Java has become a trendy language for programming on Internet which interconnects different kind of systems worldwide. Java certifies portability in two ways. First way is, Java compiler generates the bytecode and that can be executed on any machine. Second way is, size of primitive data types are machine independent.



3. **Object- oriented**

Java is truly object-oriented language. In Java, almost everything is an Object. All program code and data exist in objects and classes. Java comes with an extensive set of classes; organize in packages that can be used in program by Inheritance. The object model in Java is trouble-free and easy to enlarge.

4. **Robust and secure**

Java is a most strong language which provides many securities to make certain reliable code. It is design as garbage –collected language, which helps the programmers virtually from all memory management problems. Java also includes the concept of exception handling, which detain serious errors and reduces all kind of threat of crashing the system. Security is an important feature of Java and this is the strong reason that programmer use this language for programming on Internet. The absence of pointers in Java ensures that programs cannot get right of entry to memory location without proper approval.

5. **Distributed**

Java is called as Distributed language for construct applications on networks which can contribute both data and programs. Java applications can open and access remote objects on Internet easily. That means multiple programmers at multiple remote locations to work together on single task.

6. **Simple and small**

Java is very small and simple language. Java does not use pointer and header files, goto statements, etc. It eliminates operator overloading and multiple inheritance.

7. **Multithreaded and Interactive**

Multithreaded means managing multiple tasks simultaneously. Java maintains multithreaded programs. That means we need not wait for the application to complete one task before starting next task. This feature is helpful for graphic applications.

8. **High performance**

Java performance is very extraordinary for an interpreted language, majorly due to the use of intermediate bytecode. Java architecture is also designed to reduce overheads during runtime. The incorporation of multithreading improves the execution speed of program.

9. **Dynamic and Extensible**

Java is also dynamic language. Java is capable of dynamically linking in new class, libraries, methods and objects. Java can also establish the type of class through the query building it possible to either dynamically link or abort the program, depending on the reply. Java program is support functions written in other language such as C and C++, known as native methods.

1.2 Comparison of java with C++

There are many differences and similarities between the C++ programming language and Java. A list of top differences between C++ and Java are given below:

➔ Java is a true object-oriented language while C++ is basically C with object-oriented extension. That is what exactly the increment operator ++ indicates. C++ has maintained backward compatibility with C. It is therefore possible to write an old style C program and run it successfully under C++.

➔ Java appears to be similar to C++ when we consider only the "extension" part of C++. However, some object-oriented features of C++ make the C++ code extremely difficult to follow and maintain.

➔ Listed below are some major C++ features that were intentionally omitted from Java or significantly modified.

- Java does not support operator overloading.
- Java does not have template classes as in C++.
- Java does not support multiple inheritance of classes. This is accomplished using a new feature called "interface".
- Java does not support global variables. Every variable *and* method is declared within a *class and* forms part of that class.
- Java does not use pointers.
- Java *has* replaced the destructor function with **a *finalize( )*** function.
- There *are* no header files in Java.

Java also adds some *new* features. While C++ is a superset of C, Java is neither a superset nor a subset of C or C++. Java may be considered as a first cousin of C++ *and* a second cousin of C

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.

Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (/** ... */) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.

unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

Note

- Java doesn't support default arguments like C++.
- Java does not support header files like C++. Java uses the import keyword to include different classes and methods.

❖ Java environment

Java environment includes a large number of development tools and hundreds of classes and Methods. The development tools are part of the system known as ***Java Development Kit (JDK)*** and the classes and methods are part of the *Java Standard Library* (JSL), also known as the ***Application Programming Interface (API)***

Java Development Kit

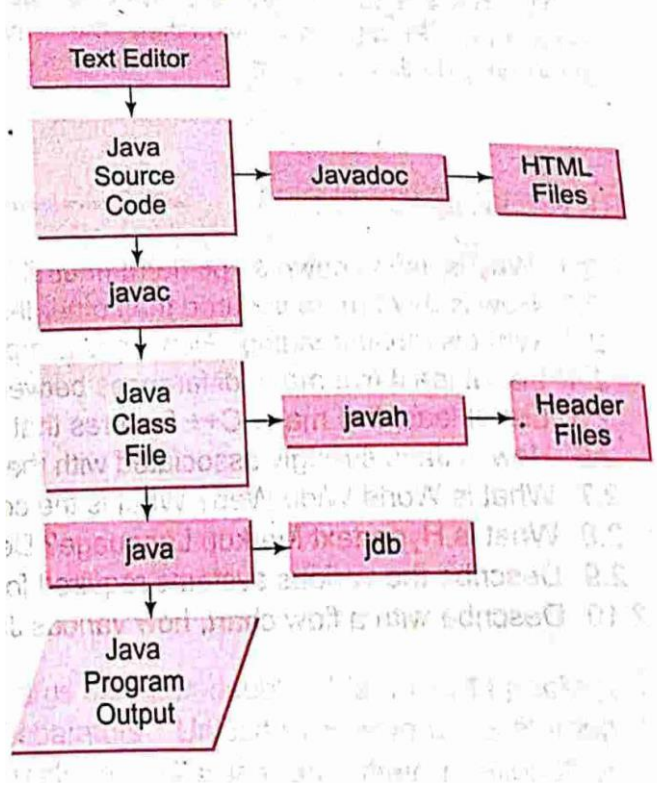
The Java Development Kit comes with a collection of tools that are used for developing and running the Java programs. They include:

- appletviewer (for viewing Java applets)
- javac (Java compiler)
- java (Java interpreter)
- javap (Java disassembler)
- javah (for C header files)
- javadoc (for creating HTML documents)
- jdb (Java debugger)

Java Development Tools

Tool	Description
appletviewer	Enables us to run Java applets (without actually using a Java-compatible browser).
java	Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
javac	The Java compiler, which translates Java sourcecode to bytecode files that the interpreter can understand.
javadoc	Creates HTML-format documentation from Java source code files.
javah	Produces header files for use with native methods.
javap	Java disassembler, which enables us to convert bytecode files into a program description.
jdb	Java debugger, which helps us to find errors in our programs.

Java programming interface



Process of building and running Java application programs

Application Programming Interface
The Java Standard Library (or API) includes hundreds of classes and methods grouped into several functional packages

Most commonly used packages are:

Language Support Package: A collection of classes and methods required for implementing basic features of Java.

Utilities Package: A collection of classes to provide utility functions such as date and time functions.

Input/Output Package: A collection of classes required for input/output manipulation.

Networking Package: A collection of classes for communicating with other computers via Internet_

AWT Package: The Abstract Window Tool Kit package contains classes that implements platform-independent graphical user interface.

Applet Package: This includes a set of classes that allows us to create Java applets.

Java Runtime Environment

The Java Runtime Environment (JRE) facilitates the execution of programs developed in Java. It primarily comprises the following:

Java Virtual Machine (JVM): It is a program that interprets the intermediate Java byte code and generates the desired output. It is because of byte code and JVM concepts that programs written in Java are highly portable.
Runtime class libraries: These are a set of core class libraries that are required for the execution of Java programs.
User interface toolkits: AWT and Swing are examples of toolkits that support varied input methods for the users to interact with the application program.
Deployment technologies: JRE comprises the following key deployment technologies
1.Java plug-in: Enables the execution of a Java applet on the browser.
2.Java Web Start: Enables remote-deployment of an application. With Web Start, users can launch an application directly from the Web browser without going through the installation procedure.

JAVA Bytecode

Why understands bytecode?

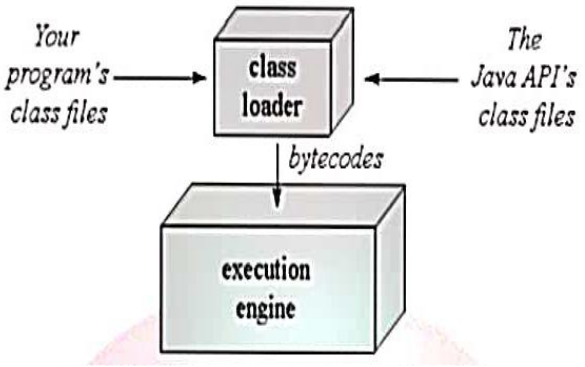
➔ Bytecode is computer object code that is processed by a program, usually referred to as virtual, rather than by the "real" computer machine, the hardware processor. The virtual machine converts each generalized machine instruction into a specific machine instruction or instructions that this computer's processor will understand.

➔ Java code is written in .java file. This code contains one or more Java language attributes like Classes, Methods, Variable, Objects etc. Javac is used to compile this code and to generate .class file. Class file is **also known as "byte code".**

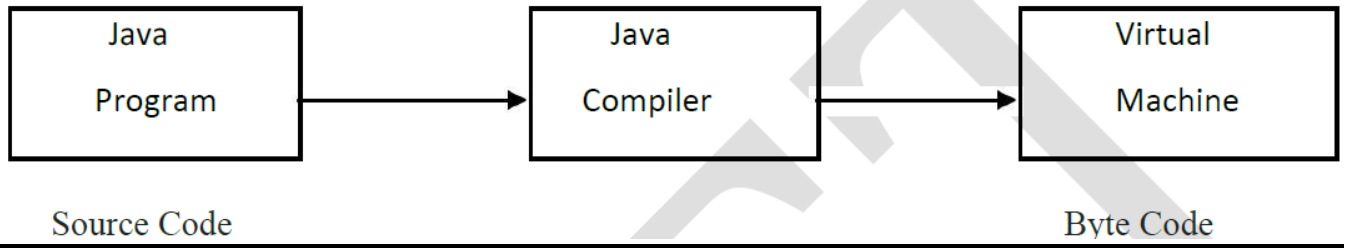
JAVA VIRTUAL MACHINE (JVM)

JVM is the main component of Java architecture and it is **the** part **of** the JRE (Java Runtime Environment). it provides the cross-platform functionality to java. This is a software process that converts the compiled Java byte code to machine code.

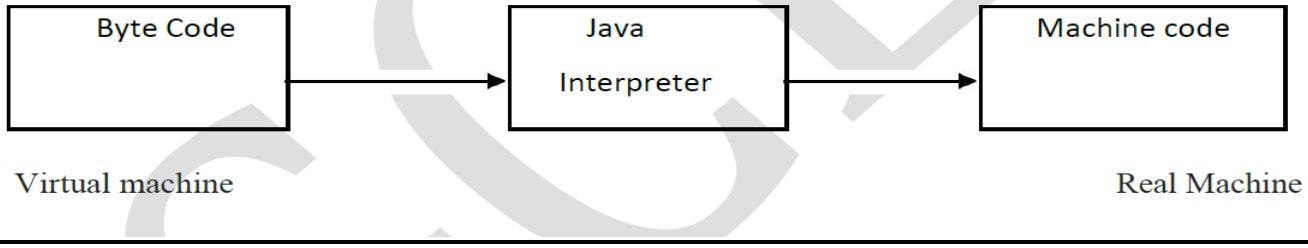
A Java virtual machine's main job is to load class files and execute the bytecodes they contain. Java virtual machine contains a class loader, which loads class files from both the program and the Java API. Only those class files from the Java API that are actually needed by a running program **are** loaded into the virtual machine, The bytecodes are executed in an execution engine.



As we know that all programming language compilers convert the source code to machine code Same job done by Java Compiler to run a Java program, but the difference is that Java compiler convert the source code into Intermediate code is called as bytecode. This machine is called the *Java Virtual machine* and it exits only inside the computer memory.
Following figure shows the process of compilation.



The Virtual machine code is not machine specific. The machine specific code is generated. By Java interpreter by acting as an intermediary between the virtual machine and real machines shown below



1.3 Java Compiler, Java Interpreter

- **javac (Java compiler)**

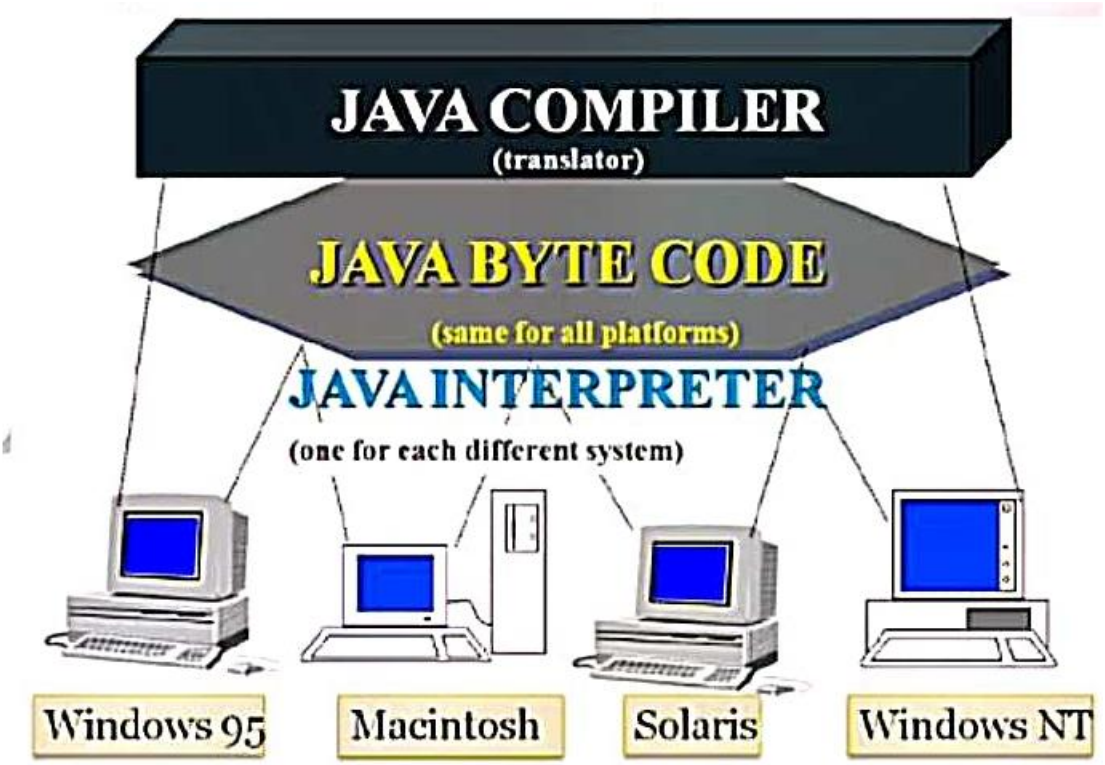
In java, we can use any text editor for writing program and then save that program with .java extension. Java compiler convert the source code or program in bytecode and interpreter convert .java file in .class file.

Syntax:
C:\javac filename.java
If my filename is —abc.java then the syntax will be
C:\javac abc.java

- **java(Java Interpreter)**

As we learn that, we can use any text editor for writing program and then save that program with .java extension. Java compiler convert the source code or program in bytecode and interpreter convert .java file in .class file.

Syntax:
C:\java filename
If my filename is abc.java then the syntax will be
C:\java abc



1.4 Identifier, Literals, Operators, Variables, Keywords, Data Types

JAVA identifier

- Identifier means any legal names of classes, variables, methods , functions etc. when we define names for classes, variables, and methods are called identifiers.
- In Java, followings are rules to define valid identifiers:
  a. All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
  b. After the first character, identifiers can have any combination of characters.
  c. A key word cannot be used as an identifier.
  d. Most importantly, identifiers are case sensitive.

Examples of legal identifiers: age, $charges, _point_1_value
Examples of illegal identifiers: 1a, -ve

Java Literals

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

1. Integer Literals

- Most Commonly used type in typical program is any whole number value is an integer literal.
- For Example: 1, 2, 3 and 25
- Note: byte, short, long literals value is also type as same Integer literals

2. Floating Point Literals

- Floating point number represents decimal values with fractional components,
- For Example: 6.023E23, 3.1415

3. Boolean Literals

- There are only two logical values that a Boolean value can have true or false

4. Character Literals

- Characters in JAVA are indicates into the Unicode character set. Character literal is represent inside a pair of single quote (‘ ‘)
- For Example: ‘x’ ,’y’

5. String Literals

- String literals in JAVA are specified like they are in most other languages by enclosing a sequence of character between a pair of double quotes (" ")
- For Example: "Hello World" , "SYBCA division 3 and 4"

6. The Null Literal

- The null type has one value, the null reference, represented by the literal null, which is formed from ASCII characters. A null literal is always of the null type.
- For Example: null

Operators in Java

It is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

i. **Unary Operators:** Unary operators need only one operand. They are used to increment, decrement or negate a value.

1. **– :Unary minus**, used for negating the values.
2. **+ :Unary plus**, indicates positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is byte, char, or short. This is called unary numeric promotion.
3. **++ :Increment operator**, used for incrementing the value by 1. There are two varieties of increment operator.
   - **Post-Increment:** Value is first used for computing the result and then incremented.
   - **Pre-Increment:** Value is incremented first and then result is computed.
4. **— : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operator.
   - **Post-decrement :** Value is first used for computing the result and then decremented.
   - **Pre-Decrement :** Value is decremented first and then result is computed.
5. **! : Logical not operator**, used for inverting a boolean value.

ii. **Arithmetic Operators:** They are used to perform simple arithmetic operations on primitive data types.

1. *** :** Multiplication
2. **/ :** Division
3. **% :** Modulo
4. **+ :** Addition
5. **– :** Subtraction

iii. **Assignment Operator : '='** Assignment operator is used to assign a value to any variable. It has a right to left associativity, i.e value given on right hand side of operator is assigned to the variable on the left and therefore right hand side value must be declared before using it or should be a constant.
General format of assignment operator is,

variable = value;

- In many cases assignment operator can be combined with other operators to build a shorter version of statement called **Compound Statement**. For example, instead of a = a+5, we can write a += 5.
    - +=, for adding left operand with right operand and then assigning it to variable on the left.
    - -=, for subtracting left operand with right operand and then assigning it to variable on the left.
    - *=, for multiplying left operand with right operand and then assigning it to variable on the left.
    - /=, for dividing left operand with right operand and then assigning it to variable on the left.
    - %=, for assigning modulo of left operand with right operand and then assigning it to variable on the left.

iv. **Relational Operators :** These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements. General format is,

variable **relation_operator** value

Some of the relational operators are-

1. **==, Equal to :** returns true if left hand side is equal to right hand side.
2. **!=, Not Equal to :** returns true if left hand side is not equal to right hand side.
3. **<, less than :** returns true if left hand side is less than right hand side.
4. **<=, less than or equal to :** returns true if left hand side is less than or equal to right hand side.
5. **>, Greater than :** returns true if left hand side is greater than right hand side.
6. **>=, Greater than or equal to:** returns true if left hand side is greater than or equal to right hand side.

v. **Logical Operators :** These operators are used to perform "logical AND" and "logical OR" operation, i.e. the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e. it has a short-circuiting effect. Used extensively to test for several conditions for making a decision.
Conditional operators are-

1. **&&, Logical AND :** returns true when both conditions are true.
2. **||, Logical OR :** returns true if at least one condition is true.

vi. **Ternary operator :** Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary. General format is-

condition ? if true : if false

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

vii. **Bitwise Operators :** These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.

- **&, Bitwise AND operator:** returns bit by bit AND of input values.
- **|, Bitwise OR operator:** returns bit by bit OR of input values.
- **^, Bitwise XOR operator:** returns bit by bit XOR of input values.
- **~, Bitwise Complement Operator:** This is a unary operator which returns the one's complement representation of the input value, i.e. with all bits inversed.

viii. **Shift Operators :** These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two. General format-

number **shift_op** number_of_places_to_shift;

- **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.
- **>>, Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

Precedence and Associativity of Operators

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude with the top representing the highest precedence and bottom shows the lowest precedence.

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ! (type)	Right to left	Unary prefix
/ * %	Left to right	Multiplicative
+ -	Left to right	Additive
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

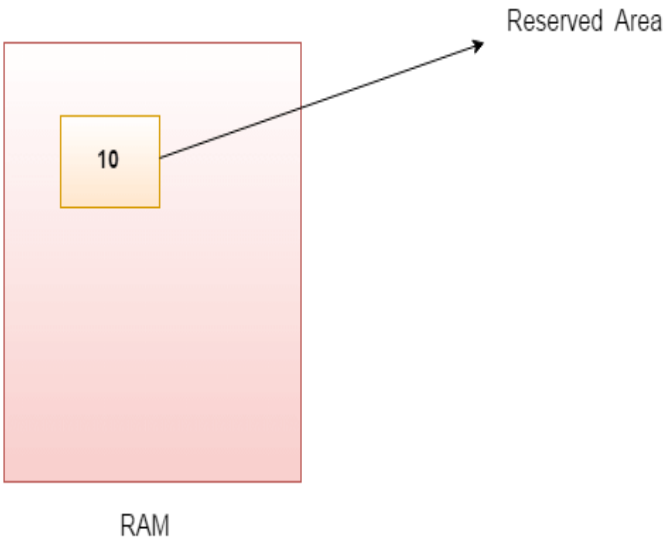
Java Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.




```
int data=50;//Here data is variable
```

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
public class A
{
  static int m=100;//static variable
  void method()
  {
    int n=90;//local variable
  }
  public static void main(String args[])
  {
    int data=50;//instance variable
  } }//end of class
```

Data Types in Java

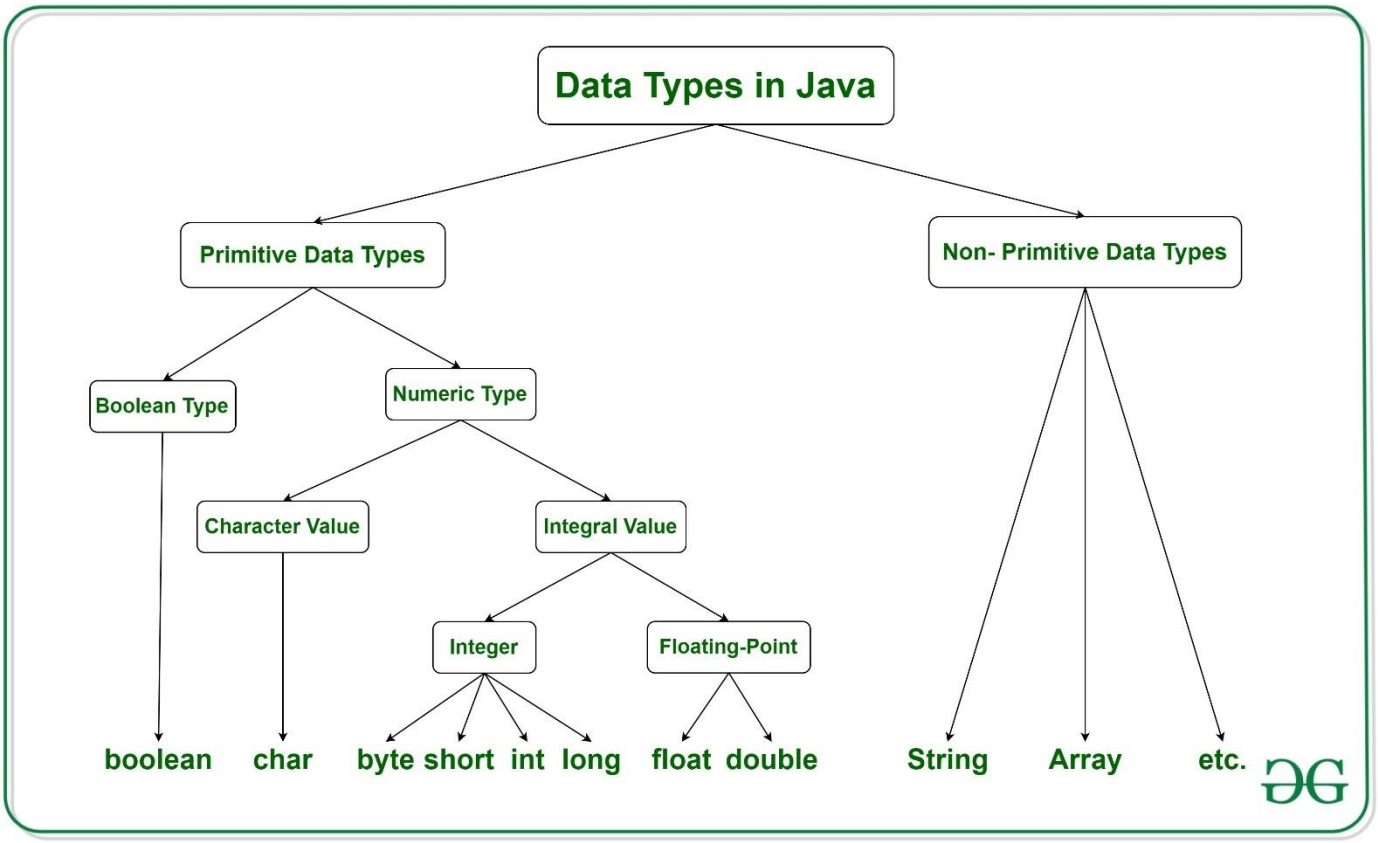
Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

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

Java Primitive Data Types

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

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



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

Java Keywords

- Keywords are reserved words which are used by lava programming language for its own feature and functionality. These keywords cannot be used as an Identifier'
- The keywords const and goto are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.

Keywords In Java

1. abstract
2. assert
3. boolean
4. break
5. byte
6. case
7. catch
8. char
9. class
10. continue
11. default
12. do
13. double
14. else
15. enum
16. extends
17. final
18. finally
19. float
20. for
21. if
22. implements
23. import
24. instanceof
25. int
26. interface
27. long
28. native
29. new
30. package
31. private
32. protected
33. public
34. return
35. short
36. static
37. strictfp
38. super
39. switch
40. synchronized
41. this
42. throw
43. throws
44. transient
45. try
46. void
47. volatile
48. while

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. Jump statements
    - break statement
    - continue statement

1.5 Branching: If – Else, Switch (Decision Making statements)

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

Example:

```
public class IfExample {
public static void main(String[] args) {
   //defining an 'age' variable
   int age=20;
   //checking the age
   if(age>18){
      System.out.print("Age is greater than 18");
   }
}
}
```

Output:

```
Age is greater than 18
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Example:

```
public class IfElseExample {
public static void main(String[] args) {
   //defining a variable
   int number=13;
   //Check if the number is divisible by 2 or not
   if(number%2==0){
      System.out.println("even number");
   }else{
      System.out.println("odd number");
   }
}
}
```

Output:

```
odd number
```

Using Ternary Operator

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

Example:

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

Output:

```
odd number
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Example:

```
public class PositiveNegativeExample {
public static void main(String[] args) {
   int number=-13;
   if(number>0){
   System.out.println("POSITIVE");
   }else if(number<0){
   System.out.println("NEGATIVE");
   }else{
   System.out.println("ZERO");
  }
}
}
```

Output:

```
NEGATIVE
```

Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Example:

```
public class JavaNestedIfExample {
public static void main(String[] args) {
   //Creating two variables for age and weight
   int age=20;
   int weight=80;
   //applying condition on age and weight
   if(age>=18){
      if(weight>50){
         System.out.println("You are eligible to donate blood");
      }
   }
}}
```

Output:

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

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

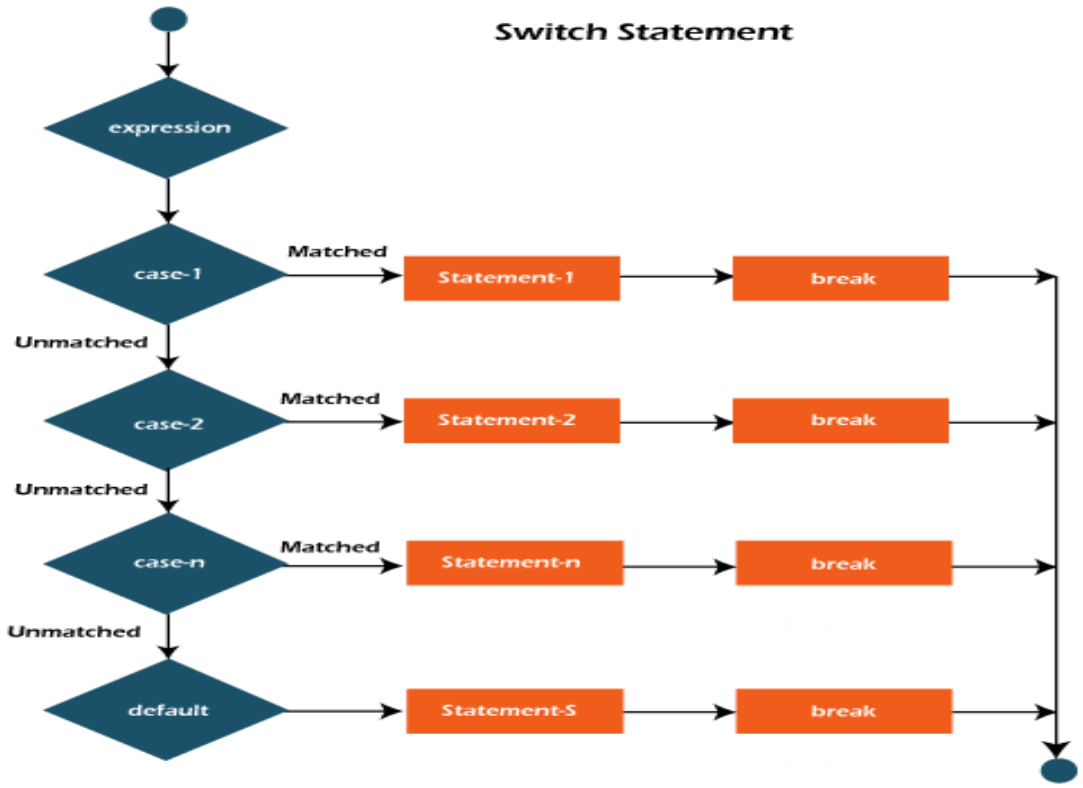
Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
   case value1:
    statement1;
    break;
   .
   .
   .
   case valueN:
    statementN;
    break;
   default:
    default statement;
}
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.



Example:

```
public class SwitchExample {
public static void main(String[] args) {
    //Declaring a variable for switch expression
    int number=20;
    //Switch expression
    switch(number){
    //Case statements
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Output:

```
20
```

Java Nested Switch Statement

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

Example:

```
//Java Program to demonstrate the use of Java Nested Switch
public class NestedSwitchExample {
   public static void main(String args[])
    {
    //C - CSE, E - ECE, M - Mechanical
     char branch = 'C';
     int collegeYear = 4;
     switch( collegeYear )
     {
        case 1:
           System.out.println("English, Maths, Science");
           break;
        case 2:
           switch( branch )
           {
              case 'C':
                 System.out.println("Operating System, Java, Data Structure");
                 break;
              case 'E':
                 System.out.println("Micro processors, Logic switching theory");
                 break;
              case 'M':
                 System.out.println("Drawing, Manufacturing Machines");
                 break;
           }
           break;
        case 3:
           switch( branch )
           {
              case 'C':
                 System.out.println("Computer Organization, MultiMedia");
                 break;
```

```
                case 'E':
                    System.out.println("Fundamentals of Logic Design, Microelectronics");
                    break;
                case 'M':
                    System.out.println("Internal Combustion Engines, Mechanical Vibration");

                    break;
            }
            break;
        case 4:
            switch( branch )
            {
                case 'C':
                    System.out.println("Data Communication and Networks, MultiMedia");
                    break;
                case 'E':
                    System.out.println("Embedded System, Image Processing");
                    break;
                case 'M':
                    System.out.println("Production Technology, Thermal Engineering");
                    break;
            }
            break;
        }
    }
}
```

Output:

```
Data Communication and Networks, MultiMedia
```

1.6 Looping: While, Do-while, For (Loop statements)

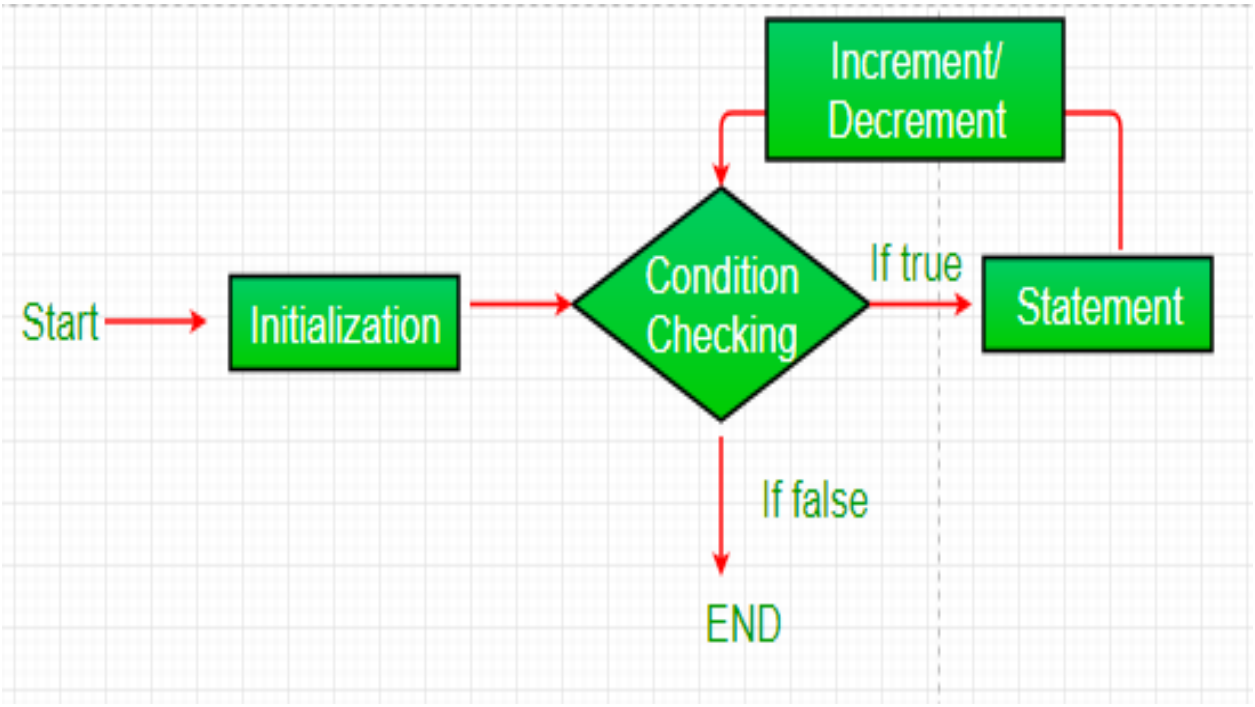
Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true. Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

1. **for loop**: for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

Syntax:

```
for (initialization condition; testing condition;
                        increment/decrement)
{
   statement(s)
}
```

Flowchart:



1. **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
2. **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
3. **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
4. **Increment/ Decrement:** It is used for updating the variable for next iteration.
5. **Loop termination:**When the condition becomes false, the loop terminates marking the end of its life cycle.

Example

//Java Program to demonstrate the example of for loop which prints table of 1

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

Output:

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

Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

Example:

```
public class NestedForExample {
public static void main(String[] args) {
//loop of i
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
      System.out.println(i+" "+j);
}//end of i
}//end of j
}
}
```

Output:

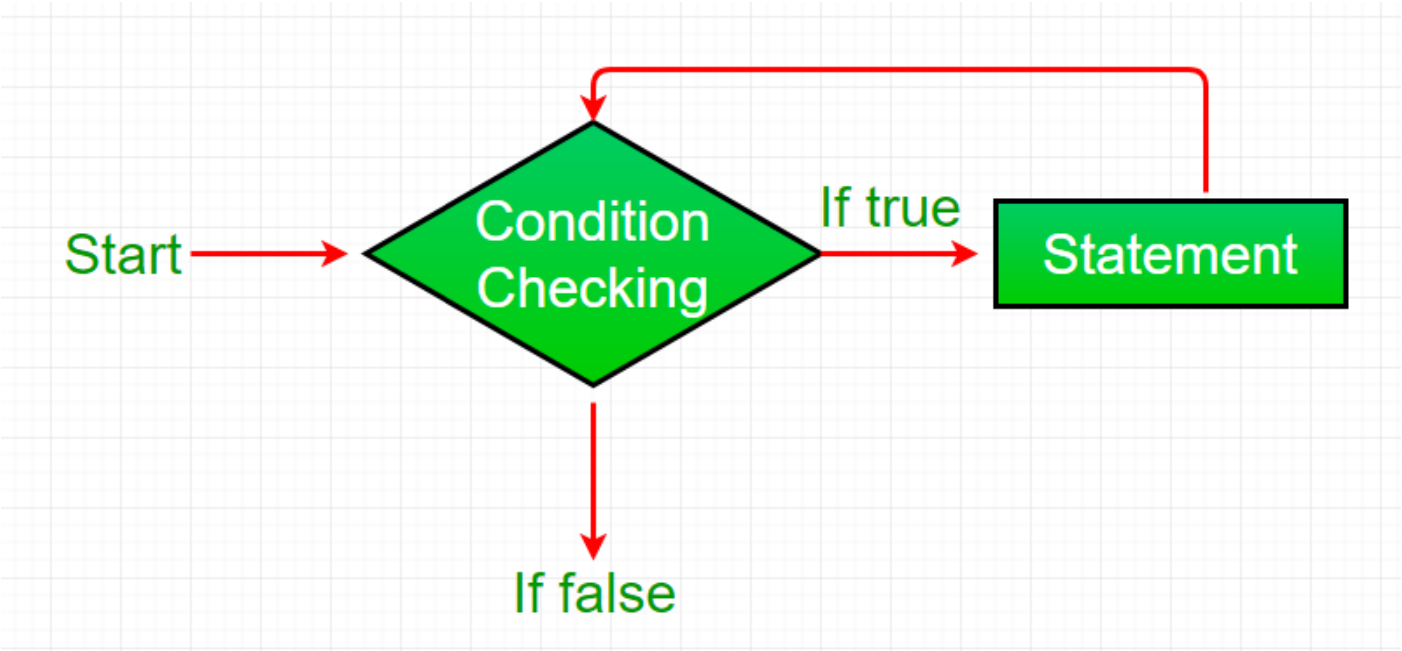
```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

2. **while loop:** A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax :

```
while (boolean condition)
{
  loop statements...
}
```

Flowchart:



- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**
- Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

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

Output:

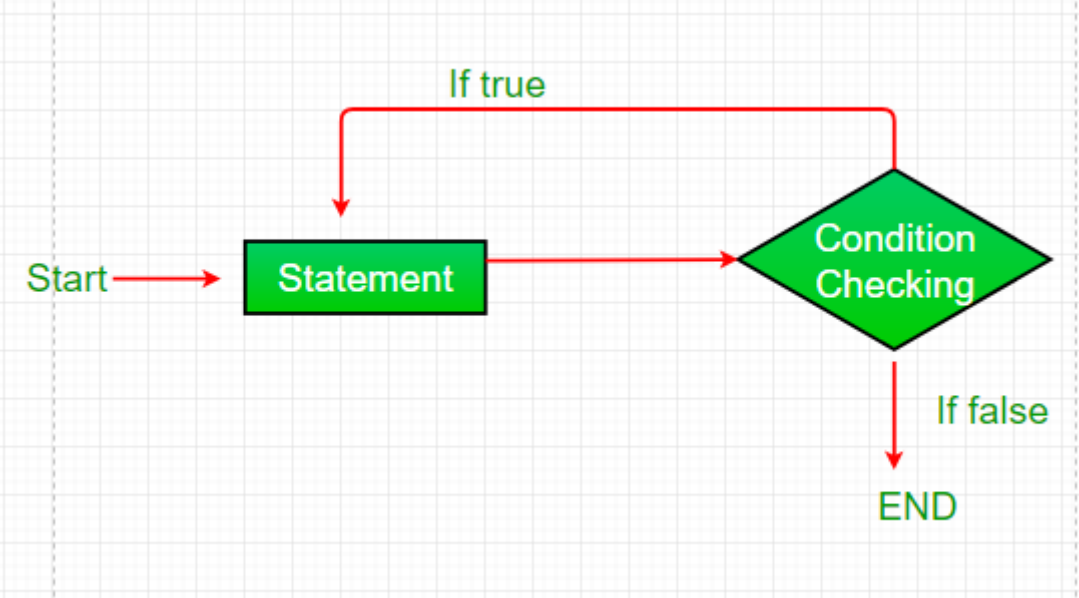
```
2
3
4
5
6
7
8
9
10
```

3. **do while:** do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop.**

Syntax:

```
do
{
   statements..
}
while (condition);
```

Flowchart:



1. do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
2. After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
3. When the condition becomes false, the loop terminates which marks the end of its life cycle.
4. It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

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

Output:

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

Java for-each Loop

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

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

Syntax:

```
for(data_type variable : array_name)
{
    //code to be executed
}
```

Example:

//Java For-each loop example which prints the elements of the array

```
public class ForEachExample {
public static void main(String[] args) {
    //Declaring an array
    int arr[]={12,23,44,56,78};
    //Printing array using for-each loop
    for(int i:arr){
        System.out.println(i);
    }
} }
```

Output:

```
12
23
44
56
78
```

Infinite loop: One of the most common mistakes while implementing any sort of looping is that that it may not ever exit, that is the loop runs for infinite time. This happens when the condition fails for some reason.

`for(;;){`
`//code to be executed`
`}`
	`while(true){`
	`//code to be executed`
	`}`
		`do{`
		`//code to be executed`
		`}while(true);`

Jump statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

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

Output:

```
1
2
3
4
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

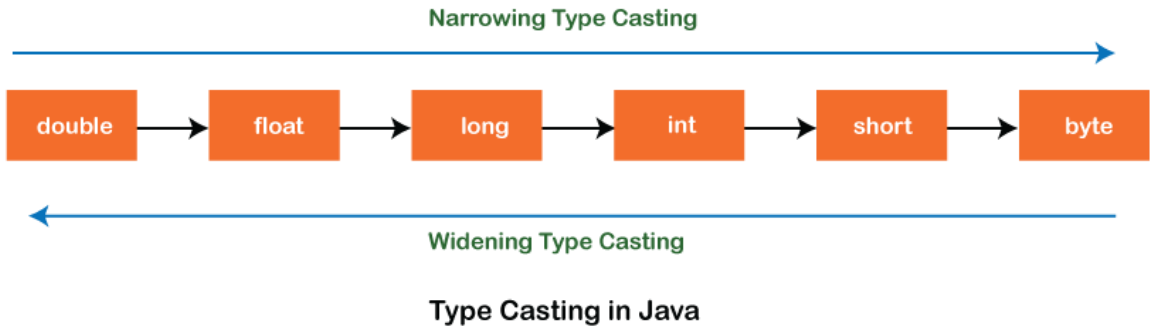
```
/Java Program to demonstrate the use of continue statement inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
   //for loop
   for(int i=1;i<=10;i++){
      if(i==5){
         //using continue statement
         continue;//it will skip the rest statement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

Output:

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

1.7 Type Casting

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.



Type Casting in Java

1. Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

1. **byte** -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

Example:

```
int myInt = 9;
double myDouble = myInt; // Automatic casting: int to double
System.out.println(myInt);      // Outputs 9
System.out.println(myDouble);   // Outputs 9.0
```

2. Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

1. **double** -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

Example:

```
double myDouble = 9.78d;
int myInt = (int) myDouble; // Manual casting: double to int
System.out.println(myDouble);   // Outputs 9.78
System.out.println(myInt);      // Outputs 9
```

Structure of Java Program

Java is an object-oriented programming, **platform-independent,** and **secure** programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications. So, before diving in depth, it is necessary to understand the **basic structure of Java program** in detail. In this section, we have discussed the basic **structure of a Java program**. At the end of this section, you will able to develop the Hello world Java program, easily.

A typical structure of a Java program contains the following elements:

- Documentation Section
- Package Declaration
- Import Statements
- Interface Section
- Class Definition
- Class Variables and Variables
- Main Method Class
- Methods and Behaviours

Documentation Section

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. To write the statements in the documentation section, we use **comments**. The comments may be **single-line, multi-line,** and **documentation** comments.

Single-line Comment: It starts with a pair of forwarding slash **(//)**. For example:

```
//First Java Program
```

Multi-line Comment: It starts with a **/*** and ends with ***/.** We write between these two symbols. For example:

```
/*It is an example of
multiline comment*/
```

Documentation Comment: It starts with the delimiter **(/**)** and ends with ***/**. For example:

```
/**It is an example of documentation comment*/
```

Package Declaration

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. Note that there can be **only one package** statement in a Java program. It must be defined before any class and interface declaration. It is necessary because a Java class can be placed in different packages and directories based on the module they are used. For all these classes package belongs to a single parent directory. We use the keyword **package** to declare the package name. For example:

```
package mypack; //where mypack is the package name
package com.mypack; //where com is the root directory and mypack is the subd irectory
```

Import Statements

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the **import** keyword to import the class. It is written before the class declaration and after the package statement. We use the import statement in two ways, either import a specific class or import all classes of a particular package. In a Java program, we can use multiple import statements. For example:

```
import java.util.Scanner; //it imports the Scanner class only
import java.util.*; //it imports all the class of the java.util package
```

Interface Section

It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An interface is a slightly different from the class. It contains only **constants** and **method** declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the **implements** keyword. An interface can also be used with other interfaces by using the **extends** keyword. For example:

```
interface car
{
void start();
void stop();
}
```

Class Definition

In this section, we define the class. It is **vital** part of a Java program. Without the class, we cannot create any Java program. A Java program may conation more than one class definition. We use the **class** keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the main() method. For example:

```
class Student //class definition
{
}
```

Class Variables and Constants

In this section, we define variables and **constants** that are to be used later in the program. In a Java program, the variables and constants are defined just after the class definition. The variables and constants store values of the parameters. It is used during the execution of the program. We can also decide and define the scope of variables by using the modifiers. It defines the life of the variables. For example:

```
class Student //class definition
{
String sname;  //variable
int id;
double percentage;
}
```

Main Method Class

In this section, we define the **main() method.** It is essential for all Java programs. Because the execution of all Java programs starts from the main() method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods. We use the following statement to define the main() method:

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

For example:

```
public class Student //class definition
{
public static void main(String args[])
{
//statements
}
}
```

Methods and behaviour

In this section, we define the functionality of the program by using the methods. The methods are the set of instructions that we want to perform. These instructions execute at runtime and perform the specified task. For example:

```
public class Demo //class definition
{
public static void main(String args[])
{
void display()
{
System.out.println("Hello World…..!");
}
//statements
}
}
```

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity** − Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.

- **Class Names** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

  Example: *class MyFirstJavaClass*

- **Method Names** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

  Example: *public void myMethodName()*

- **Program File Name** − Name of the program file should exactly match the class name.

  When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

  But please make a note that in case you do not have a public class present in the file then file name can be different than class name. It is also not mandatory to have a public class in the file.

  Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*

- **public static void main(String args[])** − Java program processing starts from the main() method which is a mandatory part of every Java program.

First Java Program | Hello World Example

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, the following software or application must be properly installed.

- Install the JDK if you don't have installed it, 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:

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

Save the above file as Simple.java.

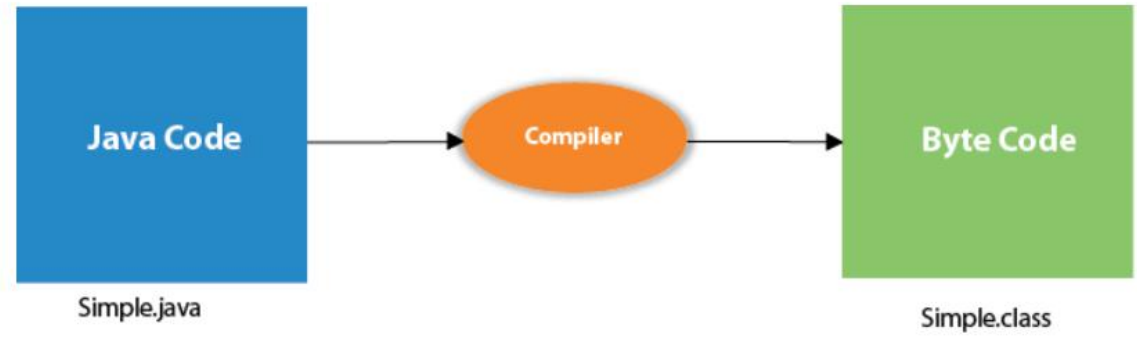
To compile:	javac Simple.java
To execute:	java Simple

Output:

```
Hello Java
```

Compilation Flow:

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



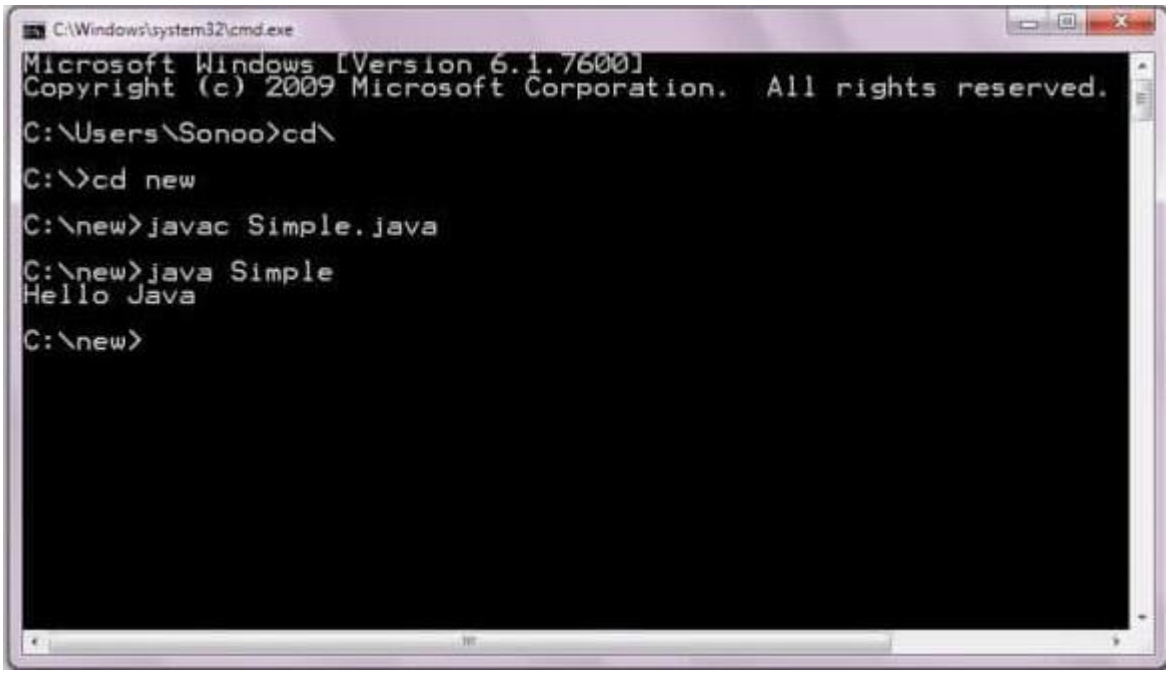
Parameters used in First Java Program

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 that 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 creating 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** or **String args[]** is used for command line argument
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class.

As displayed in the above diagram, write the simple program of Java in notepad and saved it as Simple.java. In order to compile and run the above program, you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt**. When we have done with all the steps properly, it shows the following output:



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

In how many ways we can 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.

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

2) The subscript notation in the 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.

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

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

Let's see the simple code.

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

Valid Java main() method signature

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

Invalid Java main() method signature

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

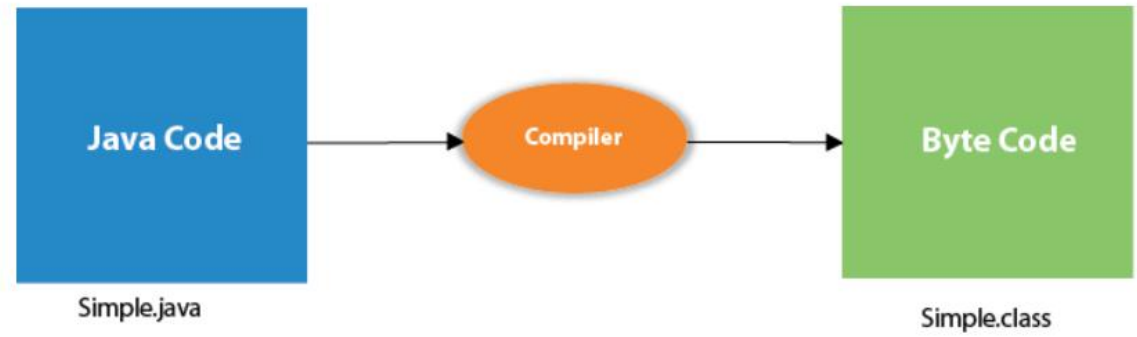
Resolving an error "javac is not recognized as an internal or external command"?

If there occurs a problem like displayed in the below figure, you need to set a path. Since DOS doesn't recognize javac and java as internal or external command. To overcome this problem, we need to set a path. The path is not required in a case where you save your program inside the JDK/bin directory. However, it is an excellent approach to set the path.

- **Internal Details of Hello Java Program**

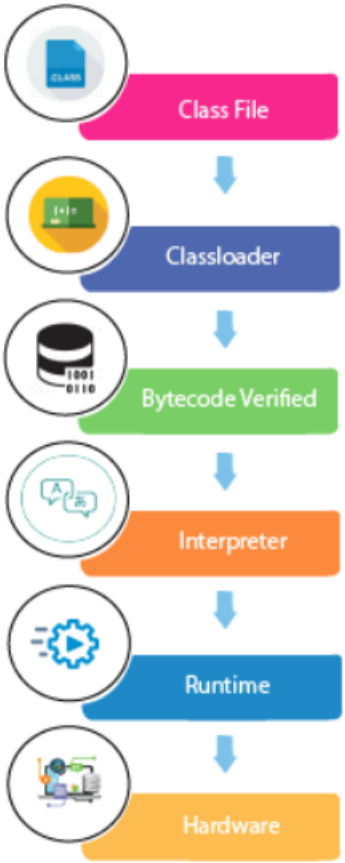
What happens at compile time?

At compile time, the Java file is compiled by Java Compiler (It does not interact with OS) and converts the Java code into bytecode.



What happens at runtime?

At runtime, the following steps are performed:



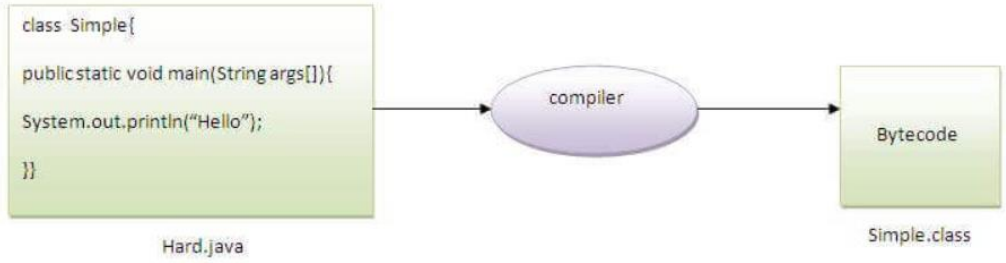
Classloader: It is the subsystem of JVM that is used to load class files.

Bytecode Verifier: Checks the code fragments for illegal code that can violate access rights to objects.

Interpreter: Read bytecode stream then execute the instructions.

Q) Can you save a Java source file by another name than the class name?

Yes, if the class is not public. It is explained in the figure given below:

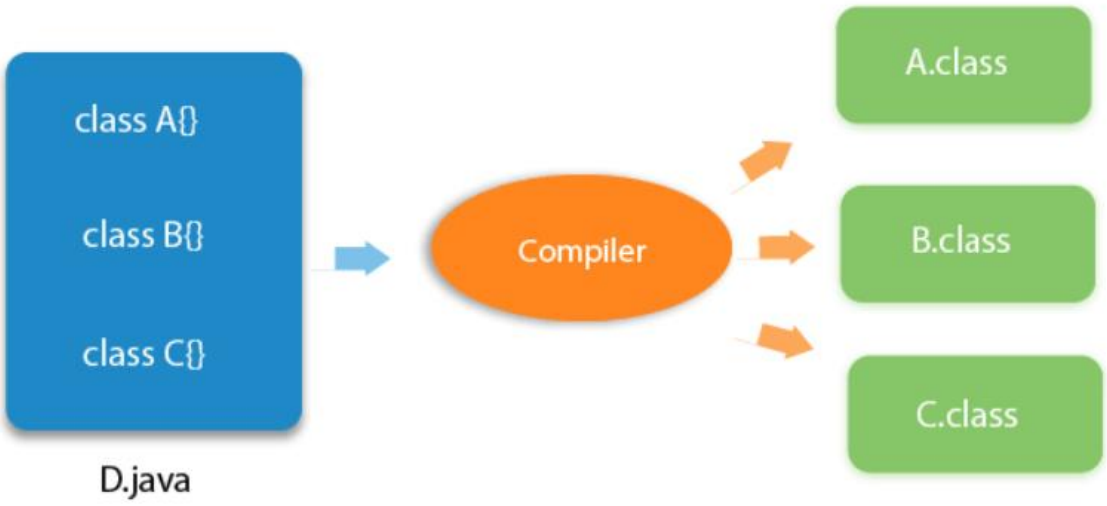


To compile:	javac Hard.java
To execute:	java Simple

Observe that, we have compiled the code with file name but running the program with class name. Therefore, we can save a Java program other than class name.

Q) Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:



How to set path in Java

The path is required to be set for using tools such as javac, java, etc.

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

1) How to set the Temporary Path of JDK in Windows

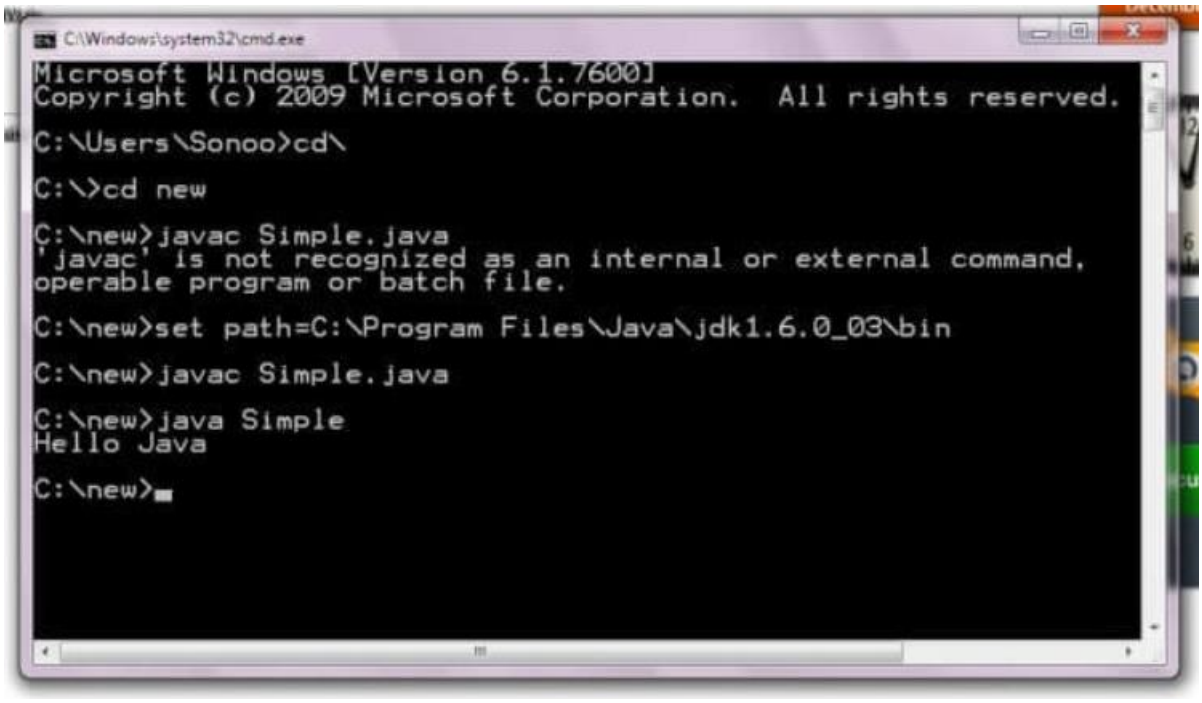
To set the temporary path of JDK, you need to follow the following steps:

- Open the command prompt
- Copy the path of the JDK/bin directory
- Write in command prompt: set path=copied_path

For Example:

set path=C:\Program Files\Java\jdk1.6.0_23\bin

Let's see it in the figure given below:



2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

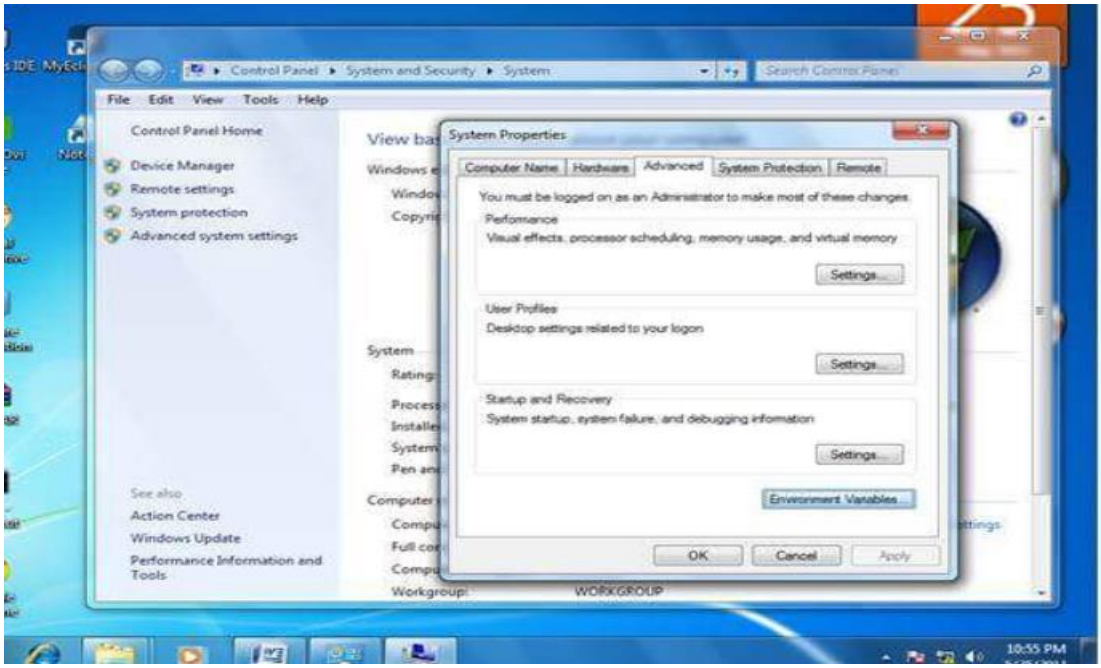
- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

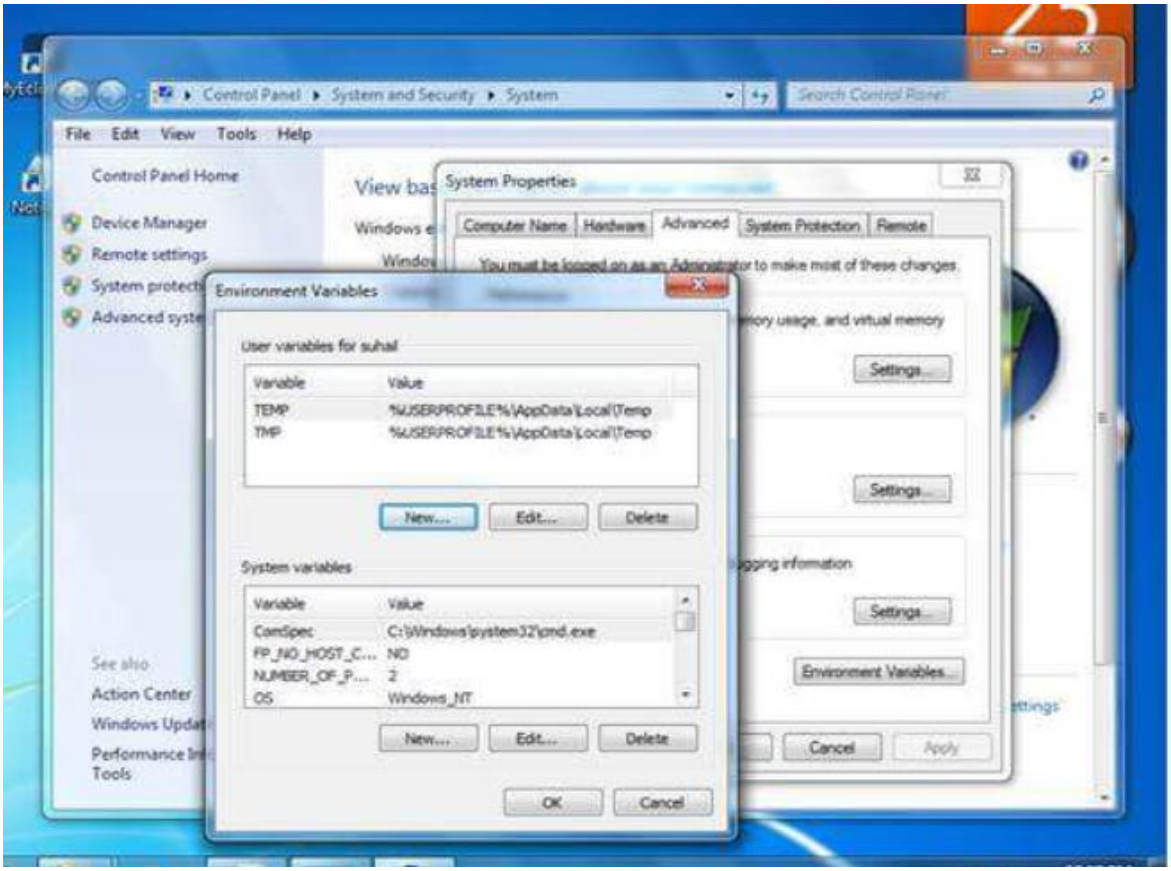
1) Go to MyComputer properties

2) Click on the advanced tab

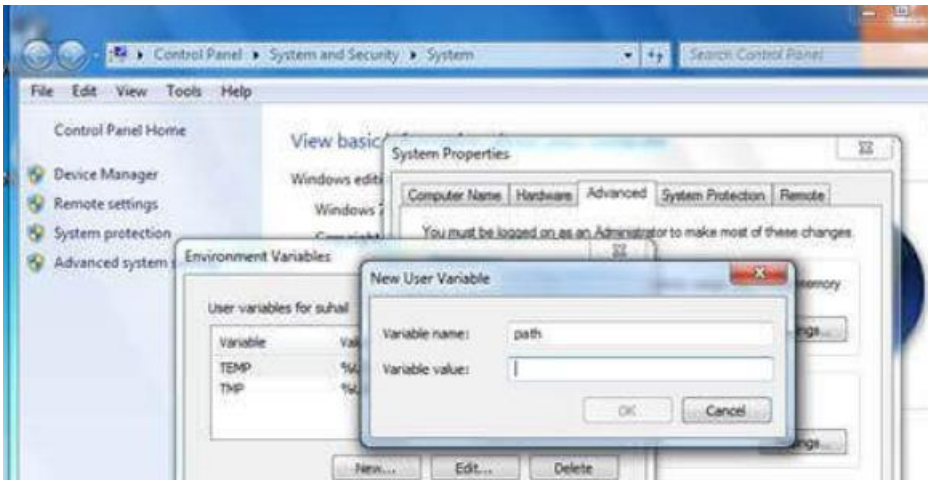
3) Click on environment variables



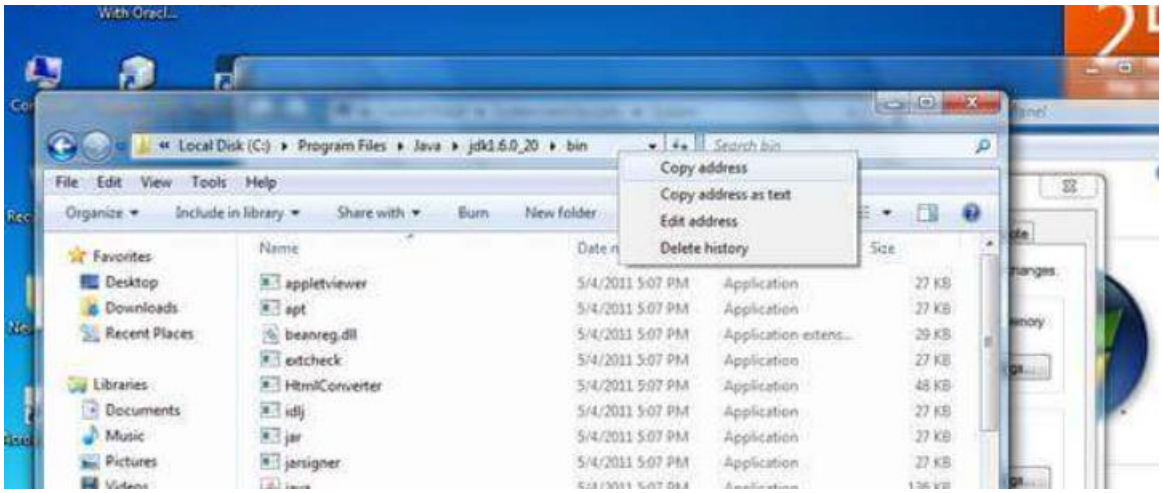
4) Click on the new tab of user variables



5) Write the path in the variable name



6) Copy the path of bin folder



7) Paste path of bin folder in the variable value

8) Click on ok button

9) Click on ok button

Now your permanent path is set. You can now execute any program of java from any drive.

How to Set CLASSPATH in Java

CLASSPATH: CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files. The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH.

You need to set the CLASSPATH if:

- You need to load a class that is not present in the current directory or any sub-directories.
- You need to load a class that is not in a location specified by the extensions mechanism.

The CLASSPATH depends on what you are setting the CLASSPATH. The CLASSPATH has a directory name or file name at the end. The following points describe what should be the end of the CLASSPATH.

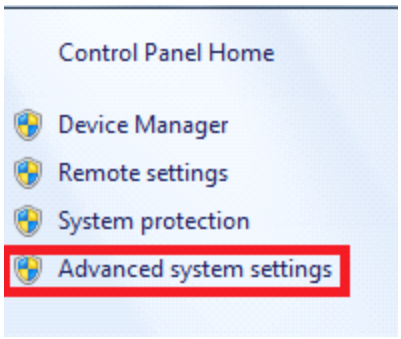
- If a JAR or zip, the file contains class files, the CLASSPATH end with the name of the zip or JAR file.
- If class files placed in an unnamed package, the CLASSPATH ends with the directory that contains the class files.
- If class files placed in a named package, the CLASSPATH ends with the directory that contains the root package in the full package name, that is the first package in the full package name.

If CLASSPATH finds a class file which is present in the current directory, then it will load the class and use it, irrespective of the same name class presents in another directory which is also included in the CLASSPATH.

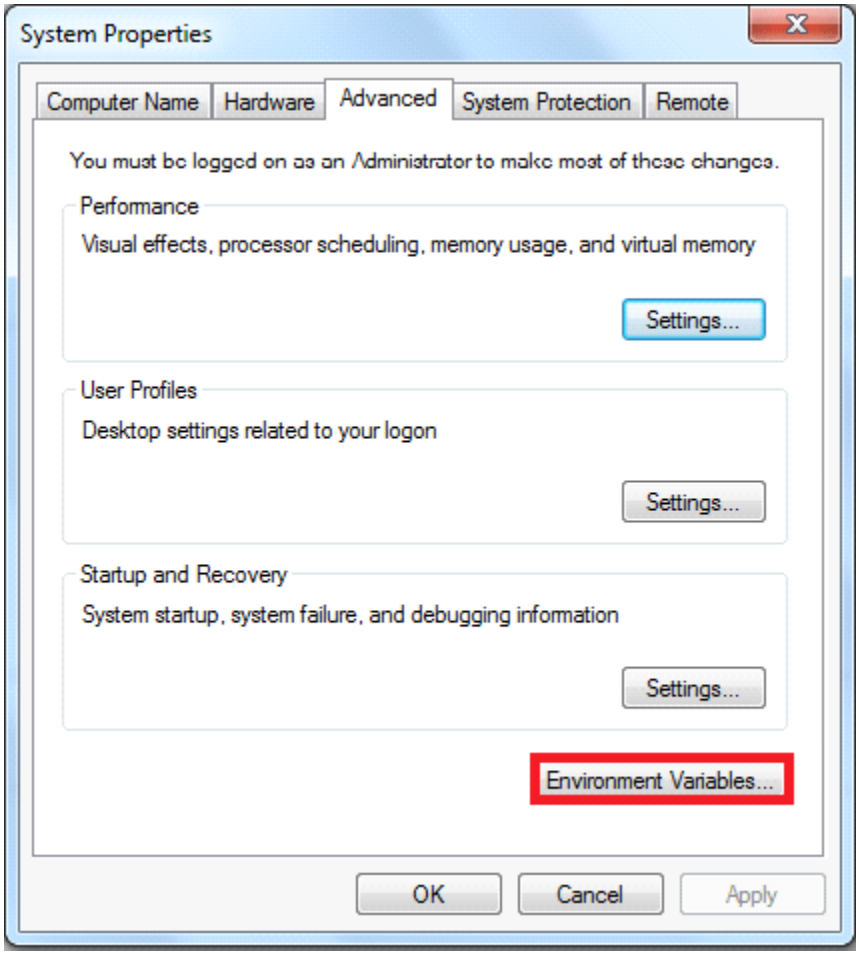
If you want to set multiple classpaths, then you need to separate each CLASSPATH by a semicolon (;).

Step 1: Click on the Windows button and choose Control Panel. Select System.

Step 2: Click on **Advanced System Settings**.



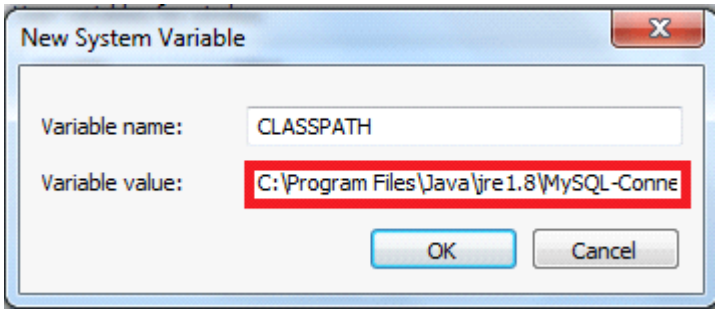
Step 3: A dialog box will open. Click on Environment Variables.



Step 4: If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

f the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as *C:\Program Files\Java\jre1.8\MySQL-Connector Java.jar;.;*

Remember: Put ;.; at the end of the CLASSPATH.



Difference between PATH and CLASSPATH

PATH	CLASSPATH
PATH is an environment variable.	CLASSPATH is also an environment variable.
It is used by the operating system to find the executable files (.exe).	It is used by Application ClassLoader to locate the .class file.
You are required to include the directory which contains .exe files.	You are required to include all the directories which contain .class and JAR files.
PATH environment variable once set, cannot be overridden.	The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command.

Difference between JDK, JRE, and JVM

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

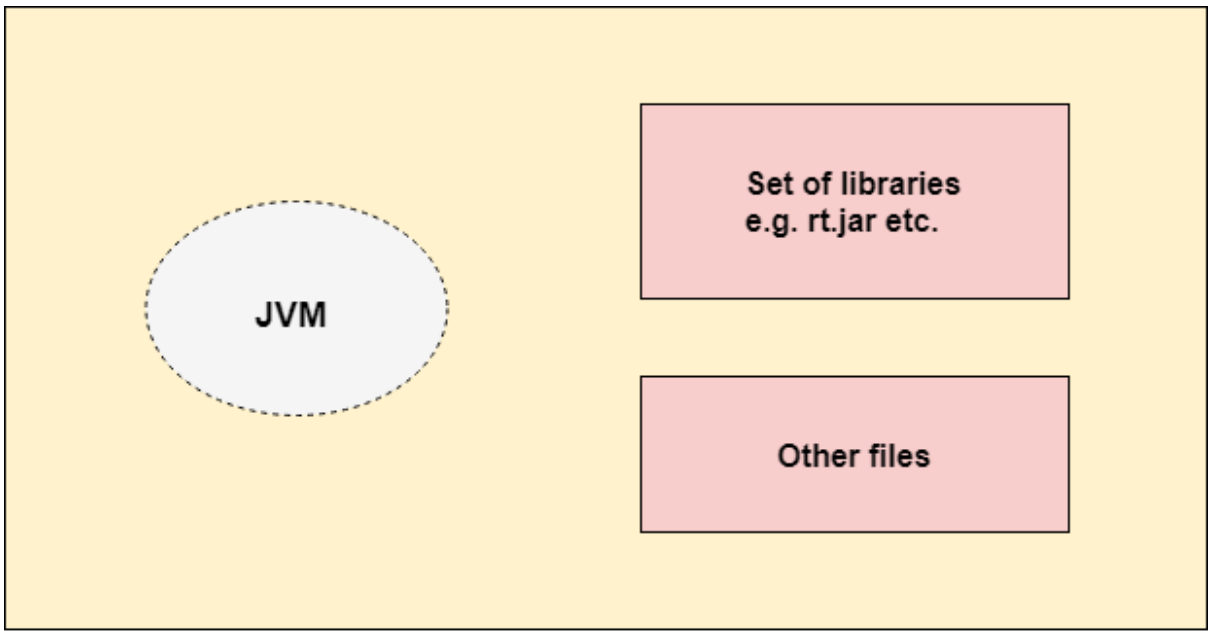
The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JRE

JDK

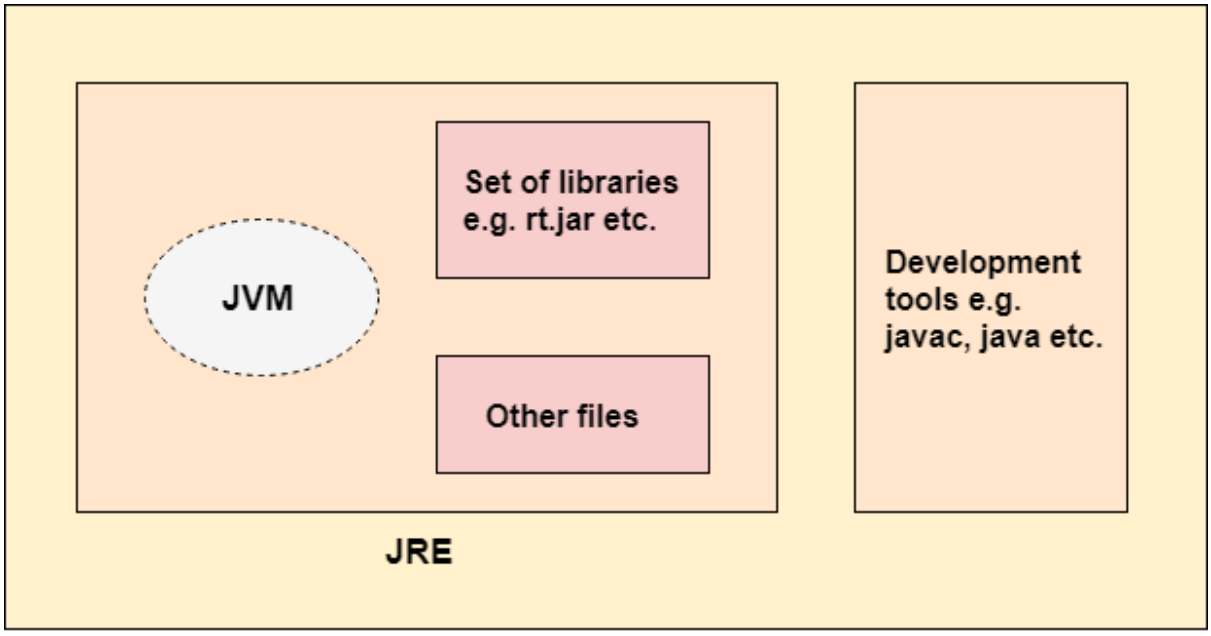
JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets

It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



JDK

Java Arrays

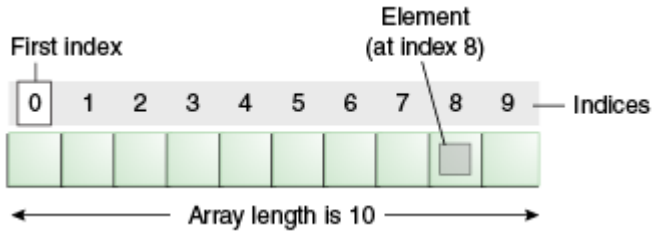
Normally, an array is a collection of similar type of elements which has contiguous memory location.**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar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

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

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

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

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



Advantages

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

Disadvantages

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

Types of Arrays in java

There are two types of arrays.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

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

Instantiation of an Array in Java

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

Example: (Method 1)

```
int a[]=new int[5];//declaration and instantiation

a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;

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

```
10
20
70
40
50
```

Example: (Method 2)

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

//printing array

for(int i=0;i<a.length;i++)//length is the property of array

{

System.out.println(a[i]);  }
```

```
33
3
4
5
```

For-each Loop for Java Array

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

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

```
for(data_type variable:array)
{
//body of the loop
}
```

Example:

```
int arr[]={33,34,35};

//printing array using for-each loop

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

```
33
34
35
```

Anonymous Array in Java

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

Example:

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

public class Array3
{

//creating a method which receives an array as a parameter

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

```
public static void main(String args[])
{
printArray(new int[]{10,22,44,66});//passing anonymous array to method
}

}
```

```
10
22
44
66
```

Multidimensional Array in Java

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

Syntax to Declare Multidimensional Array in Java

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

Example to instantiate Multidimensional Array in Java

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

Example to initialize Multidimensional Array in Java

1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;

Example:

```
//Java Program to illustrate the use of multidimensional array

public class Array5
{

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

```
{

//declaring and initializing 2D array
int arr[][]={{1,2,4},{2,4,6},{4,4,7}};

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

}
}
```

```
1 2 4
2 4 6
4 4 7
```

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

Example:

```
//Java Program to illustrate the jagged array

public class Array6
{

public static void main(String[] args)
{
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];

        //initializing a jagged array
        int count = 0;

        for (int i=0; i<arr.length; i++)
            {
        for(int j=0; j<arr[i].length; j++)
            {
                arr[i][j] = count++;
            }
            }
```

```
        //printing the data of a jagged array
        for (int i=0; i<arr.length; i++)
           {
        for (int j=0; j<arr[i].length; j++)
           {
              System.out.print(arr[i][j]+" ");
        }
           System.out.println();//new line
        }
    }
}
```

Example: Java Program to demonstrate the way of passing an array to method. And find minimum number from it

```
class Array2
{

//creating a method which receives an array as a parameter

static void min(int arr[])
{

int temp=arr[0];

for(int i=1;i<arr.length;i++)
{
 if(temp>arr[i])
{
  temp=arr[i];
}
}
System.out.println(temp);

}

public static void main(String args[])
{
int a[]={10,05,20,02};//declaring and initializing an array

min(a);//passing array to method
}}
```

Example: Java Program to return an array from the method

```
class Array4
{

//creating method which returns an array

static int[] get()
{
return new int[]{10,30,50,90,60};
}

public static void main(String args[])
{

//calling method which returns an array
int arr[]=get();

//printing the values of an array

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

```
10
30
50
90
60
```

Example: Java Program to demonstrate the case of ArrayIndexOutOfBoundsException in a Java Array.

```
public static void main(String args[])
{
int arr[]={50,60,70,80};

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

```
50
60
70
80
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
        at Array4.main(Array4.java:37)
```

Java Enums

The **Enum in Java** is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

Enums are used to create our own data type like classes. The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more *powerful*. Here, we can define an enum either inside the class or outside the class.

Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Points to remember for Java Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Simple Example of Java Enum

```
class EnumExample1{
//defining the enum inside the class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//main method
public static void main(String[] args) {
//traversing the enum
for (Season s : Season.values())
System.out.println(s);
}}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
```

Example of Java enum where we are using value(), valueOf(), and ordinal() methods of Java enum.

```
class EnumExample1{

//defining enum within class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//creating the main method
public static void main(String[] args) {
//printing all enum
for (Season s : Season.values()){
System.out.println(s);
}
System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());


}}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2
```

Note: Java compiler internally adds values(), valueOf() and ordinal() methods within the enum at compile time. It internally creates a static and final class for the enum.

What is the purpose of the values() method in the enum?

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

What is the purpose of the valueOf() method in the enum?

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

What is the purpose of the ordinal() method in the enum?

The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

Defining Java Enum

The enum can be defined within or outside the class because it is similar to a class. The semicolon (;) at the end of the enum constants are optional. For example:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

Or,

```
enum Season { WINTER, SPRING, SUMMER, FALL; }
```

Both the definitions of Java enum are the same.

Java Enum Example: Defined outside class

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumExample2{
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}}
```

Output:

```
WINTER
```

Java Enum Example: Defined inside class

```
class EnumExample3{
enum Season { WINTER, SPRING, SUMMER, FALL; }//semicolon(;) is optional here
public static void main(String[] args) {
Season s=Season.WINTER;//enum type is required to access WINTER
System.out.println(s);
}}
```

Output:

```
WINTER
```

Java Enum Example: main method inside Enum

If you put main() method inside the enum, you can run the enum directly.

```
enum Season {
WINTER, SPRING, SUMMER, FALL;
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}
}
```

Output:

```
WINTER
```

Initializing specific values to the enum constants

The enum constants have an initial value which starts from 0, 1, 2, 3, and so on. But, we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors, and methods.

Example of specifying initial value to the enum constants

```
class EnumExample4{
enum Season{
WINTER(5), SPRING(10), SUMMER(15), FALL(20);

private int value;
private Season(int value){
this.value=value;
}
}
public static void main(String args[]){
for (Season s : Season.values())
System.out.println(s+" "+s.value);
}}
```

Output:

```
WINTER 5
SPRING 10
SUMMER 15
FALL 20
```

Constructor of enum type is private. If you don't declare private compiler internally creates private constructor.

Can we create the instance of Enum by new keyword?

No, because it contains private constructors only.

Can we have an abstract method in the Enum?

Yes, Of course! we can have abstract methods and can provide the implementation of these methods.

Java Enum in a switch statement

We can apply enum on switch statement as in the given example:

Example of applying Enum on a switch statement

```
class EnumExample5{

enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, S
ATURDAY}

    public static void main(String args[]){
    Day day=Day.MONDAY;
    switch(day){
    case SUNDAY:
     System.out.println("sunday");
     break;
    case MONDAY:
     System.out.println("monday");
     break;
    default:
    System.out.println("other day");
    }
    }}
```

```
monday
```

Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

Syntax:

```
returntype methodname(){
//code to be executed
methodname();//calling same method
}
```

Java Recursion Example 1: Infinite times

```
public class RecursionExample1 {
static void p(){
System.out.println("hello");
p();
}
  public static void main(String[] args) {
p();
}
}
```

Output:

```
hello
hello
...
java.lang.StackOverflowError
```

Java Recursion Example 2: Finite times

```
public class RecursionExample2 {
static int count=0;
static void p(){
count++;
if(count<=5){
System.out.println("hello "+count);
p();}}
```

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

Output:

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

Java Recursion Example 3: Factorial Number

```
public class RecursionExample3 {
   static int factorial(int n){
        if (n == 1)
          return 1;
        else
          return(n * factorial(n-1));
   }
 public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+factorial(5));
} }
```

Output:

```
Factorial of 5 is: 120
```

Working of above program:

```
factorial(5)
  factorial(4)
    factorial(3)
      factorial(2)
        factorial(1)
          return 1
        return 2*1 = 2
      return 3*2 = 6
    return 4*6 = 24
  return 5*24 = 120
```

Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects):

Example

```
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid


ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

The eight classes of the *java.lang* package ae known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Example

```
//Java program to convert primitive into objects

//Autoboxing example of int to Integer

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

//Converting int into Integer
int a=20;
```

```
Integer i=Integer.valueOf(a);//converting int into Integer explicitly

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);

}
}
```

```
20 20 20
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Example

```
//Java program to convert object into primitives
//Unboxing example of Integer to int

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

//Converting Integer to int

Integer a=new Integer(3);

int i=a.intValue();//converting Integer to int explicitly

int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);

}
}
```

```
3 3 3
```

Creating Wrapper Objects

***To* create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:**

Example

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt);
    System.out.println(myDouble);
    System.out.println(myChar);
  }
}
```

Since you're now working with objects, you can use certain methods to get information about the specific object.

For example, the following methods are used to get the value associated with the corresponding wrapper object: intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

This example will output the same result as the example above:

Example

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt.intValue());
    System.out.println(myDouble.doubleValue());
    System.out.println(myChar.charValue());}}
```

Another useful method is the toString() method, which is used to convert wrapper objects to strings.

In the following example, we convert an Integer to a String, and use the length() method of the String class to output the length of the "string":

Example

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 100;
    String myString = myInt.toString();
    System.out.println(myString.length());
  }
}
```

Java I/O

System is a predefined class within java.lang that encapsulates different aspects of the JVM and provides access to the underlying system and out is the *output stream* connected to the console. The println() method displays the message passed to it on a new line. So when joined together a message is passed to the console and appears on a new line.

But what do we mean by *output stream*? To investigate this we need to take a closer look at the System class which contains three predefined fields named err, in and the field we are most familiar with out. Further inspection of these fields shows that they are public static and so can be used without an instance and are of the following types:

Name	Type	Description	Default I/O Device
err	PrintStream	The "standard" error output stream.	Console
in	InputStream	The "standard" input stream.	Keyboard
out	PrintStream	The "standard" output stream.	Console

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and an error** message to the console.

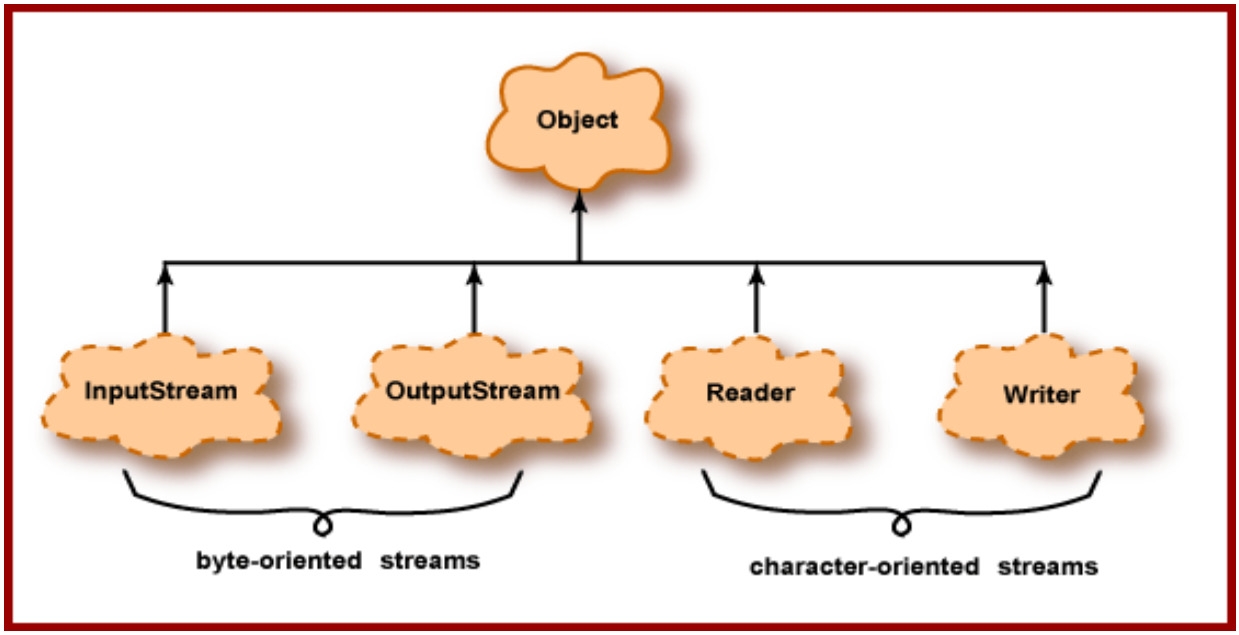
```
1. System.out.println("simple message");
2. System.err.println("error message");
```

So we can now see that these fields are of types InputStream and PrintStream and this is what Java I/O constitutes, class hierarchies of *streams*.

Java is made up of two types of streams, these being *byte streams* used for handling byte input and and output and *character streams* used for handling character input and and output.

The primary reason Java uses streams for I/O is to make Java code independent of the input and output devices involved, thus making the peripheral device used and

thus the coding to use it, transparent.these streams are represented as classes of the java.io package



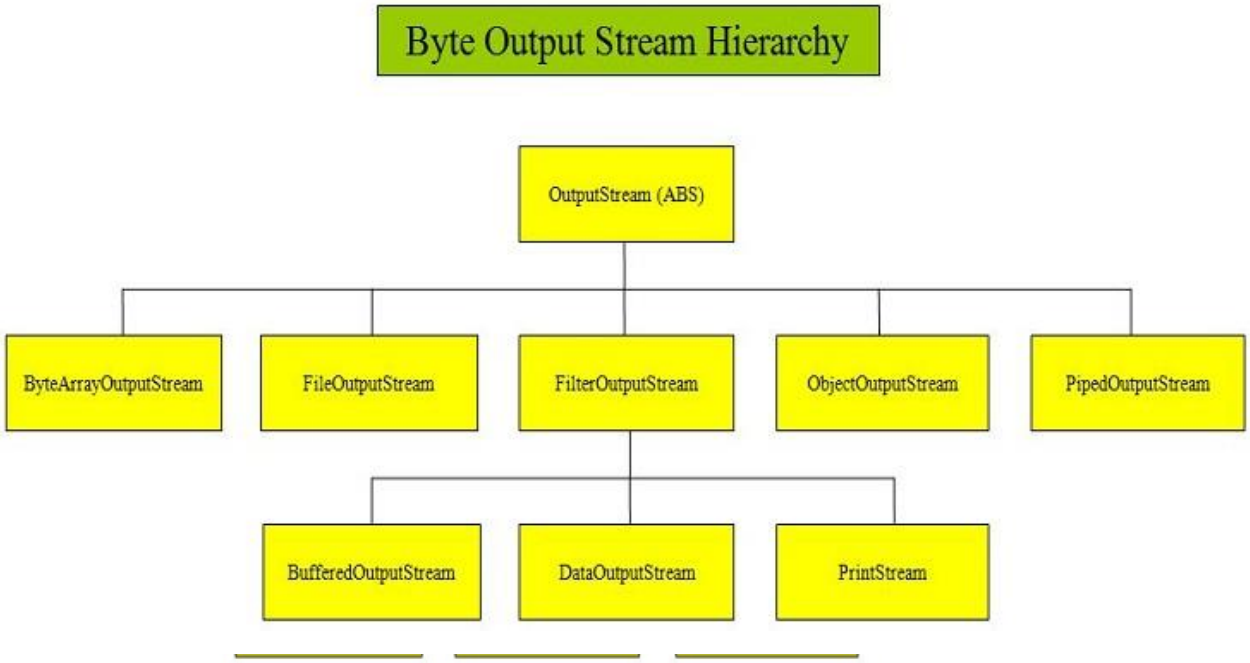
1. Byte Streams

Byte streams are defined within two class hierarchies, one for input and one for output and represent byte stream classes which provide the tools to read and write binary data as a sequence of *bytes*.

- The OutputStream class is the *abstract superclass* of all byte output streams
- The InputStream class is the *abstract superclass* of all byte input streams

Byte Output Stream Hierarchy

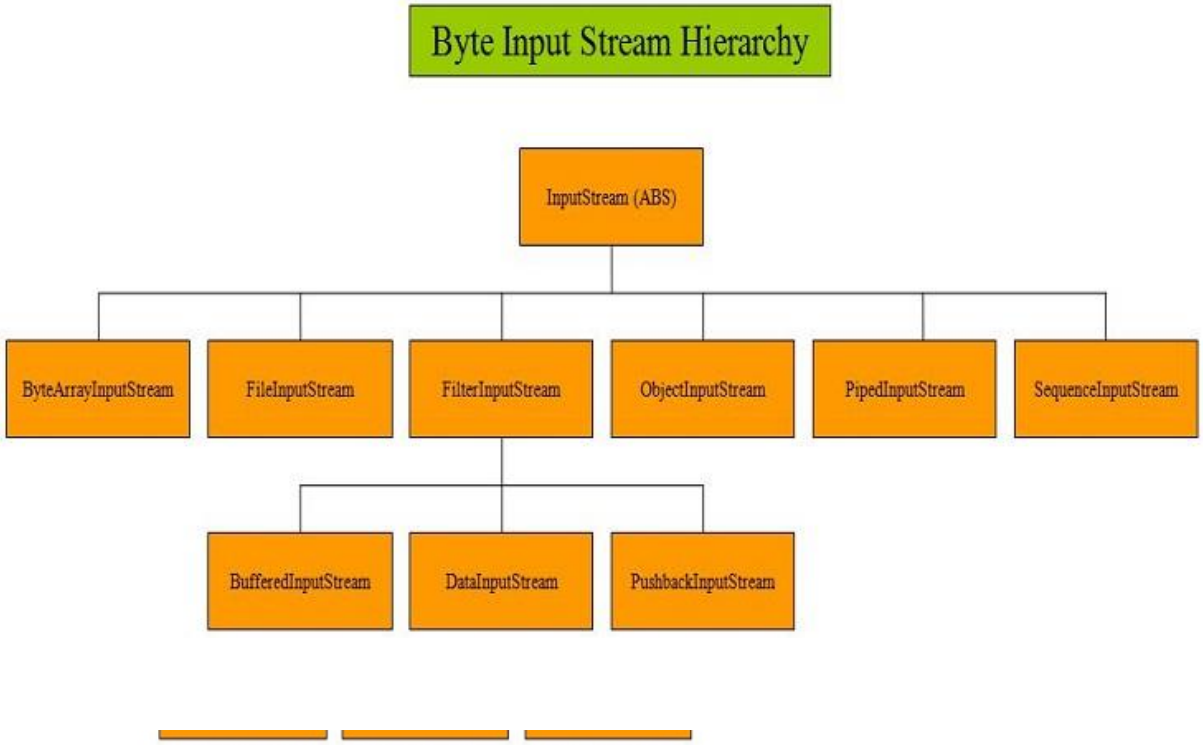
The diagram below shows most of the classes in the *byte output stream hierarchy* of which OutputStream class is the *abstract superclass*. Some subclasses of the FilterOutputStream class are not shown.



Class	Description
`OutputStream`	Abstract byte stream superclass which describes this type of output stream.
`ByteArrayOutputStream`	Output byte stream that writes data to a byte array.
`FileOutputStream`	Output byte stream that writes bytes to a file in a file system.
`FilterOutputStream`	Output byte stream that implements `OutputStream`.
`BufferedOutputStream`	Output byte stream that writes bytes to a buffered output stream.
`DataOutputStream`	Output stream to write Java primitive data types.
`PrintStream`	Convenience output byte stream to add functionality to another stream, an example being to print to the console using `print()` and `println()`.
`ObjectOutputStream`	Output stream to write and serialize objects for reading using `ObjectInputStream`.
`PipedOutputStream`	Piped Output stream that is connected to a piped input stream to create a communication pipe.

Byte Input Stream Hierarchy

The diagram below shows most of the classes in the *byte input stream hierarchy* of which InputStream class is the *abstract superclass*. Some subclasses of the FilterInputStream class are not shown.



Class	Description
`InputStream`	Abstract byte stream superclass which describes this type of input stream.
`ByteArrayInputStream`	Input byte stream that reads bytes from an internal byte array.
`FileInputStream`	Input byte stream that reads bytes from a file in a file system.
`FilterInputStream`	Input byte stream that implements `InputStream`.
`BufferedInputStream`	Input byte stream that reads bytes into an internal buffer before use.
`DataInputStream`	Input stream to reads Java primitive data types.
`PushbackInputStream`	Input byte stream containing functionality to return bytes to the input stream.
`ObjectInputStream`	Input stream to read and deserialize objects output and serialized using `ObjectOutputStream`.
`PipedInputStream`	Piped input stream that is connected to a piped output stream to create a communication pipe.
`SequenceInputStream`	Concatenation of two or more input streams read sequentially.

DataInputStream Class:

Java DataInputStream class allows an application to read primitive data from the input stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

Java DataInputStream class declaration

public class DataInputStream **extends** FilterInputStream **implements** DataInput

Java DataInputStream class Methods

Method	Description
int read(byte[] b)	It is used to read the number of bytes from the input stream.
int read(byte[] b, int off, int len)	It is used to read **len** bytes of data from the input stream.
int readInt()	It is used to read input bytes and return an int value.
byte readByte()	It is used to read and return the one input byte.
char readChar()	It is used to read two input bytes and returns a char value.
double readDouble()	It is used to read eight input bytes and returns a double value.
boolean readBoolean()	It is used to read one input byte and return true if byte is non zero, false if byte is zero.
int skipBytes(int x)	It is used to skip over x bytes of data from the input stream.
String readUTF()	It is used to read a string that has been encoded using the UTF-8 format.
void readFully(byte[] b)	It is used to read bytes from the input stream and store them into the buffer array.

void readFully(byte[] b, int off, int len)	It is used to read **len** bytes from the input stream.

Example: (taking input from user using DataInputStream Class)

```
import java.io.*;

class InputIo1
{
	public static void main(String args[]) throws IOException
	{

	DataInputStream dis=new DataInputStream(System.in);

	System.out.println("ENTER NAME:-");
	String st=dis.readLine();

	System.out.println("ENTER AGE:-");
	int age= Integer.parseInt(dis.readLine());


	System.out.println("welcome "+st);
	System.out.println("your age is "+age);

	}
}
```

```
ENTER NAME:-
khyati
ENTER AGE:-
29
welcome khyati
your age is 29
```

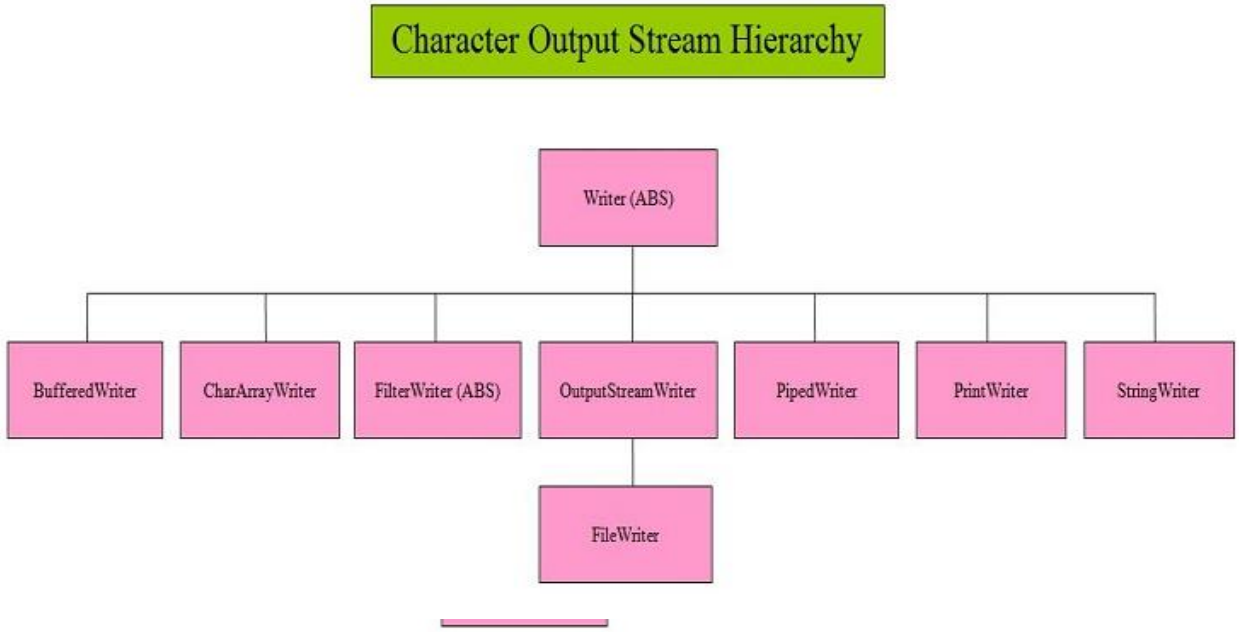
2. Character Stream

Character streams are defined within two class hierarchies, one for input and one for output:

- The Writer class is the *abstract superclass* of all character output streams
- The Reader class is the *abstract superclass* of all character input streams

Character Output Stream Hierarchy

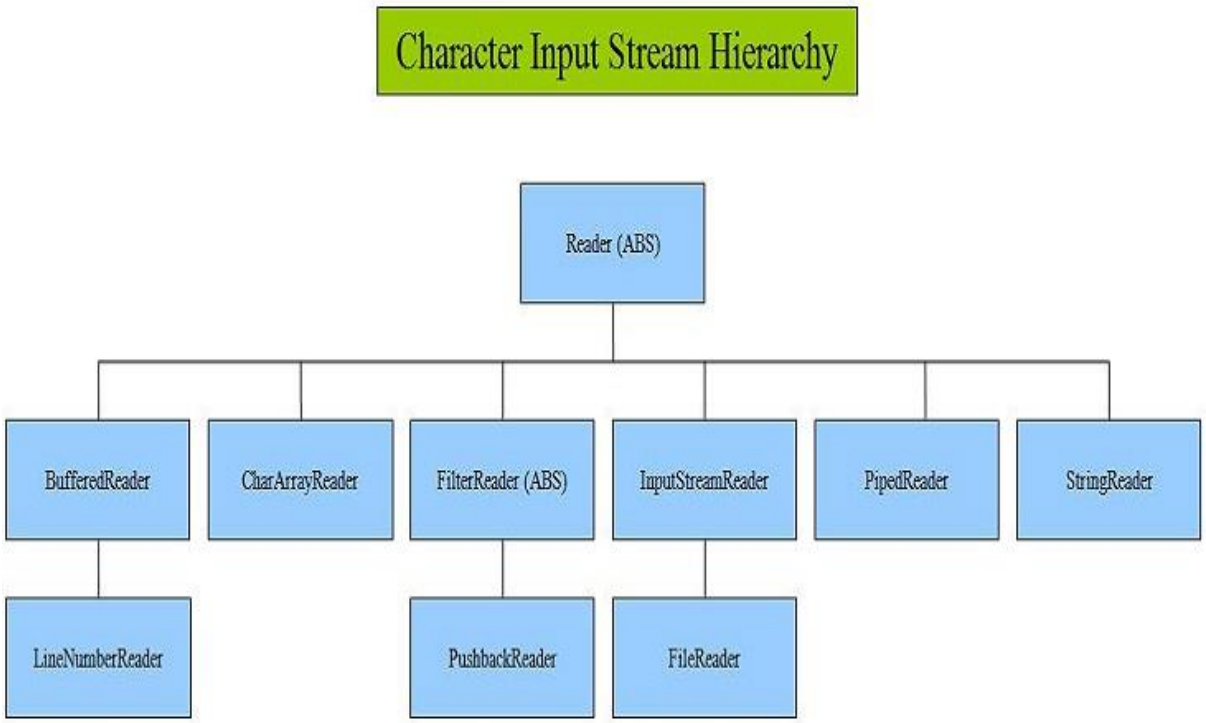
The diagram below shows the classes in the *character output stream hierarchy* of which the Writer class is the *abstract superclass*.:



Class	Description
`Writer`	Abstract character stream superclass which describes this type of output stream.
`BufferedWriter`	Buffered output character stream.
`CharArrayWriter`	Character buffer output stream.
`FilterWriter`	Abstract character stream for writing filtered streams.
`OuputStreamWriter`	Output Stream that acts as a bridge for encoding byte streams from character streams.
`FileWriter`	Output stream for writing characters to a file.
`PipedWriter`	Piped character output stream.
`PrintWriter`	Convenience output character stream to add functionality to another stream, an example being to print to the console using `print()` and `println()`.
`StringWriter`	Output stream for writing characters to a string.

Character Input Stream Hierarchy

The diagram below shows the classes in the *character input stream hierarchy* of which the Reader class is the *abstract superclass*.:



Class	Description
Reader	Abstract character stream superclass which describes this type of input stream.
BufferedReader	Buffered input character stream.
LineNumberReader	Input character stream that keeps a count of line numbers.
CharArrayReader	Character buffer input stream.
FilterReader	Abstract character stream for reading filtered streams.
PushbackReader	Character stream reader containing functionality to return characters to the input stream.
InputStreamReader	Input Stream that acts as a bridge for decoding byte streams into character streams.
FileReader	Input stream for reading characters from a file.
PipedReader	Piped character input stream.
StringReader	Input stream for reading characters from a string.

BufferedReader Class

Java **BufferedReader** class is used to read the text from a character-based input stream. It provides the method **readLine()** to read data line by line. It makes the performance fast. It inherits the Reader class. It is defined in the **java.io** package so, we must import the package at the starting of the program. The disadvantage to use this class is that it is difficult to remember.

To read a number, first, create a constructor of the **BufferedReader** class and parse a **Reader** as a parameter. We have parsed an object of the **InputStreamReader** class. After that, we have invoked the **parseInt()** method of the Integer class and parses the **readLine()** method of the BufferedReader class. The **readLine()** method reads a line of text.

java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.

An InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

Example: (taking input from user using BufferedReader Class)

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Buffer1
{
   public static void main(String[] args)throws IOException
   {

//creating an object of InputStreamReader class

InputStreamReader read=new InputStreamReader(System.in);

//creating a constructor of the BufferedReader class

BufferedReader br=new BufferedReader(read);

      /*
      //Enter data using BufferReader

       BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
*/


   System.out.println("Enter your name");
```

```
        // Reading data using readLine
        String name = br.readLine();

System.out.print("Enter your age: ");

    // Reading data using readLine
    int age=Integer.parseInt(br.readLine());


        // Printing the read line
    System.out.println("Welcome  "+name);
    System.out.println("Age: "+age);

    }
}
```

```
Enter your name
khyati
Enter your age: 29
Welcome  khyati
Age: 29
```

Java Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample
{
public static void main(String args[])
{
System.out.println("Your first argument is: "+args[0]);
}
}
```

1. compile by > javac CommandLineExample.java
2. run by > java CommandLineExample sonoo

```
Your first argument is: sonoo
```

Example 2:

```
class InputCommand1
{
	public static void main(String args[])
	{
		for(int i=0;i<args.length;i++)
		{
			System.out.println("hello " + args[i]);
		}
	}
}
```

```
C:\Users\Admin\OneDrive\Desktop\UAC\java\Khyati_Uac_Java\unit-1\unit1_prog>java InputCommand1 khyati nisha arti
hello khyati
hello nisha
hello arti
```

Example 3:

```
class InputCommand2
{
	public static void main(String args[])
	{
		int x=Integer.parseInt(args[0]);
		int y=Integer.parseInt(args[1]);
		int z=Integer.parseInt(args[2]);
		int ans= x+y+z;
		System.out.print("ans is"+ans);

	}
}
```

```
C:\Users\Admin\OneDrive\Desktop\UAC\java\Khyati_Uac_Java\unit-1\unit1_prog>java InputCommand2 5 7 8
ans is20
```

Java Scanner Class

Java **Scanner class** allows the user to take input from the console. It belongs to **java.util** package. It is used to read the input of primitive types like int, double, long, short, float, and byte. It is the easiest way to read input in Java program.

Syntax

```
1. Scanner sc=new Scanner(System.in);
```

The above statement creates a constructor of the Scanner class having **System.in** as an argument. It means it is going to read from the standard input stream of the program. The **java.util** package should be import while using Scanner class.

It also converts the Bytes (from the input stream) into characters using the platform's default charset.

Methods of Java Scanner Class

Java Scanner class provides the following methods to read different primitives types:

Method	Description
int nextInt()	It is used to scan the next token of the input as an integer.
float nextFloat()	It is used to scan the next token of the input as a float.
double nextDouble()	It is used to scan the next token of the input as a double.
byte nextByte()	It is used to scan the next token of the input as a byte.
String nextLine()	Advances this scanner past the current line.
boolean nextBoolean()	It is used to scan the next token of the input into a boolean value.
long nextLong()	It is used to scan the next token of the input as a long.
short nextShort()	It is used to scan the next token of the input as a Short.
BigInteger nextBigInteger()	It is used to scan the next token of the input as a BigInteger.
BigDecimal nextBigDecimal()	It is used to scan the next token of the input as a BigDecimal.

Example 1:(taking srting input from user using scanner class)

```
import java.util.*;

class InputScanner1
{
public static void main(String[] args)
{
Scanner sc= new Scanner(System.in); //System.in is a standard input stream

System.out.println("Enter a string: ");

String str= sc.nextLine();              //reads string

System.out.println("You have entered: "+str);

}
}
```

```
Enter a string:
hello khyati
You have entered: hello khyati
```

Example 2:(taking integer input from user using scanner class)

```
import java.util.*;

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

Scanner sc= new Scanner(System.in);    //System.in is a standard input stream

System.out.print("Enter first number- ");
int a= sc.nextInt();

System.out.print("Enter second number- ");
int b= sc.nextInt();

System.out.print("Enter third number- ");
int c= sc.nextInt();
```

```
int d=a+b+c;

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

```
Enter first number- 3
Enter second number- 4
Enter third number- 5
Total= 12
```

Why is Scanner skipping nextLine() after use of other next functions?

The nextLine() method of java.util.Scanner class advances this scanner past the current line and returns the input that was skipped. This function prints the rest of the current line, leaving out the line separator at the end. The next is set to after the line separator. Since this method continues to search through the input looking for a line separator, it may search all of the input searching for the line to skip if no line separators are present.

Example:

// Java program to show the issue with nextLine() method of Scanner Class

```
import java.util.Scanner;

public class ScannerProb
{
   public static void main(String[] args)
   {
      // Declare the object and initialize with predefined standard input object

     Scanner sc = new Scanner(System.in);

      // Taking input

      String name = sc.nextLine();
      char gender = sc.next().charAt(0);
      int age = sc.nextInt();
      String fatherName = sc.nextLine();
      String motherName = sc.nextLine();
```

```
        // Print the values to check if the input was correctly obtained.

        System.out.println("Name: " + name);
        System.out.println("Gender: " + gender);
        System.out.println("Age: " + age);
        System.out.println("Father's Name: " + fatherName);
        System.out.println("Mother's Name: " + motherName);
    }
}
```

Expected Output:

Name: abc

Gender: m

Age: 1

Father's Name: xyz

Mother's Name: pqr

Actual Output:

Name: abc

Gender: m

Age: 1

Father's Name:

Mother's Name: xyz

As you can see, the nextLine() method skips the input to be read and takes the name of Mother in the place of Father. Hence the expected output does not match with the actual output. This error is very common and causes a lot of problems.

Why does this issue occur?

This issue occurs because, when nextInt() method of Scanner class is used to read the age of the person, it returns the value 1 to the variable age, as expected. But the cursor, after reading 1, remains just after it.

abc

m

1_ // Cursor is here
xyz
pqr

So when the Father's name is read using nextLine() method of Scanner class, this method starts reading from the cursor's current position. In this case, it will start reading just after 1. So the next line after 1 is just a new line, which is represented by '\n' character. Hence the Father's name is just '\n'.

How to solve this issue?

This issue can be solved by either of the following two ways:

1. reading the complete line for the integer and converting it to an integer, or

Syntax:

// Read the complete line as String

// and convert it to integer

int var = Integer.parseInt(sc.nextLine());

Although this method is not applicable for input String after Byte character(Byte.parseByte(sc.nextLine()). Second method is applicable in that case.

2. by consuming the leftover new line using the nextLine() method.

Syntax:

```
// Read the integer
int var = sc.nextInt();


// Read the leftover new line
sc.nextLine();
```

Problem Solution

```
// Taking input

    String name = sc.nextLine();
    char gender = sc.next().charAt(0);

// Consuming the leftover new line using the nextLine() method

    sc.nextLine();

// reading the complete line for the integer and converting it to an integer

   int age = Integer.parseInt(sc.nextLine());

    String fatherName = sc.nextLine();
    String motherName = sc.nextLine();
```

Scanner Vs. Bufferreader

1. Scanner is a much more powerful utility than BufferedReader. It can parse the user input and read int, short, byte, float, long and double apart from String. On the other hand BufferedReader can only read string in java.

   2. BuffredReader has significantly large buffer (8KB) than Scanner (1KB), which means if you are reading long String from file, you should use BufferedReader but for short input and input other than String, you can use Scanner class.

   3. BufferedReader is older than Scanner. It's present in Java from JDK 1.1

onward but Scanner is only introduced in JDK 1.5 release.

4. Scanner uses regular expression to read and parse text input. It can accept custom delimiter and parse text into primitive data type e.g. int, long, short, float or double using nextInt(), nextLong(), nextShort(), nextFloat(), and nextDouble() methods, while BufferedReader can only read and store String using readLine() method.

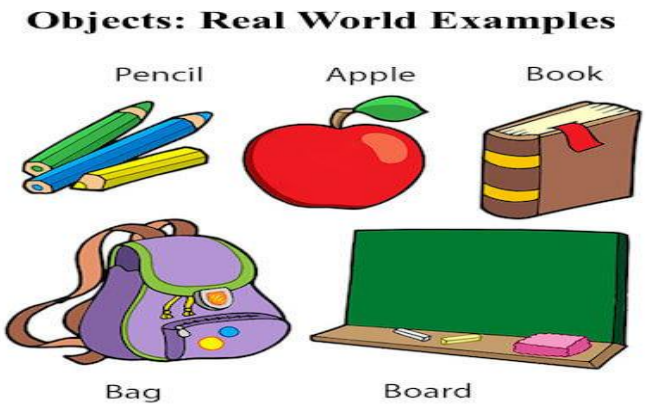
5. Another major difference between BufferedReader and Scanner class is that BufferedReader is **synchronized** while Scanner is not. This means, you cannot share Scanner between multiple threads but you can share the BufferedReader object.

What is Object Oriented programming?

Object oriented programming treats data as a critical element in the program development and does not allow it to flow freely around the system .it ties data more closely to the function that operate on it and protects it from accidental modification.

- In object-oriented programming technique, we design a program using objects and classes.
- An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java



• Objects are the basic runtime entities in object-oriented system. Objects are also called Instance of the class.

• An object represents a person, Bank-account, Vehicle, Book, products, Employee, etc…

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

An object has three characteristics:

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

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

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

Object Definitions:

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

What is a class in Java

• The general meaning of class is category. class is pro-forma or blue-print that represents particular category.

- Class contains data and code that operate on that data It is user defined data type and object are variable of class.

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

A class in Java can contain:

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

Syntax to declare a class:

```
class <class_name>
{
        field;
        method;
}
```

Instance variable in Java

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

Method in Java

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

Advantage of Method

- Code Reusability
- Code Optimization

new keyword in Java

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

Class vs. Object

S. No.	Class	Object
1	Class is used as a template for declaring and creating the objects.	An object is an instance of a class.
2	When a class is created, no memory is allocated.	Objects are allocated memory space whenever they are created.
3	The class has to be declared only once.	An object is created many times as per requirement.
4	A class cannot be manipulated as they are not available in the memory.	Objects can be manipulated.
5	A class is a logical entity.	An object is a physical entity.
6	It is declared with the class keyword	It is created with a class name in C++ and with the new keywords in Java.
7	Class does not contain any values which can be associated with the field.	Each object has its own values, which are associated with it.
8	A class is used to bind data as well as methods together as a single unit.	Objects are like a variable of the class.

Object and Class Example: main within the class

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

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

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Object and Class Example: main outside the class

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

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

File: TestStudent1.java

```
//Java Program to demonstrate having the main method in another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

3 Ways to initialize object (storing data into object)

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor (will discuss in constructor topic)

1) Object and Class Example: Initialization through reference

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

File: TestStudent2.java

```
class Student{
 int id;
 String name;
}
class TestStudent2{
```

```
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="krs";
  System.out.println(s1.id+" "+s1.name);//printing members with a white spa
ce
 }
}
```

Output:

```
101 krs
```

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

File: TestStudent3.java

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="krs";
  s2.id=102;
  s2.name="ajp";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Output:

```
101 krs
102 ajp
```

2) Object and Class Example: Initialization through method

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

File: TestStudent4.java

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Output:

```
111 Karan
222 Aryan
```

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

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

File: TestEmployee.java

```
class Employee{
   int id;
   String name;
   float salary;
   void insert(int i, String n, float s) {
      id=i;
      name=n;
      salary=s;
   }
   void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
   Employee e1=new Employee();
   Employee e2=new Employee();
   Employee e3=new Employee();
   e1.insert(101,"ajeet",45000);
   e2.insert(102,"irfan",25000);
   e3.insert(103,"nakul",55000);
   e1.display();
   e2.display();
   e3.display();
}
}
```

Output:

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

Object and Class Example: Rectangle

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

File: TestRectangle1.java

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Output:

```
55
45
```

Data Encapsulation And Data Abstraction :-

- The wrapping up of data and function into a single unit (called class) is known as **Data encapsulation.**

- Data is not access to the outside word and only those functions which are wrapped in the class can access it. Thus, It provides interface between the object's data and the program. It is also called **"Data hiding" or "Information hiding".**

- **Data abstraction** refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

- "Data Abstraction", as it gives a clear separation between properties of data type and the associated implementation details. There are two types; they are "function abstraction" and "data abstraction". Functions that can be used without knowing how its implemented is function abstraction. Data abstraction is using data without knowing how the data is store

- Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to implement the desired level of abstraction.

- class student

{

public int roll_no;

private char name;

public void read();

}

- here you will see "class" student contains both data members like roll_no and name and member function read().because both data members and factions are included in a single class ,this supports encapsulation and when other classes try to use this class student ,they will not be permitted to see the complete inside of class this feature is called abstraction.

- Encapsulation is implemented by using access specifiers. An access specifier defines the scope and visibility of a class member.

Access Modifiers in Java

The access modifiers in Java specifies 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.

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

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.

```
class A{
private int data=40;
private void msg()
{System.out.println("Hello java");}
}

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

Role of Private Constructor

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

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

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.

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

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

3) Protected

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.

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

```
//save by B.java
package mypack;
import pack.*;

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

```
Output:Hello
```

4) Public

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

Example of public access modifier

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

```
//save by B.java
 package mypack;
import pack.*;
 class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();   }  }
```

```
Output:Hello
```

Java Access Modifiers with Method Overriding

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

```
class A{
protected void msg()
{
System.out.println("Hello java");
}
}

public class Simple extends A{
//default access specifier not supported
void msg()
{
System.out.println("Hello java");
}//C.T.Error

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

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

Constructors in Java

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

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

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

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

There are two types of constructors in Java: no-arg constructor(default), 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

Types of Java constructors

There are two types of constructors in Java:

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

1. Java Default Constructor

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

Syntax of default constructor:

1. <class_name>(){}

Example of default constructor

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

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

Output:

```
Bike is created
```

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

What is the purpose of a default constructor?

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

Example of default constructor that displays the default values

```
//Let us see another example of default constructor
//which displays the default values
class Student3{
int id;
String name;
//method to display the value of id and name
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
} }
```

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.

2. 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.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();  }}
```

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

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;

    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }

    //creating three arg constructor
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();  }}
```

Output:

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

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

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

```
//Java program to initialize the values from one object to another object.
class Student6{
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
    id = i;
```

```
    name = n;
    }
    //constructor to initialize another object
    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

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.

```
class Student7{
    int id;
    String name;
    Student7(int i,String n){
    id = i;
    name = n;
    }
    Student7(){}
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student7 s1 = new Student7(111,"Karan");
    Student7 s2 = new Student7();
```

```
        s2.id=s1.id;
        s2.name=s1.name;
        s1.display();
        s2.display();
       }
    }
```

Output:

```
111 Karan
111 Karan
```

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java?

Yes.

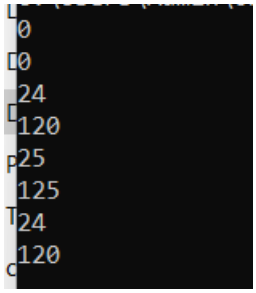
What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

Example (Rectangle class):

```
class prog7
{
        public static void main(String args[])
        {
                rect r1=new rect();
                rect r2=new rect(6,4,5);
                rect r3=new rect(5);
                rect r4=r2;
                System.out.println(r1.area());
                System.out.println(r1.vol());
                System.out.println(r2.area());
                System.out.println(r2.vol());
                System.out.println(r3.area());
                System.out.println(r3.vol());
                System.out.println(r4.area());
                System.out.println(r4.vol());
```

```
        }
}

class rect
{
        int l,b,h;
        rect()
        {
                l=b=h=0;
        }
        rect(int t)
        {
                l=b=h=t;
        }
        /*rect(int l,int b,int h)
        {
                this.l=l;
                this.b=b;
                this.h=h;
        }*/
        rect(int l1,int b1,int h1)
        {
                l=l1;
                b=b1;
                h=h1;    }
        int area()
        {
                return l*b;
        }
        int vol()
        {
                return l*b*h;
        }
}
```



Java Destructor

In Java, when we create an object of the class it occupies some space in the memory (heap). If we do not delete these objects, it remains in the memory and occupies unnecessary space that is not upright from the aspect of programming. To resolve this problem, we use the destructor. we

will see the alternate option to the destructor in Java. Also, we will see how to use the finalize() method as a destructor.

The destructor is the opposite of the constructor. The constructor is used to initialize objects while the destructor is used to delete or destroy the object that releases the resource occupied by the object.

Remember that there is no concept of destructor in Java. In place of the destructor, Java provides the garbage collector that works the same as the destructor.

The garbage collector is a program (thread) that runs on the jvm It automatically deletes the unused objects (objects that are no longer used) and free-up the memory.

The programmer has no need to manage memory, manually. It can be error-prone, vulnerable, and may lead to a memory leak.

What is the destructor in Java?

It is a special method that automatically gets called when an object is no longer used. When an object completes its life-cycle the garbage collector deletes that object and deallocates or releases the memory occupied by the object.

it is also known as finalizers that are non-deterministic. In java, the allocation and deallocation of objects handled by the garbage collector. The invocation of finalizers is not guaranteed because it invokes implicitly.

Advantages of Destructor

- It releases the resources occupied by the object.
- No explicit call is required, it is automatically invoked at the end of the program execution.
- It does not accept any parameter and cannot be overloaded.

How does destructor work?

When the object is created it occupies the space in the heap. These objects are used by the threads. If the objects are no longer is used by the thread it becomes eligible for the garbage collection. The memory occupied by that object is now available for new objects that are being created. It is noted that when the garbage collector destroys the object, the JRE calls the finalize() method to close the connections such as database and network connection.

From the above, we can conclude that using the destructor an garbage collector is the level of developer's interference to memory management. It is the main difference between the two. The destructor notifies exactly when the object will be destroyed. While in Java the garbage collector does the same work automatically. These two approaches to memory management have positive

and negative effects. But the main issue is that sometimes the developer needs immediate access to memory management.

Java finalize() Method

It is difficult for the programmer to forcefully execute the garbage collector to destroy the object. But Java provides an alternative way to do the same. The Java Object class provides the finalize() method that works the same as the destructor. The syntax of the finalize() method is as follows:

Syntax:

```
protected void finalize throws Throwable()
{
//resources to be close
}
```

It is not a destructor but it provides extra security. It ensures the use of external resources like closing the file, etc. before shutting down the program. We can call it by using the method itself or invoking the method System.runFinalizersOnExit(true).

- It is a protected method of the Object class that is defined in the java.lang package.
- It can be called only once.
- We need to call the finalize() method explicitly if we want to override the method.
- The gc() is a method of JVM executed by the Garbage Collector. It invokes when the heap memory is full and requires more memory for new arriving objects.
- Except for the unchecked exceptions, the JVM ignores all the exceptions that occur by the finalize() method.

Example of Destructor

```
public class Bus
{
Bus()
{
System.out.println("Object of bus class is created");
}
```

```
protected void finalize()
{
System.out.println("Object of bus class is destroyed  by the Garbage Collector");
}
}

Public class BusDemo
{
public static void main(String[] args)
{
Bus  b = new Bus ();
b = null;
System.gc();   // manually call Garbage collector
}
}
```

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

1) Java static variable

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

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

Advantages of static variable

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

Understanding the problem without static variable

```
class Student
{
    int rollno;
    String name;
    String college="UAC";
}
```

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

Java static property is shared to all objects.

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student
{
  int rollno;//instance variable
  String name;
  static String college ="UAC";//static variable

  //constructor
  Student(int r, String n)
{
  rollno = r;
  name = n;
  }
  //method to display the values
 void display ()
{
System.out.println(rollno+" "+name+" "+college);}
}

//Test class to show the values of objects
public class TestStaticVariable1
{
 public static void main(String args[])
{
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");

 //we can change the college of all objects by the single line of code
 //Student.college="UAC";
 s1.display();
 s2.display();  }}
```

Output:

```
111 Karan UAC
222 Aryan UAC
```

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.

```
//Java Program to demonstrate the use of an instance variable
//which get memory each time when we create an object of the class.

class Counter
{
int count=0;//will get memory each time when the instance is created

Counter()
{
count++;//incrementing value
System.out.println(count);
}

public static void main(String args[])
{
//Creating objects
Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}
```

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.

```
//Java Program to illustrate the use of static variable which
//is shared with all objects.
class Counter2
{
static int count=0;//will get memory only once and retain its value

Counter2()
{
count++;//incrementing the value of static variable
System.out.println(count);
}
public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

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

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "UAC";
```

```
    //static method to change the value of static variable
    static void change(){
    college = "UACC";
    }

    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }

    //method to display values
    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,"Sonu");
    //calling display method
    s1.display();
    s2.display();
    s3.display();  }
```

Output:

```
111 Karan UACC
222 Aryan UACC
333 Sonu UACC
```

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.

Q) Why is the Java main method static?

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.

3) Java static block

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

Example of static block

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

Output:

```
static block is invoked
Hello main
```

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

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

Example of Static function:

```
class maths{
	static int square(int x)
	{
		return x*x;
	}

	static int cube(int x)
	{
		return x*x*x;
	}
	void show()
	{
		System.out.println("HELLO...");
	}
}
class Static1
{
	public static void main(String args[])
	{
		System.out.println(maths.square(3));
		System.out.println(maths.cube(5));

		maths m1=new maths();
		m1.show();
	}
}
```

```
9
125
```

Inheritance in Java

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

The idea behind inheritance in Java is that you can create new classes

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

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

In Oop, the concept of inheritance provides the ideas of Reusability.

This means that we can add additional features to an existing class without modifying it.

Why use inheritance in java:

1. Method Overriding (achieve run-time polymorphism)
2. Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

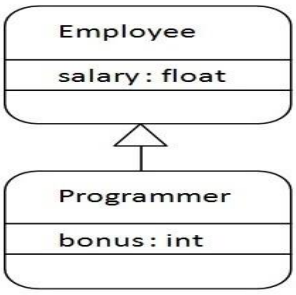
The syntax of Java Inheritance

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

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

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

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

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

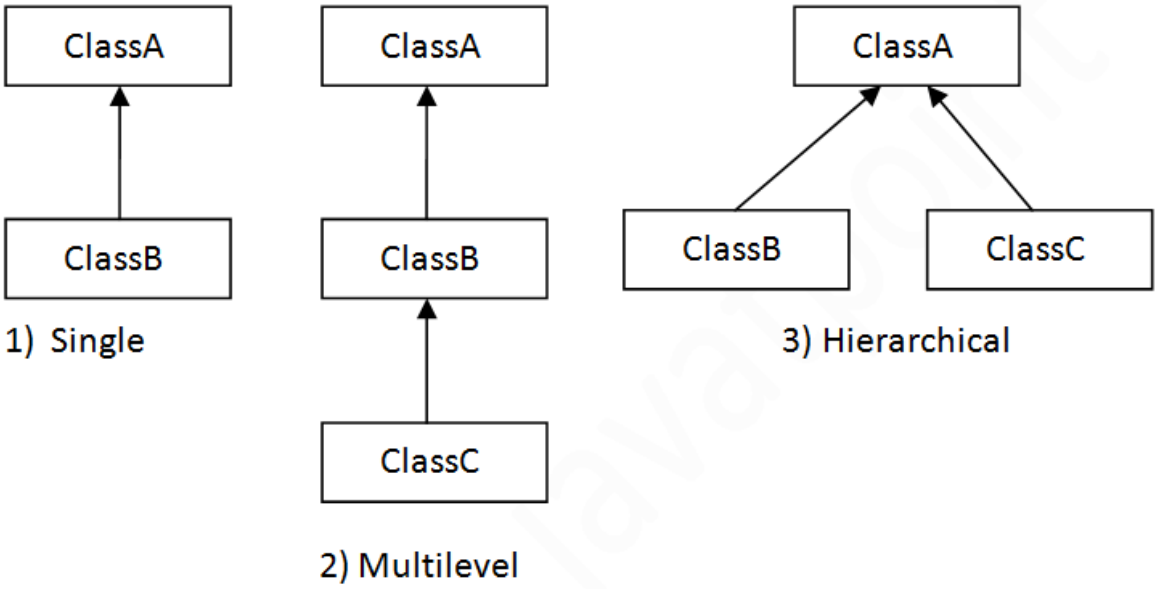
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

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

Types of inheritance in java

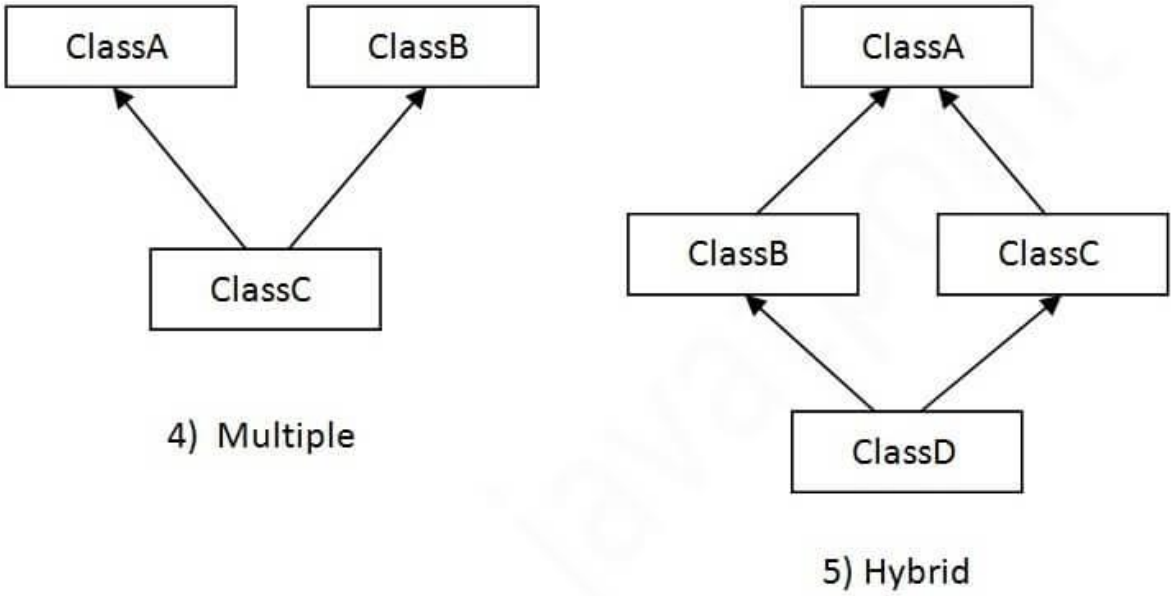
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



.

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}  }   //Compile time error
```

1. Single Inheritance:

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

Output:

```
barking...
eating...
```

Example-1:

```
class a
{
	int x;
	a()
	{
	x=0;}
	a(int x)
	{
		this.x=x;	}

	void show()
	{
		System.out.println("value of x:-"+x);
	}
}

class b extends a
{
	int y;
	b()
	{
		y=0;
	}
	b(int x,int y)
	{

	super(x);
	this.y=y;
	}

	void show()
	{
		super.show();
		System.out.println("value of y:-"+y);
	}
}
```

```
class in1
{
	public static void main(String args[])
	{
		b ob=new b();
		ob.show();
		b ob1= new b(5,10);
		ob1.show();
	}
}
```

```
value of x:-0
value of y:-0
value of x:-5
value of y:-10
```

Example-2:

```
import java.io.*;
class emp
{
	String nm;
	int basic;
	emp()
	{
		nm="";
		basic=0;
	}

	emp(String nm,int basic)
	{
		this.nm=nm;
		this.basic=basic;
	}

	void get()throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	System.out.println("enter name,basic:-");
	nm=dis.readLine();
	basic=Integer.parseInt(dis.readLine());
	}

	void show()
	{
		System.out.println("name:-"+nm);
		System.out.println("basic:-"+basic);
	}
}
```

```
class calsal extends emp
{
	int ndays;
	float hra,da,pf,ns,sal;
	calsal()
	{
		ndays=0;
	}

	calsal(String nm,int basic,int ndays)
	{
	super(nm,basic);
	this.ndays=ndays;
	}

	void get() throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	super.get();
	System.out.println("enter no.days:");
	ndays=Integer.parseInt(dis.readLine());
	}

	void show()
	{
	hra=0.15f*basic;
	da=0.10f*basic;
	pf=0.05f*basic;
	sal=ndays*basic/30;
	ns=sal+hra+da-pf;

	super.show();
	System.out.println("hra:-"+hra);
	System.out.println("da:-"+da);
	System.out.println("pf:-"+pf);
	System.out.println("ns:-"+ns);
	}
}

class in3
{
	public static void main(String args[])throws IOException
	{
		calsal c1=new calsal();
		c1.get();
		c1.show();
		calsal c2=new calsal("a",3000,15);
		c2.show();
	}
}
```

```
enter name,basic:-
khyati
30000
enter no.days:
26
name:-khyati
basic:-30000
hra:-4500.0
da:-3000.0
pf:-1500.0
ns:-32000.0
name:-a
basic:-3000
hra:-450.00003
da:-300.0
pf:-150.0
ns:-2100.0
```

Example-3:

```
import java.io.*;
class stud
{
	String nm;
	float per;

	void get()throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	System.out.println("enter name,per:-");
	nm=dis.readLine();
	per=Float.parseFloat(dis.readLine());
	}

	void show()
	{
		System.out.println("name:-"+nm);
		System.out.println("per:-"+per);
	}
}

class scholarship extends stud
{
	int samt;
	void show()
	{
	super.show();
	if(per >=70)
		System.out.println("samt is 10000");

	else if(per >= 60)
		System.out.println("samt iS 8000");
	else
		System.out.println("NO SCHOLARSHIP...");}}
```

```
class in4
{
	public static void main(String args[])throws IOException
	{
		scholarship s1=new scholarship();
		s1.get();
		s1.show();
	}
}
```

```
enter name,per:-
khyati
70
name:-khyati
per:-70.0
samt is 10000
```

2. Multilevel inheritance

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

Example-1:

```
import java.io.*;
class cust
{
	String nm,mtype; //  mtype : a,b,c

	void get()throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	System.out.println("enter name,type:-");
	nm=dis.readLine();
	mtype=dis.readLine();
	}

	void show()
	{
		System.out.println("name:-"+nm);
		System.out.println("type:-"+mtype);
	}
}

class bill extends cust
{
	int bamt;

	void get() throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	super.get();
	System.out.println("enter bill amt:-");
	bamt=Integer.parseInt(dis.readLine());
	}

	void show()
	{
	super.show();
	System.out.println("bill amt..."+bamt);
	}
}

class dis extends bill
{	float damt,namt;
	void show()
	{
	super.show();
	if(mtype.equals("a"))
		damt=0.40f*bamt;
	else if(mtype.equals("b"))
		damt=0.30f*bamt;
	else
		damt=0.15f*bamt;
```

```
        namt=bamt-damt;
        System.out.println("discount..."+damt);
        System.out.println("net bill..."+namt);
        System.out.println("thank u...");
        }
}


class in6
{
        public static void main(String args[])throws IOException
        {
                dis d1=new dis();
                d1.get();
                d1.show();
                }
}
```

```
enter name,type:-
khyati
b
enter bill amt:-
30000
name:-khyati
type:-b
bill amt...30000
discount...9000.0
net bill...21000.0
thank u...
```

3. Hierarchical inheritance:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Example-1:

```
import java.io.*;
class emp
{
	String nm;
	int basic;

	void get()throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	System.out.println("enter name,basic:-");
	nm=dis.readLine();
	basic=Integer.parseInt(dis.readLine());
	}

	void show()
	{
		System.out.println("name:-"+nm);
		System.out.println("basic:-"+basic);
	}
}

class ftime extends emp
{
	float hra,da,pf,ns;

	void get() throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	super.get();
	hra=0.15f*basic;
	da=0.10f*basic;
	pf=0.05f*basic;
	ns=hra+da-pf+basic;
	}

	void show()
	{
	super.show();
	System.out.println("hra:-"+hra);
	System.out.println("da:-"+da);
	System.out.println("pf:-"+pf);
	System.out.println("ns:-"+ns);
	}
}
```

```
class ptime extends emp
{
	int hw;
	float ns;
	void get()throws IOException
	{
	super.get();
	DataInputStream dis=new DataInputStream(System.in);
	System.out.println("enter working hours...");
	hw=Integer.parseInt(dis.readLine());
	ns= hw*basic;
	}

	void show()
	{
	super.show();
	System.out.println("working hours..."+hw);
	System.out.println("ns..."+ns);
	}
}


class in5
{
	public static void main(String args[])throws IOException
	{
		ftime f1=new ftime();
		f1.get();
		f1.show();
		ptime p1=new ptime();
		p1.get();
		p1.show();
	}
}
```

```
enter name,basic:-
khyati
40000
name:-khyati
basic:-40000
hra:-6000.0
da:-4000.0
pf:-2000.0
ns:-48000.0
enter name,basic:-
jyoti
3000
enter working hours...
4
name:-jyoti
basic:-3000
working hours...4
ns...12000.0
```

Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve* ***abstraction***. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

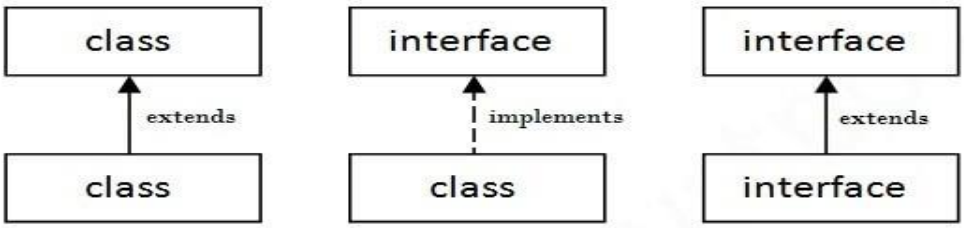
Syntax:

```
interface <interface_name>{

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

Since Java 8, Interface fields are public, static and final by default, and the methods are public and abstract.

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
interface printable
{
void print();
}
class A6 implements printable
{
public void print()
{
System.out.println("Hello");
}
 public static void main(String args[])
{
A6 obj = new A6();
obj.print();
 } }
```

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

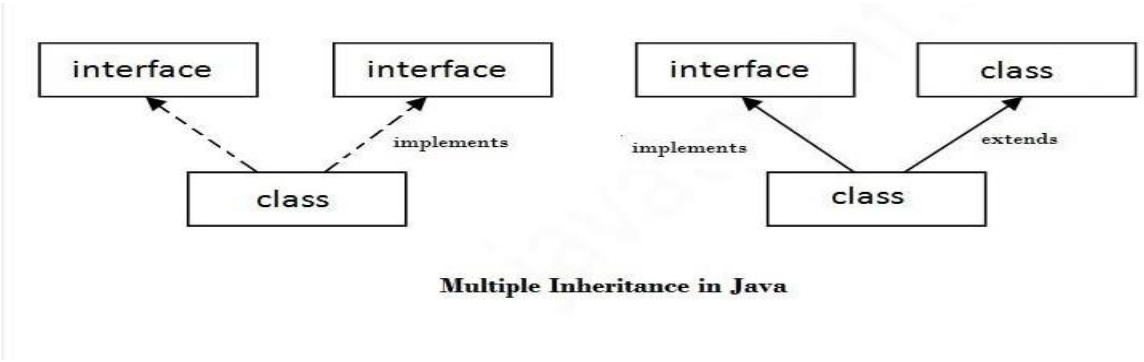
```
//Interface declaration
interface Drawable{
void draw();
}
//Implementation
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();
d.draw();
}}
```

Output:

```
drawing circle
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

```
Output:Hello
     Welcome
```

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
interface Printable{
void print();
}
interface Showable{
```

```
void print();
}
class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();   }  }
```

Output:

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}  }
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

Output:

```
drawing rectangle
default method
```

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
```

```
    d.draw();
    System.out.println(Drawable.cube(3));
    }}
```

Output:

```
drawing rectangle
```

Nested Interface in Java

Note: An interface can have another interface which is known as a nested interface.

```
interface printable
{
  void print();
 interface MessagePrintable
{
   void msg();
 }
}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

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

4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example-1:

```
import java.io.*;

interface shape
{
        public void area();
        public void vol();
}

class rect implements shape
{
        int l,b,h;
        void get()throws IOException
        {
        DataInputStream dis=new DataInputStream(System.in);
```

```
        System.out.println("enter l,b,h...");
        l=Integer.parseInt(dis.readLine());
        b=Integer.parseInt(dis.readLine());
        h=Integer.parseInt(dis.readLine());
        }


        public void area()
        {
        System.out.println("area is:-"+(l*b));   }

        public void vol()
        {
        System.out.println("vol is:-"+(l*b*h));
        }
}


class circle implements shape
{
        float r;
        void get()throws IOException
        {
        DataInputStream dis=new DataInputStream(System.in);
        System.out.println("enter r...");
        r=Float.parseFloat(dis.readLine());
        }

        public void area()
        {
        float a=3.14f*r*r;
        System.out.println("area is:-"+a);
        }

        public void vol()
        {
        float v =4/3*3.14f*r*r*r;
        System.out.println("vol is:-"+v);
        }
}

class inf1
{
```

```
        public static void main(String args[])throws IOException
        {
                shape s;
                rect r=new rect();
                circle c=new circle();
                r.get();
                c.get();
                s=r;
                s.area();
                s.vol();
                s=c;
                s.area();
                s.vol();
}
}
```

```
enter l,b,h...
3
4
6
enter r...
7
area is:-12
vol is:-72
area is:-153.86002
vol is:-1077.0201
```

Example-2:

```
interface test
{
        public void show();
}
class easy implements test
{

        public void show()
        {
        System.out.println("easy class");
        }
}
class hard implements test
{

        public void show()
        {
```

```
        System.out.println("hard class");
        }
}
```

```
class inf2
{
        public static void main(String args[])
        {
                test t;
                easy e=new easy();
                hard h=new hard();
                t=e;
                t.show();
                t=h;
                t.show();
        }
}
```

```
easy class
hard class
```

Multiple Inheritance Example-1

```
class stud
{
        void score()
        {
                System.out.println("passing marks for theoary is 50");
        }
}

interface sports
{
        public void test();
}

class result extends stud implements sports
{

        public void test()
        {
```

```
        System.out.println("sport is optional for it student");
        }
}
```

```
class minh1
{
        public static void main(String args[])
        {
                result r1=new result();
                r1.score();
                r1.test();
        }
}
```

```
passing marks for theoary is 50
sport is optional for it student
```

Multiple Inheritance Example-2

```
interface shape
{
        public void fshape();
}
interface color
{
        public void fcolor();
}
class grapes implements shape,color
{
        public void fshape()
        {
        System.out.println("SMALL IN SIZE & CIRCULAR &OVAL...");
        }
        public void fcolor()
        {
        System.out.println("GREEN COLOR....");
        }
}
class minh2
{
        public static void main(String args[])
```

```
{
        grapes g1=new grapes();
        g1.fshape();
        g1.fcolor();
}}
```

```
SMALL IN SIZE & CIRCULAR &OVAL...
GREEN COLOR....
```

Polymorphism:-

- **polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- Polymorphism simply means "**One name And Multiple behaviour**".
- Polymorphism plays an important role in allowing objects having different internal structure to share the same external interface.
- The overloaded member functions are selected for invoking by matching the arguments. In these both, data type and no. of arguments are matched. This information is known to the compiler at the time of compilation and compiler is able to select appropriate function for particular called at compile time this concept is called **Compile time polymorphism.**
- When appropriate member function is selected while program is running this is called **Runtime polymorphism**
- We can perform polymorphism in java by method overloading and method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism.

❖ Method Overloading in Java (Compile time polymorphism)

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program

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.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments

2. By changing the data type

In Java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods

so that we don't need to create instance for calling methods.

```
class Adder
{
static int add(int a,int b)
{
return a+b;
}
static int add(int a,int b,int c)
{
return a+b+c;
}
}

class TestOverloading1
{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}
}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type

The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder

    {
    static int add(int a, int b)
    {
    return a+b;
    }
    static double add(double a, double b)
    {
    return a+b;
    }
    }

    class TestOverloading2
    {
    public static void main(String[] args)
    {
    System.out.println(Adder.add(11,11));
    System.out.println(Adder.add(12.3,12.6));
    }
    }
```

Output:

```
22
24.9
```

Q) Why Method Overloading is not possible by changing the return type of method only?

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

```
class Adder
{
static int add(int a,int b)
{
return a+b;
}

static double add(int a,int b)
{
return a+b;
}
}

class TestOverloading3
{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));//ambiguity
}
}
```

Output:

```
Compile Time Error: method add(int,int) is already defined in class Adder
```

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But jvm calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4
{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

Output:

```
main with String[]
```

❖ Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding
    //Creating a parent class.
    class Vehicle{
      //defining a method
      void run(){System.out.println("Vehicle is running");}
    }
```

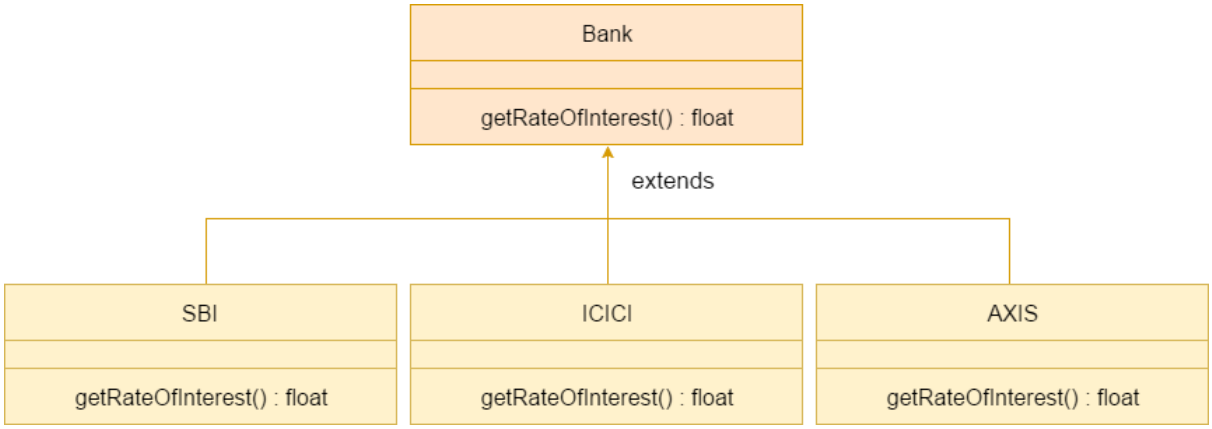
```
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



Java method overriding is mostly used in Runtime Polymorphism which we will learn later on.

```
//Java Program to demonstrate the real scenario of Java Method Overriding
//where three classes are overriding the method of a parent class.
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
```

```
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

Difference between method Overloading and Method Overriding in java

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example

```
class OverloadingExample{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
```

Java Method Overriding example

```
class Animal{
void eat(){System.out.println("eating...");}  }
```

```
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
}
```

❖ Runtime Polymorphism in Java

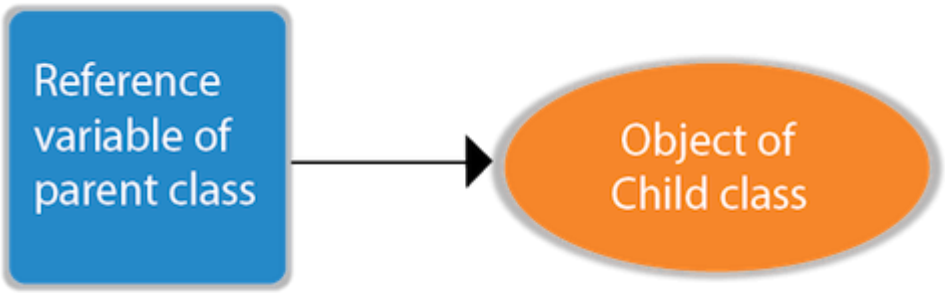
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:




```
class A{}
class B extends A{}
A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike
{
  void run()
{
System.out.println("running");
}
}
class Splendor extends Bike
{
  void run()
{
System.out.println("running safely with 60km");
}

 public static void main(String args[])
{
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output:

```
running safely with 60km.
```

Java Runtime Polymorphism Example: Shape

```
class Shape
{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape
{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape
{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape
{
void draw(){System.out.println("drawing triangle...");}
}

class TestPolymorphism2
{
public static void main(String args[]){
Shape s;
s=new Rectangle();
s.draw();

s=new Circle();
s.draw();

s=new Triangle();
s.draw();
}
}
```

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
class Dog
{
 private void eat()
{
System.out.println("dog is eating...");
}

 public static void main(String args[])
{
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal
{
 void eat(){System.out.println("animal is eating...");}
}
```

```
class Dog extends Animal
{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Output:dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

```
abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class (Run-time Polymorphism)

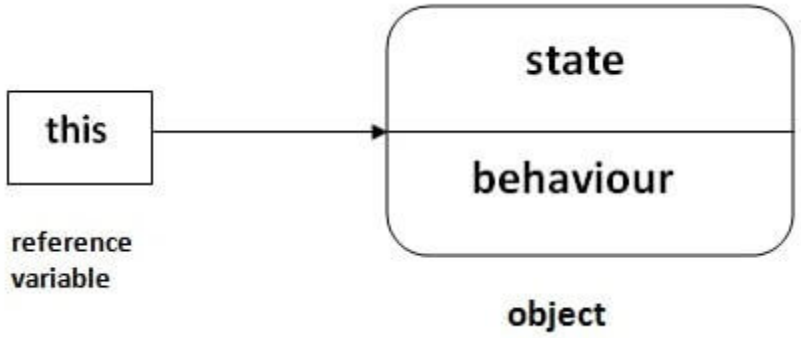
```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() met
hod
s.draw();
```

```
/*
Or
Shape s;
Rectangle r=new Rectangle();

S=r;
s.draw();
*/
}
}
```

```
drawing circle
```

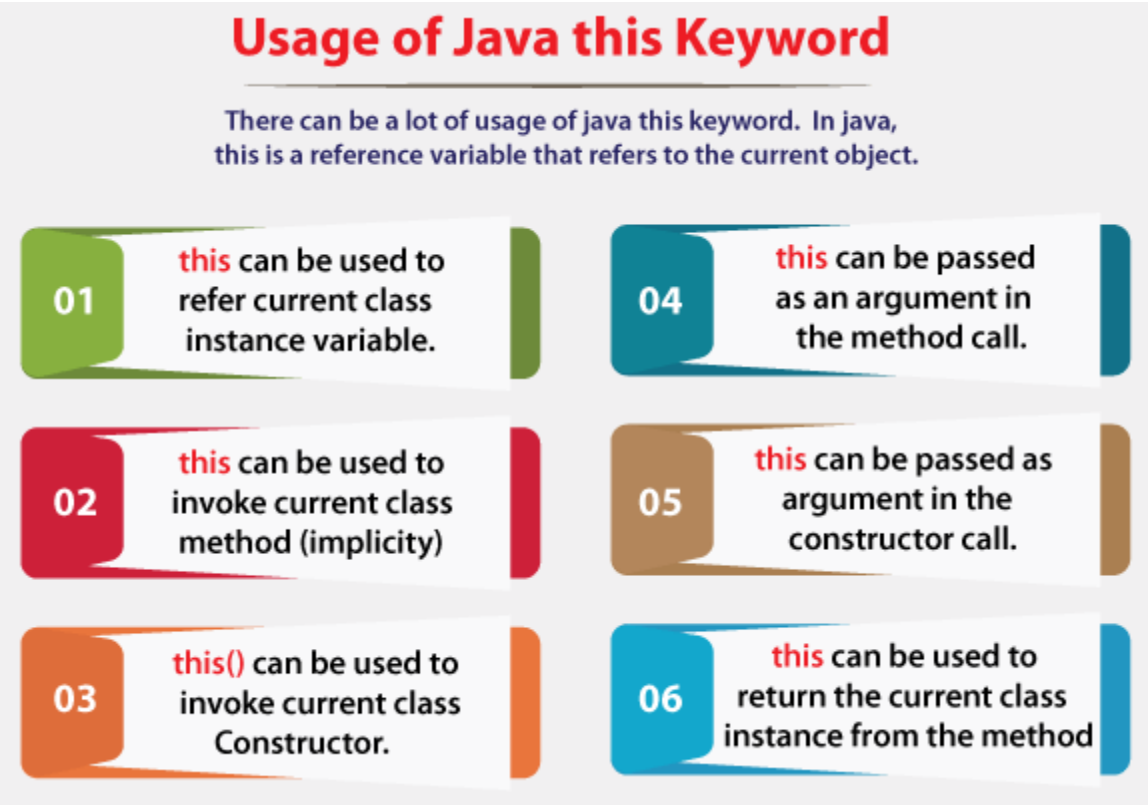
this keyword in Java

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



Usage of Java this keyword

Here is given the 6 usage of java this keyword



Note: we will see the first 3 usage of this keyword

1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

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

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

Output:

```
0 null 0.0
0 null 0.0
```

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

Solution of the above problem by this keyword

```
class Student{

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

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

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
class Student{

int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
```

```
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis3{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

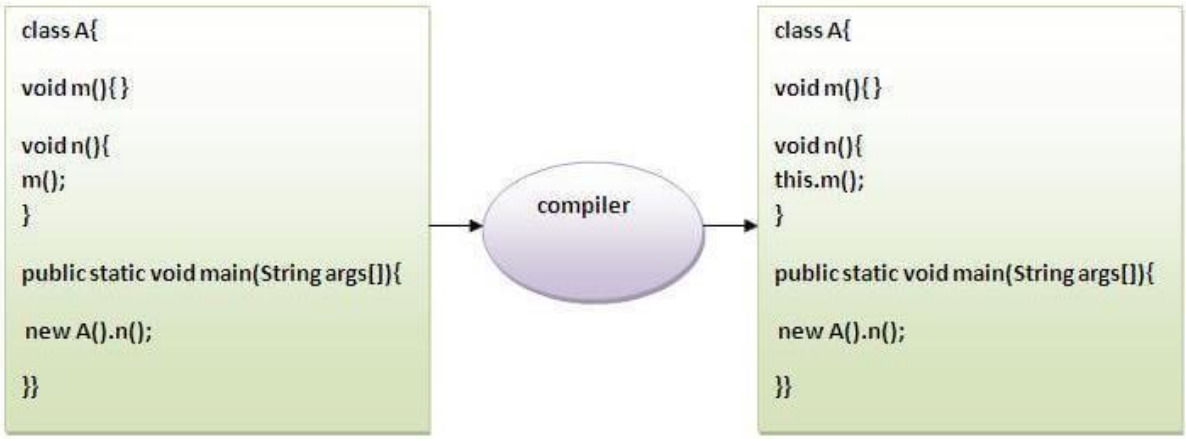
Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

```
hello n
hello m
```

3) 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:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor:

```
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

Output:

```
5
hello a
```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class Student{

int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;  }
```

```
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit java 0.0
112 sumit java 6000.0
```

Rule: Call to this() must be the first statement in constructor.

```
Student(int rollno,String name,String course,float fee){

this.fee=fee;
this(rollno,name,course);//C.T.Error
}
```

Output:

```
Compile Time Error: Call to this must be first statement in constructor
```

Super Keyword in Java

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

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

Usage of Java super Keyword

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

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

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

```
class Animal{
String color="white";
}

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

Output :

```
black
white
```

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

2) super can be used to invoke parent class method

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

```
class Animal
{
void eat(){System.out.println("eating...");}
}
```

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

Output:

```
eating...
eating bread...
```

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

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

3) super is used to invoke parent class constructor.

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

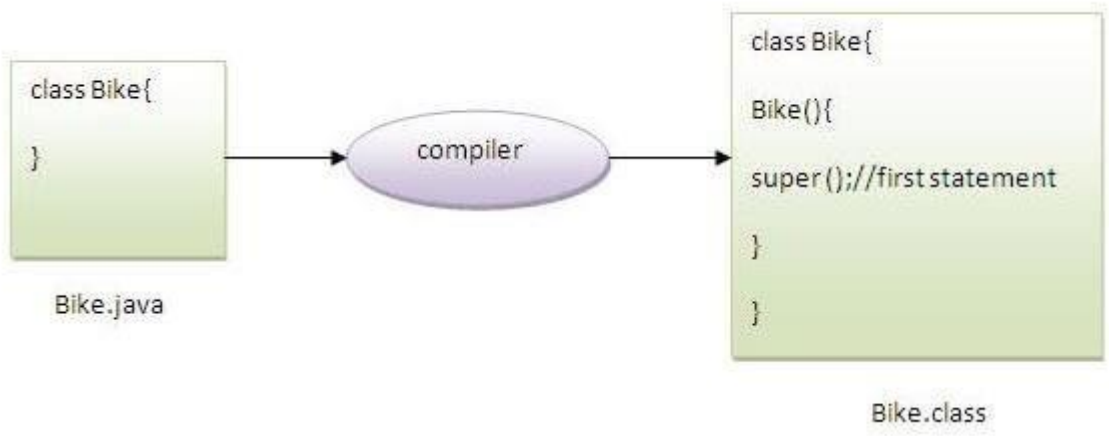
```
class Animal
{
Animal()
{System.out.println("animal is created");}
}
class Dog extends Animal
{
Dog(){
super();
System.out.println("dog is created");  }}
```

```
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}
class TestSuper4{
public static void main(String args[]){
```

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

Output:

```
animal is created
dog is created
```

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person
{
int id;
String name;
Person(int id,String name)
{
this.id=id;
this.name=name;
}
}

class Emp extends Person
{
float salary;
Emp(int id,String name,float salary)
{
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display()
{
System.out.println(id+" "+name+" "+salary);
}
}
```

```
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}
```

Output:

```
1 ankit 45000
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

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

Output:Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}
 class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}
       public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }}
```

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
  }
}
```

Output: 70

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A{
  static final int data;//static blank final variable
  static{ data=50;}
```

```
  public static void main(String args[]){
    System.out.println(A.data);
  }
}
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
  int cube(final int n){
   n=n+2;//can't be changed as n is final
   n*n*n;
   }
  public static void main(String args[]){
    Bike11 b=new Bike11();
    b.cube(5);
  }
}
```

Output: Compile Time Error

Q) Can we declare a constructor final?

No, because constructor is never inherited.

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:

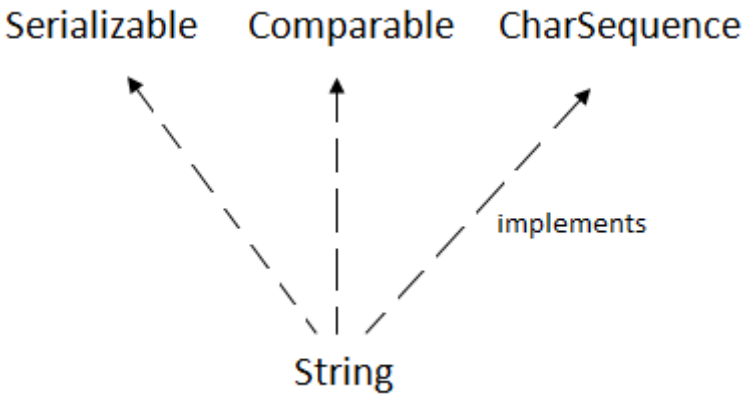
```
char[] ch={'k','h','y','a','t','i'};
String s=new String(ch);
```

is same as:

```
String s="khyati";
```

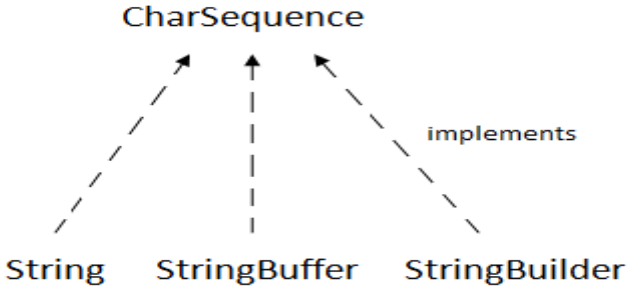
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), substring() etc.

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



CharSequence Interface

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



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

What is String in Java?

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

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

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

```
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:

```
String s1="Welcome";
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

```
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

```
public class StringExample
{
public static void main(String args[])
{
String s1="java";//creating string by Java string literal

char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string

String s3=new String("example");//creating Java string by new keyword

System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:

```
java
strings
example
```

The above code, converts a ***char*** array into a **String** object. And displays the String objects ***s1, s2***, and ***s3*** on console using ***println()*** method.

Immutable String in Java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. 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.

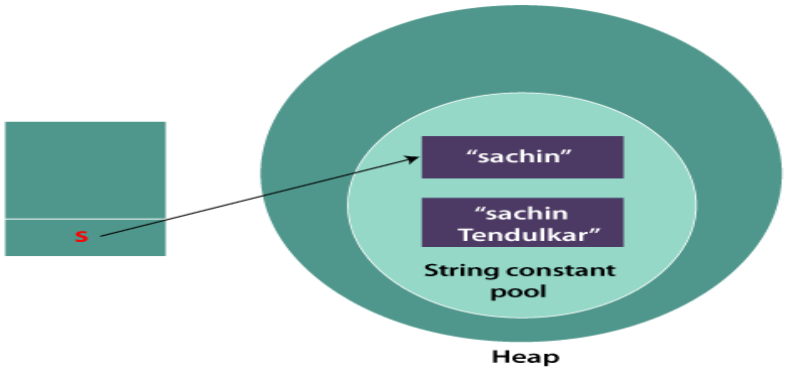
Let's try to understand the concept of immutability by the example given below:

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

Output:

```
Sachin
```

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. 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 explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

```
String s="Sachin";
s=s.concat(" Tendulkar");
System.out.println(s);
```

```
Sachin Tendulkar
```

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

Why String objects are immutable in Java?

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Why String class is Final in Java?

The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.

Java String class methods

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

1. charAt()

The **Java String class charAt()** method returns *a char value at the given index number*.

The index number starts from 0 and goes to n-1, where n is the length of the string. It returns **StringIndexOutOfBoundsException,** if the given index number is greater than or equal to this string length or a negative number.

Syntax

```
public char charAt(int index)
```

The method accepts **index** as a parameter. The starting index is 0. It returns a character at a specific index position in a string. It throws **StringIndexOutOfBoundsException** if the index is a negative value or greater than this string length.

Specified by CharSequence interface, located inside java.lang package.

```
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
```

```
S
H
```

2. concat()

The **Java String class concat()** method *combines specified string at the end of this string*. It returns a combined string. It is like appending another string.

Signature

The signature of the string concat() method is given below:

```
public String concat(String anotherString)
```

Parameter

anotherString : another string i.e., to be combined at the end of this string.

Returns

combined string

```
String s1="Sachin ";
String s2="Tendulkar";
String s3=s1.concat(s2);
System.out.println(s3);//Sachin Tendulkar
```

```
Sachin Tendulkar
```

The above Java program, concatenates two String objects ***s1*** and ***s2*** using ***concat()*** method and stores the result into ***s3*** object.

3. equals()

The **Java String class equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of the Object class.

Signature public boolean equals(Object anotherObject)

Parameter anotherObject **: another object, i.e., compared with this string.**

Returns

true if characters of both strings are equal otherwise **false**.

```
String s1="khyati";
String s2="khyati";
String s3="KHYATI";
String s4="ARTI";
System.out.println(s1.equals(s2));//true because content and case is same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
```

```
true
false
false
```

4. indexOf()

The **Java String class indexOf()** method returns the position of the first occurrence of the specified character or string in a specified string.

Signature

There are four overloaded indexOf() method in Java. The signature of indexOf() methods are given below:

No.	Method	Description
1	int indexOf(int ch)	It returns the index position for the given char value
2	int indexOf(int ch, int fromIndex)	It returns the index position for the given char value and from index
3	int indexOf(String substring)	It returns the index position for the given substring
4	int indexOf(String substring, int fromIndex)	It returns the index position for the given substring and from index

Parameters

ch: It is a character value, e.g. 'a'

fromIndex: The index position from where the index of the char value or substring is returned.

substring: A substring to be searched in this string.

Returns

Index of the searched string or character.

```
String s1="this is index of example";

//passing substring
int index1=s1.indexOf("is");//returns the index of is substring
int index2=s1.indexOf("index");//returns the index of index substring
System.out.println(index1+"  "+index2);//2 8

//passing substring with from index
int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index
System.out.println(index3);//5 i.e. the index of another is

//passing char value
int index4=s1.indexOf('s');//returns the index of s char value
System.out.println(index4);//3
```

```
2  8
5
3
```

We observe that when a searched string or character is found, the method returns a non-negative value. If the string or character is not found, -1 is returned. We can use this property to find the total count of a character present in the given string.

5. lastIndexOf()

The **Java String class lastIndexOf()** method returns the last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

Signature

There are four types of lastIndexOf() method in Java. The signature of the methods are given below:

No.	Method	Description
1	int lastIndexOf(int ch)	It returns last index position for the given char value
2	int lastIndexOf(int ch, int fromIndex)	It returns last index position for the given char value and from index
3	int lastIndexOf(String substring)	It returns last index position for the given substring
4	int lastIndexOf(String substring, int fromIndex)	It returns last index position for the given substring and from index

Parameters

ch: char value i.e. a single character e.g. 'a'

fromIndex: index position from where index of the char value or substring is retured

substring: substring to be searched in this string

Returns

last index of the string

Java String lastIndexOf() method example

```
String s1="this is index of example";//there are 2 's' characters in this sentence
int index1=s1.lastIndexOf('s');//returns last index of 's' char value
System.out.println(index1);//6
```

Java String lastIndexOf(int ch, int fromIndex) Method Example

Here, we are finding the last index from the string by specifying *fromIndex*.

```
String str = "This is index of example";
int index = str.lastIndexOf('s',5);
System.out.println(index);
```

```
3
```

Java String lastIndexOf(String substring) Method Example

It returns the last index of the substring.

```
String str = "This is last index of example";
int index = str.lastIndexOf("of");
System.out.println(index);
```

```
19
```

Java String lastIndexOf(String substring, int fromIndex) Method Example

It returns the last index of the substring from the *fromIndex*.

```
String str = "This is last index of example";
int index = str.lastIndexOf("of", 25);
System.out.println(index);
index = str.lastIndexOf("of", 10);
System.out.println(index); // -1, if not found
```

```
19
-1
```

6. isEmpty()

The **Java String class isEmpty()** method checks if the input string is empty or not. Note that here empty means the number of characters contained in a string is zero.

Signature The signature or syntax of string isEmpty() method is given below:

```
public boolean isEmpty()
```

Returns

true if length is 0 otherwise false.

```
String s1="";
String s2="khyati";

System.out.println(s1.isEmpty());
System.out.println(s2.isEmpty());
```

```
true
false
```

7. join()

The Java String class join() method returns a string joined with a given delimiter. In the String join() method, the delimiter is copied for each element. The join() method is included in the Java string since JDK 1.8.

Signature

The signature or syntax of the join() method is given below:

```
public static String join(CharSequence delimiter, CharSequence... elements)
```

Parameters

delimiter : char value to be added with each element

elements : char value to be attached with delimiter

Returns

joined string with delimiter

Exception Throws

NullPointerException if element or delimiter is null.

```
String joinString1=String.join("-","welcome","to","javaworld");
System.out.println(joinString1);
```

```
welcome-to-javaworld
```

8. length()

The **Java String class length()** method finds the length of a string. The length of the Java string is the same as the Unicode code units of the string.

Signature

The signature of the string length() method is given below:

```
public int length()
```

Specified by

CharSequence interface

Returns

Length of characters. In other words, the total number of characters present in the string.

```
String s1="khyatisolanki";
String s2="python";
System.out.println("string length is: "+s1.length());//13 is the length of khyatisolanki string
System.out.println("string length is: "+s2.length());//6 is the length of python string
```

```
string length is: 13
string length is: 6
```

9. split()

The **java string split()** method splits this string against given regular expression and returns a char array.

Signature

There are two signature for split() method in java string.

```
public String split(String regex)
and,
public String split(String regex, int limit)
```

Parameter

regex : regular expression to be applied on string.

limit : limit for the number of strings in array. If it is zero, it will returns all the strings matching regex.

Returns

array of strings

```
String s1="java string split method by khyati";
String[] words=s1.split("\\s");//splits the string based on whitespace
//using java foreach loop to print elements of string array
for(String w:words){
System.out.println(w);
```

```
java
string
split
method
by
khyati
```

Split With Regular Expression

```
String Str = new String("Welcome-to-java-Tutorials");
     System.out.println("Return Value :" );

     for (String retval: Str.split("-")) {
        System.out.println(retval);
     }
   }
}
```

Output

```
Return Value :
Welcome
To
java
```

Tutorials

10.substring()

The **Java String class substring()** method returns a part of the string.

We pass beginIndex and endIndex number position in the Java substring method where beginIndex is inclusive, and endIndex is exclusive. In other words, the beginIndex starts from 0, whereas the endIndex starts from 1.

There are two types of substring methods in Java string.

Signature

```
public String substring(int startIndex)  // type - 1
and
public String substring(int startIndex, int endIndex)  // type - 2
```

If we don't specify endIndex, the method will return all the characters from startIndex.

Parameters

startIndex : starting index is inclusive

endIndex : ending index is exclusive

Returns

specified string

Exception Throws

StringIndexOutOfBoundsException is thrown when any one of the following conditions is met.

- if the start index is negative value
- end index is lower than starting index.
- Either starting or ending index is greater than the total number of characters present in the string.

```
String s1="khyati";
```

```
String substr = s1.substring(0); // Starts with 0 and goes to end
System.out.println(substr);
String substr2 = s1.substring(2,5); // Starts from 2 and goes to 5
System.out.println(substr2);
String substr3 = s1.substring(5,15); // Returns Exception
```

```
khyati
yati
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: begin 5, end 15, length 6
```

11.trim()

The **Java String class trim()** method eliminates leading and trailing spaces. The Unicode value of space character is '\u0020'. The trim() method in Java string checks this Unicode value before and after the string, if it exists then the method removes the spaces and returns the omitted string.

The string trim() method doesn't omit middle spaces.

Signature

The signature or syntax of the String class trim() method is given below:

```
public String trim()
```

Returns

string with omitted leading and trailing spaces

```
String s1="  hello string   ";
System.out.println(s1+"khyati");//without trim()
System.out.println(s1.trim()+"khyati");//with trim()
```

```
hello string   khyati
hello stringkhyati
```

12.toUpperCase() and toLowerCase()

The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

```
String s="Sachin";
```

```
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
```

```
SACHIN
sachin
Sachin
```

13. startsWith() and endsWith()

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

```
String s="Sachin";
System.out.println(s.startsWith("Sa"));//true
System.out.println(s.endsWith("n"));//true
```

```
true
true
```

14. replace()

The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

```
String s1="Java is a programming language. Java is a platform. Java is an Island.";
String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava
"
System.out.println(replaceString);
```

```
Kava is a programming language. Kava is a platform. Kava is an Island.
```

String comparision

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 Using equals() Method
2. By Using == Operator
3. By compareTo() Method

1) By Using equals() Method

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

```
String s1="Sachin";
String s2="Sachin";
String s3=new String("Sachin");
String s4="Saurav";
System.out.println(s1.equals(s2));//true
System.out.println(s1.equals(s3));//true
System.out.println(s1.equals(s4));//false
```

```
true
true
false
```

In the above code, two strings are compared using **equals()** method of **String** class. And the result is printed as boolean values, **true** or **false**.

```
String s1="Sachin";
String s2="SACHIN";

System.out.println(s1.equals(s2));//false
System.out.println(s1.equalsIgnoreCase(s2));//true
```

```
false
true
```

In the above program, the methods of **String** class are used. The **equals()** method returns true if String objects are matching and both strings are of same case. **equalsIgnoreCase()** returns true regardless of cases of strings.

2) By Using == operator

The == operator compares references not values.

```
String s1="Sachin";
String s2="Sachin";
String s3=new String("Sachin");
System.out.println(s1==s2);//true (because both refer to same instance)
System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
```

```
true
false
```

3) By Using compareTo() method

The String class 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 objects. If:

- **s1 == s2** : The method returns 0.
- **s1 > s2** : The method returns a positive value.
- **s1 < s2** : The method returns a negative value.

```
String s1="Sachin";
String s2="Sachin";
String s3="Ratan";
System.out.println(s1.compareTo(s2));//0
System.out.println(s1.compareTo(s3));//1(because s1>s3)
System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
```

```
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 concatenate strings in Java:

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

1) String Concatenation by + (String concatenation) operator

```
String s="Sachin"+" Tendulkar";
System.out.println(s);//Sachin Tendulkar
```

Output:

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

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

In Java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and it's 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 concatenate not only String but primitive values also. For Example:

```
String s=50+30+"Sachin"+40+40;
System.out.println(s);//80Sachin4040
1.
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:

```
  public String concat(String another)
String s1="Sachin ";
String s2="Tendulkar";
String s3=s1.concat(s2);
System.out.println(s3);//Sachin Tendulkar
```

```
Sachin Tendulkar
```

The above Java program, concatenates two String objects ***s1*** and ***s2*** using ***concat()*** method and stores the result into ***s3*** object.

There are some other possible ways to concatenate Strings in Java,

1. String concatenation using StringBuilder class

```
StringBuilder s1 = new StringBuilder("Hello");    //String 1
StringBuilder s2 = new StringBuilder(" World");    //String 2
StringBuilder s = s1.append(s2);   //String 3 to store the result
   System.out.println(s.toString());  //Displays result
```

```
Hello World
```

In the above code snippet, **s1, s2** and **s** are declared as objects of **StringBuilder** class. **s** stores the result of concatenation of **s1** and **s2** using **append**() method.

2. **String concatenation using format() method**

String.format() method allows to concatenate multiple strings using format specifier like %s followed by the string values or objects.

```
String s1 = new String("Hello");    //String 1
String s2 = new String(" World");    //String 2
String s = String.format("%s%s",s1,s2);   //String 3 to store the result
   System.out.println(s.toString());  //Displays result
```

```
Hello World
```

Here, the String objects **s** is assigned the concatenated result of Strings **s1** and **s2** using **String.format()** method. format() accepts parameters as format specifier followed by String objects or values.

3. String concatenation using String.join() method (Java Version 8+)

The String.join() method is available in Java version 8 and all the above versions. String.join() method accepts arguments first a separator and an array of String objects.

```
String s1 = new String("Hello");    //String 1
String s2 = new String(" World");    //String 2
String s = String.join("",s1,s2);   //String 3 to store the result
   System.out.println(s.toString()); //Displays result
```

```
Hello World
```

In the above code snippet, the String object **s** stores the result of **String.join("",s1,s2)** method. A separator is specified inside quotation marks followed by the String objects or array of String objects.

toString() Method

If you want to represent any object as a string, **toString() method** comes into existence.

The toString() method returns the String representation of the object.

If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.

Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

How to use the toString() method

The toString() method in Java has two implementations;

- The first implementation is when it is called as a method of an object instance. The example below shows this implementation

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

```
    //Creating an integer of value 10
    Integer number=10;
    // Calling the toString() method as a function of the Integer variable
    System.out.println( number.toString() );
}
```

Output

10

- The second implementation is when you call the member method of the relevant class by passing the value as an *argument*. The example below shows how to do this

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

        // The method is called on datatype Double
        // It is passed the double value as an argument
        System.out.println(Double.toString(11.0));
        // Implementing this on other datatypes

        //Integer
        System.out.println(Integer.toString(12));

        // Long
        System.out.println(Long.toString(123213123));

        // Booleam
        System.out.println(Boolean.toString(false));
    }
}
```

Output

11.0

12

123213123

false

Example 1: (write a program to count number of vowels in inputted string)

```
import java.io.*;
class vowel
{
	public static void main(String args[])throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	int cnt=0;
	System.out.println("enter string");
	String st=dis.readLine();
	for(int i=0;i<st.length();i++)
	{
		char t=st.charAt(i);
		if(t=='a' || t=='e' || t=='o' || t=='i' || t=='u' ||
		  t=='A' || t=='E' || t=='O' || t=='I' || t=='U')
			cnt++;
	}
	System.out.println("total vowels:-  "+cnt);
	}
}
```

```
enter string
khyati
total vowels:-  2
```

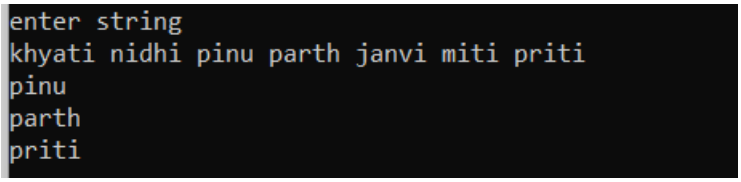
Example 2: (write a program to enter 5 string and sort it)

```
class sort1
{
	public static void main(String args[])
	{
	String st[]={"w","p","a","k","z"};
		for(int i=0;i<st.length;i++)
		{
		for(int j=i+1;j<st.length;j++)
		{
			if(st[i].compareTo(st[j])>0)
			{
				String t=st[i];
				st[i]=st[j];
				st[j]=t;
			}
```

```
                }
                }
                for(String t:st)
                {
                        System.out.println(t);
                }
        }
}
```

Example 3: (write a program to find words beginning with 'p' chatacter in inputted string)

```
import java.io.*;
class search1
{
        public static void main(String args[])throws IOException
        {
        DataInputStream dis=new DataInputStream(System.in);
        System.out.println("enter string");
        String st=dis.readLine();
        String t[]=st.split(" ");
        for(int i=0;i<t.length;i++)
        {
                if(t[i].startsWith("p"))
                        System.out.println(t[i]);
        }
        }
}
```



StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

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

Important Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16. StringBuffer s = **new** StringBuffer();
StringBuffer(String str)	It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation. StringBuffer s = **new** StringBuffer("khyati");
StringBuffer(int capacity)	It accepts an integer argument that explicitly sets the size of the buffer. StringBuffer s = **new** StringBuffer(20);

Important methods of StringBuffer class

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

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

What is a mutable String?

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

1) append()

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

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

```
System.out.println(sb);//prints Hello Java
} }
```

Output:

```
Hello Java
```

2) insert()

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

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

```
HJavaello
```

3) replace()

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

```
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
```

```
HJavalo
```

4) delete()

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

```
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
```

```
Hlo
```

5) reverse()

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

```
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
```

```
olleH
```

6) capacity()

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

```
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
```

```
16
16
34
```

6) length()

```
StringBuffer s = new StringBuffer("khyati");

// Getting the length of the string

int p = s.length();

System.out.println("Length of string="+ p);
```

Output: Length of string= 6

6) deleteCharAt()

```
StringBuffer s = new StringBuffer("khyatisolanki");

    s.deleteCharAt(7);

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

Output: khyatislanki

Difference between String and StringBuffer

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

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate the strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

Exception Handling in Java

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

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

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

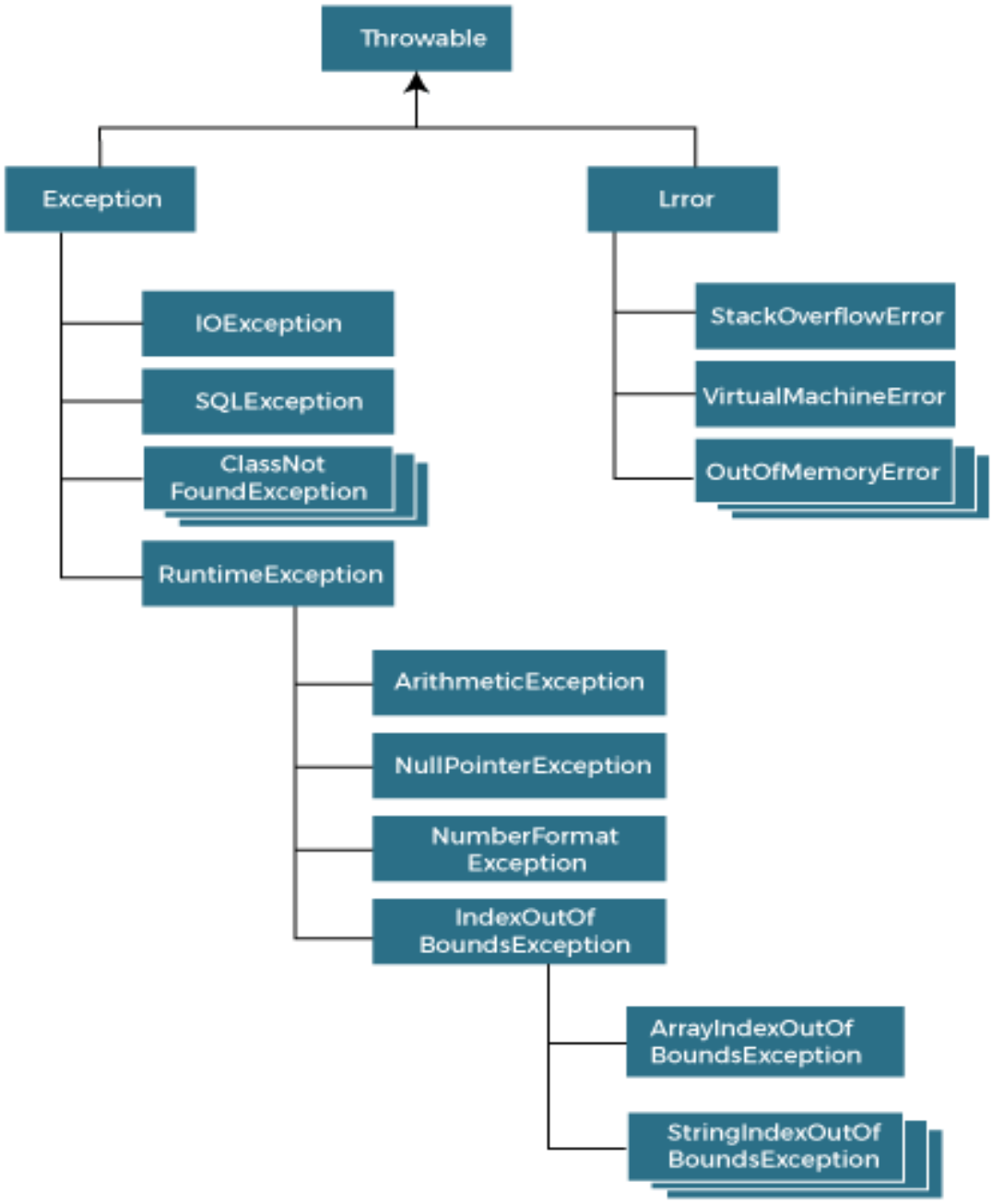
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

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

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

Hierarchy of Java Exception classes

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



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

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

Java Exception Handling Example

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

```
public class JavaExceptionExample
{
  public static void main(String args[])
{
   try
{
      //code that may raise exception
      int data=100/0;
   }
catch(ArithmeticException e)
{
System.out.println(e);
}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

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

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

Common Scenarios of Java Exceptions

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

1) A scenario where ArithmeticException occurs

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

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

2) A scenario where NullPointerException occurs

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

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

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

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

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

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

❖ Java try-catch block

Java try block

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

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

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

Syntax of Java try-catch

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

Syntax of try-finally block

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

Java catch block

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

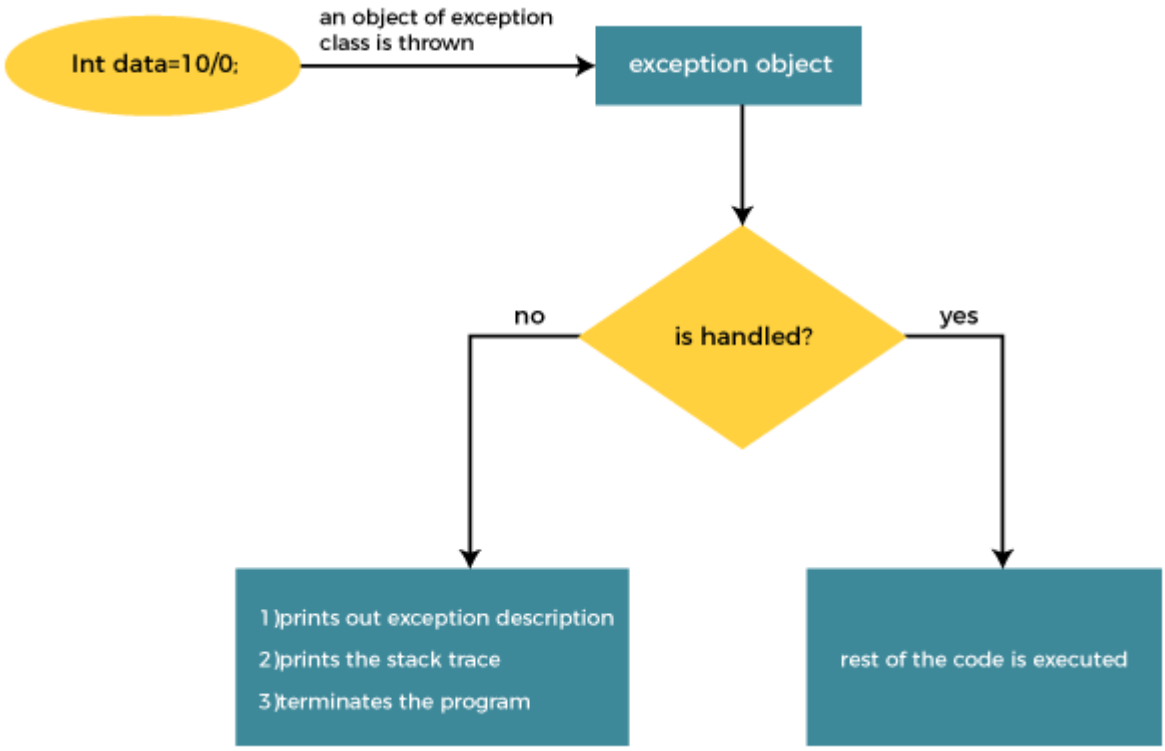
The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block

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

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

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



Problem without exception handling

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

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception
            System.out.println("rest of the code");
        }   }
```

Output:

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

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

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

```
public class TryCatchExample2 {
    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
           //handling the exception
        catch(ArithmeticException e)
        {
           System.out.println(e);
        }
        System.out.println("rest of the code");    }}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

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

Example 3

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

```
public class TryCatchExample3 {
    public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
                    // if exception occurs, the remaining statement will not exceute
      System.out.println("rest of the code");
      }
          // handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }       }  }
```

Output:

```
java.lang.ArithmeticException: / by zero
```

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

Example 4

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

```
public class TryCatchExample4 {

    public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
```

```
    // handling the exception by using Exception class
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Example 5

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

```
public class TryCatchExample5 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
                // displaying the custom message
            System.out.println("Can't divided by zero");
        }
    }
}
```

Output:

```
Can't divided by zero
```

Example 6

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

```
public class TryCatchExample6 {

    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output:

```
25
```

Example 7

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

```
public class TryCatchExample7 {
public static void main(String[] args) {
try
{
int data1=50/0; //may throw exception
}
```

```
            // handling the exception
        catch(Exception e)
        {
            // generating the exception in catch block
        int data2=50/0; //may throw exception
        }
        System.out.println("rest of the code");
        }
        }
```

Output:

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

Example 8

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

```
public class TryCatchExample8 {

    public static void main(String[] args) {
        try
        {
        int arr[]= {1,3,5,7};
        System.out.println(arr[10]); //may throw exception
        }
            // handling the array exception
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

❖ Java Catch Multiple Exceptions

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

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

Example 1

```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println(" Exception occurs");
             }
           System.out.println("rest of the code");       }}
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 2

```
public class MultipleCatchBlock2 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Example 3:

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

```
public class MultipleCatchBlock3 {

    public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
            System.out.println(a[10]);
           }
```

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

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 4

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

```
public class MultipleCatchBlock4 {

    public static void main(String[] args) {

        try{
            String s=null;
            System.out.println(s.length());
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
```

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

Output:

```
Parent Exception occurs
rest of the code
```

Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

MultipleCatchBlock5.java

```
class MultipleCatchBlock5{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

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

Output:

```
Compile-time error
```

❖ Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Why use nested try block

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

Syntax:

```
....
//main try block
try
{
   statement 1;
   statement 2;
//try catch block within another try block
   try
   {
      statement 3;
      statement 4;
//try catch block within nested try block
      try
      {
         statement 5;
         statement 6;
    }
      catch(Exception e2)
      {
//exception message
      }

   }
```

```
    catch(Exception e1)
    {
//exception message
    }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

Java Nested try Example

Example 1

Let's see an example where we place a try block within another try block for two different exceptions.

```
public class NestedTryBlock
{
 public static void main(String args[])
{
 //outer try block
  try
{
  //inner try block 1
    try
{
    System.out.println("going to divide by 0");
    int b =39/0;
   }
   //catch block of inner try block 1
   catch(ArithmeticException e)
   {
    System.out.println(e);
   }
```

```
    //inner try block 2
    try{
    int a[]=new int[5];

    //assigning the value out of array bounds
     a[5]=4;
     }

    //catch block of inner try block 2
    catch(ArrayIndexOutOfBoundsException e)
    {
       System.out.println(e);
    }

    System.out.println("other statement");
  }
  //catch block of outer try block
  catch(Exception e)
  {
    System.out.println("handled the exception (outer catch)");
  }

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

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

Example 2

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

```
public class NestedTryBlock2 {

   public static void main(String args[])
   {
      // outer (main) try block
      try {

         //inner try block 1
         try {

            // inner try block 2
            try {
               int arr[] = { 1, 2, 3, 4 };

               //printing the array element out of its bounds
               System.out.println(arr[10]);
            }

            // to handles ArithmeticException
            catch (ArithmeticException e) {
               System.out.println("Arithmetic exception");
               System.out.println(" inner try block 2");
            }
         }

         // to handle ArithmeticException
         catch (ArithmeticException e) {
```

```
            System.out.println("Arithmetic exception");
            System.out.println("inner try block 1");
        }
    }

    // to handle ArrayIndexOutOfBoundsException
    catch (ArrayIndexOutOfBoundsException e4) {
        System.out.print(e4);
        System.out.println(" outer (main) try block");
    }
    catch (Exception e5) {
        System.out.print("Exception");
        System.out.println(" handled in main try-block");
    }
  }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock2
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 outer
 (main) try block
```

❖ Java finally block

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

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Usage of Java finally

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

Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");  }
    System.out.println("rest of the code...");      }}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

Case 2: When an exception occurr but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

```
public class TestFinallyBlock1{
    public static void main(String args[]){

    try {

      System.out.println("Inside the try block");

      //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    //cannot handle Arithmetic type exception
    //can only accept Null Pointer type exception
    catch(NullPointerException e){
      System.out.println(e);
    }

    //executes regardless of exception occured or not
    finally {
      System.out.println("finally block is always executed");
    }

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

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Case 3: When an exception occurs and is handled by the catch block

Example:

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

```
public class TestFinallyBlock2{
    public static void main(String args[]){
    try {
        System.out.println("Inside try block");
        //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    //handles the Arithmetic Exception / Divide by zero exception
    catch(ArithmeticException e){
      System.out.println("Exception handled");
      System.out.println(e);
    }
    //executes regardless of exception occured or not
    finally {
      System.out.println("finally block is always executed");
    }
    System.out.println("rest of the code...");
    }    }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

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

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

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

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

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

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

```
public class TestThrow1
 {
   //function to check if person is eligible to vote or not
   public static void validate(int age)
{
       if(age<18)
     {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
       else
     {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[])
{
       //calling the function
       validate(13);
       System.out.println("rest of the code...");
  }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

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

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Which exception should be declared?

Ans: Checked exception only, because:

- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Example 1:

```
public class TestThrows
 {
   //defining a method
   public static int divideNum(int m, int n) throws ArithmeticException {
      int div = m / n;
      return div;
   }
```

```
    //main method
    public static void main(String[] args) {
        TestThrows obj = new TestThrows();
        try {
            System.out.println(obj.divideNum(45, 0));
        }
        catch (ArithmeticException e){
            System.out.println("\nNumber cannot be divided by 0");
        }

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

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrows

Number cannot be divided by 0
Rest of the code..
```

Difference between throw and throws keyword:

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception handled	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.

		exception cannot be propagated using throw only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Example of throw and throws

```
public class TestThrowAndThrows
{
   // defining a user-defined method
   // which throws ArithmeticException
   static void method() throws ArithmeticException
   {
      System.out.println("Inside the method()");
      throw new ArithmeticException("throwing ArithmeticException");
   }
   //main method
   public static void main(String args[])
   {
      try
      {
         method();
      }
```

```
        catch(ArithmeticException e)
        {
           System.out.println("caught in main() method");
        }
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows
Inside the method()
caught in main() method
```

Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

:Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.

3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.

Java Custom Exception

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Using the custom exception, we can have your own exception and message.

Why use custom exceptions?

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Example 1 (Custom Exception)

```
import java.io.*;

class uexc1
{
	public static void main(String args[]) throws IOException
	{
	try
	{

	DataInputStream dis=new DataInputStream(System.in);
	System.out.println("ENTER No:-");
	int n= Integer.parseInt(dis.readLine());

	if(n<0)
	{
		throw new myexp();
	}
	System.out.println("your no is accepted...");
	}

	catch(myexp e)
	{
		System.out.println(e);
	}

	catch(IOException e1)
	{
		System.out.println(e1);
	}

	catch(Exception ex)
	{
		System.out.println(ex);
	}
}
}
```

```
class myexp extends Exception
{
	public String toString()
	{
		return("number should be +ve");

	}
}
```

```
ENTER No:-
-5
number should be +ve
```

Example 2 (Custom Exception)

```
import java.io.*;

class uexc2
{
	public static void main(String args[]) throws IOException,myexp
	{
	DataInputStream dis=new	DataInputStream(System.in);
	System.out.println("ENTER No:-");
	int n= Integer.parseInt(dis.readLine());
	if(n<0)
		throw new myexp();
	System.out.println("your no is accepted...");
	}

}
class myexp extends Exception
{
	public String toString()
	{
		return("number should br +ve");

	}
}
```

```
ENTER No:-
9
your no is accepted...

C:\Users\Admin\OneDrive\Desktop\java\Khyati_Uac_Java\unit-2\java_prog_unit2>java uexc2
ENTER No:-
-7
Exception in thread "main" number should br +ve
        at uexc2.main(uexc2.java:10)
```

Example 1 (in-built exception)

```
import java.io.*;

class exc1
{
public static void main(String args[])
{
        try
        {
        DataInputStream dis=new DataInputStream(System.in);
        System.out.println("ENTER 2 NO:-");
        int x= Integer.parseInt(dis.readLine());
        int y= Integer.parseInt(dis.readLine());
        int ans=x/y;
        System.out.println("ans is"+ans);
        }
        catch(NumberFormatException ex)
        {
                System.out.println(ex);

        }
        catch(ArithmeticException ex)
        {
                System.out.println(ex);
        }
        catch(IOException ex)
        {
                System.out.println(ex);
        }
        catch(Exception ex)
        {
```

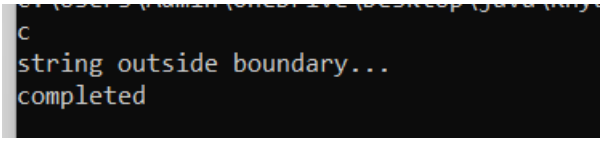
```
            System.out.println(ex);
        }
        finally
        {
            System.out.println("completed");
        }
}
}
```

```
C:\Users\Admin\OneDrive\Desktop\java\Khyati_Uac_Java\unit-2\java_prog_unit2>java exc1
ENTER 2 NO:-
3
4
ans is0
completed

C:\Users\Admin\OneDrive\Desktop\java\Khyati_Uac_Java\unit-2\java_prog_unit2>java exc1
ENTER 2 NO:-
5
0
java.lang.ArithmeticException: / by zero
completed

C:\Users\Admin\OneDrive\Desktop\java\Khyati_Uac_Java\unit-2\java_prog_unit2>java exc1
ENTER 2 NO:-
f5
java.lang.NumberFormatException: For input string: "f5"
completed
```

Example 2 (in-built exception)

```
class exc3
{
public static void main(String args[])
{
        try
        {
        String st="abcd";
        System.out.println(st.charAt(2));
        System.out.println(st.charAt(6));
        }
        catch(StringIndexOutOfBoundsException ex)
        {
            System.out.println("string outside boundary...");

        }
```

```
        catch(Exception ex)
        {
                System.out.println(ex);
        }
        finally
        {
                System.out.println("completed");
        }
}
}
```



Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

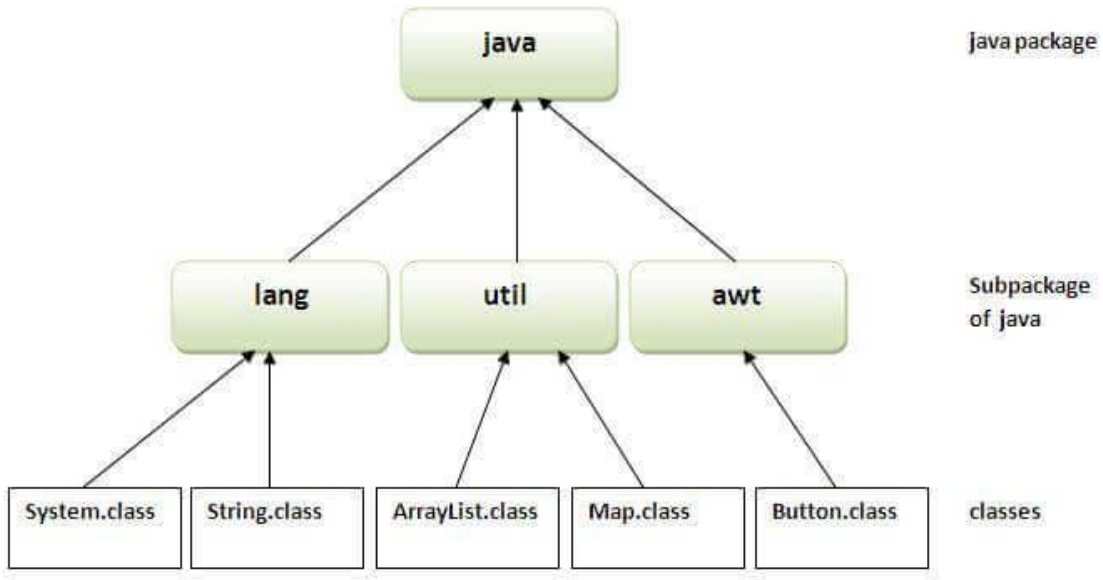
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

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

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[])
{
   System.out.println("Welcome to package");
  }
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

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

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

*1) Using packagename.**

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

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

```
///save by B.java

package mypack;
import pack.*;

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

```
Output:Hello
```

2) Using packagename.classname

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

Example of package by import package.classname

```
        /save by A.java

    package pack;
    public class A{
      public void msg(){System.out.println("Hello");}
    }

//save by B.java
package mypack;
import pack.A;

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

```
Output:Hello
```

3) Using fully qualified name

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

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

Example of package by import fully qualified name

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

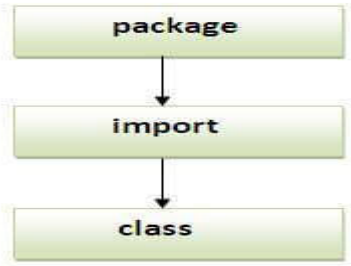
```
//save by B.java
package mypack;
class B
{
  public static void main(String args[])
{
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package

e.g. com.java1.bean or org.sssit.dao.

Example of Subpackage

```
package com.java1.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

To Compile: javac -d . Simple.java

To Run: java com.java1.core.Simple

```
output:Hello subpackage
```

Example

```
//save with MathUtill.java (Compile)

package mypack1;
public class MathUtill
{
        public static int square(int n)
        {
                        return n*n;
        }
```

```
	public static int cube(int n)
	{
			return n*n*n;
	}
}
```

//save with Customer.java (Compile)

```
package mypack1;
import java.io.*;
public class Customer
{
	String nm;
	int amt;

	public void get()throws IOException
	{
	DataInputStream dis=new DataInputStream(System.in);
	System.out.println("enter name,amt:-");
	nm=dis.readLine();
	amt=Integer.parseInt(dis.readLine());
	}

	public void show()
	{
		System.out.println("name:-"+nm);
		System.out.println("amt:-"+amt);
	}
}
```

//save with P1.java (Compile and run)

```
import mypack1.*;
import java.io.*;
class P1
{
	public static void main(String args[])throws IOException
	{
		customer c=new customer();
		c.get();//Non-Static method call
		c.show();//Non-Static method call
		c.nm // non-public property of Customer class
				//so not allow to access
```

```
            System.out.println(mathutill.square(2)); //Static method call
            System.out.println(mathutill.cube(2)); //Static method call
        }
}
```

Compile MathUtill class

javac -d . MathUtill.java (creating .class file Mathutill.class in mypack1 package)

Compile Customer class

javac -d . Customer.java (creating .class file Customer.class in mypack1 package)

Compile p1 class which is outside mypack1 package

javac p1.java

Run p1 class which is outside mypack1 package

java p1

```
enter name,amt:-
khyati
6000
name:-khyati
amt:-6000
4
8
```

THREADING IN JAVA

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java.

A **thread** is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavy-weight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

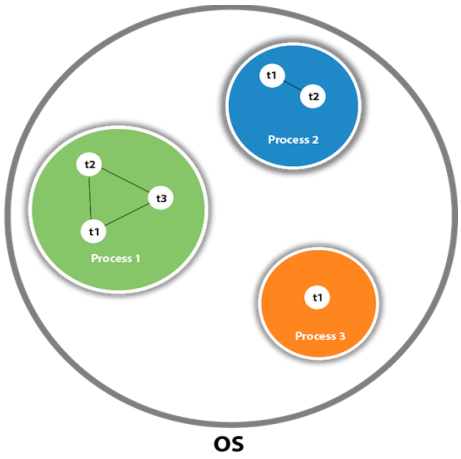
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

❖ What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the os, and one process can have multiple threads.

Note: At a time one thread is executed only.

❖ Difference between Thread and Process

Process	Thread
A process is an instance of a program that is being executed or processed.	Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory.

Each process is treated as a new process by the operating system.	The operating system takes all the user-level threads as a single process.
If one process gets blocked by the operating system, then the other process can continue the execution.	If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process.
Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fast because they are very lightweight.
The data segment and code segment of each process are independent of the other.	Threads share data segment and code segment with their peer threads; hence are the same for other threads also.
The operating system takes more time to terminate a process.	Threads can be terminated in very little time.
New process creation is more time taking as each new process takes all the resources.	A thread needs less time for creation.

❖ Life cycle of a Thread (Thread States)

Life Cycle of Thread in Java is basically state transitions of a thread that starts from its birth and ends on its death.

When an instance of a thread is created and is executed by calling start() method of Thread class, the thread goes into runnable state.
When sleep() or wait() method is called by Thread class, the thread enters into non-runnable state.

From non-runnable state, thread comes back to runnable state and continues execution of statements. When the thread comes out of run() method, it dies. These state transitions of a thread are called **Thread life cycle in Java**.

Thread States in Java

A thread is a path of execution in a program that enters in any one of the following five states during its life cycle. The five states are as follows:

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead

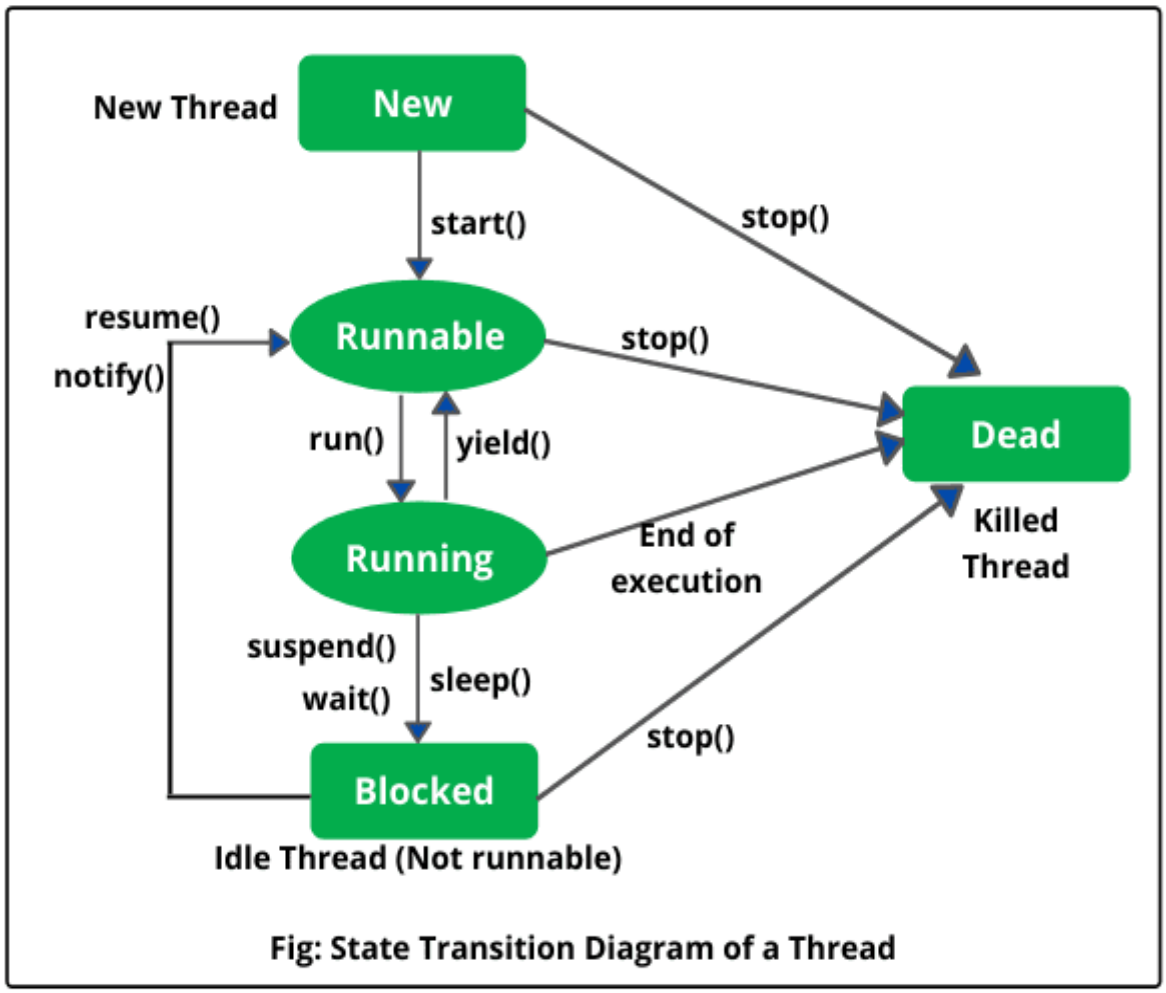
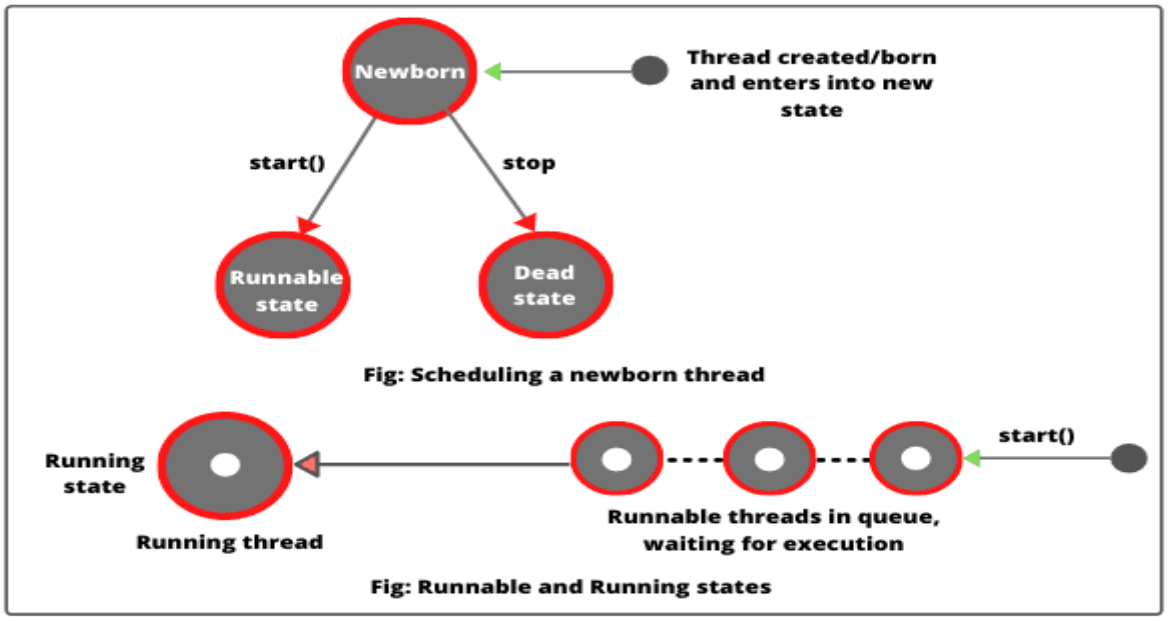


Fig: State Transition Diagram of a Thread

1. **New (Newborn State):** When we create a thread object using Thread class, thread is born and is known to be in Newborn state. That is, when a thread is born, it enters into new state the start() method has not been called yet on the instance.



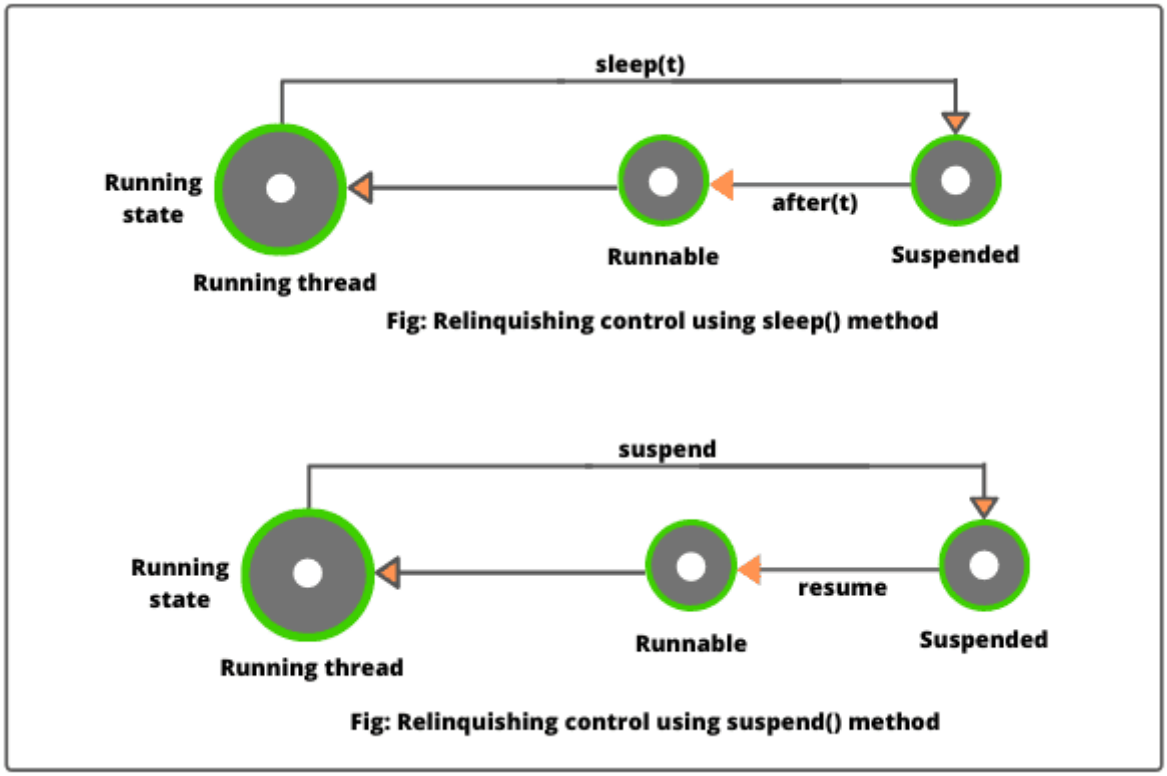
In other words, Thread object exists but it cannot execute any statement because it is not an execution of thread. Only start() method can be called on a new thread; otherwise, an **IllegalThreadStateException** will be thrown.

2. Runnable state: Runnable state means a thread is ready for execution. When the start() method is called on a new thread, thread enters into a runnable state.

In runnable state, thread is ready for execution and is waiting for availability of the processor (CPU time). That is, thread has joined queue (line) of threads that are waiting for execution. If all threads have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner. The process of allocating time to threads is known as **time slicing**. A thread can come into runnable state from running, waiting, or new states.

3. Running state: Running means Processor (CPU) has allocated time slot to thread for its execution. When thread scheduler selects a thread from the runnable state for execution, it goes into running state. Look at the above figure.
In running state, processor gives its time to the thread for execution and executes its run method. This is the state where thread performs its actual functions. A thread can come into running state only from runnable state.

A running thread may give up its control in any one of the following situations and can enter into the blocked state.



1. When sleep() method is invoked on a thread to sleep for specified time period, the thread is out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.

2. When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method.

3. When wait() method is called on a thread to wait for some time. The thread in wait state can be run again using notify() or notifyAll() method.

4. Blocked state: A thread is considered to be in the blocked state when it is suspended, sleeping, or waiting for some time in order to satisfy some condition.

5. Dead state: A thread dies or moves into dead state automatically when its run() method completes the execution of statements. That is, a thread is terminated or dead when a thread comes out of run() method. A thread can also be dead when the stop() method is called.

During the life cycle of thread in Java, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads are executing, only one thread can use CPU at a time.

All other threads live in some other states, either waiting for their turn on CPU or waiting for satisfying some conditions. Therefore, a thread is always in any of the five states.

❖ How to create a thread in Java

There are two ways to create a thread:

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

1. 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 void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.

5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

2. 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().

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

Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs the 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) Java Thread Example by extending Thread class

```
class Multi extends Thread
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();
t1.start();
 }
}
```

Output:

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable
{
public void run()
{
System.out.println("thread is running...");
}

public static void main(String args[])
{
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
 }
```

Output:

```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

```
public class MyThread1
{
// main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");
// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}  }
```

```
My first thread
```

4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

```
public class MyThread2 implements Runnable
{
public void run()
{
System.out.println("Now the thread is running ...");
}
// main method
public static void main(String argvs[])
{
// creating an object of the class MyThread2
Runnable r1 = new MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");

// the start() method moves the thread to the active state
th1.start();
```

```
// getting the thread name by invoking the getName() method
String str = th1.getName();
System.out.println(str);
}
}
```

Output:

```
My new thread
Now the thread is running ...
```

Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones.

There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread.sleep()

The Java Thread class provides the two variant of the sleep() method. First one accepts only an one arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

The sleep() Method Syntax:

Following are the syntax of the sleep() method.

1. **public static void** sleep(**long** mls) **throws** InterruptedException
2. **public static void** sleep(**long** mls, **int** n) **throws** InterruptedException

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is

accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

Parameters:

The following are the parameters used in the sleep() method.

mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

Example of the sleep() method in Java : on the custom thread

```
class TestSleepMethod1 extends Thread
{
    public void run()
    {
        for(int i=1;i<5;i++)
        {
            // the thread will sleep for the 500 milli seconds
                try
                {
                Thread.sleep(500);
                }
            catch(InterruptedException e)
```

```
                                        {
                                        System.out.println(e);
                                        }
                System.out.println(i);
            }
        }
        public static void main(String args[])
        {
         TestSleepMethod1 t1=new TestSleepMethod1();
         TestSleepMethod1 t2=new TestSleepMethod1();

         t1.start();
         t2.start();
        }
}
```

Output:

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

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Example of the sleep() Method in Java : on the main thread

```
// important import statements
import java.lang.Thread;
import java.io.*;

public class TestSleepMethod2
{
   // main method
public static void main(String argvs[])
{
```

```
try
 {
for (int j = 0; j < 5; j++)
{
// The main thread sleeps for the 1000 milliseconds, which is 1 sec
// whenever the loop runs
Thread.sleep(1000);

// displaying the value of the variable
System.out.println(j);
}
}
catch (Exception expn)
{
// catching the exception
System.out.println(expn);  } }  }
```

Output:

```
0
1
2
3
4
```

Example of the sleep() Method in Java: When the sleeping time is -ive

when the time for sleeping is negative it throws IllegalArguementException

```
Thread.sleep(-100);
```

Output : java.lang.IllegalArgumentException: timeout value is negative

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

```
public class TestThreadTwice1 extends Thread
{
 public void run()
{
   System.out.println("running...");
 }
 public static void main(String args[])
{
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

Output:

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

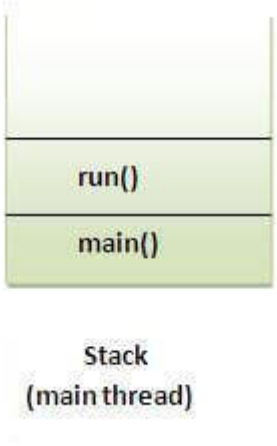
What if we call Java run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
class TestCallRun1 extends Thread
{
 public void run()
{
   System.out.println("running...");
 }
 public static void main(String args[])
{
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```

Output:

```
running...
```



Problem if you direct call run() method

```
class TestCallRun2 extends Thread
{
 public void run()
{
  for(int i=1;i<5;i++)
{

    try
{
Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[])
{
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();

  t1.run();
  t2.run();  }
}
```

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

As we can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

Java join() method

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution.

Suppose *th* be the object the class Thread whose thread is doing its execution currently, then the *th.join();* statement ensures that *th* is finished before the program does the execution of the next statement.

When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the developer or programmer to mention the waiting period.

However, similar to the sleep() method in Java, the join() method is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters. The following are the three overloaded join() methods.

Description of The Overloaded join() Method

1. **join():** When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

Syntax:

```
public final void join() throws InterruptedException
```

2. **join(long mls):** When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds) is over.

Syntax:

```
public final synchronized void join(long mls) throws InterruptedException, where m
ls is in milliseconds
```

3. **join(long mls, int nanos):** When the join() method is invoked, the current thread stops its execution and go into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

Syntax:

1. **public final synchronized void** join(**long** mls, **int** nanos) **throws** InterruptedExceptio n, where mls is in milliseconds.

Example of join() Method in Java

```
// A Java program for understanding  the joining of threads
  // import statement
import java.io.*;

// The ThreadJoin class is the child class of the class Thread

class ThreadJoin extends Thread
{
// overriding the run method
public void run()
{
for (int j = 0; j < 2; j++)
{
try
{
// sleeping the thread for 300 milli seconds
Thread.sleep(300);
System.out.println("The current thread name is: " + Thread.currentThread().getName());
}
// catch block for catching the raised exception
catch(Exception e)
{
System.out.println("The exception has been caught: " + e);
}
System.out.println( j );
}
}
}

public class ThreadJoinExample
{
// main method
public static void main (String argvs[])
{
// creating 3 threads
```

```
ThreadJoin th1 = new ThreadJoin();
ThreadJoin th2 = new ThreadJoin();
ThreadJoin th3 = new ThreadJoin();

// thread th1 starts
th1.start();

// starting the second thread after when
// the first thread th1 has ended or died.
try
{
System.out.println("The current thread name is: "+ Thread.currentThread().getName());

// invoking the join() method
th1.join();
}

// catch block for catching the raised exception
catch(Exception e)
{
System.out.println("The exception has been caught " + e);
}

// thread th2 starts
th2.start();

// starting the th3 thread after when the thread th2 has ended or died.
try
{
System.out.println("The current thread name is: " + Thread.currentThread().getName());
th2.join();
}

// catch block for catching the raised exception
catch(Exception e)
{
System.out.println("The exception has been caught " + e);
}
// thread th3 starts
th3.start();  }  }
```

Output:

```
The current thread name is: main
The current thread name is: Thread - 0
0
The current thread name is: Thread - 0
1
The current thread name is: main
The current thread name is: Thread - 1
0
The current thread name is: Thread - 1
1
The current thread name is: Thread - 2
0
The current thread name is: Thread - 2
1
```

Explanation: The above program shows that the second thread th2 begins after the first thread th1 has ended, and the thread th3 starts its work after the second thread th2 has ended or died.

Some More Examples of the join() Method

```
class TestJoinMethod1 extends Thread
{
 public void run()
{
  for(int i=1;i<=5;i++)
{
   try
  {
    Thread.sleep(500);
   }
catch(Exception e)
{
System.out.println(e);
}
  System.out.println(i);
  }
 }
public static void main(String args[])
{
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
```

```
 try
{
  t1.join();
 }
catch(Exception e)
{
System.out.println(e);
}
 t2.start();
 t3.start();
 }
}
```

Output:

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

We can see in the above example, when t1 completes its task then t2 and t3 starts executing.

join(long miliseconds) Method Example

```
class TestJoinMethod2 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }    }
```

```
    public static void main(String args[]){
     TestJoinMethod2 t1=new TestJoinMethod2();
     TestJoinMethod2 t2=new TestJoinMethod2();
     TestJoinMethod2 t3=new TestJoinMethod2();
     t1.start();
     try{
      t1.join(1500);
     }catch(Exception e){System.out.println(e);}

     t2.start();
     t3.start();
     }
    }
```

Output:

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

In the above example, when t1 completes its task for 1500 milliseconds(3 times), then t2 and t3 start executing.

Naming Thread and Current Thread

❖ Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name, i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using the setName() method. The syntax of setName() and getName() methods are given below:

1. **public** String getName(): is used to **return** the name of a thread.
2. **public void** setName(String name): is used to change the name of a thread.

We can also set the name of a thread directly when we create a new thread using the constructor of the class.

Example of naming a thread : Using setName() Method

```
class TestMultiNaming1 extends Thread
{
  public void run()
{
   System.out.println("running...");
  }
 public static void main(String args[])
{
  TestMultiNaming1 t1=new TestMultiNaming1();
  TestMultiNaming1 t2=new TestMultiNaming1();

 System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());

  t1.start();
  t2.start();

  t1.setName("Sonoo Jaiswal");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

Output:

```
Name of t1:Thread-0
Name of t2:Thread-1
After changing name of t1:Sonoo Jaiswal
running...
running...
```

Example of naming a thread : Without Using setName() Method

One can also set the name of a thread at the time of the creation of a thread, without using the setName() method. Observe the following code.

```
// A Java program that shows how one can set the name of a thread at the time  of creation
 of the thread
  // import statement
import java.io.*;
 // The ThreadNameClass is the child class of the class Thread
class ThreadName extends Thread
{
  // constructor of the class
ThreadName(String threadName)
{
// invoking the constructor of  the superclass, which is Thread class.
super(threadName);
}



// overriding the method run()
public void run()
{
System.out.println(" The thread is executing....");
}
}

public class ThreadNamingExample
{
// main method
public static void main (String argvs[])
{
// creating two threads and settting their name  using the contructor of the class
ThreadName th1 = new ThreadName("Java1");
ThreadName th2 = new ThreadName("Java2");

// invoking the getName() method to get the names  of the thread created above
System.out.println("Thread - 1: " + th1.getName());
System.out.println("Thread - 2: " + th2.getName());

  // invoking the start() method on both the threads
th1.start();
th2.start();
}  }
```

Output:

```
Thread - 1: Java1
Thread - 2: Java2
 The thread is executing....
 The thread is executing....
```

❖ Current Thread

The currentThread() method returns a reference of the currently executing thread.

```
public static Thread currentThread()
```

Example of currentThread() method

```
class TestMultiNaming2 extends Thread
{
 public void run()
{
  System.out.println(Thread.currentThread().getName());
 }
 public static void main(String args[])
{
  TestMultiNaming2 t1=new TestMultiNaming2();
  TestMultiNaming2 t2=new TestMultiNaming2();
  t1.start();
  t2.start();
 }
}
```

Output:

```
Thread-0
Thread-1
```

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
public class ThreadPriorityExample extends Thread
{
  // Method 1
// Whenever the start() method is called by a thread  the run() method is invoked
public void run()
{
// the print statement
System.out.println("Inside the run() method");
}

// the main method
public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

// We did not mention the priority of the thread.  Therefore, the priorities of the thread is 5
, the default value

// 1st Thread
// Displaying the priority of the thread  using the getPriority() method
```

```
System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 2nd Thread
// Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 3rd Thread
// // Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
// Setting priorities of above threads by  passing integer arguments
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);

// 6
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
// 3
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
// 9
System.out.println("Priority of the thread th3 is : " + th3.getPriority());

// Main thread
// Displaying name of the currently executing thread
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getNa
me());

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority(
));

// Priority of the main thread is 10 now
Thread.currentThread().setPriority(10);

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority(
));
}
}
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
```

```
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler.

Note: If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

Example of IllegalArgumentException

We know that if the value of the parameter *newPriority* of the method getPriority() goes out of the range (1 to 10), then we get the IllegalArgumentException

```
//priority of the thread is set to 17, which is greater than 10
Thread.currentThread().setPriority(17);
```

```
on in thread "main" java.lang.IllegalArgumentException
        at java.base/java.lang.Thread.setPriority(Thread.java:1141)
        at IllegalArgumentException.main(IllegalArgumentException.java:12)
```

yield() method

The **yield()** method of thread class causes the currently executing thread object to temporarily pause and allow other threads to execute.

Syntax

```
public static void yield()
```

Example

```
public class JavaYieldExp extends Thread
{
   public void run()
   {
      for (int i=0; i<3 ; i++)
         System.out.println(Thread.currentThread().getName() + " in control");
   }
```

```
    public static void main(String[]args)
    {
        JavaYieldExp t1 = new JavaYieldExp();
        JavaYieldExp t2 = new JavaYieldExp();
        // this will call run() method
        t1.start();
        t2.start();
        for (int i=0; i<3; i++)
        {
            // Control passes to child thread
            t1.yield();
            System.out.println(Thread.currentThread().getName() + " in control");
        }
    }
}
```

Output:

```
main in control
main in control
main in control
Thread-0 in control
Thread-0 in control
Thread-0 in control
Thread-1 in control
Thread-1 in control
Thread-1 in control
```

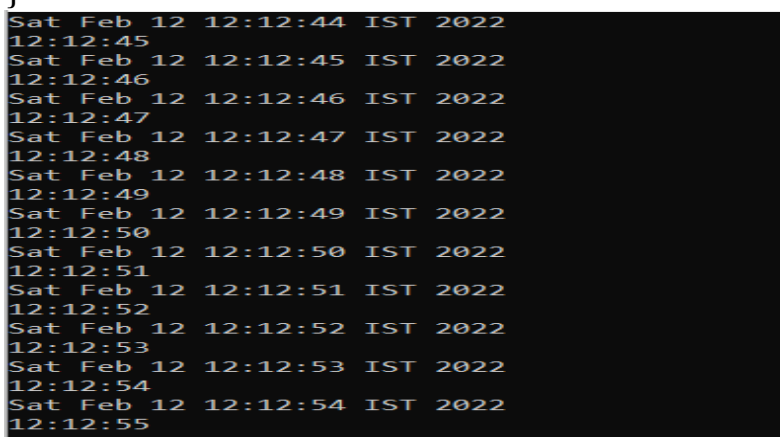
Example 1 : (print 1-1 char from user inputted string at 1000 milisecond interval)

```
class sthd
{
    public static void main(String args[])
    {
        try
        {
            String st="KHYATI";
            for(int i=0;i<st.length();i++)
            {
                char t=st.charAt(i);
                System.out.println(t);
                Thread.sleep(1000);
            }
        }
```

```
            catch(InterruptedException ex)
            {
                System.out.println(ex);
            }
        }
}
```

Example 2 : (print current date-time at 1000 milisecond interval)

```
import java.util.*;
class sthd2
{
        public static void main(String args[])
        {
            try
            {
                while(true)
                {
                    Date d=new Date();
        System.out.println(d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds());
                    System.out.println(d);
                    Thread.sleep(1000);
                }
            }
            catch(InterruptedException ex)
            {
                System.out.println(ex);
            }
        }
}
```



Example 3 : (multi threaded prog. print 1 to 5 and 10 to 15 using thread class)

```
class thd2 extends Thread
{
	String st;
	int p;

	thd2(String st, int p)
	{
		this.st=st;
		this.p=p;
		setName(st);
		setPriority(p);
		start();
	}

	public void run()
	{
		try
		{
			for(int i=1;i<=5;i++)
			{
				System.out.println("i=" + i +"" + getName());
				Thread.sleep(100);
			}
		}
		catch(InterruptedException ex)
		{
			System.out.println(ex);
		}
	}
}

class thd1
{
	public static void main(String args[])
	{
		try
		{
			thd2 t1=new thd2("A",3);
			thd2 t2=new thd2("B",7);
			thd2 t3=new thd2("C",9);

			for(int j=10;j<=15;j++)
			{
				System.out.println("main"+j);
				Thread.sleep(500);
			}
```

```
            }
            catch(InterruptedException ex)
            {
                System.out.println(ex);
            }
        }
}
```

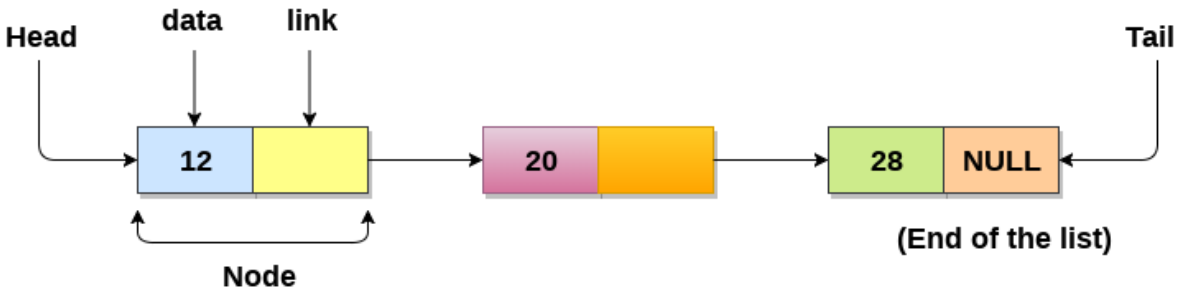
Example 4 : (multi threaded prog. print 1 to 5 and 10 to 15 using runnable interface)

```
class thd4 implements Runnable
{
        Thread t1= new Thread(this);
        String st;
        int p;
        thd4(String st, int p)
        {
            this.st=st;
            this.p=p;
            t1.setName(st);
            t1.setPriority(p);
            t1.start();
        }
        public void run()
        {
            try
            {
                for(int i=1;i<=5;i++)
                {
                    System.out.println("i=" + i +"" + t1.getName());
                    Thread.sleep(100);
                }
            }
            catch(InterruptedException ex)
            {
                System.out.println(ex);
            }
        }
}

class thd3
{
        public static void main(String args[])
        {
            try
            {
                thd4 t1=new thd4("A",3);
```

```
                thd4 t2=new thd4("B",7);
                thd4 t3=new thd4("C",9);
                for(int j=10;j<=15;j++)
                {
                        System.out.println("main"+j);
                        Thread.sleep(500);
                }
        }
        catch(InterruptedException ex)
        {
                System.out.println(ex);
        }
    }
}
```

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

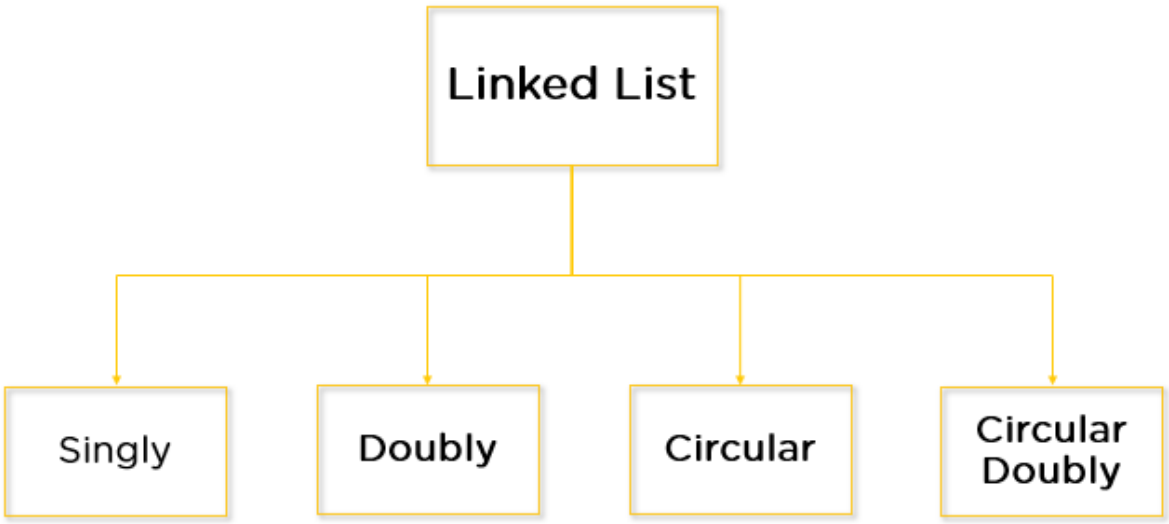
1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.

3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

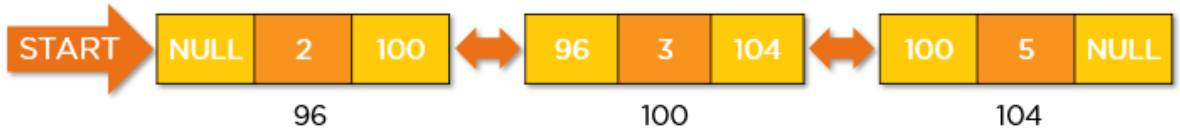
Types Of Link List



What is a Singly Linked List?

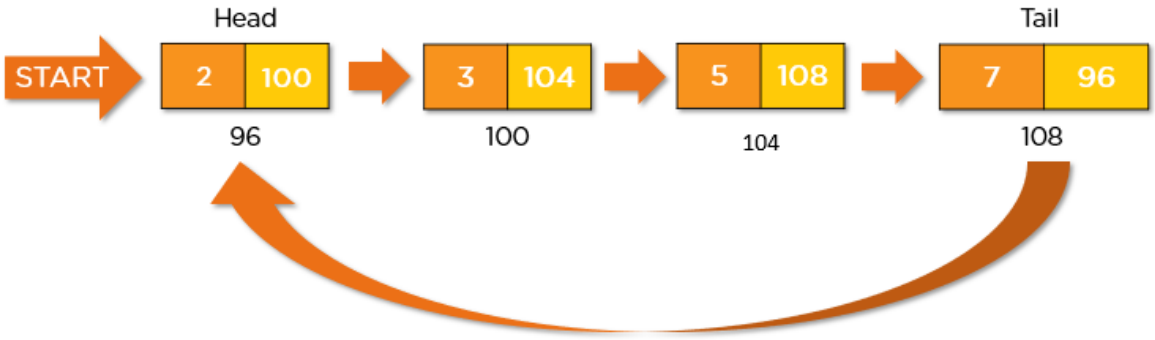
A singly linked list is a unidirectional linked list. So, you can only traverse it in one direction, i.e., from head node to tail node.

What is a Doubly Linked List?



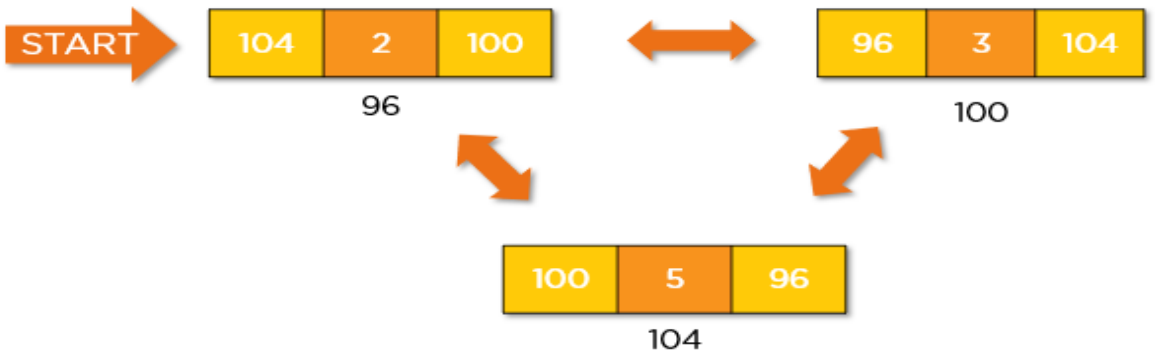
A doubly linked list is a bi-directional linked list. So, you can traverse it in both directions. Unlike singly linked lists, its nodes contain one extra pointer called the previous pointer. This pointer points to the previous node.

What is a Circular Linked List?



A circular Linked list is a unidirectional linked list. So, you can traverse it in only one direction. But this type of linked list has its last node pointing to the head node. So while traversing, you need to be careful and stop traversing when you revisit the head node.

What is a Circular Doubly Linked List?



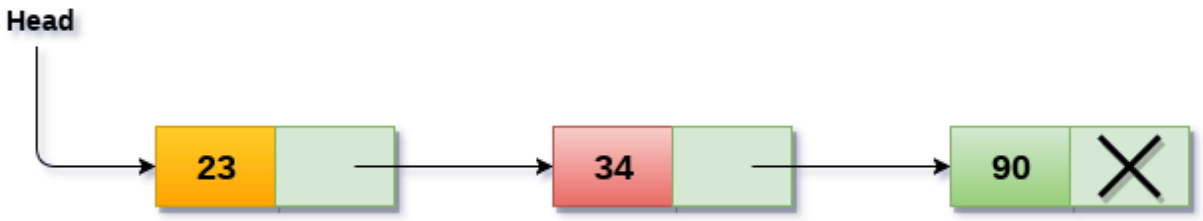
A circular doubly linked list is a mixture of a doubly linked list and a circular linked list. Like the doubly linked list, it has an extra pointer called the previous pointer, and similar to the circular linked list, its last node points at the head node. This type of linked list is the bi-directional list. So, you can traverse it in both directions.

1. Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.

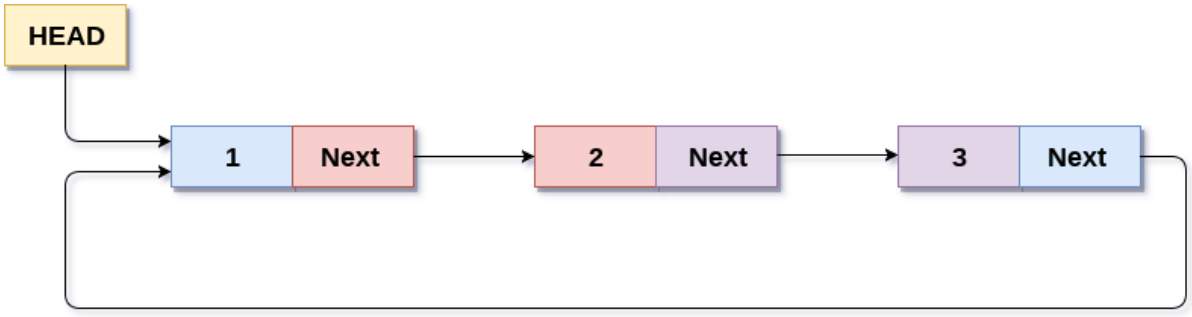
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

2. Circular Singly Linked Lis

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



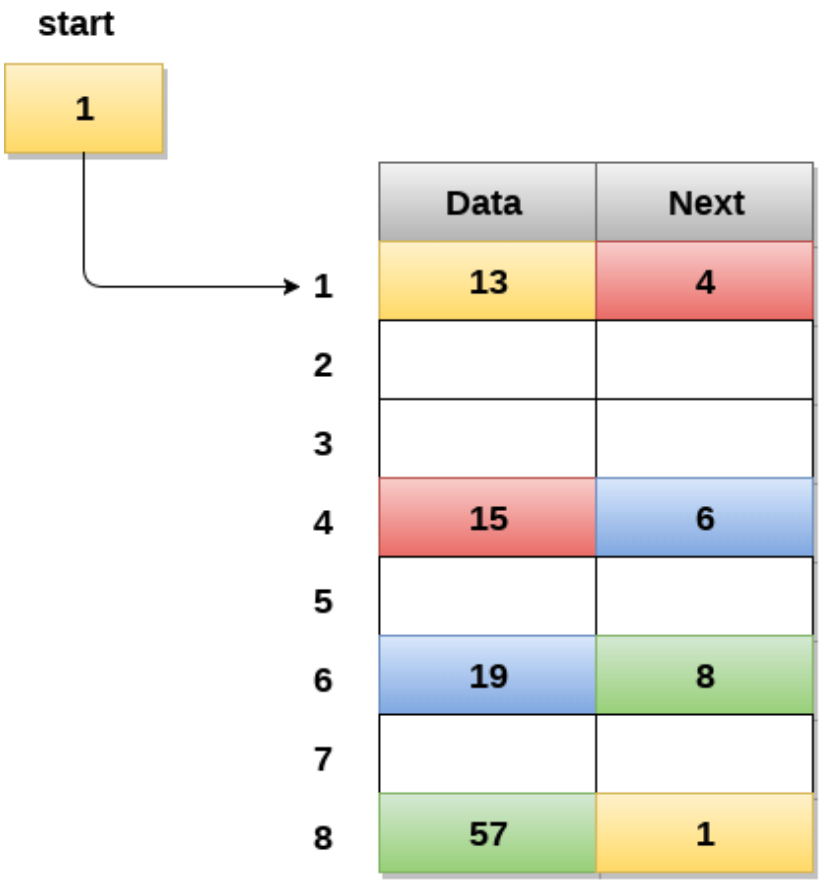
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can

find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Array vs. Link List

Array	Linked list
An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address.
Array elements store in a contiguous memory location.	Linked list elements can be stored anywhere in the memory or randomly stored.
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list takes less time while performing any operation like insertion, deletion, etc.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement.