CORE JAVA

Praveen

INetSolv Technologies

SecondFloor, SwathiManners,

Ameerpet,Hyderabad

Cell: 9849688861

www.inetsolv.info

mail: admin@inetsolv.info

Index

Introduction to Java

Program: A program is a set of instructions, which are executed by a machine to reduce the burden of a user or a human being by performing the operations faster without any mistakes.

Software: It is a set of programs, which can perform multiple tasks.

The softwares are classified into two types:

1) **System software:** The softwares which are designed to interact or communicate with the hardware devices, to make them work are called as system software. These softwares are generally developed in languages like C, C++ etc.
   Example: Operating System, drivers, compilers etc.

2) **Application Software:** The softwares which are called designed to store data provide entertainment, process data, do business, generate reports etc. are called as application software. These softwares are generally developed in languages like java, .net, etc.
   The application softwares are further classified into two types.
   i. **Stand Alone Software:** The software which can execute in the context of a single machine are called as standalone software.
      Example: MS-Word, media player, etc.
   ii. **Web Based Software:** This software can execute on any machine in the context of a browser are called as web based software.
      Example: Gmail, Facebook, etc

The java language is released by SUN Micro Systems in the year 1995 in three editions:

1) **JSE (Java Standard Edition):**
   This edition can be used for developing stand alone software.
2) **JEE (Java Enterprise Edition):**
   This edition can be used for developing web based software.
3) **JME (Java Mobile Edition):**
   This edition can be used for developing applications for mobile devices, wireless devices, embedded controllers etc, where memory is limited.

Features of Java

The features of any programming language are the services or the facilities provided by that language. The various features of java language are:

1) **Simple:** The java language is called as simple programming language because of the following reasons:
   a) The syntax of java language is similar to other programming languages like C, C+ etc. & therefore, simple to migrate from other languages.

b) The complex topics like pointers, templates etc. are eliminated from java making it simple.

c) In the java language the programmer is responsible for only allocation of memory. The deallocation of the memory is done by the garbage collector.

2) **Object Oriented:** The java language is called as object oriented language. Any language can be called as object oriented if the development of the application is based on objects and classes.

Object: Any entity that exists physically in this real world which requires some memory is called as object.

Every object contains some properties and some actions. The properties are the data which describes the object and the actions are the tasks or the operations performed by the objects. **Class:** A class is a collection of common properties, and common actions of a group of objects. A class can be considered as a plan or a model or a blue print for creating the objects. For every class we can create any number of objects and without a class object can't be created.

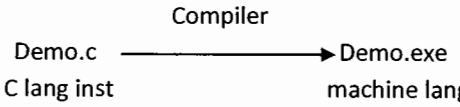
Example: Class: Student
Object: rajesh, amit

3) **Secured:** Security is one of the most important principles of any programming language. The java language contains in-built security programs for protecting the data from unauthorized usage.

4) **Distributed:** Using the distributed feature we can access the data available in multiple machines and provide it to the user. Using this feature we can improve the performance of the application by making the data more available and more accessible.

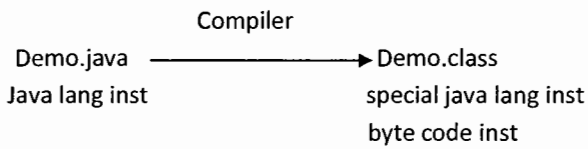
5) **Platform Independent or Machine Independent or Architecture Neutral:** The java program can be executed on any machine irrespective of their hardware, software, architecture, operating system etc therefore it is called as platform independent language.

C Language: When we compile a C program, the compiler verifies whether, the 'C' language instructions are valid not, if valid the compiler generates .exe file containing machine language instructions.



The machine language instructions available in the .exe files generated by the compiler can be executed only in that machine, where it is compiled. If we want to execute the C program in another machine, then we need to recompile and then execute. This nature of C language makes it machine dependent or platform dependent language.

Java Language: When we compile a java program, the compiler verifies whether the java language instructions are valid or not, if valid the compiler will generate .class file containing special java instructions (byte code instructions).



The special java instructions available in the .class file generated by the compiler can be executed on any machine with help of JVM, without recompiling it. This nature of java language makes it platform independent.

6) **Interpreted:** The java language is said to interpreted language as the execution of the program is done by the interpreter available inside the JVM.

7) **High Performance:** The execution of a java program is done by an interpreter along with a special compiler called JIT compiler, thereby reducing the execution time and improving the performance of the application.

8) **Portable:** The java language is said to be portable language using which we can develop an application which is a collection of small components which can be replaced and reused.

9) **Multithreaded:** A language is said to be multithreaded, if it supports multithreading. Every thread in java program is a control. If the program contains multiple controls then we can reduce the waiting time, and provide response faster and thereby improving the performance.

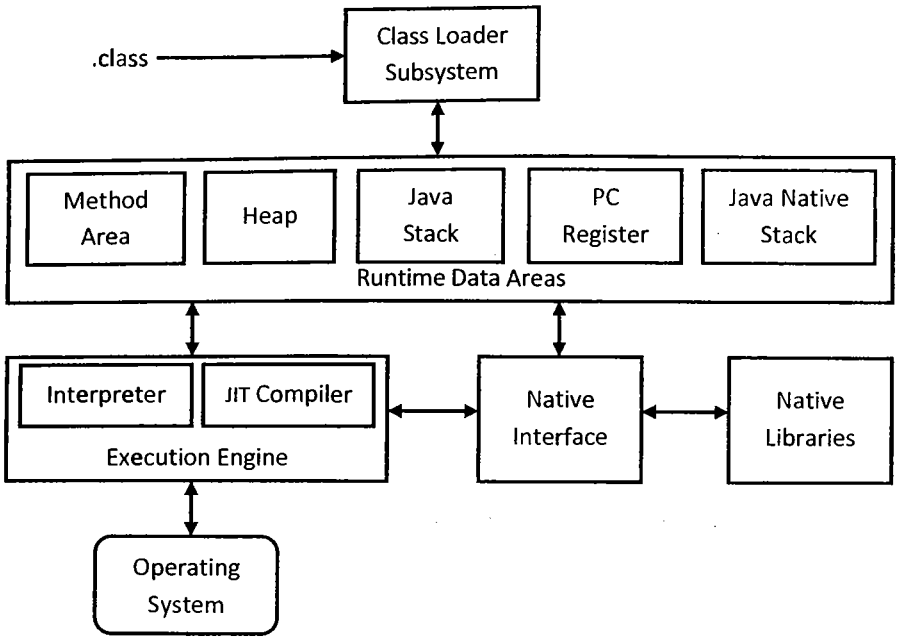
10) **Dynamic:** The java language is said to be dynamic because the allocation of memory is done at execution time according to the requirement.

11) **Robust:** The java language is said to be strong programming language, because of the following reasons:

a) Memory management: In java language the allocation of memory and deallocation of memory, both are efficient. During the memory allocation time, there will be no wastage of memory and deallocation is done by garbage collector which is also efficient as the unused memory will be removed.

b) Exception handling: The errors that occur at runtime because of the logical failure or invalid inputs are called as exceptions. When an exception occurs, the application will be terminated abnormally and executed incompletely. In order to execute the code completely and terminate normally, we take the help of exception handling. The process of exception handling in java is simple and efficient.

JVM Architecture



Class loader subsystem: The class loader subsystem will take .class file as the input and performs the following operations:

a) It will load the .class file into the JVM's memory

b) Before loading the .class file, it will verify whether the byte code instructions are valid or not with the help of byte code verifier.

c) If the byte code instructions are valid then it will load a byte code instruction into different areas of the JVM called runtime data areas.

Runtime data areas: These areas are available at runtime to store different code. The various runtime data areas are:

1) **Method area:** This area can be used for storing all the class code along with their method code.
2) **Heap:** This area can be used for storing the objects.
3) **Java Stack:** This area can be used for storing the methods that are under execution. The java stack can be considered as a collection of frames, where each frame will contain the information of only one method.
4) **PC (Program Counter) Register:** This register will contain the address of the next instruction. that has to be executed.
5) **Java Native Stack:** This area can be used for storing non-java code during the migration of the application from non java code to java code. Non-Java code will be called as native code.

Execution Engine: The execution engine is responsible for executing the java program. It contains two parts:

1) **Interpreter**
2) **JIT (Just-In-Time) Compiler**

Both the parts of the execution engine are responsible for executing the code parallely, reducing the execution time, and thereby improving the performance of the application.

Note: The JIT compiler is designed by Hot Spot Technologies and the code executed by a JIT compiler will be called as hot spots.

Library: A library is a collection of pre-defined programs of a language.

Native Libraries: The collection of libraries of the non java languages will be together called as native libraries.

Native Interface: The native interface will help to load the native code from native libraries into the java native stack.

OS (Operating System): To execute a java program the JVM requires some resources like memory, processor etc from the machine. To get those resources from the machine, the JVM has to communicate with the operating system.

Datatypes

The datatypes represent the type of data that we store into the memory. The datatypes in java language are classified into three types:

1) **Primitive data type:** The primitive datatypes are predefined and designed to store a single value.
   Example: int, char etc.
2) **Derived data type:** The derived datatypes are predefined and designed to store multiple values.
   Example: array
3) **User defined data type:** If a datatype is created by the user or the programmer, then it is called as user defined datatype. Using the user defined datatype we can store any number of values and any type of values, according to the application requirement.
   Example: class

Primitive data types: The primitive data types are designed to store a single value and they are used to store the basic inputs required for a program. The primitive data types are also called as fundamental data types.

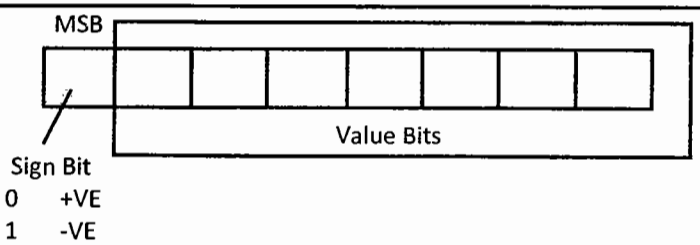
The java language provides 8 primitive data types and they are classified into 4 categories:

1) Integer Category
2) Floating-Point Category
3) Character Category
4) Boolean Category

1) **Integer Category:** This category can be used for storing numbers, either positive or negative without a decimal point. Under the integer category we have 4 primitive data types and they are:
   1. byte
   2. short
   3. int
   4. long

All the 4 primitive data types are used for storing same kind of data, but their sizes and ranges are different so that the memory can be utilized efficiently without any wastage.

Datatype	Size	Range
byte	1	-128 to 127
short	2	-32768 to 32767
int	4	-2147483648 to 2147483647
long	8	-9223372036854775808 to 9223372036854775807



2) **Floating-Point Category:** This category used for storing numbers either positive or negative with decimal point. Under the floating point category we have two primitive data types and they are:
   1. float
   2. double

Both the primitive data types under floating-point category are used for storing same kind of data, but their sizes and ranges are different so that the memory can be utilized efficiency without wastage.

Datatype	Size	Range	Number of decimal digits
float	4	1.4E-45 to 3.4E38	7
double	8	4.9E-324 to 1.79E308	15

3) **Character Category:** This category can be used for storing a single character. A character can be represented by either one alphabet or one digit or one special symbol. Under the character category there is only on primitive data type and it is char.

Datatype	Size	Range
char	2	0 to 65535

Q) Why is the size of char 1 byte in C language and 2 bytes in Java?
A) When an application is developed in C language, it uses the characters of only english language. To store the English language characters 1 byte of memory is sufficient. The C language uses ASCII Character Set (0-255).

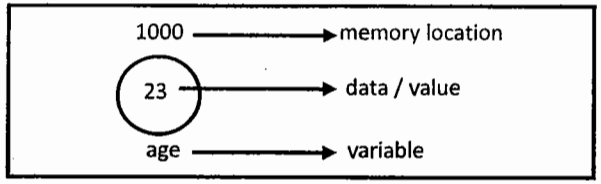
When an application is developed in Java language, it uses the characters of all the foreign languages. To store the characters of all languages 1 byte of memory is not sufficient, therefore the size is increased to 2 bytes in Java language. The Java language uses UNICODE Character Set (0-65535).

4) **Boolean Category:** This category is used for storing either true or false. Under the boolean category we have only one primitive type i.e. boolean.

Datatype	Size	Range
boolean	JVM Dependent	true false

Variable

A variable is a name given to a memory location where we can store some data.



Variable Declaration: The process of specifying what type of data is stored into the memory is called as variable declaration.

Syntax:
datatype variableName;
datatype variable1, variable2, variable3, ...;

Example:
int rollNo;
double marks;
char grade;
boolean result;
int custId, accountNumber, pinNo, age;

When a variable is declared, the memory for that variable will be allocated. The amount of memory allocated to a variable will depend upon the data type that is specified.

Once the memory for that variable is allocated and if we do not specify any value to that variable, then the variable will automatically contain its default value.

byte	0	float	0.0
short	0	double	0.0
int	0	char	space
long	0	boolean	false

If we declare a variable and if don't provide a value to that variable then that variable will be initialized automatically with default values. If we do not want the variable to contain default values, then we can initialize the variable with our own values.

Syntax for initialing a variable during declaration time:

datatype variableName = value;
datatype var1 = value1, var2 = value2, ...;

Example:
```
int rollNo = 123;
double marks = 89.5;
char grade = 'A';
boolean result = true;
int custId = 1234, accountNumber = 45633, pinNo = 1912, age = 23;
```

Initialization: The process of specifying a value to a variable for the first time is called initialization.

The value of a variable can be changed any number of times during the execution of the program.

Assignment: The process of specifying a value to a variable from the second time onwards or after the initialization is called assignment.

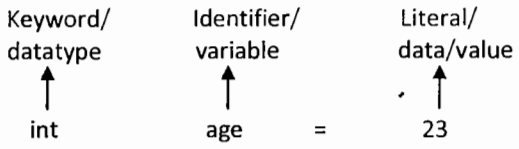
Example: double dollar = 61.45; initialization
dollar = 62.05; assignment
dollar = 58.34; assignment

Keyword: The words that have predefined meaning in java or the words whose meaning is reserved in java are called keywords.

Note: The keywords of java language must be specified in lowercase only.

Literal: The values that we store into a variable are called as literals.

Identifier: Any name that is used for the purpose of identification is called identifier.



Rules for writing an Identifier:

1) An identifier can be a combination of alphabets, digits, underscore(_) and dollar($).
2) An identifier must not begin with a digit. It can begin with either an alphabet or underscore or dollar.
3) Identifier should not contain any spaces.
4) The keywords of the java language should not be used as identifier.
5) There is no restriction on the length of the identifier, but it is recommended to write an identifier having not more than 16 characters.
6) The name of the identifier should be meaningful and appropriate to the application.
   int x = 23 ; valid but not recommended
   int custId = 23; valid and recommended

Structure of Java Program

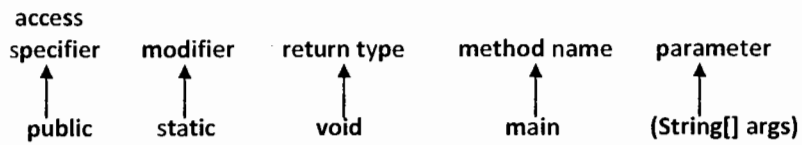
```
package statement;
import statements;
class ClassName {
    variables
    methods
    public static void main(String[] args) {
        statements;
    }
}
```

Specifying the package statement in a java program is optional. A java program can contain at most one package statement. The package statement should be the first executable statement in a program.

Specifying the import statement in a java program is optional. A java program can contain any number of import statements. The import statements should be specified after the package statement and before the class.

A java program can contain any number of classes and every class can contain variables and methods, which are together called as member of the class. A class can contain any number of members.

Execution of a java program is done by JVM and begins from main method, whose syntax must be as follows:



The main method can contain any number of statements and every statement must be terminated by a semicolon.

A java program can contain any number of comments, and they can be specified anywhere in the program. The comments are non executable statements.

Procedure to develop, save, compile, and execute a java program

Step 1 Developing a java program: To develop a java program we require an editor like, notepad, wordpad, vi editor etc.

Program 1:

```
class FirstProgram {
    public static void main(String [] args) {
        System.out.println ("Welcome to Java @ INetsolv");
    }
}
```

Step 2 Saving a java program: A java program can be saved with any name but the extension must be .java

Ex: FirstProgram.java

Step 3 Compiling a java program: To compile a java program we require a command prompt and we use javac command.

Syntax: javac programname/filename with extension
Example: javac FirstProgram.java

java compiler will take the java program and verifies whether the java code is valid or not, if valid the compiler generates .class file. The .class file generated by the compiler will be given to the JVM for execution.

Step 4 Executing a java program: To execute a java program we require a command prompt and we use java command.

Syntax: java ClassName without extension
Example: java FirstProgram

Output: Welcome to Java @ INetsolv

Comments

The comments are used for explaining the code. The comments make the understandability of a program faster. The comments improve the readability of the code.
The java language provides 3 types of comments:

1. **Single Line Comments**
2. **Multi Line Comments**
3. **Documentation Comments**

1. **Single Line Comments:** Using the single line comments we can write a message in a single line. The single line comments begin with // symbols and they end in the same line.
   Example: //This is a single line comment
2. **Multi Line Comments:** Using the multi line comments we can write the message in multiple lines. Multi line comments begin with /* and ends with */. In between these symbols we can specify any numbers of lines.
   Example:
   ```
   /* This is line one comment
      This is line two comment
      This is line three comment  */
   ```
3. **Documentation Comments:** Using the documentation comments we can create the manual for the project that is developed. Using these comments we can create the API (Application Programming Interface). The documentation comments begins with /** and ends with */.
   Example:
   ```
   /** author: inetsolv
       created on: 1/1/2014
       last modified on: 28/11/2014
       ....                                    */
   ```

To develop the java program we need to install java software. The java software is a freeware, which can be downloaded from internet. Once the java software is installed, we need to set the path. The PATH is called an environment variable. The variables of an operating system are called as environment variables.

Setting the path is a process of specifying the address or location of the java commands to the OS (operating system).
Syntax:

```
set PATH=java installation folder;
set PATH=c:\Program Files\Java\jdk1.6.0_21\bin;
```

Program 2:

```
class SecondProgram {
    public static void main(String[] args) {
        System.out.println("this is my second");
        System.out.print("java program ");
        System.out.println("at inetsolv");
    }
}
```

Java Coding Conventions (Hungarian Notations)

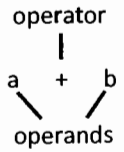
Every predefined class, interface, method etc. will follow the java coding conventions and we are recommended to follow the same conventions. It is always a good programming practice to develop a program by following the coding conventions.

1. **Conventions for a class:** A class name can contain any number of words and the first letter of every word should be specified in uppercase.
   Example: System StringBuffer SecondProgram
2. **Convention for an interface:** An interface name can contain any number of words and the first letter of every word should be specified in uppercase.
   Example: Runnable ActionListener MyInterface
3. **Convention for a method:** A method name can contain any number of words and the first word should be specified completely in lowercase. The first letter of the remaining words if available should be specified in uppercase.
   Example: main() toCharArray() areaOfSquare()
4. **Convention for a variable:** A variable name can contain any number of words and the first word should be specified completely in lowercase. The first letter of the remaining words, if available should be specified in uppercase.
   Example: length numberOfStudents accountNumber
5. **Convention for a constant:** A constant can contain any number of words. All the letters of all the words should be specified in uppercase, if multiple words are available then separate them by underscore (_).
   Example: MIN_VALUE MAX_PRIORITY PI
6. **Convention for a package:** A package name should be specified completely in lowercase.
   Example: java.lang java.net inetsolv.example

Operators

Operator: Any symbol that performs an operation will be called as operators.

Operand: The values on which the operations are performed are called as operands.



Based on the number of operands, the operators are classified into the following categories:

1) **Unary Operators:** These operators will perform operations on one operand.
2) **Binary Operators:** These operators will be performed operations on two operands.
3) **Ternary Operators:** These operators will be performed operators on three operands.

Based on the task that is performed, the operators are classified into the following categories:

1) **Arithmetic Operators:** These operators perform simple mathematical calculations. The various arithmetic operators are +, -, *, /, %

   If both the operands are of integer type, then the result will be of integer type. If at least one of the operand is of floating point type, then the result will be of floating point type.

   Example: 7+2 = 9, 7-2 = 5, 7*2 = 14, 7/2 = 3, 7/2.0 = 3.5, 7%2 = 1

2) **Unary Operators:** These operators will perform operations on a single operand. The various unary operators are -, ++, --

   Unary Minus(-): It can be used to convert values from positive to negative or negative to positive.

   int temp = 23;
   int temp = -23;
   int temp = - (-23);

   Increment Operator(++): The increment operator will increase the value of a variable by 1. Based on the position of the increment operator, the increment operator is classified into 2 types.

   PreIncrement(++x): If the increment operator is placed before the variable then it is called preincrement operator. The preincrement operator will use the value after increasing.

   PostIncrement(x++): If the increment operator is placed after the variable then it is called postincrement operator. The postincrement operator will use the value before increasing.

   Decrement Operator(--): The decrement operator will decrease the value of a variable by 1. Based on the position of the decrement operator, the decrement operator is classified into 2 types.

   PreDecrement(--x): If the decrement operator is placed before the variable then it is called predecrement. The predecrement operator will use the value after decreasing.

   PostDecrement(x--): If the decrement operator is placed after the variable then it is called postdecrement. The postdecrement operator will use the value before decreasing.

Program 3:

```
class Operators {
    public static void main (String[] args) {
        int a = 6;
        int b = ++a;
        int c = b--;
        int d = a++ + --b - --c;
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
    }
}
```

Rule 1: The increment and decrement operators can be applied to all numeric datatypes (byte, short, int, long, double, float, char)

Rule 2: The increment and decrement operators can be applied to only variables, not to constants.

Rule 3: The increment and decrement operators cannot be nested.

3) **Assignment Operators:** This operator can be used for assigning a value to a variable. The assignment operator is =

The assignment operator will copy the value from right side to left side. On the right side we can specify either a variable or a value or an expression etc. but on the left side we must specify only a variable.

Example: x = 5, y = x, z = x+y

The assignment operators can be nested after the declaration of variable but not during the declaration of the variables.

```
int a=b=c=d=6;          not valid because nested during declaration

int a, b, c, d;
a=b=c=d=6;              valid because nested after declaration

int a=6,b=6, c=6,d=6;   valid because there is no nesting
```

If the assignment operator is combined with other operators then it is called as compound assignment operator(+= -= *= /= %=).

4) **Relational Operators:** These operators can be used for comparing the values. These operators are also called as comparison operators. The various relational operators are <, <=, >, >=, ==, !=

The relational operators can be used for creating conditions.

Example: x<y x>y x>=y x==y

The result of the condition will be of boolean type, i.e. if the condition is satisfied, then the result will be true, otherwise the result will be false.

Rule 1: The relational operators <, <=, >, >=, can be applied to any combination of numeric types only.

Rule 2: The relational operators ==, != also called as equality operators can be applied to any combination of numeric types or any combination of boolean types, but not to mixed types, i.e. one boolean and one numeric data.

5) **Logical Operators:** The logical operators can be used for combining the conditions or complimenting the result of a condition.

The various logical operators are &, &&, |, ||, ^, !

AND Operator(&): This operator can be used for combining multiple conditions.

Syntax: (cond1) & (cond2)

F	F	F
F	T	F
T	F	F
T	T	T

The result of & operator will be of boolean type i.e. the result will be either true or false. The result will be true only if all the conditions are true.

The & operator will evaluate all the conditions and then decide the result.

And Operator(&&): This operator can be used for combining multiple conditions.

Syntax: (cond1) && (cond2)

F	F	F
F	T	F
T	F	F
T	T	T

The result of && operator will be of boolean type i.e. the result will be either true or false. The result will be true only if all the conditions are true.

The && operator will evaluate the first condition. If the result of the first condition is false, then it will skip the evaluation of remaining conditions and directly decide the result as false, but if the result of first condition is true, then it will evaluate the next condition and then decide the result.

Note: The && operator is designed to improve the performance of the application sometimes, when the result of the first condition is false.

OR Operator(|): This operator can be used for combining multiple conditions.

Syntax: (cond1) | (cond2)

F	F	F
F	T	T
T	F	T
T	T	T

The result of | operator will be of boolean type i.e. the result will be either true or false. The result will be true, if at least one of the condition is true.

The | pipe operator will evaluate all the conditions and then decide the result.

OR Operator(||): This operator can be used for combining multiple conditions.

Syntax: (cond1) || (cond2)

F	F	F
F	T	T
T	F	T
T	T	T

The result of || operator will be of boolean type i.e. the result will be either true or false. The result will be true, if at least one of the conditions is true.

The || pipe operator will evaluate the first condition, if the result of the first condition is true, then it will skip the evaluation of remaining conditions and directly decide the result as true, but if the result of the first condition is false, then it will evaluate the next condition and then decide the result.

Note: The || operator is designed to improve the performance of the application sometimes, when the result of first condition is true.

X-OR Operator(^): This operator can be used for combining multiple conditions.
Syntax: (cond1) ^ (cond2)

The result of ^ operator will be of boolean type i.e. the result will be either true or false. The result will be true if the inputs are different and if the inputs are same the result will be false.

F	F	F
F	T	T
T	F	T
T	T	F

Not operator(!): This operator can be used for complimenting the result of a condition i.e. converts true to false or false to true.
Syntax: ! (condition)

Rule: The logical operators can be applied to any combination of boolean type only.

6) **Bitwise Operators:** This operator will perform the operation on the bits of a number. Th various bit wise operators are ~ & | ^ << >> >>>

Negation Operator(~)(tilde): This operator will convert the bits from $0^{'s}$ to $1^{'s}$ and $1^{'s}$ to $0^{'s}$

x = 5	0	0	0	0	0	1	0	1
~x	1	1	1	1	1	0	1	0

If the first bit is '1' then it represents a negative number. The value of the negative number is calculated by performing 2's compliment.

2's compliment = 1's compliment + 1

	1	1	1	1	1	0	1	0
	0	0	0	0	0	1	0	1
								1
	0	0	0	0	0	1	1	0 = -6

Note: ~x = x + 1

Bitwise AND Operator(&): This operator will perform AND operation on the bits of a number.

0	0	0
0	1	0
1	1	0
1	1	1

x=5	0	0	0	0	0	1	0	1	
y=6	0	0	0	0	0	1	1	0	
x&y	0	0	0	0	0	1	0	0	= 4

Bitwise OR Operator(|): This operator will perform OR operation on the bits of a number.

0	0	0
0	1	1
1	1	1
1	1	1

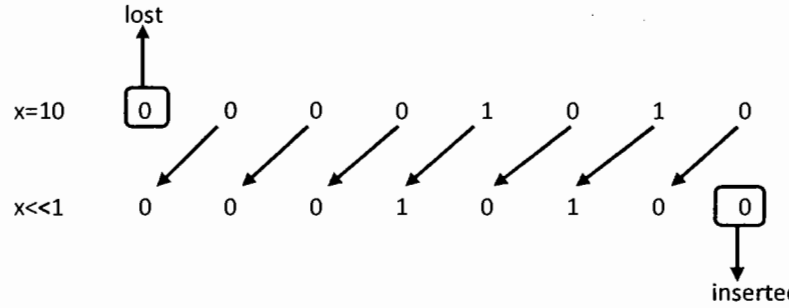
x=5	0	0	0	0	0	1	0	1	
y=6	0	0	0	0	0	1	1	0	
x|y	0	0	0	0	0	1	1	1	= 7

Bitwise X-OR Operator(^): This operator will perform X-OR operation on the bits of a number.

0	0	0
0	1	1
1	1	1
1	1	0

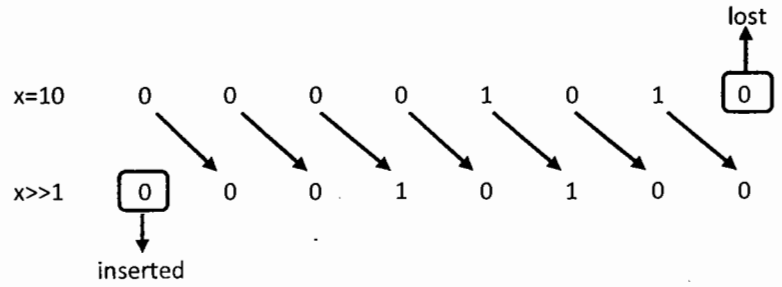
x=5	0	0	0	0	0	1	0	1	
y=6	0	0	0	0	0	1	1	0	
x^y	0	0	0	0	0	0	1	1	= 3

Left Shift Operator(<<): This operator will shift the bits of a number towards left side by the specified number of positions.



Shifting the bits of a number towards left side by one position is equivalent to multiplying a number by 2.

Right Shift Operator(>>): This operator will shift the bits of a number towards right side by the specified number of positions.



Shifting the bits of a number towards right side by one position is equivalent to dividing a number by 2.

Unsigned Right Shift Operator(>>>): This operator will shift the bits of a number toward right side by the specified number of positions.

Difference between >> & >>>:
When we shift the bits of a number towards right side by using >> operator, the sign bit of the input number will be copied as it is into the resultant i.e. positive input will lead to positive output and negative input will lead to negative output.

When we shift the bits of a number towards right side by using >>> operator, the sign bit will be never copied into the resultant. We will always insert a bit 0 into the sign bit of the result i.e. the result will be always positive, whether the input is positive or negative.

Rule: Bitwise operators can be applied to any combination of numbers without decimal point(byte, short, int, long, char).

Note: The & | ^ operators can be applied to any combination of boolean type or any combination of numbers without decimal point.

7) **Conditional Operator:** This operator will perform a task based on a condition.
The conditional operator is **? :**

Syntax: (condition) **?** exp1 **:** exp2

If the condition is satisfied, then it will evaluate the expression(expression1) available after question mark (**?**) and if the condition is not satisfied, then it will evaluate the expression(expression2) available after colon (**:**). The conditional operator is also called a ternary operator.

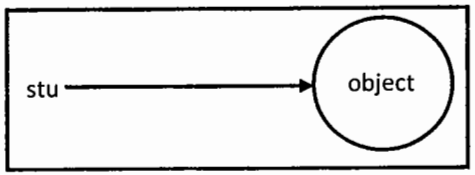
```
max = (x > y) ? x : y
diff = (x > y) ? x-y : y-x
5 > 6 ? "hai" : "bye"
```

The conditional operator can be nested any number of times.

```
2 < 3 ? 4 > 5 ? "hai" : "hello" : "bye"
```

8) **new operator:** The new operator is used for creating the object. Creating an object means allocating memory for a class in heap memory.

```
class Student {

}

Student stu = new Student();
```



Heap

stu is called as the reference variable. It is used to refer or point to an object. The name of the reference variable can be any valid java identifier.

For every class we can create any number of objects but without a class we cannot create an object. The new operator can be used for creating an object of predefined classes, and user defined classes.

9) **Dot(.) Operator:** This operator can be used for accessing the members(variables and methods) of a class or access a class from a package.

```
ClassName.variable
ClassName.method

reference.variable
reference.method

packagename.ClassName
```

Control Statements

The statement of a java program will be executed in sequential order. If we don't want to execute the statements in sequential order i.e. if we want to execute the statements in random order according to programmer choice, then we take the help of control statements.

The control statements will help the programmer to control the flow of execution in a java program. The control statements are classified as following

1. Conditional statements
   - 1.1 if else statement
   - 1.2 switch statement
2. Iterating statements
   - 2.1 for loop
   - 2.2 while loop
   - 2.3 do-while loop
   - 2.4 for each loop
3. Transfer statements
   - 3.1 break
   - 3.2 continue
   - 3.3 return

Conditional Statements:

If-else statement: if else statement can be used for executing a group of statements based on condition.

Syntax :

```
if (condition) {
   statements1;      if block
}
else {
   statements2;      else block
}
```

If the condition is satisfied then it will be executing if block and if the condition is not satisfie then it will be executing else block.

Program 4:

```
class IfElseDemo {
   public static void main (String[] args) {
      int n = 89;
      if (n%2 == 0) {
         System.out.println("Even Number");
      }
      else {
         System.out.println("Odd Number");
      }
   }
}
```

Rule 1: Specifying the condition to if else statement is mandatory and it should be of boolean typ (either a condition or boolean variable or boolean value).
Rule 2: Specifying the else block is optional.
Rule 3: Specifying the { } is optional. If we don't specify the { } then it will consider only on statement, and that one statement is mandatory. If we want to consider multiple statements the specifying the { } is mandatory. Within the { } we can specify any number of statements.
Rule 4: In if else statement we cannot execute both the blocks and we cannot skip both th blocks, it will always execute exactly one block.

witch statement: The switch statement can be used for executing a group of statements based n a value.

yntax:

```
witch (argument) {
 case label1: statements1;
              break;
 case label2: statements2;
              break;
 :
 default: default statements;
```

rogram 5:

```
lass SwitchDemo {
 public static void main (String[] args) {
    int ch = 2;
    switch(ch) {
       case 1: System.out.println("First Choice");
               break;
       case 2: System.out.println("Second Choice");
               break;
       case 3: System.out.println("Third Choice");
               break;
       case 4: System.out.println("Fourth Choice");
               break;
       default: System.out.println("Wrong Choice");
    }
 }
```

ule 1: Specifying the argument to switch statement is mandatory and it should be of either byte, hort, int or char only.
ule 2: Specifying the flower braces to switch statement is mandatory.
ule 3: Specifying the case and default statements in switch is optional.
ule 4: A switch statement can contain any number of cases and it can contain at most one efault.
ule 5: The default statement can be specified anywhere in the switch statement.
ule 6: The case labels must be unique i.e. they should not be empty and they should not be uplicated.
ule 7: Specifying the break statement is optional. The break is a transfer statement and it will ransfer the control from inside the switch to outside the switch, so that it skips the execution of emaining cases.

Rule 8: When a switch statement is executed, switch argument is compared with case labels an execution will begin from that case onwards whose label is matching with the argument an continues until it encounters a break statement or until the end of switch.

Iterating Statements:

For Loop: This loop can be used for executing the statements multiple times. A for loop has to b used when we know the exact number of iterations.

Syntax:

```
        1          2 5 8...      4 7 10...
for (initialization ; condition ; increment/decrement) {
      statements; 3 6 9...
}
```

Program 6:

```
class ForDemo {
  public static void main(String[] args) {
    int n = 6;
    for(int i=1; i<=10 ; i++) {
      System.out.println (n+" * "+ i+" = "+ (n*i));
    }
  }
}
```

Rule 1: All the three sections of the for loop are optional. If we do not specify any initializatio and if do not specify any inc/dec, then the compiler will not specify any initialization and will no specify any inc/dec, but if we do not specify any condition then the compiler will automatical specify a boolean value true.
Rule 2: Specifying the section separators(;) in a for loop are mandatory.
for(; ;) will execute for infinite times.
Rule 3: In the for loop, initialization section and the inc/dec section can contain any number o valid java statements, but the condition section must contain a value of only boolean type.
Rule 4: The initialization section can contain multiple initializations separated by a comma(,) a should be of same time and the data type should be specified only one time.
Rule 5: The condition section can contain any number of conditions, but they must be joined b using a logical operator(& && | || ^).
Rule 6: The inc/dec section can contain multiple increments or decrements but separated by a comma (,).
Rule 7: If a variable is declared inside for loop, then it can be used only in that for loop.
Rule 8: Specifying the { } is optional. If we don't specify the { } then it will consider only on statement, and that one statement is mandatory. If we want to consider multiple statements the specifying the { } is mandatory. Within the { } we can specify any number of statements.

While loop: This loop can be used for executing the statements multiple times. A while loop has to be used when we do not know the exact number of iterations.

Syntax:

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

Program 7:

```
class WhileDemo {
   public static void main(String args[]) {
      int n = 8 ,i = 1;
      while(i <=10) {
         System.out.println(n+" * "+i+" = "+(n*i));
         i++;
      }
   }
}
```

Rule 1: Specifying a condition to while loop is mandatory and it should be of boolean type.
Rule 2: Specifying the { } is optional. If we don't specify the { } then it will consider only one statement, and that one statement is mandatory. If we want to consider multiple statements then specifying the { } is mandatory. Within the { } we can specify any number of statements.

Note: A java program should not contain any unreachable statements i.e. every statement has to be executed at some point of time.

Do-While loop: This loop can be used for executing the statements multiple times. A do-while loop has to be used when we do not know the exact number of iterations.

Syntax:

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

Program 8:

```
class DoWhileDemo {
  public static void main(String args[]) {
    int n = 9 ,i = 1;
    do {
      System.out.println(n+" * "+i+" = "+(n*i));
      i++;
    } while(i <=10);
  }
}
```

Rule 1: Specifying a condition to do-while loop is mandatory and it should be of boolean type.
Rule 2: Specifying the { } is optional. If we don't specify the { } then we must specify exactly on one statement. If we want to consider multiple statements then specifying the { } is mandator Within the { } we can specify any number of statements.

Difference between while loop and do-while loop:

1) In a while loop the statements will execute after evaluating the condition, whereas in a d while loop the statements will execute before evaluating the condition.
2) In a while loop if the condition is false for the first time then the statements will not execu whereas in a do-while loop if the condition is false for the first time then the statemens w not execute.
3) In a while loop the statements will execute for 0 or more times, whereas in a do-while lo the statements will execute for 11 or times.

Nested Loop: If we specify a loop inside another loop then it is called as nested loop. Any lo can be specified inside any other loop any number of times.

Program 9:

```
class NestedForDemo {
  public static void main(String[] args) {
    for(int i=1;i<=5;i++) {
      for(int j=1;j<=5;j++) {
        System.out.print("* ");
      }
      System.out.println();
    }
  }
}
```

Program 10:

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


Transfer Statements:

Break: break is a transfer statement which can be used either inside switch statement or inside a loop.

- The break statement, when used in switch will transfer the control from inside the switch to outside the switch, so that we skip the execution of removing cases.
- The break statement, when used in loop, will transfer the control from inside the loop to outside the loop, so that we will skip the execution of remaining iterations.

Note: When we are specifying the break statement in a loop we are recommended to specify the break statement along with condition.

Program 11:

```
class BreakDemo {
  public static void main (string[]args) {
    int sum = 0, capacity = 15;
    for(int i=1,i<=100;i++) {
      System.out.println(i);
      sum = sum + i;
      if(sum>=capacity)
        break;
    }
    System.out.println(sum);
  }
```

Continue: continue is a transfer statement which has to be used only in loops. The continue statements will skip the current iteration and continues with the remaining iterations.

Note: When we are specifying a continue statement in a loop, then we are recommended to specify the continue statement along with a condition.

Program 12:

```
class ContinueDemo {
  public static void main(String[] args)
    for(int i=1;i<=20;i++) {
      if(i==7 | i==13)
        continue;
      System.out.println(i);
    }
  }
```

Arrays

An array is a derived data type, which can be used for storing multiple values.

If an application requires multiple values, then we can store those multiple values by declari multiple variables. If we declare multiple variables in a program then the code size will increas and readability will be reduced.

In order to reduce the code size, and improve the readability of the code, we take the help arrays. Arrays in java language are classified into two types. They are

1) Single dimension array
2) Multi dimension array

Single dimension array: single dimension array is a collection of multiple values represented the form of a single row or single column.

Syntax for declaring a single dimension array:
datatype arrayName[]; each pair of square bracket represents one dimension.

The name of the array can be any valid java identifier

Rule 1: At the time of array declaration we can specify the pair of [] either before the array nam or after the array name.

Example:
int rollNo[];
double[] marks;
char []grade;
boolean result[];

Rule 2: At the time of array declaration we should not specify the size of the array.

Syntax for Creation of single dimension Array:
datatype arrayName[] = new datatype[size];
Or
datatype arrayName[];
arrayName = new datatype[size];

Example:
int arr[] = new int[10];
Or
int arr[];
arr=new int[10];

Rule: Specifying the size of the array at the time of array creation is mandatory and it should be byte, short, int, char type only.

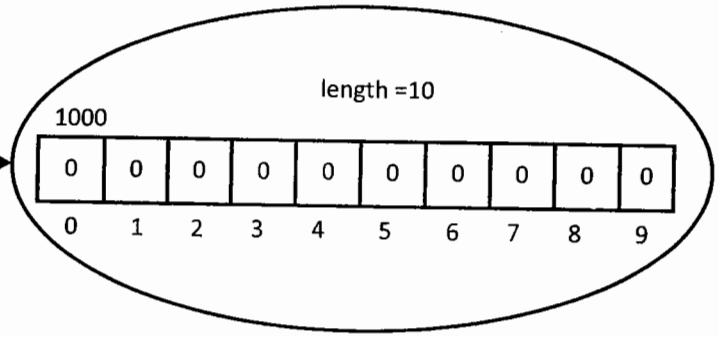
The size of the array must be a positive number. If a negative number is specified, then we get runtime error called NegativeArraySizeException.

When an array is created, the memory for that array will be allocated in sequential memory oca`tions. The amount of memory allocated to an array depends upon the size of the array and he size of the datatype of the array. Once the memory for the array is allocated, all the array lements will be initialized automatically with default values.

n array contains multiple values and for the entire array we have only one name. In order to ccess the array elements we take the help of index position. The index position will always begin ith 0. The range of the index position will be 0 to size - 1.

very array is internally an object and it contains a default variable called length, which represents he size of the array.

t arr[] = new int[10];



rrayName[index] = starting address of the array + index * size of array type
rr[0] = 1000 + 0 * 4 = 1000
rr[1] = 1000 + 1 * 4 = 1004
rr[5] = 1000 + 5 * 4 = 1020

yntax for accessing the array elements:
rrayName[index]

xample:
rr[0]
rr[1]

hen we are accessing the array elements, we are supposed to specify the index position within he range otherwise we get runtime error called ArrayIndexOutOfBoundsException.

yntax to assign a value to an array element:
rrayName[index] = value;

xample:
rr[0] = 10;
rr[1] = 20;

For each loop(enhanced for loop): This loop is introduced in java 1.5 version and it designe for accessing the elements from arrays(collection). It is also called as enhanced for loop.

Syntax:

```
for(declaration : arrayName) {
  statements;
}
```

The declaration of a variable in for each loop must be same as that of the type of elemen availiable in the array. In for each loop, the statements will be executed one time for each eleme available in the array and therefore called for each loop.

Program 13:

```
class ArrayDemo {
  public static void main(String[] args) {
    int[] arr;
    arr = new int[5];

    System.out.println(arr[0]);
    System.out.println(arr[1]);
    System.out.println(arr[2]);
    System.out.println(arr[3]);
    System.out.println(arr[4]);

    arr[0] = 11;
    arr[1] = 22;
    arr[2] = 33;
    arr[3] = 44;
    arr[4] = 55;

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

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


eclaring, creating, and assigning the array elements in a single line.

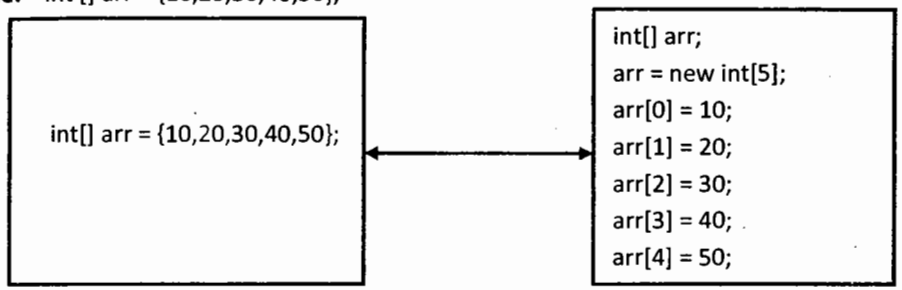
yntax:

atatype arrayName[] = { list of values };

ithin the { } we can specify any number of values separated by a comma.

- When we are declaring, creating and assigning the values in a single line, then we need not specify the size of the array.
- The creation of array will be done by JVM and the size of the array is also decided by JVM based on number o f values specified within the { }.
- Once the size of the array is decided the JVM itself will assign the values to the array element.

xample: int [] arr = {10,20,30,40,50};



rogram 14:

```
ass ArrayDemo {
 public static void main (String[] args) {
   int[] iarr = {12,34,45,67,89};
   for(int x : iarr) {
     System.out.println(x);
   }
   double []darr = {1.1,2.2,.3.3,4.4,5.5,6.6};
   for(double y : darr) {
     System.out.println(y);
   }
   char carr[] = {'a','b','c','d'};
   for(char z:carr) {
     System.out.println(z)
   }
}
```

Multi dimensional arrays: The multi dimension arrays in java will be represented in the form array of arrays.

Syntax for declaring a two dimension array:
datatype arrayName[][];

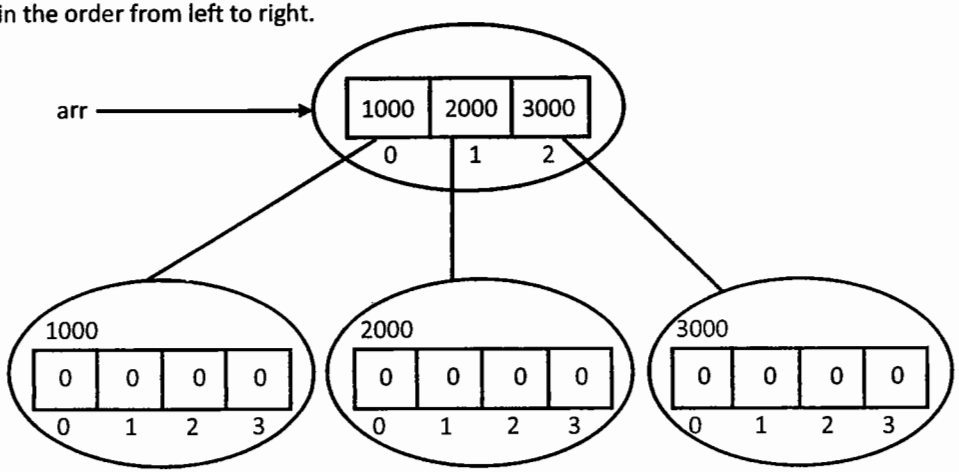
Rule 1: At the time of array declaration we can specify the pair of [][] either before the array name or after the array name.
int arr1[][];
int[][] arr2;
int [][]arr3;
int[] []arr4;
int []arr5[];
int[] arr6[];

Rule 2: At the time of array declaration we should not specify size of the array.

Syntax for creating 2 dimensional array :
datatype arrayName[][] = new datatype[size1][size2];
Or
datatype arrayName[][];
arrayName = new datatype[size1][size2];

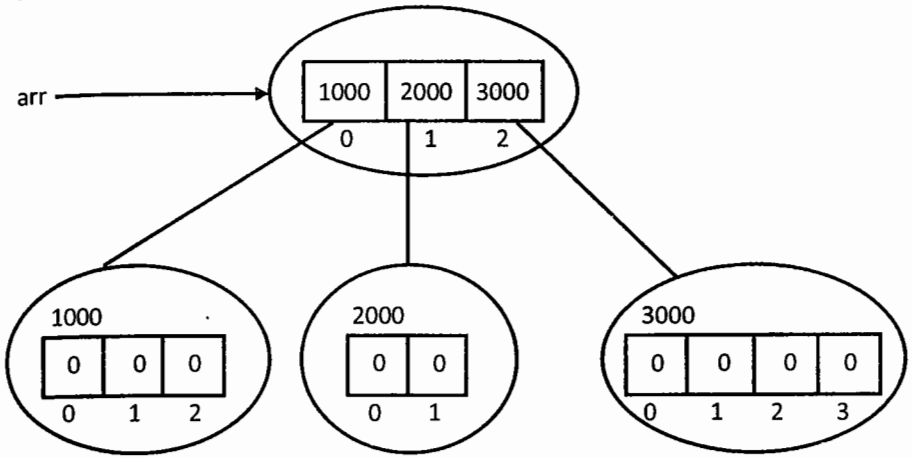
Example:
int arr[][] = new int[3][4];
Or
int arr[][];
arr = new int[3][4];

Rule: In a multi dimension array specifying the first dimension of array is mandatory and remaining are optional. If we are specifying the remaining dimensions then they must be specified in the order from left to right.



gged array: In multi dimension array of the arrays have unequal size then they are called gged array.

ample of Jagged array:



eation of the above jagged array

[][] arr = new int[3][];
r[0] = new int[3];
r[0] = new int[2];
r[0] = new int[4];

ogram 15:

```
ss ArrayDemo {
public static void main(String [] args) {
  int arr[][] = {{1,2,3},{4,5,6},{7,8,9}};
  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();
  }
  for(int[] x : arr) {
    for(int y : x) {
      System.out.print (y+" ");
    }
    System.out.println();
  }
```

Program 16:

```
class Student {
  public static void main(String[] args) {
    int[] marks = {91,95,100,89,79,99};
    int total = 0;
    boolean result = true;

    System.out.println("student report card");

    for(int i=0; i<marks.length; i++) {
      System.out.println("subject " + (i+1)) + " marks : "+marks[i])
      total = total + marks[i];
      if(marks[i] < 35)
        result = false;
    }

    System.out.println("total marks : " + total);

    int avg = total / marks.length;
    System.out.println("average : "+avg);

    if(result) {
      if(avg >= 75) {
        System.out.println("Grade : Distinction");
      }
      else if(avg >= 60) {
        System.out.println("Grade : First Class");
      }
      else if(avg >= 50) {
        System.out.println("Grade : Second Class");
      }
      else {
        System.out.println("Grade : Third Class");
      }
    }
    else {
      System.out.println("Welcome Again");
    }
  }
}
```


String

String is a predefined class, which can be used for storing a group of characters.

To store a group of characters we need to create an object of String class. The String class object can be created in two ways.

1. The String class object can be created by using new operator
   String str = new String("hello");
2. The String class object can be created by specifying a group of characters enclosed directly in a pair of double quotes(" ").
   String str = "hello";

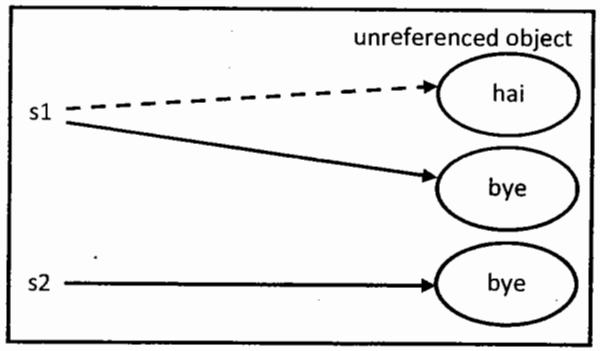
The String objects created by both the mechanisms are called as immutable object, which means once the String object is created we cannot modify the content of the object.

Difference between the two mechanisms of creating String objects:

String Object Created by using new Operator:

1) **Location:** The String objects created by using new operator are stored in heap memory.
2) **Allocation:** If the String objects are stored in heap memory then, it will never verify, whether the heap memory contains objects with same content or not, it will always create new object and store the content.

```
String s1 = new String("hai");
s1 = new String("bye");
String s2 = new String("bye");
```



heap

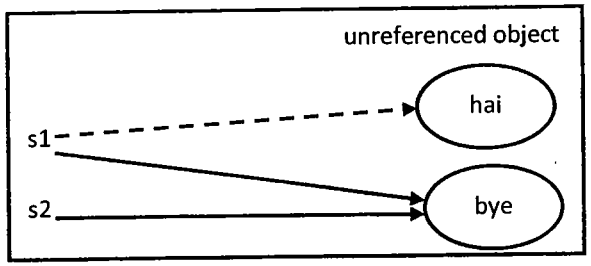
Unreferenced Object: If an object does not have any live reference i.e. if no reference variable is referring to the object then, the object should be called as unreferenced object.

3) **Deallocation:** If the heap memory contains unreferenced objects then, they will be deallocated by garbage collector.

String object created by specifying a group of characters enclosed directly in a pair of dou quotes:

1) **Location:** If a String object is created by enclosing a group of characters in a pair of dou quotes (" ") then, they are stored in string constant pool.
2) **Allocation:** If the String objects are stored in string constant pool then, it will always ve whether the string constant pool contain objects with same content or not if available th refer to existing object. If not available then create a new object.

```
String s1 = "hai";
s1 = "bye";
String s2 = "bye";
```

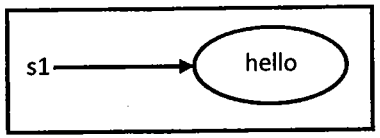


string constant pool

3) **Deallocation:** If the string constant pool contains unreferenced object then, they will deallocated by the string constant pool itself, when the string constant pool is complet filled.

If a String object is created by using new operator then the content will be stored in both he and string constant pool but the reference will refer to the object available in heap memory.

```
String s1 = new String("hello");
```

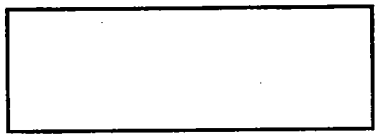


heap

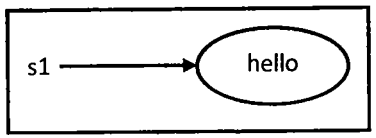
string constant pool

If a String object is created by specifying a group of characters enclosed directly in a pair of dou quotes then the content will be stored in only in string constant pool.

```
String s1 = "hello";
```



heap



string constant pool

Method Of String Class:

1) **int length():** This method will return the count of the number of characters available in a String.

Program 17:

```
class StringDemo {
    public static void main(String[] args) {
        String str = new String("java program");
        System.out.println (str.length());
    }
}
```

2) **char charAt(int index):** This method will return a character that is available at the specified index position.

The index position starts from 0. If the specified index is not within the range then, it will generate a runtime error called StringIndexOutOfBoundsException.

Example:

```
String str = new String("java program");
System.out.println(str.charAt(5));
System.out.println(str.charAt(15));
```

3) **String concat(String):** This method can be used to append the contents of one string to another string.

Example:

```
String s1 = new String("java");
String s2 = new String("program");
s1= s1.concat (s2);
System.out.println(s1);
System.out.println(s2 );
```

4) **int compareTo(String):** This method can be used to compare the unicode values of the characters available in a String, by considering their case. This method is designed for sorting the strings.

```
S1 > S2     +ve
S1 < S2     -ve
S1 == S2    0
```

5) **int compareToIgnoreCase(String):** This method can be used to compare the unicode values of the characters available in a String by ignoring their case. This method is designed for sorting the strings.

Example:

```
String s1 = new String("abc");
String s2 = new String("bcd");
System.out.println(s1.compareTo(s2));
System.out.println(s1.compareToIgnoreCase(s2));
```

6) **boolean equals(Object):** This method can be used to compare the contents of the stri objects by considering their case.
7) **boolean equalsIgnoreCase(String):** This method can be used to compare the content of t string objects by ignoring their case.

Example:

```
String s1 = new String("abc");
String s2 = new String("ABC");
System.out.println(s1.equals(s2));
System.out.println(s1.equalsIgnoreCase(s2));
```

8) **boolean startsWith(String):** This method can be used to check whether a string begins with specified group of characters or not.
9) **boolean endsWith(String):** This method can be used to check whether a string ends with specified group of characters or not.

Note: The startsWith() and endsWith() will consider the case.
Example:

```
String str = new String("java program");
System.out.println(str.startsWith("ja"));
System.out.println(str.startsWith("jab"));
System.out.println(str.startsWith("JAVA"));
System.out.println(str.endsWith("ram"));
System.out.println(str.endsWith("Gram"));
```

10) **int indexOf(char):** This method will return the index of the first occurrence of the specifi character.
11) **int lastIndexOf:** This method will return the index of the last occurrence of the specifi character.
Note: The indexOf() and lastIndexOf() will return -1, if the specified character is not availab
Example:

```
String str = new String("java program");
System.out.println(str.indexOf('a'));
System.out.println(str.indexOf('p'));
System.out.println(str.indexOf('q'));
System.out.println(str.lastIndexOf('a'));
System.out.println(str.lastIndexOf('p'));
System.out.println(str.lastIndexOf('z'));
```

12) **String replace(char old, char new):** This method can be used to replace all the occurrences the specified character with a new character.
Example:

```
String str = new String("java jug jar jungle");
System.out.println(str.replace('j', 'b'));
```


13) **String substring(int index):** This method will return a group of characters beginning from the specified index position up to the end of the string.
14) **String substring(int index, int offset):** This method will return a group of characters beginning from the specified index position up to the specified offset.

Note: The offset represents the position of a character in a string and it begins with 1.
Example:

```
String str = new String("java program");
System.out.println(str.substring(3));
System.out.println(str.substring (2,9));
```

15) **String toLowerCase():** This method will convert the contents of a string to completely lower case.
16) **String toUpperCase():** This method will convert the contents of a string to completely upper case.
Example:

```
String str = new String("java program");
System.out.println(str);
System.out.println(str.toLowerCase());
System.out.println(str.toUpperCase());
```

17) **String trim():** This method can be used to remove the leading and trialing spaces available in a string
Example:

```
String str = new String("java program");
System.out.println(str+"bye");
System.out.println(str.trim()+"bye");
```

18) **String intern():** This method will refer to an object available in string constant pool which was created at the time of object creation in heap memory.
Example:

```
String s1 = new String("hello");
String s2 = s1.intern();
System.out.println(s1);
System.out.println(s2);
```

haining Of Methods:

ogram 18:

```
ass Sample {
 public static void main(String[] args) {
    String str = new String("core");
    System.out.println(str.concat("java").substring (4).concat("program")
          .substring(4,8).concat("test").replace('z', 'n').toUpperCase().charAt(2));
 }
```

Rule: If both the operands are of numeric type then + operator will perform addition and if atle one operand is of string type then + operator will perform concatenation.

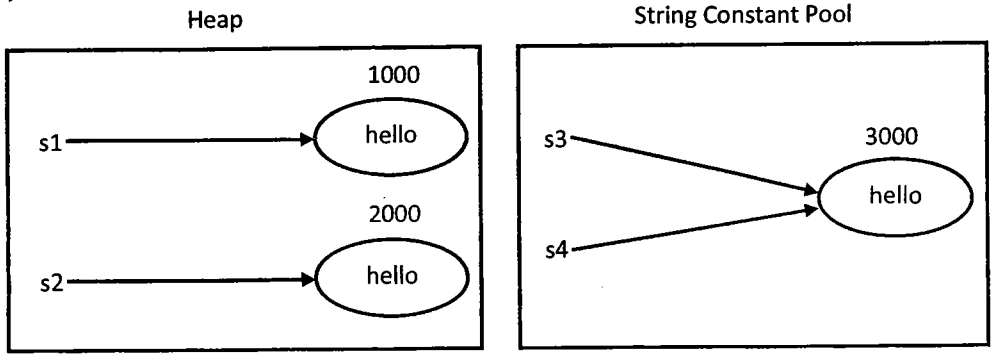
Program 19:

```
class Sample {
    public static void main(String[] args) {
        String str = new String("hello");
        System.out.println(str+1+2+3);
        System.out.println(1+str+2+3);
        System.out.println(1+2+str+3);
        System.out.println(1+2+3+str);
        System.out.println((1+2)+str+3);
        System.out.println(1+str+(2+3));
    }
}
```

Rule: When the string objects are compared by using equals(), then it will compare the conte of the string objects and if we compare the string objects by using == operator, then it compare the references(hashcodes) of the objects.

Program 20:

```
class Sample {
    public static void main(String[] args) {
        String s1 = new String("hello");
        String s2 = new String("hello");
        String s3 = "hello";
        String s4 = "hello";
        System.out.println(s1.equals(s2));
        System.out.println(s1.equals(s3));
        System.out.println(s3.equals(s4));
        System.out.println(s1==s2);
        System.out.println(s1==s3);
        System.out.println(s3==s4);
    }
}
```



tringBuffer

tringBuffer is a predefined class, used for storing group of characters. StringBuffer object can be reated in only one way and that is by using new operator.

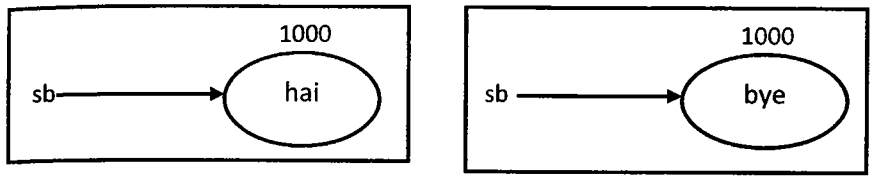
yntax:

```
tringBuffer sb = new StringBuffer("welcome");
```

he StringBuffer objects are mutable which means we can modify the content of the object.

xample:

```
ringBuffer sb = new StringBuffer("hai");
 = new StringBuffer("bye");
```



ethods Of StringBuffer:

) **int length():** This method will return the count of the number of characters available in StringBuffer.

Example:

```
StringBuffer sb = new StringBuffer("welcome");
System.out.println(sb);
System.out.println( sb.length());
```

) **StringBuffer append(xxxx):** This method can be used to append the specified content to the existing StringBuffer object.

Example:

```
StringBuffer sb = new StringBuffer("java");
System.out.println(sb.append(1.7));
System.out.println(sb.append("program"));
```

) **StringBuffer deleteCharAt(int index):** This method can be used to delete a character that is available specified index position.

) **StringBuffer delete(int index, int offset):** This method can be used to delete a group of characters beginning with the specified index position up to the specified offset.

Example:

```
StringBuffer sb = new StringBuffer("java1.7program");
System.out.println(sb.deleteCharAt(5));
System.out.println(sb.delete(3,10));
```

5) **String substring(int index):** This method can be used to retrieve a part of a StringBuffe beginning from the specified index position up to the end of the StringBuffer .
6) **String substring(int index, int offset):** This method can be used to retrieve a part of th StringBuffer beginning from the specified index position up to the specified offset.

Note: The substring() will not modify the content of the StringBuffer.

Example:
```
StringBuffer sb = new StringBuffer("java 1.7program");
System.out.println(sb.substring(6));
System.out.println(sb.substring(1,4));
```

7) **StringBuffer insert(int index, xxx):** This method can be used to insert the specified content a the specified index position.

Example:
```
StringBuffer sb = new StringBuffer("program");
System.out.println(sb.insert(0,"java"));
System.out.println(sb.insert(4,1.7));
```

8) **StringBuffer replace(int index, int offset, String):** This method can be used to replace group of characters beginning with the specified index position up to the specified offse with the specified String.

Example:
```
StringBuffer sb = new StringBuffer("java1.8program");
System.out.println(sb.replace (0,7,"jse"));
```

9) **StringBuffer reverse():** This method can be used to reverse the contents of StringBuffer.

Example:
```
StringBuffer sb = new StringBuffer("java1.7program");
System.out.println(sb.reverse());
```

Program 21:
```
class Sample {
   public static void main(String[] args) {
      StringBuffer sb = new StringBuffer("java");
      System.out.println(sb.append("program").insert(0,"core")
      .delete(4,8) .append("test").replace(4,8,"frog").substring (7)
      .concat("example") .substring (4).replace ('e', 'n').toUpperCase()
      .chatAt (8));
   }
}
```


StringBuilder

he StringBuilder class is introduced in java 1.5 version and it can be used for storing a group of haracters.

yntax:
```
tringBuilder sb = new StringBuilder("java program");
```

tringBuilder objects are mutable that is we can modify or update the content of the StringBuilder bject.

ifference between StringBuffer & StringBuilder:
he StringBuffer object is synchronized (thread-safe) i.e. it can be accessed by only one thread at time, whereas StringBuilder objects is not synchronized i.e. it can be accessed by multiple hreads at a time.

ommand Line Arguments

he values that we pass into the program during the execution time in the command prompt are alled as command line arguments.
he purpose of command line arguments is to pass values dynamically and to avoid hard coding.
ard Coding: The process of providing a value to a variable within the program is called hard oding.
s a programmer we are recommended to avoid hard coding during the application development.

rogram 22:
```
lass Addition {
   public static void main(String[] abc) {
      int x = Integer.parseInt(abc[0]);
      int y = Integer.parseInt(abc[1]);
      int z = x + y;
      System.out.println(z);
   }
```

```
vac Addition.java
va Addition 11 22
```

e are supposed to pass the command line arguments in the command prompt during the xecution time after the class name.

ll the command line arguments will be stored into the string array of main method in string rmat. We can convert the values from string format to original format by using a technique is alled parsing.

Parsing: It is a process of converting the value from string type to primitive type.

Parsing can be applied to all primitives except character.

Byte.parseByte()
Short.parseShort()
Integer.parseInt()
Long.parseLong ()
Float.parseFloat()
Double.parseDouble()
Boolean.parseBoolean()

As a programmer we are responsible for only declaring the array.
String[] abc;

The name of the array can be any valid java identifier.
The type of the array must be only string so that we can store any type of values and any numbe of values.

It is the responsibility of the JVM to create an array and JVM itself will decide the size of the arr based on the number of values specified. Once the array is created the JVM will only assign t values to the array elements.
abc = new String[2];
abc[0] = "11";
abc[1] = "22";

If we don't specify any command line arguments then also the JVM will create any array of size 0
abc = new String[0];

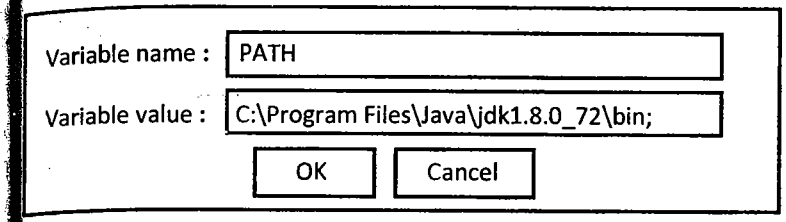
BufferedReader: A program to read data from the keyboard using BufferedReader class.

Program 23:

```
import java.io.*;
class Addition {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(
                    new InputStreamReader (System.in));

        System.out.println("Enter first number: ");
        String s1 = br.readLine();
        int fno = Integer.parseInt(s1);

        System.out.println("Enter second number: ");
        int sno = Integer.parseInt(br.readLine());

        System.out.println("Result : "+(fno+sno));
    }
}
```


Procedure To Set The Path:

Right clink on computer icon → select properties → click on advanced system settings → select advanced tab → click on environment variables → click on either new (or) edit button



Object Oriented Programming Concepts

To develop an application we require a programming language which uses one of the following concepts.

1. Procedure oriented programming concepts
2. Object oriented programming concepts

Procedure Oriented Programming Concepts: A language is said to be procedure oriented language, if the applications are developed with the help of procedures or functions.

Examples of procedure oriented languages: C, Pascal, Cobol, Fortran etc.

Limitations or Drawbacks of procedure oriented programming languages:

The applications that are developed by using procedure oriented programming concepts are difficult to maintain and debugging of such application is time consuming.
The applications that are developed by using procedure oriented programming concepts do not provide security to the data.
The applications that are developed by using procedure oriented programming concepts gives more importance to the functions than to the data.
The data available in the applications developed by using procedure oriented programming concepts is open and therefore they are not suitable for developing distributed application.
The applications that are developed by using procedure oriented programming concepts are difficult to enhance i.e. it does not support the integration of new modules.
The procedures and functions are the fundamental concepts of procedure oriented programming concepts. The design of these fundamental concepts is very weak therefore they are not suitable for developing real time and complex applications.

Note: Procedure oriented languages also called as structured programming languages.

Object Oriented Programming Concepts: The applications that are developed with help of objects and classes are said to follow object oriented programming concepts.

Example of object oriented languages: Java, C++, .NET, Smalltalk etc.

Even though C++ is an object oriented programming language, but according to the programming language experts it is called as partial object programming language because of the following reasons:

a) In a C++ application we can write some code inside the class and some code outside the class.
b) An application in C++ can be developed without following any object oriented programming concepts.
c) The friend function concept can violate or break any level of security provide to the data.

The objects and classes are the fundamentals concepts of object oriented programming.

Object: Any entity that exists physically in the real word which requires some memory will be called as an object. Every object will contain some properties and some actions. The properties are the data or information which describes the object and they are represented by variables. Actions are the tasks or the operations performed by the object and they are represented by methods.

Class: A class is a collection of common properties and common actions of a group of objects.

A class can be considered as a plan or a model or a blue print for creating an object. For a class we can create any number of objects, without the class object creation is not possible. An Object is an instance of a class.

All the object oriented programming concepts are derived from the real world, from the human being lives so that programming becomes simpler. The programmer can understand the concepts easily an implement them without any difficulty. The design of the object oriented programming concepts is very strong and they are suitable for developing real time and complex applications.

The various object oriented concepts are:

1) Encapsulation
2) Inheritance
3) Polymorphism

Encapsulation: It is a process of binding the variables and methods into a single entity.

- Encapsulation can be achieved with help of a class.
- Using encapsulation we can improve the maintenance of the application
- Using encapsulation we can implement abstraction (data hiding) which will decide what to hide and what to present that is we can implement security.
- Using encapsulation we can isolate or separate the members from one class to another class and thereby reduce the debugging time.

Inheritance: It is a process of acquiring the members from one class to another class.

- Using Inheritance we can achieve reusability and thereby reduce the code size and reduce the development time of the application.

Polymorphism: If a single entity shows multiple behaviors or multiple forms, then it is said to be polymorphism.

- Using polymorphism we can achieve flexibility where single entity can perform different operations according to the requirements.

Encapsulation

Class: Using a class we can achieve encapsulation and using a class we can create user defined data type where we can store any number of values and any type of values, according to the application requirement.

A java program can contain any number of classes. A class can contain variables and methods which are together called as members of the class.

Syntax for a Class:

```
class ClassName {
 datatype variableName1;
 datatype variableName2;
 datatype variableName3;
 returntype methodName1(list of parameters) {
   statements;
 }
 returntype methodName2(list of parameters) {
   statements;
 }
```

Program 24:

```
class Student {
 int rollNo;
 double marks;
 String name;
 void display() {
   System.out.println("RollNo : "+rollNo);
   System.out.println("Marks : "+marks);
   System.out.println("Name : "+name);
 }
 public static void main(String[] ar) {
   System.out.println("Student Information");
   Student st = new Student();
   st.display();
 }
```

When the above java program Student.java is compiled, the compiler will verify whether the ja code available in the program is valid or not, if valid the compiler will generate the .class file. Th .class file generated by the compiler will be provided to the JVM for execution.

The execution of the java program will be done by the JVM and it begins from the main(). Initia the java stack will be empty. The JVM will begin the execution of the program by calling main When a method is called for execution then that method will be pushed into the java stack a then begins the execution.

Student st = new Student();

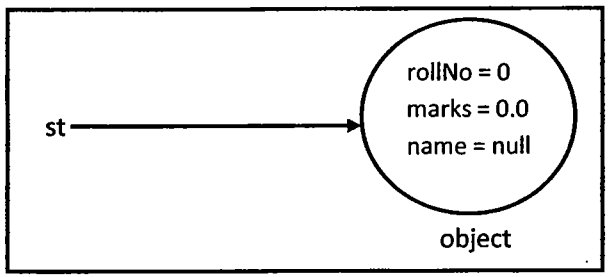
The above statement will create an object of Student class. Creating an object means allocati memory for the instance variables in heap memory.

Instance Variable: If a variable is declared inside the class and outside the methods, then they a called as instance variables.

If an instance variable is declared and not initialized then it will be initialized automatically wi default values of that type.

byte	0
short	0
int	0
long	0
float	0.0
double	0.0
char	space
boolean	false

Employee	null
Customer	null
String	null
AnyClass	null



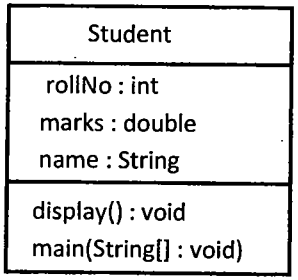
A class can contain any number of methods, but the JVM can call only main() for execution. If w want other methods to be executed, then it is the responsibility of the programmer to call th other method which has to be executed.

st.display();

The above statement will call display() for execution. When the display() is called for executio the display() will be pushed on to the top of the java stack and then begins the execution. Whe a method is completely executed then that method will be poped(removed) from the java stac

Every class in a java application will be represented in the form of class diagram. The class diagram will be represented in rectangle shape with 3 partitions providing the information of class name, variables and methods respectively.

Class Diagram



A class in a java program can contain variables and methods.

Variable: The purpose of a variable is to store some value.

In a java program we can declare any number of variables. The declaration of the variable in a java program is dynamic i.e. we can declare the variables anywhere in the program.

The java language is called as **strongly typed language**. Any language can be called as strongly typed, if the variables are first declared and then used and whose compiler checks for type compatibility.

Method: The purpose of a method is to perform a task or an operation.

Syntax of method:

```
returntype methodName (list of parameters) {
   statements;
```

Every method will be divided into two parts:

1) Method Declaration
2) Method Definition

Method Declaration / Header / Prototype: The method declaration consists of three parts they are returntype, methodName and list of parameters.

- Specifying the return type and method name is mandatory, whereas specifying the list of parameters is optional.
- We can specify any number of parameters and they can be of any type separated by a comma.
- The return type of the method should be specified just before the method name.
- If we don't want the method to return any value, then specify the return type as void. If we want the method to return a value then they do not specify the return type as void.

Method Definition / Body / Implementation: The method definition is a group of statement that we specify with in a pair of { }.

```
{
    statements;
}
```

In the method definition we can specify any number of statements. If we do not specify an statements in the method definition, then it should be called as **empty definition or nu implementation.**

If the return type of the method is specified as void, then we should not specify any retur statement but if the return type of the method is not void then we must specify the retur statement in the method definition.

Syntax for return statement:

```
return value;
```

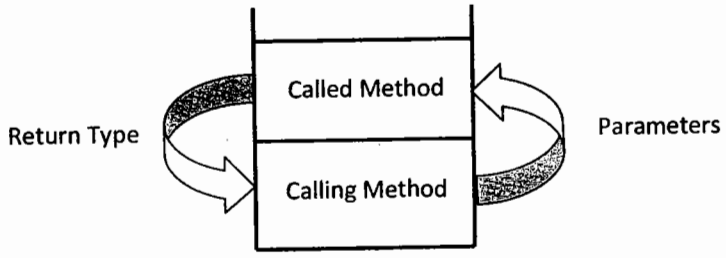
Example:

```
return 123;
return 3.14;
return 'a';
return true;
return "hello";
```

- The value that we return must be compatible with the return type that is specified.
- The return statement can be specified anywhere in the method definition.
- A method can return at must one value.

Method Invocation: The process of calling a method for the purpose of execution will be calle as invocation.

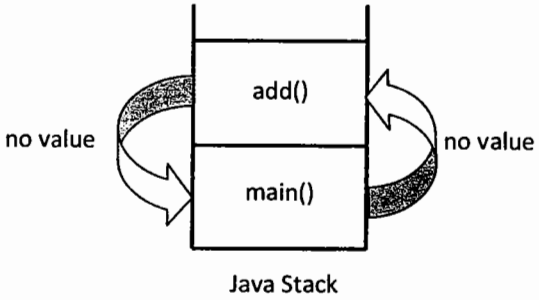
If a method is calling another method, then it is called as **Calling method** and if a method i called by another method, then it is called as **Called method.**
The values that we transfer from calling method to called method are known as **parameters** an the values that we transfer from called method to calling method are known as **return type.** W can transfer any number of parameters and we can return atmost one value.



Method without return type and without parameters:

Program 25:

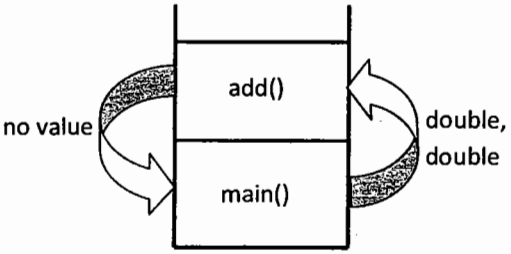
```
class Addition {
    void add() {
        int a = 10;
        int b = 20;
        int c = a + b;
        System.out.println(c);
    }
    public static void main(String[] args) {
        Addition ad = new Addition();
        ad.add();
    }
}
```



Method without return type and with parameters:

Program 26:

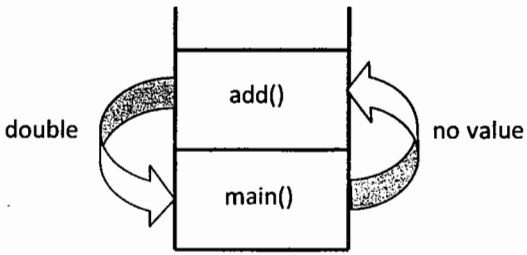
```
class Addition {
    void add(double a, double b) {
        double c = a + b;
        System.out.println(c);
    }
    public static void main(String[] args) {
        Addition ad = new Addition();
        double x = 1.5;
        double y = 1.6;
        ad.add(x,y);
    }
```



Method with return type and without parameters:

Program 27:

```
class Addition {
    double add() {
        double a = 1.4;
        double b = 5.5;
        double c = a + b;
        return c;
    }
```

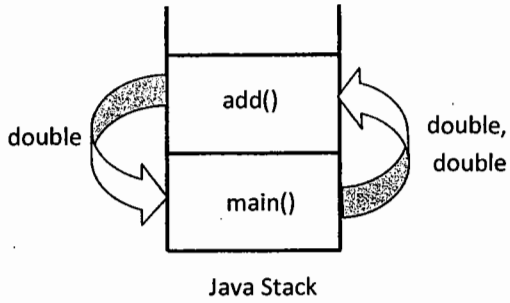


```
    public static void main(String[] args) {
        Addition ad = new Addition();
        double res = ad.add();
        System.out.println(res);
    }
}
```

Method with return type and with parameters:

Program 28:

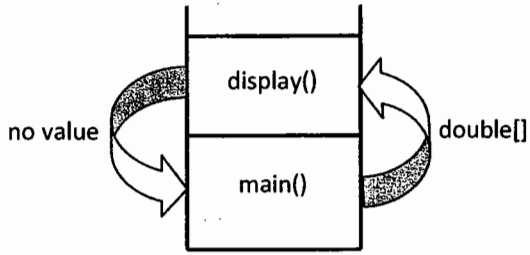
```
class Addition {
    double add(double a, double b) {
        double c = a + b;
        return c;
    }
    public static void main(String[] args) {
        Addition ad = new Addition();
        double res = ad.add(2.3, 4.5);
        System.out.println(res);
    }
}
```



Method taking array as parameter:

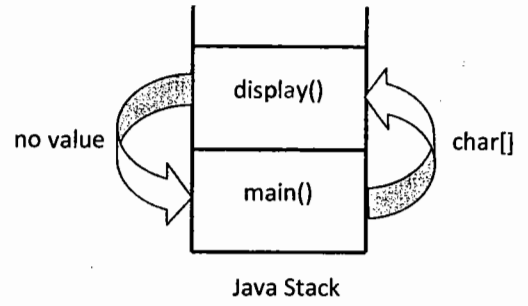
Program 29:

```
class ArrayDemo {
    void display(double[] temp) {
        for(double x : temp) {
            System.out.println(x);
        }
    }
    public static void main(String[] args) {
        ArrayDemo ad = new ArrayDemo();
        double arr[] = {1.2, 3.4, 5.6, 7.8};
        ad.display(arr);
        double nos[] = {1.1, 2.2, 3.3, 4.4, 5.5};
        ad.display(nos);
    }
}
```



Program 30:

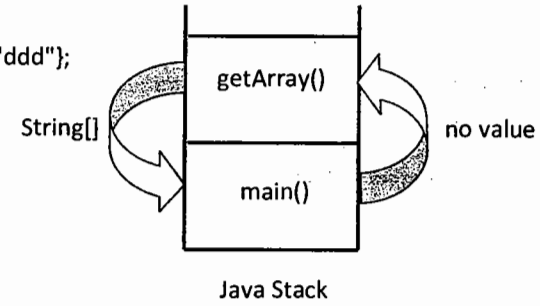
```
class ArrayDemo {
    void display(char[] temp) {
        for(char x : temp) {
            System.out.println(x);
        }
    }
    public static void main(String[] args) {
        ArrayDemo ad = new ArrayDemo();
        char[] arr = {'a', 'b' ,'c' ,'d'};
        ad.display(arr);
    }
}
```



Method returning an array:

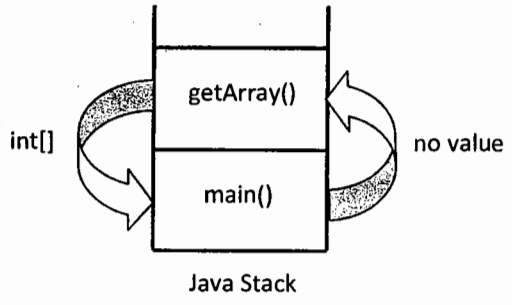
Program 31:

```
class ArrayDemo {
    String[] getArray() {
        String[] name = {"aaa", "bbb", "ccc", "ddd"};
        return name;
    }
    public static void main(String[] args) {
        ArrayDemo ad = new ArrayDemo();
        String[] abc = ad.getArray();
        for(String x : abc) {
            System.out.println(x);
        }
    }
}
```



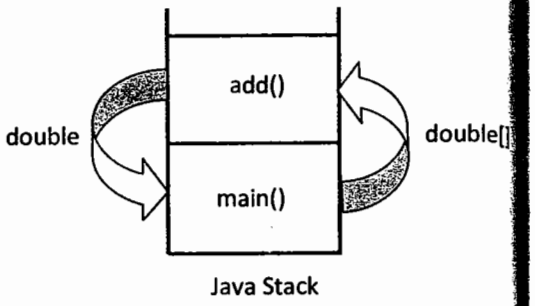
Program 32:

```
class ArrayDemo {
    int[] getArray() {
        int arr[] = {10, 20, 30, 40, 50};
        return arr;
    }
    public static void main(String[] args) {
        ArrayDemo ad = new ArrayDemo();
        int[] abc = ad.getArray();
        for(int x : abc) {
            System.out.println(x);
        }
    }
}
```



Program 33:

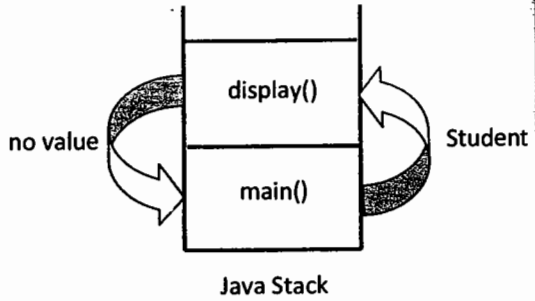
```
class ArrayDemo {
    double add(double[] abc) {
        double sum = 0.0;
        for(double x : abc) {
            sum = sum + x;
        }
        return sum;
    }
    public static void main(String[] args) {
        ArrayDemo ad = new ArrayDemo();
        double[] arr= {1.1, 2.2, 3.3, 4.4};
        double res = ad.add(arr);
        System.out.println(res);
    }
}
```



Method taking Object(user defined data type) as parameter:

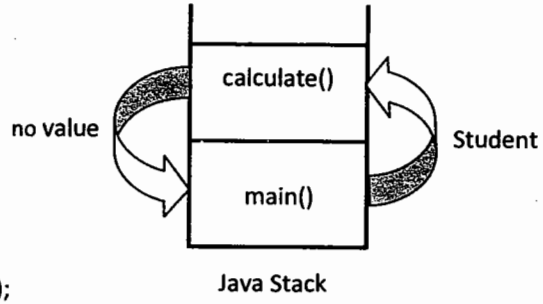
Program 34:

```
class Student {
    int rollNo = 123;
    double marks = 78.9;
    String name = "abcd";
}
class ObjectDemo {
    void display(Student stu) {
        System.out.println(stu.rollNo);
        System.out.println(stu.marks);
        System.out.println(stu.name);
    }
    public static void main(String[] args) {
        Student st = new Student();
        ObjectDemo od = new ObjectDemo();
        od.display(st);
    }
}
```



Program 35:

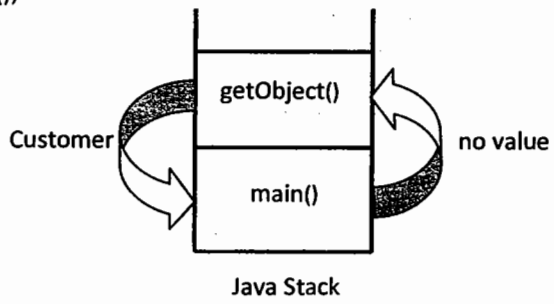
```
class Student {
    int rollNo = 123, m1 = 60, m2 = 70, m3 = 80;
}
class ObjectDemo {
    void calculate(Student stu) {
        System.out.println(stu.rollNo);
        int sum = stu.m1 + stu.m2 + stu.m3;
        System.out.println(sum);
        int avg = sum / 3;
        System.out.println(avg);
    }
    public static void main(String[] args) {
        Student st = new Student();
        ObjectDemo od = new ObjectDemo();
        od.calculate(st);
    }
}
```



Method Returning An Object:

Program 36:

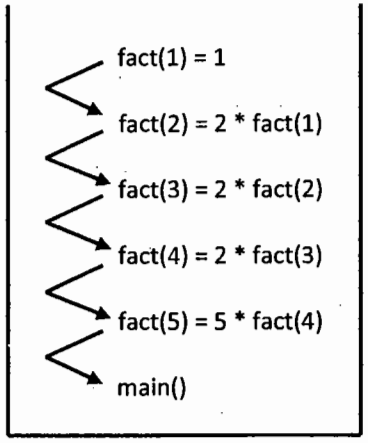
```
class Customer {
    int custId = 1234;
    String name = "pqrs";
}
class ObjectDemo {
    public static void main(String[] args) {
        ObjectDemo od = new ObjectDemo();
        Customer c2 = od.getObject();
        System.out.println(c2.custId);
        System.out.println(c2.name);
    }
    Customer getObject() {
        Customer c1= new Customer();
        return c1;
    }
```



Recursion: If a method is calling itself multiple times, then it is called as recursion. Such method are called as recursive methods.

Program 37:

```
class Factorial {
    public static void main(String[] args) {
        Factorial f = new Factorial();
        for(int i=1;i<=10;i++) {
            int res = f.fact(i);
            System.out.println(i+" ! = "+res);
        }
    }
    int fact(int n) {
        if(n==1)
        return 1;
        int x = n * fact(n-1);
        return x;
    }
}
```



Java Stack

Program 38:

```
class FibonacciSeries {
    int fibonacci(int n) {
        if ((n == 0) || (n == 1))
            return n;
        else
            return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static void main(String[] args) {
        FibonacciSeries fs = new FibonacciSeries();
        for(int i=0;i<=10;i++) {
            int res = fs.fibonacci(i);
            System.out.println("Fibonacci of "+i+" is "+res);
        }
    }
}
```


Instance variable: If a variable is declared inside the class and outside the methods, then it is called as instance variable.

If an instance variable is declared and not initialized, then it will be initialized automatically with default value. If we do not want the instance variable to contain default values, then we can initialize the instance variable with our own values in the following two locations:

1. At the time of declaration
2. By using a constructor.

Constructor

A constructor is a block, which is used for initializing the instance variables.

- The name of the constructor must be same as that of the class name.
- A constructor should not have any return type, not even void.
  Note: If we specify any return type to a constructor then the code will be valid, but it will be considered as a method instead of a constructor.
- The constructor will be executed during the object creation time.
- The constructor will execute one time for every object that is created.
- A constructor can have parameters. Based on the number of parameters, the constructors are classified into two types:
  1. Zero parameterized constructor
  2. Parameterized constructor

1) **Zero parameterized constructor:** If a constructor does not have any parameters, then it is called as zero parameterized constructor.

Syntax:

```
class ClassName {
    ClassName() {
    }
}
```

2) **Parameterized constructor:** If a constructor contains some parameters, then it is called as parameterized constructor. A parameterized constructor can contain any number of parameters and any type of parameters separated by comma(,).

Syntax:

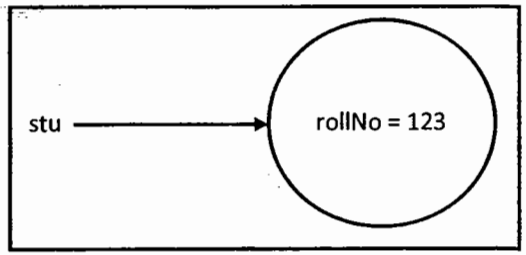
```
class ClassName {
    ClassName(list of parameters) {
    }
}
```

- Every class in java will contain a constructor whether we specify or not. The compiler will provide a zero parameterized constructor, only if the class does not have any constructor.

Note: Compiler cannot provide a parameterized constructor.

Program 39:

```
class Student {
    int rollNo;
    Student() {
        rollNo =123;
    }
    void display() {
        System.out.println(rollNo);
    }
    public static void main(String[] args) {
        Student stu = new Student();
        stu.display();
    }
}
```



Heap

Instance Variable

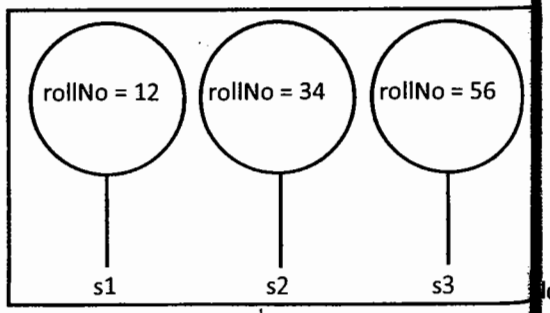
If a variable is declared inside the class and outside the methods and outside the constructor then it is called as instance variable.

- The memory for the instance variable will be allocated during object creation time.
- The memory for the instance variable will be allocated in heap memory.
- The memory for the instance variable will be allocated one time for every object that is created.
- Every object will contain its own copy of the instance variable.

- ❖ If we want all the objects to contain instance variable with same value, then initialize that instance variable either at the time of declaration or by using a zero parameterized constructor.
- ❖ If we want all the objects to contain instance variable with different value, then initialize that instance variable by using a parameterized constructor.

Program 40:

```
class Student {
  int rollNo;
  Student(int x) {
    rollNo = x;

  }
  void display () {
    System.out.println(rollNo);
  }
  public static void main(String args[]) {
    Student s1 = new Student(12);
```



heap


```
    s1.display();
    Student s2 = new Student(34);
    s2.display();
    Student s3 = new Student(56);
    s3.display();
  }
}
```

this: this keyword refers to current instance(object) of a class.
This keyword can be used to access instance members (instance variables, instance methods) of a class.

Local variable: If a variable is declared inside the class and inside the method or inside the constructor, then it is called as local variable.

Specifying this keyword is sometimes optional and sometimes mandatory.

Optional: If there is no confusion between instance variable and local variable, then specifying this keyword is optional. In such case, if we don't specify this keyword then the compiler will automatically specify this keyword and access the instance variable.
Mandatory: If there is confusion between instance variable and local variable, then specifying this keyboard is mandatory. In such case if we don't specify this keyword then the compiler will also not specify this keyword and access local variable. If we want to access instance variable, then the program has to explicitly specify this keyword.

Program 41:

```
class Sample {
  int a = 11;
  int b = 22;

  void display() {
    int b = 33;
    int c = 44;
    System.out.println(this.a);
    System.out.println(this.b);
    System.out.println(b);
    System.out.println(c);
  }

  public static void main(String args[]) {
    Sample s = new Sample();
    s.display();
  }
```

Note: This keyword should not be applied to local variables.

Instance Variable: If a variable is declared inside the class and outside the methods and outside the constructors without static keyword, then it is called as instance variable or non static variable.

The memory for the instance variable will be allocated multiple times i.e. one time for every object that is created. If we do not want to allocate the memory for a variable multiple times then declare the variable with static keyword.

- static keyword can be applied to both variables and methods.

Static Variable

If a variable is declared inside the class, outside the methods and outside the constructors with static keyword, then it is called as static variable.

Syntax for static variable:
static datatype variableName;

- The memory for the static variable will be allocated during the class loading time.
- The memory for static variable will be allocated in method area.
- The memory for static variable will be allocated one time for entire class.
- All the objects will share the same copy of the static variable.

Syntax for static method:

```
static returntype methodName(list of parameters) {
   statements;
}
```

- The instance members(instance variables and instance methods) can be accessed only by using reference(object) whereas the static members(static variables and static methods) can be accessed either by using a class name or by using a reference(object).

Note: It is recommended to access the static members by using class name, because we cannot guarantee the existence of the object.

The class can contain both instance members(instance variables, instance methods) and static members(static variables, static methods).

Program 42:

```
class Student {
    int rollNo = 123;
    static int code = 456;
    public static void main (String []args) {
        Student stu = new Student();
        System.out.println(stu.rollNo);
        System.out.println(stu.code);
        System.out.println(Student.code);
    }
}
```

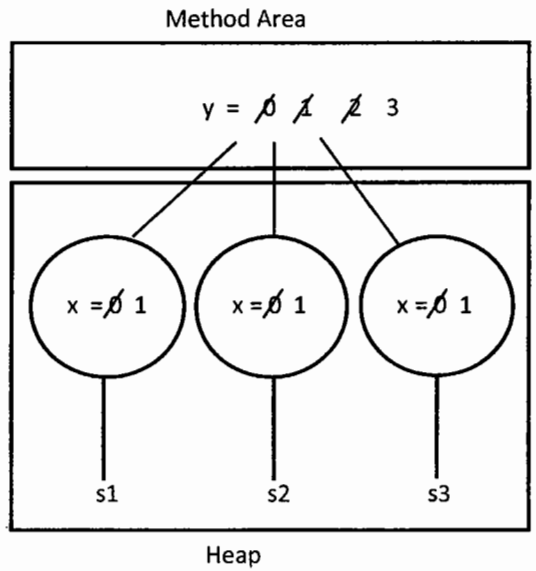

- An instance method can access both instance members and static members directly whereas a static method can access only static members directly.
- If a static variable is declared and not initialized, then it will be initialized automatically with default value. If we don't want static variable to contain default value, then we can initialize the static variable with our own value at the time of declaration.

Program 43:

```
class Sample {
  int x = 11;
  static int y = 22;
  void show() {
     System.out.println(x);
     System.out.println(y);
  }
  static void display() {
     //System.out.println(x);
     System.out.println(y);
  }
  public static void main(String[] args) {
     Sample s = new Sample();
     s.show();
     Sample.display();
  }
}
```

Program 44:

```
class Sample {
  int x;
  static int y;
  public static void main(Stirng[] args) {
     Sample s1 = new Sample();
     s1.x++;
     s1.y++;
     System.out.println(s1.x+" : "+s1.y);
     Sample s2 = new Sample();
     s2.x++;
     s2.y++;
     System.out.println(s2.x+" : "+s2.y);
     Sample s3 = new Sample();
     s3.x++;
     s3.y++;
     System.out.println(s3.x+" : "+s3.y);
  }
}
```



Rules for accessing instance and static members

instance-instance combination: An instance method can access instance members(instance variables and instance methods) directly provided they belong to the same class, otherwise they must be accessed only by using reference(object).

instance-static combination: An instance method can access static members(static variables and static methods) directly provided they belong to the same class, otherwise they must be accessed either by using a class name or a reference(object).

static-static combination: A static method can access static members(static variables, static methods) directly provided they belong to the same class, otherwise they must be accessed either by using a class name or a reference(object).

static-instance combination: A static method can access instance members(instance variables and instance methods) only by using a reference(object) whether they belongs to same class or not.

Program 45:

```
class Access1 {
    int x = 11;
    static int y = 22;

    void m1() {
        System.out.println("instance method m1");
    }

    void m2() {
        System.out.println(x);    // rule 1
        m1();                     //rule 1
        System.out.println(y);    // rule 2
        m3();                     // rule 2
        System.out.println("instance method m2");
    }

    static void m3() {
        System.out.println("instance method m3");
    }

    public static void main(Stirng[] args) {
        System.out.println(y);        // rule 3
        m3();                         // rule 3
        Access1 a = new Access1();
        System.out.println(a.x);      // rule 4
        a.m2();                       // rule 4
    }
}
```



```
class Access2 {
    void m4(){
        Access1 c = new Access1();
        System.out.println(c.x);            // rule 1
        c.m1();                             // rule 1
        System.out.println(Access 1.y);     // rule 2
        c.m3();                             // rule 2
        System.out.println("instance method m4");
    }
    public static void main(Stirng[] args) {
        Access1 d = new Access1();
        System.out.println(Access1.y);      // rule 3
        d.m3();                             // rule 3
        System.out.println(d.x);            // rule 4
        d.m1();                             // rule 4
        Access2 b = new Access2();
        b.m4();                             // rule 4
    }
}
```

Explain public static void main(String[] args)

- **public:** The main method should be declared as public, so that the JVM can access the main method from any location.
- **static:** The main method should be declared as static, so that the JVM can invoke the main method directly by using class name.
- **void:** The caller of the main method is JVM and the JVM does not expect any value from the main method, therefore the main method should not return any value to JVM hence we specify the return type as void.
- **main():** main is the name of the method, a valid java identifier, following the java coding conventions and the name is fixed.
- **String[]:** It is used for storing command line arguments.

Note: The name of the string array can be any valid java identifier.

Rule: If a declaration contains multiple modifiers, then we can specify them in any sequence.

```
tatic public void main(String[] abc)
ublic static void main(String[] abc)
```

xplain System.out.println()

- System is a predefined class available in java.lang package.
- out is a reference variable of PrintStream class declared as static in System class.
- println() is predefined method available in PrintStream. PrintStream is a predefined class available in java.io package.

ut is a reference variable declared as static in System class and therefore it can be accessed irectly by using class name(System.out).

System.out will provide an object of PrintStream class, using which we can access the method of PrintStream class and therefore we write System.out.println().

```
java.io
class PrintStream{
    void println() {
    }
}
```

```
java.lang
class System{
    static PrintStream out = new PrintStream();
}
```

Local Variable

The variable that is declared inside the class and inside the method or inside the constructor is called as local variable.

- The memory for the local variable will be allocated during the method or constructor invocation time.
- The memory for the local variable will be allocated in the java stack.
- The memory for the local variable will be allocated one time for every invocation. But only one copy of the local variable will be available.
- If a local variable is declared and not initialized, then it will not be initialized automatically. It is the responsibility of the programmer to initialize the local variable either at the time of declaration or any where before using it.

Note: If a local variable is not used, then it need not be initialized.
Note: static keyword cannot be applied to local variable.

Program 46:

```
class Sample {
    int a = 11;
    static int b = 22;
    void show() {
        int c = 33;
        int d;
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        d = 44;
        System.out.println(d);
    }
    public static void main(String[] args) {
        Sample s = new Sample();
        s.show();
    }
}
```


Parameter

The values that are passed to a method or a constructor are called as parameters. The parameters of the method or a constructor will be initialized during their invocation time.

The parameters of the method or constructor should be considered as local variables.

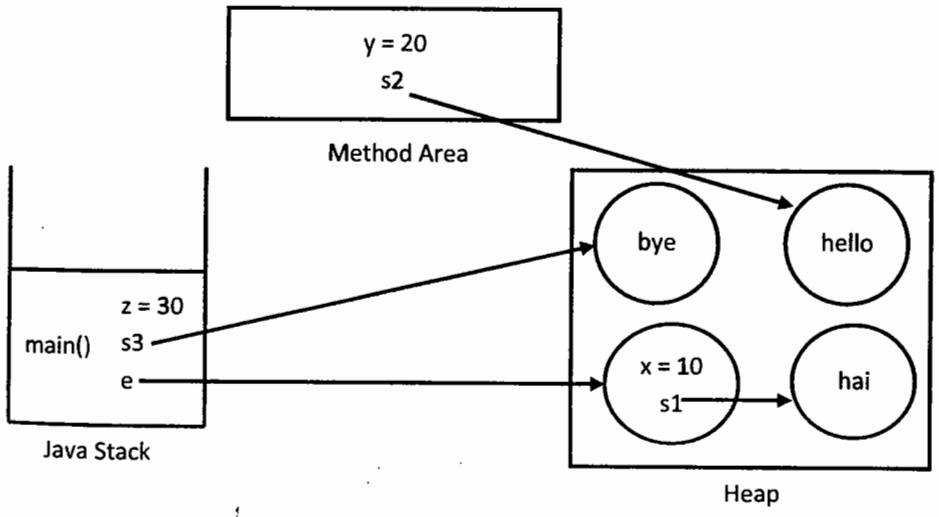
Reference Variable

If a variable is referring or pointing to an object, then it is called as a reference variable.

The reference variable of any class can be declared as either instance or static or local.

Program 47:

```
class Example {
  int x = 10;
  String s1 = new String("hai");
  static int y = 20;
  static String s2 = new String("hello");
  public static void main(String[] args) {
     int z = 30;
     String s3 = new String ("bye");
     Example e = new Example();
  }
```



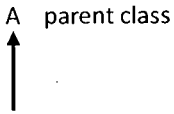
Inheritance

Inheritance is a concept of acquiring the members from one class to another class.

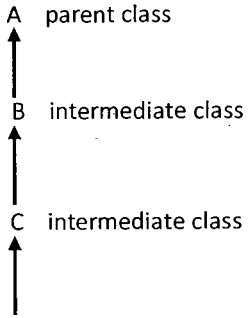
Using inheritance we can achieve reusability there by reducing the code size and reduce the development time of the application.

The Java language supports two types of inheritance

1. Single level inheritance
2. Multi level inheritance



Single level inheritance:
If the inheritance concept contains one parent class and one child class, then it is called as single level Inheritance.



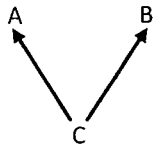
Multi level inheritance:
If the inheritance concept contains one parent class, one child class and one or more intermediate classes, then it is called multi level inheritance.

Parent class / Super class / Base class: If a class provides members to another class, then it is called as parent class.

Child class / Sub class / Derived class: If a class is receiving members from another class, then it is called as child class.

Intermediate class: If a class acts as both parent class and child class, then it is called as intermediate class.

Java doesn't support multiple Inheritance.



Multiple Inheritance: If a class is inheriting from two or more classes then it is called as multiple inheritance.

Q) Why the java language is not supporting multiple inheritance.
A) If both the parent classes contains same members, then it will be confusion to the child class whether to access the members from first parent or from the second parent. To avoid this confusion, there is no concept in java language and therefore the java does not support multiple Inheritance.

To achieve inheritance concept in java language we use **extends** keyword.

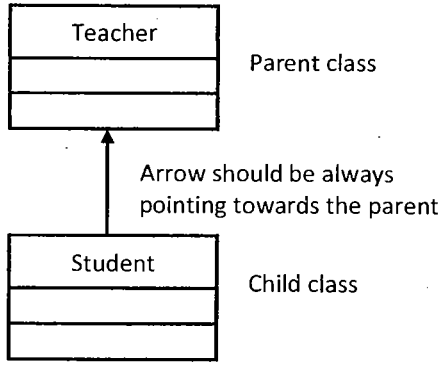
Syntax:

```
class ParentClass {

}
class ChildClass extends ParentClass {

}
```

Example:

```
class Teacher {

}
class Student extends Teacher {

}
```

Class Diagram showing Inheritance



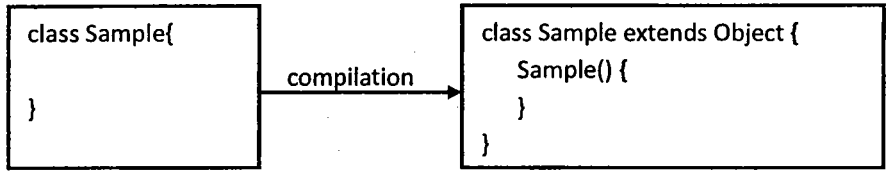
Program 48:

```
class Cone {
   int x = 11;
   void show() {
      System.out.println("cone class show()");
   }
}

class Ctwo extends Cone {
   int y = 22;
   void display() {
      System.out.println("ctwo class display()");
   }
}

class Inheritance {
   public static void main(String[] args) {
      Ctwo c = new Ctwo();
      System.out.println(c.x);
      c.show();
      System.out.println(c.y);
      c.display();
   }
}
```

If we create an object of parent class, then we can access the members of only parent class, but if we create an object of child class, then we can access the members of both parent class and child class.

Every class in java either predefined or user defined will be subclass of Object class either directly or indirectly. The Object class is called as super most class of all classes in java.

When a java program is compiled and if the class does not contain any parent then the compile will automatically specify extends Object and makes that class as subclass of Object class.



When we create an object of any class, the constructor of that class has to be executed. Before executing the constructor of that class it will verify whether the class contains any parent class or not, if available then it will execute the parent class constructor. But before executing the constructor of parent class it will verify whether the parent class contain any parent or not and this process will continue until it reaches the Object class. Once the control reaches the Object class, it will begin the execution of the constructors.

The invocation of the constructor will be in the order of child to parent class and the execution of the constructor will be in the order of parent to child class and it will always begin from Object class.

Program 49:

```
class Cone {
    Cone() {
        System.out.println("Cone constructor");
    }
}
class Ctwo extends Cone {
    Ctwo() {
        System.out.println("Ctwo constructor");
    }
}
class Cthree extends Ctwo {
    Cthree() {
        System.out.println("Cthree constructor");
    }
}
class Inheritance {
    public static void main(String[] args) {
        Cthree c = new Cthree();
    }
}
```


Types of Relationships

1) **IS-A Relationship:** If a class is inheriting another class then it is called as IS-A relationship.

   Example:

```
class Vehicle {
}
class Car extends Vehicle {
}
```

   Car IS-A Vehicle
   Using IS-A relationship we can achieve reusability.

2) **HAS-A Relationship:** If a class contains an object of another class, then it is called as HAS-A relationship.

   Example:

```
class Engine {
}
class Car {
    Engine object
}
```

   Car HAS-A Engine
   Using HAS-A relationship we can achieve reusability.

In a java application we can achieve reusability in two ways.

1. IS-A Relationship
2. HAS-A Relationship

Polymorphism

If a single entity shows multiple forms or multiple behaviors, then it is called as polymorphism.

Using polymorphism we can achieve flexibility, where a single entity can perform different operations according to the requirement.

Polymorphism is classified into two types and they are

1) Compiler Time Polymorphism
2) Run Time Polymorphism

1) **Compiler Time Polymorphism:** If the polymorphic nature of an entity is decided by the compiler during the compilation time, then it is called as compiler time polymorphism.
   To achieve the compile time polymorphism, we take the help of a concept called method overloading.

Method Overloading: The process of specifying multiple methods, having different signature with same method name is called as method overloading.

Method Signature: The method signature includes 4 parts, they are

1) Method Name
2) Number Of Parameters
3) Type Of Parameters
4) Order Of Parameters

A class can contain any number of methods and no two methods must have same signature i.e. every method must have different signature.

In method overloading, the methods must have same name, having either different no of parameters or different type of parameters or different order of parameters. The methods that participate in method overloading are called as overloaded methods.

Note: The method signature does not include parameter names and does not include return type of the method.

- In method overloading, the compiler will decide which method has to be executed and therefore called **compile time polymorphism**.
- When a method is invoked, the compiler will perform a process of linking or binding the method invocation with the method definition, whose signature is matching. Once the compiler performs the binding it cannot be changed and therefore called as **static binding**.
- Since the binding is performed before the execution of the program, it is called as **early binding**.

Program 50:

```
class Addition {
    void add(int x, int y) {
        System.out.println("Result1 : "+(x+y));
    }
    void add(int x, int y, int z) {
        System.out.println("Result2 : "+(x+y+z));
    }
    void add(int x, double y) {
        System.out.println("Result3 : "+(x+y));
    }
    void add(double x, int y) {
        System.out.println("Result4 : "+(x+y));
    }
    public static void main(String[] args) {
        ad.add(3,4);
        ad.add(3,4,5);
        ad.add(3,4.5);
        ad.add(3.4,5);
    }
}
```


Q) Why should be implement method overloading.
A) We implement method overloading to provide flexibility to the user so that, the user can use one method to perform different operations according to the requirement.

- Method overloading can be applied to instance methods, static methods or both and even to main method.

Program 51:

```
class Sample {
    static void show(int x) {
        System.out.println("int parameter");
    }
    static void show(double y) {
        System.out.println("double parameter");
    }
    public static void main(String[] args) {
        show(12);
        show(1.2);
    }
}
```

Overloading of main Method:

Program 52:

```
class Sample {
    public static void main(String[] args) {
        System.out.println("string array");
    }
    public static void main(String args) {
        System.out.println("string parameter");
    }
    public static void main(int[] args) {
        System.out.println("int array");
    }
    public static void main(int  args) {
        System.out.println("int parameter");
    }
}
```

When a program is executed, the JVM can invoke only main method with string array as the parameter, but as a programmer we can invoke any method with any parameter including main method.

Program 53:

```
class Demo {
    public static void main(String[] args) {
        System.out.println("demo class main method");
        Sample.main(12);
        Sample.main("hello");
        Sample.main(new int[]{4,5,6,7,8});
        Sample.main(new String[]{"aaa", "bbb","ccc"});
    }
}
```

Anonymous Array: An array is said to an anonymous array if the array does not have any name. The anonymous array has to be used when we want to use the array only one time. The anonymous arrays will be generally used in the process of method invocation.

Method overloading can be implemented either in a single class or in two different classes that have IS-A-Relationship.

Program 54:

```
class Demo {
    void show(int x) {
        System.out.println("int value");
    }
}
class Test extends Demo {
    void show(char x) {
        System.out.println("char value");
    }
}
```

2) **Runtime polymorphism:** If the polymorphic nature of an entity is decided by the JVM during the runtime, then it is called as runtime polymorphism.

To achieve runtime polymorphism, we take the help of a concept called method overriding.

Method Overriding: The process of specifying two methods with same signature and same return type in two different classes that have IS-A relationship is called as method overriding.

- When a method is invoked, the decision of which method to be executed is taken by the JVM at run time and therefore called **run time polymorphism.**
- When a method is invoked, the JVM will perform a process of binding or linking the method invocation with method definition of that class whose object is created. Since the binding is done according to the object that is created, it is called as **dynamic binding**.
- Since the binding is performed after the beginning of execution of the program and after the object is created, it is called as **late binding.**

Program 55:

```
class Parent {
    void msg () {
        System.out.println("good morning");
    }
}
class Child extends Parent {
    void msg() {
        System.out.println("good night");
    }
}
class Polymorphism {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.msg();
        Child c = new Child();
        c.msg();
        Parent p1 = new Child();
        p1.msg();
    }
}
```

- The reference variable of any class can refer to its own object or it can refer to its child class object.

```
Object O1 = new Object();
Object O2 = new Customer();
Object O3 = new Employee();
Object O4 = new String();
```

Q) Why should we implement method overriding

A) If the child class does not want the parent class method implementation, then the child class will override that method to provide new implementation.

- The method over riding can be applied to only instance methods if the same concept is applied to static method then it is called as **method overhiding.**
- In method overriding or in method overhiding both the methods should be either instance or static.
- Method overriding or method overhiding can be implemented only in two different classes that have IS-A relationship.

Constructor Overloading: The process of specifying multiple constructors with different signature is called as constructor overloading.

Program 56:

```
class Rectangle {
    int length, breadth;
    Rectangle() {
        length = breadth = 3;
    }
    Rectangle(int x) {
        length = breadth = x;
    }
    Rectangle(int length, int breadth) {
        this.length = length;
        this.breadth = breadth;
    }
    void area() {
        System.out.println("area : "+(length * breadth));
    }
    public static void main (String[] args) {
        Rectangle r1 = new Rectangle();
        r1. area();
        Rectangle r2 = new Rectangle(4);
        r2. area();
        Rectangle r3 = new Rectangle(5,6);
        r3. area();
    }
}
```

Q) Why should implement constructor over loading

A) We implement constructor overloading to provide flexibility to the user so that, the user can create an object in different ways by passing different values.

Note: In any class, if we are specifying a parameterized constructor then we are recommended to specify a zero parameterized constructor whether required or not.

- Constructor overloading can be implemented only in a single class, it can't be implemented in two different classes even if they have IS-A relationship.
- The concept of overriding is possible only there is a concept of Inheritance. The constructors of a class can't be inherited into another class and therefore constructor overriding is not possible.

this keyword:

this keyword will refer to the current instance(object) of a class.

Using this keyword we can access instance members(instance variables and instance methods) of a class.

Specifying this keyword is sometimes optional, sometimes mandatory.

Optional: If there is no confusion between instance variable and local variable then, specifying this keyword is optional. In such case, if we don't specify this keyword then, the compiler will automatically specify this keyword and access instance variable.

Mandatory: If there is confusion between instance variable and local variable then, specifying this keyword is mandatory. In such case if we don't specify this keyword then the compiler will also not specify this keyword and it will access local variable. If we want to access the instance variable then the programmer has to explicitly specify this keyword.

Note: this keyword cannot be applied to local variable.

Note: There is no concept of local methods i.e. we cannot specify a method inside another method.

Program 57:

```
class Sample {
    int a = 11;
    int b = 22;
    void show() {
        int b = 33;
        int c = 44;
        System.out.println(this.a);
        System.out.println(this.b);
        System.out.println(b);
        System.out.println(c);
        this.msg ();
    }
    public static void main (String[] args) {
        Sample s = new Sample();
        s.show()
    }
}
```

this(): this() is used to refer to the zero parameterized constructor of current class.
this(...): this(...) is used to refer to the parameterized constructor of the current class.

Rules:

- A constructor can invoke at most one constructor.
- The invocation of the constructor must be specified as the first statement.
- The invocation of a constructor must not form a cycle or loop.

Program 58:

```
class Sample {
    Sample() {
        this(8);
        System.out.println("sample zero constructor");
    }
    Sample(int x) {
        System.out.println("sample int constructor");
    }
    Sample (double y) {
        this();
        System.out.println("sample double constructor");
    }
    public static void main (String[] args) {
        Sample s = new Sample(1.2);
    }
}
```

this:

- this keyword can be used to access the instance members of the current class.
- this keyword can be specified either in constructors or instance methods of the current class.
- this keyword cannot be specified in static methods.

this():

- this() is used to access zero parameterized constructor of the current class.
- this() can be specified in any parameterized constructor of any class.
- this() cannot be specified in methods(either instance or static).

this(...):

- this(...) is used to access parameterized constructor of the current class.
- this(...) can be specified either in zero parameterized constructor or other parameterized constructor of the current class.
- this(...) cannot be specified in methods(either instance or static).

Note: this()/this(...) are designed for calling constructors from other constructors.

super keyword:

The super keyword is used for accessing instance members(instance variables and instance methods) of parent class.
Specifying the super keyword is sometimes optional and sometimes mandatory.

Optional: If there is no confusion between parent class instance members and child class instance members then, specifying super keyword is optional. In such case, if do not specify super keyword, then the compiler will automatically specify this keyword and first search in current child class and then search in parent class.

Mandatory: If there is confusion between parent class instance members and child class instance members then, specifying super keyword is mandatory. In such case, if we do not specify super keyword, then the compiler will also not specify super keyword, instead the compiler will specify this keyword and access current class instance members. If we want to access parent class instance members, then the programmer has to explicitly specify super keyword.

Note: The compiler can never specify super keyword. The super keyword works for only one level i.e. only its parent class members.

Program 59:

```
class Test {
    int a = 11;
    int b = 22;
}
class Demo extends Test {
    int a = 33;
    int b = 44;
    void m1() {
        System.out.println("demo class m1");
    }
}
class Sample extends Demo {
    int a = 55;
    int b = 66;
    void m2() {
        System.out.println("sample class m2");
    }
    void m3() {
        System.out.println("sample class m3");
    }
```

```
    void show() {
        int a = 77;
        int c = 88;
        Test t = new Test();
        System.out.println(t.a);
        System.out.println(b);
        System.out.println(super.a);
        System.out.println(c);        //this .c     super.c
        System.out.println(this.a);
        System.out.println(d);        //this.d
        System.out.println(a);
        System.out.println(e);
        m1();        //this.m1()  super.m1()
        super.m2();
        m2();        //this.m2()
        m3();        //this.m3()
    }
    public static void main(String[] args) {
        Sample s = new Sample();
        s.show();
    }
}
```

super(): super() can be used to access zero parameterized constructor of parent class.

super(...): super(...) can be used to access parameterized constructor of parent class.

Rules:

- Every constructor can invoke atmost one constructor of current class by using this()/this(...) or atmost one constructor of parent class by using super()/super(...)
- The invocation of the current class constructor by using this()/this(...) or the invocation of the parent class constructor by using super()/super(...) must be specified as the first statement.
- If the programmer does not invoke any current class constructor by using this()/this(...) and does not invoke any parent class constructor by using super()/super(...), then the compiler will automatically specify super() to invoke parent class zero parameterized constructor.

Program 60:

```
class Demo {
    Demo() {
        System.out.println("demo zero constructor");
    }
    Demo(int x) {
        this();
        System.out.println("demo para constructor");
    }
}
class Sample extends Demo {
    Sample() {
        this(8);
        System.out.println("sample zero constructor");
    }
    Sample(int x) {
        super(9);
        System.out.println("sample para constructor");
    }
    public static void main(String[] args) {
        Sample s = new Sample();
    }
}
```

super:

- super keyword can be used for accessing instance members of the parent class.
- The super keyword can be specified either in child class constructors or child class instance methods.
- The super keyword cannot be specified in child class static methods.

super():

- super() can be used to access zero parameterized constructor of parent class
- super() can be specified in any constructor of child class.
- super() cannot be specified in methods (either instance or static)

super(...):

- super(...) can be used to access parameterized constructor of parent class.
- super(...) can be specified in any constructor of child class.
- super(...) cannot be specified in methods(either instance or static).

Note: super()/super(...) are designed for calling constructors from other constructor.

Instance Block

If we specify a group of statements within a pair of flower braces without any keyword then, it called as instance block.

Syntax:

```
{
   Statements;
}
```

- The purpose of the instance block is to initialize instance variable.
- A class can contain any number of instance blocks and they can be declared anywhere the class.
- All the instance blocks will be executed in sequence from top to bottom, one time for ever object that is created.

If an instance variable is declared, not initialized then it will initialized automatically wit default values. If we do not want the instance variable to contain default value then, we ca initialize the instance variable with our own values in the following three locations

1. At the time of declaration.
2. By using instance block
3. By using constructor

Procedure followed by the JVM during the object creation time:

- When an object is created, the JVM will go to the beginning of the class and search fo instance variables and instance blocks in sequence from top to bottom. If the JVM encounters an instance variable, then memory for that instance variable will be allocate and if the JVM encounters instance block, then it will be executed (instance variables an instance blocks have same preference.)
- Once the memory for all the instance variables is allocated and all the instance blocks ar executed then, the JVM will execute the constructor that is specified.

Static Block

If we specify a group of statements within a pair of flower braces with static keyword then, it i called as a static block.

Syntax:

```
static {
   statements;
}
```

- The purpose of the static variable is to initialize static variable.
- A class can contain any number of static blocks and they can be specified anywhere in th class.
- All the static blocks will be executed in sequence from top to bottom, one time during th class loading time.

If a static variable is declared and not initialized then, it will be initialized automatically with default values. If we don't want the static variable to contain default value then, we can initialize a static variable with our own values in the following two locations:

1. At the time of declaration
2. By using static block

Procedure followed by the JVM during class loading time:

- When a class is loaded, the JVM will go to the beginning of the class and search for static variables and static blocks in sequence from top to bottom. If the JVM encounters a static variable, then memory for that static variable will be allocated and if the JVM encounters static, then it will be executed (static variables and static blocks have same preference).
- Once the memory for the all the static variable is allocated and all the static blocks are executed then, the JVM will search for main method and then begins the execution of main method.

Program 61:

```
class Sample {
  static {
     System.out.println("static block1");
  }
  {
     System.out.println("instance block1");
  }
  Sample() {
     System.out.println("sample zero constructor");
  }
  Sample(int x) {
     System.out.println("sample para constructor");
  }
  public static void main(String[] args) {
     System.out.println("sample main method");
     Sample s1 = new Sample();
     Sample s2 = new Sample(6);
  }
  {
     System.out.println("instance block2");
  }
  static {
     System.out.println("static block2");
  }
```

Final Keyword

Final is a keyword which can be applied to variables, methods and classes.

Final variable:

The purpose of variable is to store some value. The value of the variable can changed any number of times. If we do not want to change the value of the variable, then declare the variable as final.

Syntax for final variable:

final datatype variable = value;

- Final variable must be initialized and they cannot be assigned.
- The final keyword can be applied to instance variables, static variables, local variables, parameters and reference variables.
- By declaring the variable as final we are fixing the value of that variable. i.e. the value can't be changed.
- By declaring the variable as final we are creating a constant in a java program.

Instance variable: If an instance variable is declared and not initialized, then it will be initialized automatically with default value.

Example:

```
class Sample {
   int a;
}
```

If we don't want the instance variable to contain default value, then we can initialize the instance variable with our own value either at the time of declaration or by using constructor.

Example:

```
class Sample {
   int a = 10;
   int b;
   Sample() {
      b = 20;
   }
}
```

Final instance variable: If an instance variable is declared as final and not initialized, then it will not be initialized automatically with default value.

Example:

```
class Sample {
   final int a;
}
```

It is the responsibility of the programmer to initialize the final instance variable either at the time of declaration or by using a constructor.

Example:

```
class Sample {
   final int a = 10;
   final int b = 20;
   final int c;
   final int d;
   final int e;
   Sample () {
      b = 30;
      c = 40;
   }
   void m1() {
      d = 50;
   }
}
```

Static variable: If a static variable is declared and not initialized then it will be initialized automatically with default values.

Example:

```
class Sample {
   static int a;
}
```

If don't want a static variable to contain default values then we can initialize the static variable with our own value at the time of declaration.

Example:

```
class Sample {
   static int a = 10;
}
```

Final static variable: If a static variable is declared as final and not initialized, then it will not be initialized automatically with default value.

Example:

```
class Sample {
   final static int a;
}
```

It is the responsibility of the programmer to initialize the final static variable at the time of declaration.

Example:

```
class Sample {
   final static int a =10;
   final static int b = 20;
   final static int c;
```

```
    final static int d;
    final static int e;
    Sample () {
        b = 30;
        c = 40;
    }
    void main () {
        d = 50;
    }
}
```

Local variable: If a local variable is declared and not initialized, then it will not be initialized automatically.

It is the responsibility of the programmer to initialize the local variable either at the time of declaration or any where before using it.

Note: If the local variable is not used, then it need not be initialized.

Example:

```
class Sample {
    void m1() {
        int a = 10;
        int b;
        int c;
        b = 20;
        int d = a + b;
    }
}
```

Final local variable: If a local variable is declared as final and not initialized, then it will not be initialized automatically.

It is the responsibility of the programmer to initialize the local variable either at the time of declaration or any where before using it.

Note: If the final local variable is not used, then it need not be initialized.

Example:

```
class Sample {
    void m1() {
        final int a = 10;
        final int b;
        final int c;
        b = 20;
        final int d = a + b;
    }
}
```


Parameters: The parameters are the values that are passed to a method or a constructor.

- The parameters of a method or a constructor can be initialized during their invocation.
- The values of the parameters can be changed any number of times within the method or constructor.
- If we don't want the value of the parameter to change then declare the parameter as final.
- A method or a constructor can contain any number of parameters and we can declare any of them as final.

Program 62:

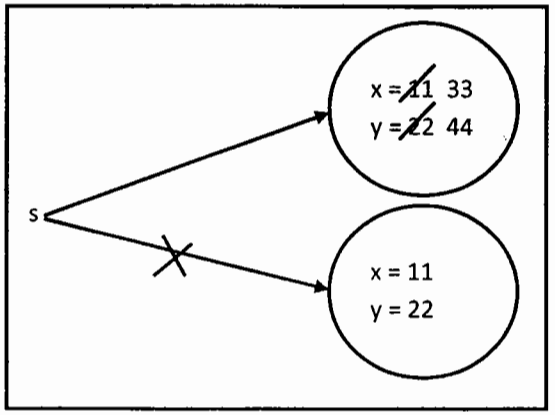
```
class Rectangle {
    void area(final int length, int breadth) {
        System.out.println("area : " +(length * breadth));
        breadth = 6;
        System.out.println("area : " +(length * breadth));
    }
    public static void main(String[] ar) {
        Rectangle r = new Rectangle();
        r.area(3,4);
    }
}
```

Reference Variable: If a variable is referring or pointing to an object, then it is called as reference variable.

- The reference of any class can refer to an object and later on refer to another object of the same class.
- If we don't want a reference variable to refer to another object, then declare the reference variable as final.
- If a reference variable is declared as final, then it can't refer to another object but content of that object can be modify.

Program 63:

```
class Sample {
    int a = 11;
    int b = 22;
    public static void main(Stirng[] ar) {
        final Sample s = new Sample();
        // s = new Sample();
        System.out.println(s.a+" : "+s.b);
        s.a = 33;
        s.b = 44;
        System.out.println(s.a+" : "+s.b);
    }
}
```



Final method: If a method is declared as final then, we cannot change the implementation of that method.

We cannot override a final method.

A method should be declared as final, when we want to protect the logic or implementation of that method.

Program 64:

```
class Parent {
    final void msg () {
        System.out.println("we are listening");
    }
}
class Child extends Parent {
    void msg () {
        System.out.println("we are sleeping");
    }
}
```

The final methods can be inherited and accessed in the child class but, they can't be overridden.

Program 65:

```
class Parent {
    final void msg() {
        System.out.println("we are listening ");
    }
}
class Child extends Parent {
    public static void main (String[] args) {
        Child c = new Child();
        c.msg ();
    }
}
```

Final class: If a class is declared as final then, that class cannot be inherited.

By declaring the class as final, we want to protect the implementation of all the methods of that class.

Example:

```
final class Parent {
}
class Child extends Parent {
}
```


- Declaring a class as final is equivalent to declaring all the methods of that class as final.
- The variables available in the final class or not final.

Program 66:

```
final class Parent {
    int x = 12;
    void msg() {
        System.out.println("we are listening");
    }
}
class Child {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.msg();
        System.out.println(p.x);
        p.x = 34;
        System.out.println(p.x);
    }
}
```

By declaring the class as final, we are restricting IS-A relationship and forcing to use HAS-A relationship.

Literal:

Any value that we store into a variable will be called as literal.

int literal: If we specify any number either positive or negative without decimal point then, it will be by default considered as int literal.

Example of int literals: 123 87429 -456 0

The int literals can be stored into all numeric data types(byte, short, int, long, float, double, char).

long literal: The long literals can be created indirectly by suffixing l or L to int literal.

Example of long literals: 123l 87429L -456l 0L

The long literal can be stored into long, float and double type.

Note: we cannot create byte literals and short literals either directly or indirectly.

- Under the integer category we have 4 types and they are byte, short, int and long. The default data type of integer category is int and the default value of integer category is 0.

double literal: The double literals can be created in two ways.

1. The double literal can be created directly by specifying any number either positive or negative with decimal point.

Example of double literals: 23.4 5674.9 -0.0119 0.0
23.4d 5674.90D -0.0119d 0.00D

2. The double literal can be created indirectly by suffixing either d or D to int literal.

Example of double literals: 23D 45679d -435d 0d

The double literal can be stored only into double type.

float literal: The float literals can be created indirectly by suffexing f or F to either int literals or to double literals.

Example of float literals: 23f 45679F -435F 0f
23.4f 5674.9f -0.0119F 0.0F

The float literals can be stored into either float type or double type.

- Under the floating point category we have two primitive data types and they are float and double. The default data type of floating point category is double and the default value is 0.0

character literal: A character literal can be created by specifying exactly one character enclosed in a pair of single quotes.

Example of char literals: 'a' 'N' '8' '$'

The character literals can be stored into all numeric data (byte, short, int, long, float, double, char).

The character literals can be specified in UNICODE format. The UNICODE format must begin with \ and followed by u (lower case) and followed by exactly 4 digit hexadecimal number enclosed in a pair of single quotes.

UNICODE Format: '\uxxxx' (0-9a-f or 0-9A-F)

The character literal can be created by specifying any one of the escape sequence in a pair of single quotes.

Escape sequences of java: \n \t \r \f \b \\ \' \"

boolean literal: There are only two boolean literals and they are true and false. The boolean literals can be stored only into boolean type.

Predefined Literals: The java language provides three predefined literals and they are true, false, and null.

The true and false literals can be stored only into boolean type and null can be stored into any reference.

The three predefined literals must be specified in lower case only.

TypeCasting:

It is a process of converting a value from one type to another type.

If the values are of same type, then we do not require any typecasting. Typecasting has to be performed, if the values are of different type and if they are compatible.

Typecasting can be performed for the following two types:

1) Typecasting with respect to Primitive Types.
2) Typecasting with respect to Reference Types.

Typecasting with respect to Primitive Types: Typecasting with respect to primitive types is a process of converting a value from one primitive type to another primitive type which is compatible.

All the numeric data types are compatible to each other. The numeric data types are byte, short, int, long, float, double and char.

Widening: It is a process of converting a value from smaller primitive type to bigger primitive type.

Syntax: firsttype var = (firsttype) secondtypevalue;
typecasting

Specifying a data type with in a pair of parenthesis is called Typecasting.

In the process of widening, specifying the typecasting is optional. If we do not specify the typecasting then, the compiler will perform the typecasting automatically and therefore called **Implicit Typecasting.**

Example:

```
int x = 14;
double y = (double) x;

char c = 'd';
int i = (int) c;
```

Narrowing: It is a process of converting a value from bigger primitive type to smaller primitive type.

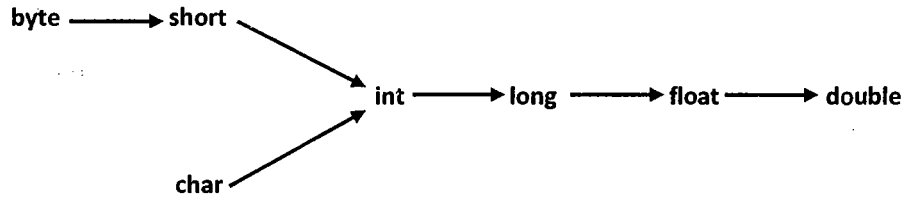
Syntax: firsttype var = (firsttype) secondtypevalue;
typecasting

In the process of narrowing, specifying the typecasting is mandatory. If we do not specify the typecasting, then the compiler cannot perform typecasting automatically because there is a chance of losing data therefore, it has to be done explicitly by the programmer hence called **Explicit Typecasting.**

Example:

```
double x = 12.34;
int y = (int) x;

int i = 97;
char c = (char) i;
```

Automatic Type Promotion or Implicit Type Conversion



Type Casting With Respect To Reference Type: It is a process of converting an object from one reference type to another reference type, which is compatible.

The reference types are said to be compatible, if their corresponding classes have IS-A relationship.

Program 67:

```
class Parent {
    void show() {
        System.out.println("parent class show method");
    }
    void display() {
        System.out.println("parent class display method");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("child class show method");
    }
    void print() {
        System.out.println("child class print method");
    }
}
```


Case 1:

```
class Typecasting {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.show();
        p.display();
        //p.print();
    }
}
```

In the above code 'p' is the reference of parent class, referring to an object of parent class. Using the 'p' reference variable we can access the members of only parent class.

Case 2:

```
class Typecasting {
    public static void main(String args[]) {
        Child c = new Child();
        c.show();
        c.display();
        c.print();
    }
}
```

In the above code 'c' is reference variable of child class referring to an object of child class. Using this 'c' reference variable we can access the members of the parent class and child class.

Upcasting: If a child class object is referred by parent class reference then, it is called as upcasting.

Syntax: ParentClass ref = (ParentClass) ChildClassObject;

typecasting

In the process of upcasting, specifying the type casting is optional. If we do not specify the typecasting then, the compiler will perform the typecasting automatically.

A reference variable of any class can refer to its own object or it can refer to any of its child class object.

Example:

```
Object o1 = new Object();
Object o2 = new Customer();
Object o3 = new Employee();
Object o4 = new String();
```

Note: When a java program is compiled, the compiler will check the code based on reference type, and when the java program is executed, the JVM will execute the program based on object type.

Case 3:

```
class Typecasting {
    public static void main(String[] args) {
        Parent p = (Parent) new Child();
        p.show();
        p.display();
        //p.print();
    }
}
```

In the above code 'p' is the reference variable of parent class, referring to an object of child class. Using the 'p' reference variable we can access all the members of parent class and only those methods of child class, which are overridden.

Downcasting: If the parent class object is referred by child class reference then, it is called as downcasting.

Syntax: ChildClass ref = (ChildClass) ParentClassObject;

typecasting

In the process of downcasting, specifying the downcasting is mandatory.

Case 4:

```
class Typecasting {
    public static void main(String[] args) {
        Child c = (Child) new Parent();
        c.show();
        c.display();
        c.print();
    }
}
```

In the above code 'c' is the reference variable of child class, referring to an object of parent class. This code is valid during compilation time, and invalid during execution time, down casting is not allowed.

Case 5:

```
class Typecasting {
    public static void main(String[] args) {
        Parent p = (parent) new Child();
        Child c = (Child) p;
        c.show();
        c.display();
    }
}
```


Program 68:

```
class Student {
}

class Employee extends Student {
    void msg() {
        System.out.println("hello friends");
    }
}

class Typecasting {
    public static void main(String[] args) {
        Student s = new Employee();
        Employee e = (Employee) s;
        e.msg ();
    }
}
```

Program 69:

```
class Typecasting {
    public static void main(Stirng[] args) {
        Object obj = new String();
        String str = (String) obj;
        System.out.println(str.length());
    }
}
```

Abstract Class

Concrete method: A method is said to be concrete, if it contains both declaration and definition.

Concrete class: A class is said to be concrete, if all the methods of that class are concrete.

A concrete class can be instantiated i.e. we can create an object of concrete class and using that object we can access the members of that class.

Abstract method: If a method contains only declaration without any definition then, it is said to be an abstract method.

Syntax of abstract method:
abstract returntype methodName(list of parameters);

- Abstract methods must end with semi colon (;) and they must be declared with abstract keyword.

Abstract class: If a class contains some abstract methods then, the class should be called as abstract class.

Syntax of abstract class:
abstract class ClassName {

}

- An abstract class can be combination of abstract methods and non-abstract (concrete) methods.
- An abstract class can contain zero or more abstract methods.
- If a class does not contain any abstract methods then, declaring the class as abstract is optional.
- If a class contains at least one abstract method then, declaring the class as abstract is mandatory.

An abstract class cannot be instantiated i.e. we cannot create an object of the abstract class and therefore we cannot access the members of that class. In order to access the members of the abstract class, we need to inherit the abstract class into another class and override all the abstract methods available in the abstract class.

Syntax:
```
abstract class AbstractClass {
}
class SubClass extends AbstractClass {
}
```

If the subclass does not override at least one abstract method then, declare the subclass also as abstract.

Program 70:
```
abstract class Operation {
   void msg() {
      System.out.println("hello friends");
   }
   abstract void twice(int a);
}
class Programmer1 extends Operation {
   void twice(int x) {
      System.out.println("result1 : "+ (x+x));
   }
}
class Programmer2 extends Operation {
   void twice(int y) {
      System.out.println("result2 : " + (y*2));
   }
}
class Programmer3 extends Operation {
   void twice(int z) {
      System.out.println("result3 : "+ (z<<1));
   }
}
class AbstractDemo {
   public static void main(Stirng[] args) {
      Programmer1 p1 = new Programmer1();
      p1.msg();
      p1.twice(5);
      Programmer p2 = new Programmer2();xmark
      p2.msg();
      p2.twice(6);
      Programmer p3 = new Programmer3();
      p3.msg();
      p3.twice(8);
   }
}
```

Q) Why we are not allowed to instantiate an abstract class ?
A) Assume we are allowed to create an object of abstract class, using that object if we invoke a concrete method then, it will be executed because it contains definition, but using that object if we invoke an abstract method then, it will lead to an unsafe operations because it does not contain any definition. To avoid the unsafe operations, we are not allowed to instantiate an abstract class.

Q) Why should be declared a method as abstract?

A) A method should be declared as abstract, when we want a method to be implemented by different programmers with different logics.

Q) Why should we declare a class as abstract even though it doesn't contain any abstract methods?

A) We declare the class as abstract even though it does not contain any abstract methods, when we don't want an object our class to be created. By declaring the class as abstract we are restricting HAS-A relationship and forcing to use IS-A relationship.

- An abstract class can contain main method and it can be executed.

Program 71:

```
abstract class Sample {
    public static void main(String[] args) {
        System.out.println("abstract class main method");
    }
}
```

- Every class in java, either predefined or user defined, either abstract or concrete will be sub class of Object class either directly or indirectly
- Every class in java, either abstract class or concrete class will contain a constructor whether we specify or not.
- The constructor of the abstract class cannot be executed directly, it can be executed indirectly by creating an object of its child class.
- An abstract class cannot be instantiated, but we can declare a reference of the abstract class. The reference of the abstract class can be used to refer to an object of any of its child class which is concrete.

Program 72:

```
abstract class Shape {
    int dim1, dim2;
    Shape(int dim1, int dim2) {
        this.dim1 = dim1;
        this.dim2 = dim2;
    }
    abstract double area();
}
class Rectangle extends Shape {
    Rectangle(int length, int breadth) {
        super(length, breadth);
    }
    double area() {
        return dim1 * dim2;
    }
}
```



```
class Triangle extends Shape {
    Triangle (int base, int height) {
        super(base, height);
    }
    double area() {
        return 0.5 * dim1 * dim2;
    }
}

class Calculation {
    public static void main(String[] args) {
        Shape s;
        s = new Rectangle(3,4);
        double res = s.area();
        System.out.println("Rectangle area : "+res);
        s = new Triangle(5,6);
        res = s.area();
        System.out.println("Triangle area : "+res);
    }
}
```

Modifiers: Modifiers are the keywords, which modify the meaning of a declaration.

- The static modifier can be applied to variables, methods and inner classes.
- The final modifier can be applied to variables, methods and classes.
- The abstract modifier can be applied to methods and classes.

Illegal combination of modifiers for a method:

1) A method can't be declared as abstract and final because the abstract keyword says we must override the method where as the final keyword says we must not override the method.
2) A method can't be declared as abstract and static because if a static method is invoked directly by using class name then, it will lead to unsafe operation because it doesn't have definition.

Illegal combination of modifiers for a class:

1) A class can't be declared as abstract and final because the abstract keyword says we must inherit the class whereas the final keyword says we must not inherit the class.

- We can restrict HAS-A relationship by declaring the class as abstract and we can restrict IS-A relationship by declaring the class as final but we can't restrict both relationships at the same time.
- An abstract class can have final method but a final class cannot have abstract method.

Interface

An interface is a collection of only abstract methods. An interface is used as a contract or an agreement for developing a project or software etc.

Syntax:

```
interface InterfaceName {
    variables ;
    methods;
}
```

The name of an interface can be any valid java identifier.

Example:

```
interface Test {
    int x =12;
    void m1();
}
```

When a java program is compiled, the compiler generates a .class file for every class and for every interface. When the above java program, Test.java is compiled, the compiler will generate Test.class

javac Test.java

javap Test

```
interface Test {
    public static final int x;
    public abstract void m1();
}
```

- The variables of an interface are declared as public, static and final whether we specify or not.
- The methods of an interface are declared as public and abstract whether we specify or not.

public: The variables and methods of an interface are declared as public, so that they can be accessed by any class from any location.

static: The variables of an interface are declared as static, so that they can be accessed directly by using interface name.

final: The variables of an interface are declared as final, so that the value of that variable does not change.

abstract: The methods of an interface are declared as abstract because they do not have any definition.

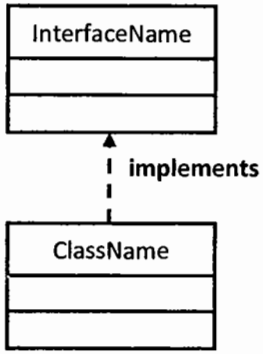
An interface can be considered as 100% abstract and therefore interface cannot be instantiated. In order to access the members of an interface, we need to inherit the interface into another class by using implements keyword.

Example:

```
interface InterfaceName {
}

class SubClass implements InterfaceName {
}
```

Inheriting can interface should also be represented by IS-A relationship.

Class Diagram showing implementing an Interface:



- If a class is implementing an interface then that class must provide implementation to all the methods available in that interface.
- It the subclass does not provide implementation to atleast one abstract method then, declare the subclass as abstract.
- If the subclass is providing implementation to all the abstract methods then, the subclass should be called as Implementation class.
- An interface can have any number of implementation classes.

Program 73:

```
interface Animal {
   void makeSound();
}
class Cat implements Animal {
   public void makeSound() {
      System.out.println("meow meow");
   }
}
class Dog implements Animal {
   public void makeSound () {
      System.out.println("bow bow");
   }
}
```

```
class InterfaceDemo {
    public static void main(String[] args) {
        Cat c = new Cat();
        c.makeSound ();
        Dog d = new Dog();
        d.makeSound();
    }
}
```

The Cat and Dog are subclasses of Animal interface.
Cat and Dog are implementation classes of Animal interface.

Cat IS-A Animal
Dog IS-A Animal

- The variables of an interface must be initialized. The variables of an interface can be accessed by using interface name or can be accessed directly by implementing the interface.
- An interface cannot be instantiated but we can declare a reference of that interface. The reference of an interface can refer to an object of its implementation class.
- The java language does not support multiple inheritance but it can be achieved with help of interface. A class can implement any number of interfaces.

Syntax:

```
class ClassName implements interface1, interface2, ... {

}
```

Program 74:

```
interface Sample {
    int x = 45;
    void msg();
}
interface Test {
    int y = 99;
    void msg();
}
class InterfaceDemo implements Sample, Test {
    public void msg() {
        System.out.println("multiple Inheritance");
    }
    public static void main(String[] args) {
        System.out.println(x);
        System.out.println(y);
        InterfaceDemo id = new InterfaceDemo();
        id.msg();
        Sample s = new InterfaceDemo();
        s.msg();
        Test t = new InterfaceDemo();
        t.msg();
    }
}
```

Rules for inheriting a class and interface

- A class can extend atmost one class.
- A class can implement any number of interfaces.
- An interface can extends any number of interfaces.
- A class can extends a class and implement any number of interfaces together at the same time (extends keyword should be followed by implements keyword).

Marked Interface(Tagged Interface): An interface is said to be marked or tagged if it is empty i.e. if the interface does not have any members.

- The purpose of a marked interface is to provide some instructions to the JVM to perform a task in a special way.

Examples of predefined marked interfaces: Cloneable Serializable RandomAccess etc.

Note: We can create user defined marked interface but they are of no use because the JVM has no meaning for user defined marked interfaces.

Differences between abstract class and interface

Abstract class	Interface
1) An abstract class is a combination of abstract methods and concrete methods.	1) An interface is a collection of only abstract methods.
2) An abstract class can contain abstract methods, concrete methods, instance methods and static methods.	2) An interface will contain only abstract methods which are instance methods.
3) Declaring an abstract method with abstract keyword in an abstract class is mandatory.	3) Declaring an abstract method with abstract keyword in an interface is optional.
4) The members of an abstract class can be either public or non public.	4) The members of an interface will be only public.
5) The value of the variables in an abstract class can be modified or fixed.	5) The value of the variables in an interface cannot be modified because they are by default fixed.
6) An abstract class can contain both instance variables and static variables.	6) An interface can contain only static variables.
7) An abstract class will contain a constructor whether we specify or not.	7) An interface will not contain a constructor.
8) An abstract class can be inherited into a class by using extends keyword.	8) An interface can be inherited into a class by using implements keyword.
9) An abstract class can extend at most one class.	9) An interface can extend any number of interfaces.
10) Using abstract class we cannot achieve multiple inheritance.	10) Using interface we can achieve multiple inheritance.
11) An abstract class can implement any number of interfaces.	11) An interface cannot implement any interface.
12) An abstract class can inherit from both class and interface.	12) An interface can inherit from only interface.
13) Object is the super most class of all the classes.	13) There is no super interface in java.
14) An abstract class can contain final methods.	14) An interface cannot have final methods.
15) An abstract class will contain abstract methods so that different programmers provide different implementation.	15) An interface will contain abstract methods given by the client, so that the programmer provides implementation class.

Package

A package is a collection of classes and interfaces which are related. The purpose of a package is to improve the performance and increase accessibility.

Packages are classified into two types:

1. Predefined Packages
2. Userdefined Packages

Predefined package: The packages which are available as part of the java software given by either SUN Microsystems or some other organizations are called predefined packages.

The predefined packages are further classified into three types:

1. **Core Packages:** The packages which are given by SUN Microsystems and which begin with "java" are called core packages.
2. **Extended Packages:** The packages which are given by SUN Microsystems and which begin with "javax" are called extended packages.
3. **Third party packages (vendor packages):** The packages which are given by some other organizations as part of the java software are called third party packages or vendor packages. The third party package may begin with any term.

Examples of Predefined package:

- **java.lang:** This package contains a set of classes and interfaces required for basic programming.
  Note: java.lang package is called as default package.
- **java.io:** This package contains a set of classes and interfaces required for performing reading and writing operations.
- **java.util:** This package contains a set of classes and interfaces required for additional usage like Date, Time, Collections, etc.
- **java.net:** This package contains a set of classes and interfaces required for developing networking applications.
- **java.text:** This package contains a set of classes and interfaces required for formatting text, numbers, date, currency etc. Using this package we can achieve Internationalization(I18N).
- **java.awt:** This package contains a set of classes and interfaces required for developing a GUI.
- **javax.swing:** This package contains a set of classes and interfaces required for developing a GUI better than awt.
- **java.applet:** This package contains a set of classes and interfaces required for developing a distributed GUI.

User defined Packages: The packages which are created by the user or the programmer are called as user defined packages.
To create a user defined package we use package keyword.

Syntax:
package packagename;
package package1[.package2[.package3]];

A package statement can contain any number of levels, but specifying atleast one level is mandatory.

Example:
package inetsolv;
package inetsolv.core;
package inetsolv.core.example;

A java program can contain atmost one package statement and it should be specified as the first executable statement in a java program.

Program 75:
```
package inetsolv;
class Sample {
    public static void main(String[] args) {
        System.out.println("hello friends");
    }
}
```

Syntax for compiling java program with a package statement:

javac -d path programname/filename with extension

-d is an option which indicates the compiler to create a directory by the name that is specified in the package statement of the program, in the specified destination.

Example:
javac -d . Sample.java
javac -d d: Sample.java
javac -d e: Sample.java
javac -d e:\check Sample.java

Syntax to execute a java program available in a package:
java packagename.ClassName
java package1[.package2[.package3]].ClassName

Example:
java inetsolv.Sample

Every java program contains two default packages:

1. current working directory /package(user defined package)
2. java.lang.package(predefined package)

All the classes and interfaces available in the default packages can be accessed directly.

To access the classes and interfaces which are not available in default packages, we need to use the following two mechanisms:

1. Fully Qualified Name
2. Import Statements

Fully Qualified Name: Fully qualified name is a process of specifying a class name or interface name along with its complete package details (absolute path).

Syntax:
packagename.ClassName
package1[.package2[.package3]].ClassName

Example:
java.util.ArrayList
java.io.BufferedReader
java.util.zip.DeflaterInputStream

Using the fully qualified name we can access either one class or one Interface. If a class or interface is used multiple times then, we need to specify the fully qualified name also multiple times, thereby increasing the code size, and reducing the readability of the code.

In order to improve the readability and reduce the code size, we take the help of import statements.

Import statements: Import statements can be used to a access a class or an interface from a package.

The import statement can be specified one time and accessed multiple times. A java language can contain any number of import statements. All the import statements has to be specified after the package statement and before the class.

Using the import statement we can access all the classes and interfaces from the package or we can access either one class or one interface from a package.

Syntax for import statement accessing all the classes and interfaces from a package:
import packagename.*;
import package1[.package2[.package3]].*;

Example:

```
import java.io.*;
import java.util.*;
import java.util.zip.*;
```

Using the above import statement we can automatically access all the classes and interfaces available in the specified package and therefore called as **implicit import statement**.

Syntax for import statement accessing a single class or a single interface from a package:

```
import packagename.ClassName;
import package1[.package2[.package3]].ClassName;
```

Example:

```
import java.io.BufferedReader;
import java.util.ArrayList;
import java.util.zip.DeflaterInputStream;
```

Using the above import statements we can access only that class or interface which is explicitly specified from a package and therefore called as **explicit import statements**.

Note: It is recommended to always use explicit import statements because they improve the readability of the code.

Difference between include statement and import statement:

In an application, if we specify an include statement then, it will load the entire predefined program into our application whether require or not, increasing the code size and compilation time. This kind of loading is called **static loading**.

In an application, if we specify an import statement then, it will load only those classes and interfaces that are required and therefore there will not be any increase in the code size and compilation time. The loading of classes and interface will be done into JVM memory(method area) during the execution time. This kind of loading is called **dynamic loading** or **load on demand**.

Static Import

static import statements are introduced in java 1.5 and it is used to access the static members from a class or interface.

Syntax:

```
import static packagename.ClassName.*;
import static package1[.package2].ClassName.*;

import static packagename.ClassName.staticmember;
import static package1[.package2].ClassName.staticmember;
```

Program 76:

```
import static java.lang.math.*;
import static java.lang.Short.MAX_VALUE;
class Sample {
   public static void main(String[] args) {
      System.out.println(min(5,8));
      System.out.println(max(5,8));
      System.out.println(floor(5.8));
      System.out.println(ceil(5.8));
      System.out.println(round(5.8));
      System.out.println(random());
      System.out.println(pow(2,4));
      System.out.println(abs(-8));
      System.out.println(Short.MAX_VALUE);
   }
}
```

Difference between general import and static import

General import	Static import
General import is used to access the classes and interfaces from a package.	Static import is used to access the static members from a class or interface.
import packagename.*;	import static package.ClassName.*;
import package.ClassName.*;	import static package.ClassName.member;

Access Specifiers

The access specifiers can be used to define the level of accessibility of either the members in a class/interface or the level of accessibility of a class/interface in a package. Using the access specifies we can define the scope (life) of a class/interface or its members and provide security.

The java language provides 4 levels of access specifiers, with three keywords.

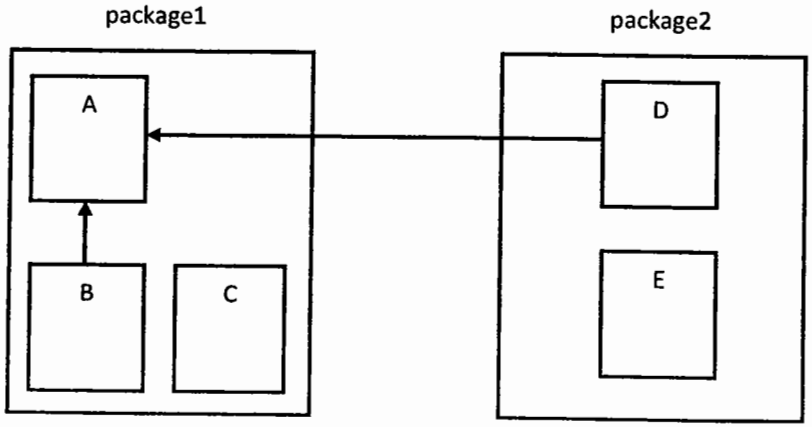
1. public
2. private
3. protected
4. default (no access specifier)

public: If a member is declared as public then, it can be accesses from all the classes of all the packages. The public access specifier can be applied to classes, interfaces, variables, methods and constructors.

private: If a member is declared as private then, it can be accessed only in that class. The private access specifier can be applied to variables, methods and constructors.

protected: If a member is declared as protected then, it can be accessed from all the classes of that package and from the sub classes available in other packages. The protected access specifier can be applied to variables, methods and constructors.

default(no access specifier): If a member is not declared with either public or private or protected then, it is called as default access specifier. If a member is declared as default then, it can be accessed from all the classes of that package only. The default access specifier is also called as package level access. The default access specifier can be applied to classes, interfaces, variables, methods and constructors.



package1

```
class A {
}
class B extends A {
}
class C {
}
```

package2

```
class D extends A {
}
class E {
}
```

- If a variable x is declared as public in class A, then it can be accessed in A, B, C, D and E.
- If a variable x is declared as private in class A, then it can be accessed in only A.
- If a variable x is declared as protected in class A, then it can be accessed in A, B, C and D.
- If a variable x is declared as default in class A, then it can be accessed in A, B, and C.

	private	default	protected	public
within the same class	✓	✓	✓	✓
within the sub class of same package	X	✓	✓	✓
within the non sub class of same package	X	✓	✓	✓
within the sub class of other package	X	X	✓	✓
within the non sub class of other package	X	X	X	✓

private < default < protected < public

The most accessible access specifier is public and the most restricted access specifier is private.

Overriding rule: When we are overriding a method, the child class method should have an access specifier, same as that of the parent class method access specifier or a bigger access specifier.

Note: The private methods cannot be inherited and therefore cannot be overridden.

Illegal combination of modifiers for a method: A method cannot be declared as abstract and private because, the abstract keyword says we must inherit and then override whereas the private keyword says we cannot inherit and therefore we cannot override.

A class or interface can be declared as either default or public. The naming of the java program is based on class declaration(access specifiers).

Note: The naming of the java program is not based on main method.

Rule: If a class is declared as default then, the name of the program can be any name but if a class is declared as public then, the name of the program must be same as that of the public class name.

A java program can have any number of classes, but it can contain atmost one public class.

Example:

```
class A {
    public static void main(String[] ar) {
        System.out.println("class A main method");
    }
}
class B {
    public static void main(String[] ar) {
        System.out.println("class B main method");
    }
}
class C {
    public static void main(String[] ar) {
        System.out.println("class C main method");
    }
}
class D {

}
```

In the above program all the classes are declared as default and therefore the name of the program can be any name.

All the below names are valid

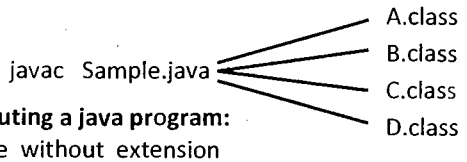
A.java	ABCD.java
B.java	Dcba.java
C.java	Anyname.java
D.java	Sample.java

Let us name it as Sample java.

Syntax for compiling a java program:
javac programname/filename with extension

When a java program is compiled, the compiler generates .class file. The number of .class files that are generated will be equal to number of classes and interfaces available in the program. The name of the .class files will be same as the class names or interface names.

Example:



Syntax for executing a java program:
java ClassName without extension

Example:
java A
class A main method

java B
class A main method

java C
class A main method

java D
RTE: NoSuchMethodError

java Sample
RTE: NoClassDefFoundError

Every class in java will have a constructor whether we specify or not. If we do not specify any constructor then, the compiler will provide either public zero parameterized constructor or default zero parameterized constructor i.e. the declaration of the constructor provided by the compiler will be based on class declaration. If the class is declared as public then, the compiler will provide public zero parameterized constructor and if the class is declared as default then, the compiler will provide default zero parameterized constructor.

Note: The compiler cannot declare the constructor as private and protected and it cannot provide parameterized constructor.

As a programmer we can specify either a zero parameterized constructor or a parameterized constructor or both and we can declare them as either public or private or protected or default.

Singleton class: A class is said to be singleton, if we are able to create only one object for that class.

To create a singleton class, the class must have private constructor and a factory method.

Factory method: If a method returns an object of its own class then, it is called as a factory method.

Note: The factory method must be declared as static.

Program 77:

```
public class Sample {
    static Sample s;
    private Sample() {
    }
    void msg() {
        System.out.println("hello friends");
    }
    static Sample getObject() {
        if(s == null)
            s = new Sample();
        return s;
    }
}
class Demo {
    public static void main(String[] args) {
        Sample s = Sample.getObject();
        s.msg();
    }
}
```

Keywords: The words that have predefined meaning in java or the words whose meaning is reserved in java are called keywords.
Example: class, import, static, public, private, final, default, abstract

Modifiers: Modifiers are the keywords, which modifies the meaning of a declaration.
Example: static, final, abstract, public, private, protected

Specifiers: Specifiers are the modifiers, which specify the level of accessibility.
Example: public, private, protected.

Program 78:

```
package abcd;
public class Sample {
    public void msg() {
        System.out.println("hello friends");
    }
    public static void main(String[] args) {
        Sample s = new Sample();
    }
}
class Demo {
    public static void main(String[] args) {
        Sample s = new Sample();
        s.msg();
    }
}
package xyz;
import abcd.Sample;
class Test {
    public static void main(String[] args) {
        Sample s = new Sample();
        s.msg();
    }
}
```

Exception Handling

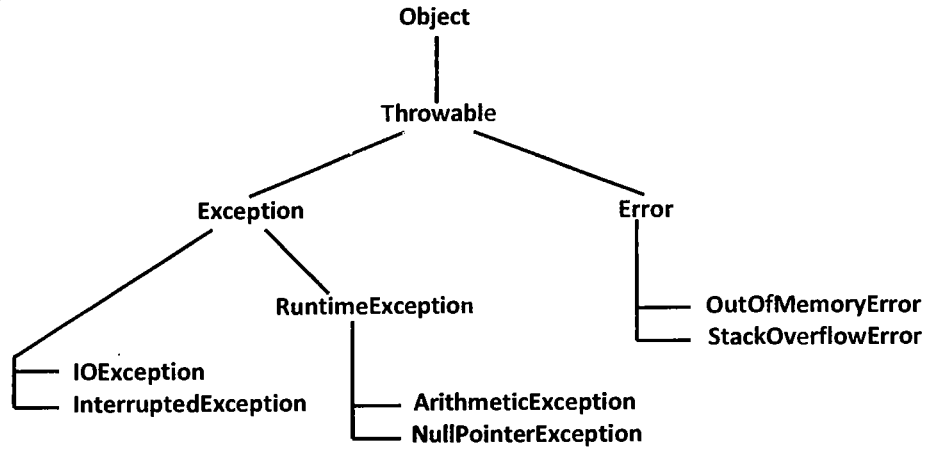
When an application is developed, the application may contain some errors. The errors that occur in the applications are classified into two types:

1. Compiler time errors
2. Runtime errors

Complier time error: The errors that occur in a program or application because of the syntactical mistakes are called as compile time errors.

Runtime errors: The errors that occur in a program at execution time, because of either programmer failure or JVM failure are called as run time errors.

Exception hierarchy:



Object: Object is the super most class of all the classes in Java.

Throwable: Throwable is the super most class of all the runtime errors in Java. Throwable is the subclass of Object class.

Exception and Error are the subclasses of Throwable.

Exception: An exception is a runtime error, which occurs because of the programmer failure(logical failure, invalid inputs etc). Exception is the super most class of all the Exceptions.

Exceptions are classified into two types based on when they are identified:

1. **Compile time exception:** The exceptions which occur at runtime and which can be identified before runtime or during compilation time are called as compiler time exceptions.
2. **Runtime exception:** The exceptions which occur at runtime and which can be identified during runtime are called as runtime exceptions.

Error: An Error is a runtime error which occurs because of the JVM failure. Error is the super most class of all the Error classes.

The exceptions are classified into two types based on whether they are handled or not:

1. **Checked exception:** The exceptions whose handling is mandatory are called as checked exception.
2. **Unchecked exception:** The exceptions whose handling is optional are called as unchecked exception.

According to the exception hierarchy, RuntimeException and its subclasses, Error and its subclasses are unchecked and the remaining are checked.

If an application contains an exception then, the application will be terminated abnormally leading to incomplete execution. In order to execute the code completely and terminate the application normally then, we need to take the help of exception handling.

Exception handling is a process of finding an alternate solution so that the remaining code execute completely and terminates normally. The code that performs exception handling will be called as **exception handler**.

Note: The exception handling process will not remove the exception from the program.

Program 79:

```
public class Sample {
   public static void main(String[] args) {
      System.out.println("line one");
      System.out.println(10/0);
      System.out.println("line three");
      System.out.println("line four");
   }
}
```

When an exception occurs in an application, an object of its corresponding exception class will be created and then, all the details related to that exception will be stored into that object and then, the object will be thrown to the JVM by that method in which an exception has occurred.

The JVM will catch the object and reads the information available in that object and then the JVM looks for exception handling code, if not available the JVM will call default exception handler.

The responsibility of the default exception handler is to display the information available in the exception object and terminate the application abnormally and therefore leading to incomplete execution.

The keywords related to exception handling concept are **try, catch, finally,** throws and **throw**.

try: A try is a block, in which we can specify a group of statements that may generate an exception.

Syntax:

```
try {
    statements generating exception;
}
```

A try block can contain any number of statements but recommended to specify only those statements which may generate the exception.

catch: A catch is a block, in which we can specify a group of statements that will display the information of the exception that has occurred.

Syntax:

```
catch(AnyException ref) {
    statements displaying exception information;
}
```

Note: Every catch block must contain a reference of any one of the exception.

finally: A finally is a block, in which we can specify a group of statements, that will perform code cleanup activities like releasing memory, resources etc.

Syntax:

```
finally {
    statements performing code cleanup activities,
}
```

Rules:

1) A try block must be followed by either a catch block or a finally block.
2) A catch block must be preceded by either try or catch block but in the hierarchy we must specify try block on the top.
3) A catch block can be followed by either a catch block or finally block.
4) A finally block must be preceded by either a try or a catch but in the hierarchy we must specify try block on the top.
5) A try block can be followed by any number of catch blocks(zero or more).
6) A try block contains multiple catch blocks and if the exceptions specified in the catch blocks have IS-A relationship then, they have to specified in the order from child to parent, but if the exceptions don't have IS-A relationship then, we can specify them in any order.
7) A try block can contain atmost one finally block.
8) We can specify the statements either before the blocks or after blocks or inside the blocks but not in between the blocks.
9) A try block may contain multiple catch blocks but, it can execute atmost one catch block, which in matching.

10) A try block may contain multiple catch blocks but, none of the catch blocks may execute in the following situation.
    i. When there is no exception in the program.
    ii. An exception has occurred but not matching.
11) The try, catch and finally blocks can be nested either in try or catch or finally and they can be nested any number of times.
12) A program can contain any number of combinations of try, catch, and finally blocks.
13) If statements in a try block generate an exception then, the control will be transferred from inside the try block to the corresponding catch block.
14) Once the control is out of the try block it cannot return back to the try block.
15) We cannot guarantee the execution of the try and catch block but the execution of finally block is guaranted.

Program 80:

```
public class Sample {
  public static void main (String[] args) {
    System.out.println("line one");
    try {
      String s1 = args[0];
      String s2 = args[1];
      int a = Integer.parseInt(s1);
      int b = Integer.parseInt(s2);
      System.out.println(a/b);
    }
    catch(ArithmeticException ae) {
      ae.printStackTrace();
    }
    catch(ArrayIndexOutOfBoundsExecption aioobe) {
      aioobe.printStackTrace();
    }
    finally {
      System.out.println("special line");
    }
    System.out.println("line four");
  }
}
```

Multi catch block: In java 1.7 version we can handle multiple exceptions by writing a single catch block.

Syntax:
```
catch(Exception1 | Exception2 | Exception3... ref) {
    statements;
}
```
Example:
```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
    e.printStackTrace();
}
```
Rule: The exceptions specified in multi catch block must not have IS-A relationship.

throws: The throws keyword is designed to transfer or delegate the responsibility of exception handling to its caller.

Syntax:
```
returntype methodName(parameters) throws Exception1, Exception2, ... {
    statements;
}
```
Note: The throws keyword will not handle exceptions.

Program 81:
```
public class Sample {
    public static void main(String[] args) {
        System.out.println("main first line");
        try {
            show();
        }
        catch(ArithmeticException ae) {
            ae.printStackTrace();
        }
        System.out.println("main last line");
    }
    static void show() throws ArithmeticException {
        System.out.println("show first line");
        System.out.println(6/0);
        System.out.println("show last line");
    }
}
```
In the above program, an exception has occurred in show() and it is the responsibility of the show() to handle the exception, but instead of show() handling the exception, it has transferred the responsibility of exception handling to its caller(main() method).

throw: The throw keyword is used to throw an exception object explicitly to the JVM.

Syntax:
```
throw AnyExceptionObject;
```
Note: Exception handling can be done only by using try-catch block.

Different ways of displaying the information of the exception:

1) **printStackTrace():** This method can be used to display the detailed information of the exception that has occurred. This method will provide details like exception name, reason for the occurrence of the exception, line number, method name, class name, program name.

```
try {
    System.out.println(10/0);
}
catch(ArithmeticException ae) {
    ae.printStackTrace();
}
```
```
java.lang.ArithmeticException : / by zero at Sample.main(Sample.java:7)
```

2) **toString():** The To String method is used to display only the exception name and the reason for the occurrence of the exception.

```
try {
    System.out.println(10/0);
}
catch(ArithmeticException ae) {
    System.out.println(ae);
    //System.out.println(ae.toString());
}
```
```
java.lang.ArithmeticException : / by zero
```

3) **getMessage():** The getMessage() can be used to display only the reason for the occurrence of the exception.

```
try {
    System.out.println(10/0);
}
catch(ArithmeticException ae) {
    System.out.println(ae.getMessage());
}
```
```
/ by zero
```

4) **User defined Message:** If we don't want the output to be provided by the predefined methods then, we can display our own message by using System.out.println().

```
try {
    System.out.println(10/0);
}
catch(ArithmeticException ae) {
    System.out.println("ArithmeticException has occurred");
    System.out.println("it is a user defined message");
}
```

```
Arithmetic has occurred
it is a user defined message
```

User defined exception:

If an exception is created by a user or a programmer then, it is called as user defined exception. The user defined exceptions has to be created when none of the predefined exceptions are matching our application requirement.

Procedure for creating user defined exception:

1. Every predefined exception is a class and therefore the user defined exception should also be a class.
2. Every predefined exception is a subclass of Exception class either directly or indirectly therefore every user defined exception should also be a subclass of Exception class either directly or indirectly.
3. If the application doesn't know when to generate the exception and which one to generate the exception and therefore it is the responsibility of the programmer to explicitly create an object and throw it to the JVM by using throw keyword.
4. Every user defined exception must have two constructors, one zero parameterized constructor and the other parameterized constructor taking one parameter of String type.

Program 82:

```
class SmallAgeException extends RuntimeException {
  SmallAgeException() {
  }
  SmallAgeException(String str) {
     super(str);
  }
}

class BigAgeException extends RuntimeException {
  BigAgeException() {
  }
  BigAgeException(String str) {
     super(str);
  }
}

public class Sample {
  public static void main(String[] args) {
     int age = Integer.parseInt(args[0]);
     try {
        if(age < 20) {
           throw new SmallAgeException("your age is less than 20");
        }
        else if(age > 30) {
           throw new BigAgeException("your age is more than 30");
        }
        else {
           System.out.println("you are eligible");
        }
     }
     catch(SmallAgeException sae) {
        sae.printstackTrace();
     }
     catch(BigAgeException bae) {
        bae.printStackTrace();
     }
     System.out.println("have a good day");
  }
```

java.lang package

The java.lang package is a predefined package available in java language and it is a default package. All the classes available in that package can be accessed directly.

Object Class

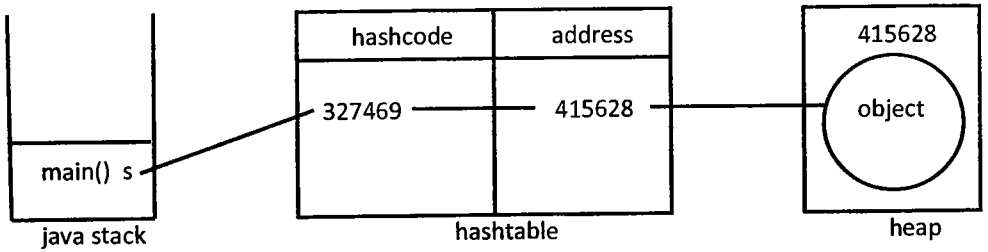
Object is the super most class of all the classes in java i.e. every class in java will be subclass of Object class either directly or indirectly. Every class in java either predefined or user defined can use the members of Object class.

Methods of Object class:

1) String toString(): The toString() method can be used to display an object in the form of a string representation.

Program 83:

```
class Sample {
    public static void main(String[] args) {
      Sample s = new Sample();
      System.out.println(s);
      // Or
      //System.out.println(s.toString());
    }
}
```



The toString() will be invoked by the compiler automatically whenever we display a reference variable of any class. The toString() of Object class will always display class name followed by hash code which is not meaningful and understandable to the user.

Predefined code of tostring() of Object class

```
public String toString() {
    return getClass().getName()+"@"+Integer.toHexString(hashCode());
}
```

Output : Sample@32a15b9

It is always recommended to override toString() in every user defined class to display meaningful and understandable message to the user.

Program 84:

```
class Sample {
   public static void main(String[] args) {
       Sample s = new Sample();
       System.out.println(s.toString());
       Employee e1 = new Employee(12,"abcd");
       System.out.println(e1.toString());
       Employee e2 = new Employee(15,"xyz");
       System.out.println(e2.toString());
   }
   public String toString() {
       return "this is sample object";
   }
}
class Employee {
   int empId;
   String ename;
   Employee (int empId, String ename) {
       this.empId = empId;
       this.ename = ename;
   }
   public string toString() {
       return empId+ " : " +ename;
   }
}
```

2) int hashCode(): The hashCode() will generate a unique number for every object that is created.

Note: Hashcode is a unique number which is used to identify the object.

If we don't want the hashcode to be generated by the predefined method then, we can override the hashCode() in our class to generate your own hash code.

If we are overriding the hashCode(), then we are recommended to override the hashCode() in such way that, we generate a unique number for every object that is created.

Program 85:

```
class Sample {
   static int count =12;
   public static void main(String[] args) {
       Sample s1 = new Sample();
       System.out.println(s1.hashCode());
```

```
        Sample s2 = new Sample();
        System.out.println(s2.hashCode());
    }
    public int hashCode() {
        return count++;
    }
}
```

3) boolean equals(Object): The equals() can be used for comparing the objects.
The equals() of Object class will by default compare the hashcodes of the objects. If we don't want to compare the hashcodes of the objects then, we must override the equal() in that class whose object has to be compared and define the meaning of the equality.

Program 86:

```
class Sample {
    public static void main(String[] args) {
        String s1 = new String("hello");
        String s2 = new String("hello");
        System.out.println(s1.equals(s2));
        Product p1 = new Product (12,90);
        Product p2 = new Product (12,90);
        System.out.println(p1.equals(p2));
    }
}

class Product {
    int pid, price;
    Product (int pid, int price) {
        this.pid = pid;
    }
    public boolean equals(Object obj) {
        int id1 = this.pid;
        int pr1 = this.price;
        Product pro = (Product) obj;
        int id2 = pro.pid;
        int pr2 = pro.price;
        if(id1 == id2 && pr1 = pr2)
            return true;
        else
            return false;
    }
}
```


4) Object clone(): This method can be used for taking the backup of an object. Using the clone() we can create duplicate copy of the existing object.

Any object created by the JVM by default cannot be duplicated.

To use the clone() method we need to follow some procedure:

1) The return type of clone() is Object, which has to be typecasted to our required type
2) The clone() throws CloneNotSupportedException which must be handled (because it is a checked exception).
3) The class whose object has to be duplicated, must implement Cloneable interface.

Cloneable is a marked interface or tagged interface which will provide some instructions to the JVM to create the object in a special way so that, we can create a duplicate copy of an object.

Program 87:

```
class Product implements Cloneable {
    int pid =12 , price = 89;
    public static void main(String[] args) throws CloneNotSupportedException {
        Product p1 = new Product();
        System.out.println(p1.pid +" : " + p1.price);
        Object obj = p1.clone();
        Product p2 = (Product) obj;
        p1.pid = 15;
        p1.price = 90;
        System.out.println(p1.pid +" : " + p1.price);
        System.out.println(p2.pid +" : " + p2.price);
    }
}
```

Reflection: It is a process of examining or introspecting the information of a class during the execution time.

All the classes and interfaces related to reflections are available in java.lang.reflect package.

5) Class getClass(): This method will return the information of the class whose object is provided.

Program 88:

```
import java.lang.reflect.*;
class Demo {
    static Object getObject() {
        return new String("hello");
    }
}
```

```
class Sample {
    public static void main(String[] args) {
        Object obj = Demo.getObject();
        Class c = obj.getClass();

        System.out.println(c.getName());
        System.out.println(c.getSuperclass());
        System.out.println(c.getPackage());

        Method[] mthds = c.getMethods();
        for(method x : mthds) {
            System.out.println(x);
        }
        Constructor[] cons = c.getConstructors();
        for(constructor y : cons) {
            System.out.println(y);
        }
        Field[] f = c.getFields();
        for(Field z : f) {
            System.out.println(z);
        }
    }
}
```

Wrapper Classes

These classes are used to wrap or convert a primitive datatypes into a class(object), so that the application is completely or pure object oriented.

For every primitive type in java language, we have a corresponding wrapper class. There are 8 wrapper classes and all are available in java.lang package.

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

Creation of wrapper class objects:

```
Byte b = new Byte(byte);
Byte b = new Byte(String);

Short s = new short(shot);
Short s = new short(String);

Integer i = new Integer(int);
Integer i = new Integer(String);

Long l = new Long(long);
Long l = new Long(String);

Float f  = new Float(float);
Float f = new Float(double);
Float f = new Float(String);

Double d = new Double (double);
Double d = new Double(String);

Character c = new Character(char);

Boolean b = new Boolean(boolean);
Boolean b = new Boolean(String);
```

Example:

```
int x = 12;
Integer y = new Integer(x);
Integer z = new Integer("12");
```

If we create a Boolean class object by specifying the boolean value true, then the result is true and if we create a Boolean class object by specifying the boolean value false then, the result is false. If we create a Boolean class object by specifying a string whose content is true in any case("true", "True", "tRUe", "TRUE") will result as true and any other content will lead to false.

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(false);
Boolean b3 = new Boolean(True);
Boolean b4 = new Boolean(False);
Boolean b5 = new Boolean(TRUE);
Boolean b6 = new Boolean(FALSE);
Boolean b7 = new Boolean("TRUE");
Boolean b8 = new Boolean("TrUe");
Boolean b9 = new Boolean("tRuE");
Boolean b10 = new Boolean("TrUE");
Boolean b11 = new Boolean("TRue");
Boolean b12 = new Boolean("true");
```

```
Boolean b13 = new Boolean("FALSE");
Boolean b14 = new Boolean("FaLsE");
Boolean b15 = new Boolean("fAlSe");
Boolean b16 = new Boolean("false");
Boolean b17 = new Boolean("hello");
Boolean b18 = new Boolean("good morning");
Boolean b19 = new Boolean (null);
Boolean b20 = new Boolean(" ");
```

Methods of wrapper classes:

1) valueOf(): This method can be used for converting a value from primitive type or String type to wrapper type.

Syntax1: static WrapperClass valueOf(primitivetype)
The above syntax can be used for converting a value from primitive type to wrapper type. This method is available in all the 8 wrapper classes.

Syntax2: static WrapperClass valueOf(String)
The above syntax can be used to convert a value from string type to wrapper type. This method is available in all wrapper classes except Character class.

Example:
```
int x = 12;
Integer y = Integer.valueOf (x);
Integer z = Integer.valueOf("12");
```

2) xxxxvalue(): This method can be used for converting a value from wrapper type to primitive type.

byteValue()
shortValue()
intValue()
longValue()
floatValue()
doubleValue()
} These 6 methods are available in Byte, Short, Integer, Long, Float and Double

charValue() This method is available only in Character class

booleanValue() This method is available only in Boolean class

Example:
```
Integer x = Integer.valueOf(12);
byte b = x.byteValue();
int i = x.intValue();
float f = x.floatValue();
```


3) parsexxxx(): This method will convert a value from string type to primitive type. This method is available in all wrapper classes except Character class.

Syntax: static primitivetype parsexxxx(String)

Example:
```
String s = "45";
short x = Short.parseShort(s);
int y = Integer.parseInt(s);
float z = Float.parseFloat(s);
```

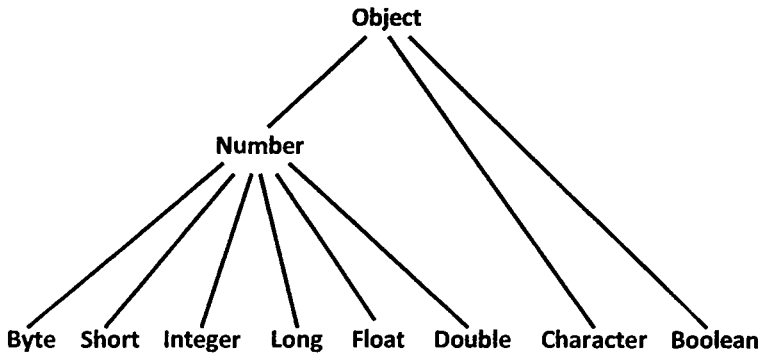
4) toxxxxString(): This method will convert a value from decimal number system to binary, octal, hexadecimal number system respectively.

Syntax:
```
static String toBinaryString(int/long)
static String toOctalString(int/long)
static String toHexString(int/long)
```

The above three methods are available in Integer and Long class only.

Example:
```
int x = 12;
System.out.println(x);
System.out.println(Integer.toBinaryString(x));
System.out.println(Integer.toOctalString(x));
System.out.println(Integer.toHexString(x));
```

Hierarchy of wrapper classes:



Boxing: It is a process of converting a value from primitive type to wrapper type.
This process is done automatically from java 1.5 version onwards and therefore called as auto boxing.

Example:
```
int x = 12;
Integer y = x;
```

Unboxing: It is process of converting a value from wrapper type to primitive type.

This process is done automatically from java 1.5 version onwards and therefore called as auto unboxing.

Example:

```
Integer x = new Integer(34);
int y = x;
```

Variable Arguments

The variable arguments is a concept introduced in Java 1.5 version using which we can pass any number of values. The variable arguments are represented by ... called as ellipsis.

A method or a constructor can contain atmost one variable argument and it should be the last argument.

Note: We can replace a single dimensional array with variable arguments.

Program 89:

```
class Demo {
    void show(double d, int . . . arr) {
        for(int x : arr) {
            System.out.println(x);
        }
    }
    public static void main(String... args) {
        Demo d = new Demo();
        d.show(1.1);
        d.show(1.2,4);
        d.show(1.3,5,6);
        d.show(1.4,7,8,9);
        d.show(1.5,new int[] {3,4,5,6,7,8});
    }
}
```

Rule: When a method is invoked or executed it will look for a method with exact type, if not matching then, it will be promoted to next primitive and perform searching, still not matching continue the process until it reaches double type, still not matching convert it to its corresponding wrapper class and perform searching, still not matching typecast to its parent type, still not matching continue the process until we reach Object class still not matching search for variable argument, still not matching it will generate a error.

1. Exact type
2. Type promotion
3. Boxing
4. Upcasting
5. Variable arguments

enum keyword

The enum keyword is introduced in java 1.5 version and it is designed to create a group of named constants.

Syntax:

```
enum <enumName> {
   Constants
}
```

Example: Day.java

```
enum Day {
   MON, TUE, WED, THU, FRI, SAT, SUN
}
```

When a java program is compiled, the compiler will generate a .class file for every class, every interface and for every enum.

When the above java program is compiled, the compiler generates Day.class

```
javac Day.java

javap Day

final class Day extends Enum {
   public static final Day MON;
   public static final Day TUE;
   :
   public static Day[] values();
}
```

Every enum is internally a final class, extending from Enum class. Enum is an abstract class and belongs to java.lang package. Enum class is subclass of Object class.

- An enum can be empty.
- An enum can contain only constants.
- An enum can contain other code along with constants
- An enum cannot contain only other code without constants.
- If an enum contains only constants then, specifying the semicolon after the constants is optional.
- If an enum contains other code along with constants then, specifying the semicolon after the constants is mandatory.
- If an enum contains other code along with constants, then the constants should be specified as the first statements.
- An enum can contain main method and it can be executed.

Program 90:

```
enum Day {
    MON, TUE, WED, THU, FRI, SAT, SUN;
    public static void main (String[] args) {
        System.out.println("enum main method");
    }
}
```

- An enum can't be instantiated explicitly by the programmer. An enum can be instantiated only by declaring a constant.
- An enum can contain a constructor and it will execute one time for every constant that is declared.

Program 91:

```
enum Day {
    MON, TUE, WED, THU, FRI, SAT, SUN;
    Day() {
        System.out.println("enum constructor");
    }
    public static void main(String[] args) {
        System.out.println("enum main method");
    }
}
```

Program 92:

```
enum Fruit {
    APPLE(40), MANGO(90), GRAPES, BANANA(30), ORANGE(60);
    int price;
    Fruit() {
        price = 50;
    }
    Fruit(int price) {
        this.price = price;
    }
    int getPrice() {
        return price;
    }
}
```



```
class Eat {
    public static void main(String[] args) {
        Fruit f = Fruit.MANGO;
        System.out.println(f+" : "+f.price);
        Fruit[] fr = Fruit.values();
        for(Fruit x : fr) {
            System.out.println(x+" : "+x.getPrice());
        }
    }
}
```

Rule: An argument to switch statement is mandatory and it should be of either byte, short, int, char upto java 1.4 version. From java 1.5 version onwards the switch statements can be 4 primitive types (byte, short, int and char) and therefore corresponding wrapper classes (Byte, Short, Integer, Character) and enum. From java 1.7 version onwards we can specify String as the argument to enum.

- If we are specifying an enum as an argument to switch then, the case labels must be exactly same as that of the constants specified in the enum.

Program 93:

```
enum Day {
    MON, TUE, WED, THU, FRI, SAT, SUN;
}
public class Work {
    public static void main(String[] args) {
        Day d = Day.MON;
        switch(d) {
            case MON : System.out.println("boring");
            break;
            case FRI : System.out.println("preparing");
            break;
            case SAT : System.out.println("sleeping");
            break;
            case SUN : System.out.println("eating");
        }
    }
}
```

- An enum can't be inherited into another enum or class.
- An enum can't inherit from another enum or class.
- An enum can implement any number of interfaces.

Collections

Array: An array is a collection of elements which are stored in continuous memory locations. Arrays can be used for storing both primitive types and objects.

Example:

```
int[] arr = new int[5];
double []marks = new double[10];

Student stu[] = new Student[25];
stu[0] = new Student();          //valid
stu[1] = new Employee();         //invalid

Employee []emp = new Employee[100];
emp[0] = new Employee();         //valid
emp[1] = new Student();          //invalid
```

In the Student array we can store only Student objects but we cannot store Employee objects and in the Employee array we can store only Employee objects but we cannot store Student objects.

To overcome the above problem we can create an array of Object class where we can store an object of any class.

```
Object obj[] = new Object[50];
obj[0] = new Object();
obj[1] = new Student();
obj[2] = new Employee();
```

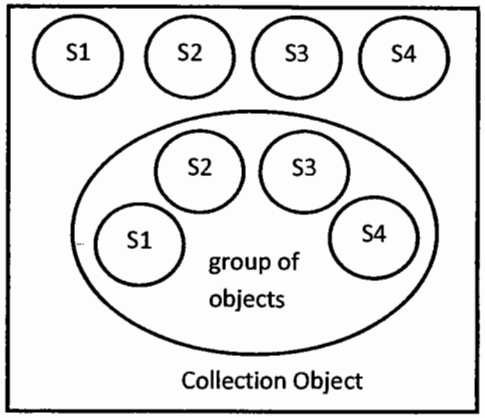
Limitations of array (or) disadvantages of array:

1. The size of an array is fixed i.e., once the array is created, the size cannot be increased or decreased. Therefore either the memory may be wasted or the memory may not be sufficient.
2. To perform the operations like insertion, deletion, searching, sorting etc, the array concept does not provide any predefined methods. The programmer has to write their own logic to perform these operations.

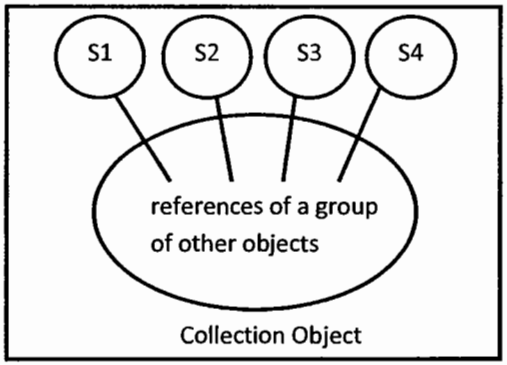
To overcome the limitations of the array concept, the java soft people have come up with a concept called collections. The collections are introduced in java 1.2 version.

The collections are designed to store only objects.

Collection object: An object is said to be a collection object if it holds or stores a group of other objects.



Assume every object requires 2 bytes of memory. To store 4 objects we require 8 bytes of memory. The objects referred by s1, s2, s3 and s4 are stored two times, one time inside the collection object and one time outside the collection object, there by wasting the memory. To save the memory, the jvm instead of storing the objects directly, stores the references of the objects in the collection object.



Collection object: An object is said to be collection object if it holds or stores a group of references of other objects. The collection object can also be called **as container object.**

Collection class: A collection class is a class whose object can hold or store a group of other objects.

Note: collections cannot store primitive type values.

All the collection classes and interfaces are together called as **collection framework**. Collection framework is a library which is implemented in java.util(utility) package.

The advantages of collection framework are:

1) **Reduces programming effort** by providing data structures and algorithms so that we don't have to write them on our own.
2) **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
3) **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
4) New data structures that conform to the standard collection interfaces are by nature **reusable**.

All the collection classes are classified into three categories:

1. **List:** This category can be used for storing a group of individual elements where the elements can be duplicated. List is an interface which cannot be instantiated and therefore we take the help of the implementation classes. The implementation classes of List interface are
   1) ArrayList 2) LinkedList 3) Vector 4) Stack
2. **Set:** This category can be used for storing a group of individual elements where the elements cannot be duplicated. Set is an interface which cannot be instantiated and therefore we take the help of the implementation classes. The implementation classes of Set interface are
   1) HashSet 2) LinkedHashSet 3) TreeSet
3. **Map:** This category can be used for storing the elements in the form of key-value pairs where the keys cannot be duplicated but the values can be duplicated. Map is an interface which cannot be instantiated and therefore we take the help of the implementation classes. The implementation classes of Map interface are
   1) HashMap 2) LinkedHashMap 3) TreeMap 4) Hashtable

List Category: List category can be used for storing a group of individual objects. List category allows duplicates.

ArrayList:

- ArrayList is an implementation class of List interface.
- ArrayList can be used for storing individual objects.
- ArrayList allows duplicates.
- ArrayList allows null value.
- ArrayList is not synchronized.

Creation of ArrayList:

Syntax:

1) ArrayList<E> al = new ArrayList<E>();
   The above syntax creates an empty list with the default initial capacity as 10.
2) ArrayList<E> al = new ArrayList<E>(int initialCapacity);
   The above syntax creates an empty list with the specified capacity as the default initial capacity.
3) ArrayList<E> al = new ArrayList<E>(Collection c);
   The above syntax creates a list with the elements that are available in the specified Collection.

Here, E represents the element data type

Methods of ArrayList:

1. **boolean add(Element obj):** This method is used to place the specified element at the end of List.
2. **void add(int index, Element obj):** This method is used to insert the specified element at the specified index position.
3. **boolean addAll(Collection c):** This method is used to append all the elements available in the specified collection into the list.
4. **boolean addAll(int index, Collection c):** This method is used to insert all the elements available in the specified collection into the list at the specified index position.
5. **boolean remove(Element obj):** This method is used to remove the first occurrence of the specified element.
6. **Element remove(int position):** This method is used to remove an element available at the specified index position.
7. **void clear():** This method will remove all the elements available in the list.
8. **int size():** This method will return the count of the number of elements available in the list.
9. **boolean contains(element obj):** This method returns true if the specified element is available in the list.
10. **Element get(int position):** This method is used to access the element that is available in the specified index position.
11. **Element set(int position, Element obj):** This method is used to replace an element at the specified index position with the specified element.
12. **boolean isEmpty():** This method returns true if the list is empty.
13. **Object[] toArray():** This method converts a list into an array of objects.

Program 94:

```
import java.util.*;
public class ArrayListDemo {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("Nokia");
        al.add("Samsung");
        al.add("Sony");
        al.add("HTC");
        al.add(3,"Motorla");
        System.out.println("List : "+al);
        al.remove("HTC");
        al.remove(1);
        System.out.println("List : "+al);
        System.out.println("size : "+al.size());
        Iterator it = al.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

LinkedList:

- LinkedList is an implementation class of List interface.
- LinkedList can be used for storing individual objects.
- LinkedList allows duplicates.
- LinkedList allows null value.
- LinkedList is not synchronized.

Creation of LinkedList:

Syntax:

1) LinkedList<E> ll = new LinkedList<E>();
   The above syntax creates an empty list.
2) LinkedList<E> ll = new LinkedList<E>(Collection c);
   The above syntax creates a list with the elements that are available in the specified Collection.

Here, E represents the element data type.

Note: LinkedList can use the same methods as specified in the ArrayList.

Difference between Array List and Linked List:

ArrayList is an implementation class which follows resizable array structure. ArrayList is faster in accessing the elements and slower in performing insertions and deletions. LinkedList is an implementation class which follows double linked list structure. LinkedList is slower in accessing the elements and faster in performing insertions and deletions.

Program 95:

```
import java.util.*;
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList<Student> ll = new LinkedList<Student>();
        ll.add(new Student(34));
        ll.add(new Student(12));
        ll.add(new Student(56));
        ll.add(new Student(90));
        ll.add(new Student(78));
        ll.add(2,new Student(45));
        System.out.println("List : "+ll);
        ll.remove(2);
        System.out.println("List : "+ll);

        Iterator it = ll.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }
}

class Student {
    int rollNo;
    Student(int rollNo) {
        this.rollNo = rollNo;
    }
    public String toString() {
        return " "+rollNo;
    }
}
```

Vector:

- Vector is an implementation class of List interface.
- Vector can be used for storing individual objects.
- Vector allows duplicates.
- Vector allows null value.
- Vector is synchronized.

Creation of Vector:

Syntax:

1) Vector<E> v = new Vector<E>();
   The above syntax creates an empty list with the default initial capacity as 10.
2) Vector<E> v = new Vector<E>(int initialCapacity);
   The above syntax creates an empty list with the specified capacity as the default initial capacity.
3) Vector<E> v = new Vector<E>(Collection c);
   The above syntax creates a list with the elements that are available in the specified Collection.

Here, E represents the element data type.

Program 96:

```
import java.util.*;
public class VectorDemo {
    public static void main(String[] xyz) {
        Vector<Integer> v = new Vector<Integer>();
        v.add (new Integer(11));
        v.add(new Integer(22));
        v.add(new Integer(33));
        v.add(44);//auto boxing
        v.add(55);
        v.add(66);
        v.add(1,99);
        System.out.println("List: "+v);
        v.remove(new Integer(22));
        v.remove(1);

        System.out.print("List using for loop: ");
        for(int i=0;i<v.size();i++){
            System.out.print(v.get(i)+" ");
        }
```



```
        System.out.print("\nList using for each loop: ");
        for(int x : v) {
            System.out.print(x+" ");
        }

        ListIterator lit = v.listIterator();
        System.out.print("\nforward direction: ");
        while(lit.hasNext()){
            System.out.print(lit.next()+ " ");
        }

        System.out.print("\nbackward direction: ");
        while(lit.hasPrevious()){
            System.out.print(lit.previous()+" ");
        }
    }
}
```

Stack:

- Stack is an implementation class of List interface.
- Stack can be used for storing individual objects.
- Stack allows duplicates.
- Stack allows null value.
- Stack is synchronized.
- Stack class can be used for implementing stack data structure.

Creation of Stack:

Syntax:

1) Stack<E> s = new Stack<E>();
   The above syntax creates an empty list with the default initial capacity as 10.

Here, E represents the element data type.

Methods of Stack:

1) **Object push(Object):** This method is used to push the specified object into the top of the stack.
2) **Object pop():** This method is used to remove the object from the top of the stack.
3) **Object peek():** This method is used to access the top most element from the stack.
4) **boolean empty():** This method is used to check whether the stack is empty or not.
5) **int search(Object):** This method is used to check whether the specified element is available or not.

Program 97:

```
import java.util.Stack;
public class StackDemo {
    public static void main(String[] args) {
        Stack<Integer> s = new Stack<Integer>();
        s.push(50);
        s.push(20);
        s.push(40);
        s.push(60);
        s.push(10);
        s.push(30);
        System.out.println("List: "+s);

        System.out.println(s.pop());
        System.out.println("List: "+s);

        System.out.println(s.peek());
        System.out.println("List: "+s);
    }
}
```

Differences between the List implementation classes

	ArrayList	LinkedList	Vector	Stack
Ordered	Ordered by Insertion	Ordered by Insertion	Ordered by Insertion	Ordered by Insertion
Duplicates	Allowed	Allowed	Allowed	Allowed
Null value	Allowed	Allowed	Allowed	Allowed
Synchronized	Not Synchronized	Not Synchronized	Synchronized	Synchronized
Data Structure	Resizable Array	Double Linked List	Resizable Array	Resizable Array
Initial Capacity	10	----	10	10

Set Category: Set category can be used for storing a group of individual objects. Set category does not allow duplicates.

HashSet:

- HashSet is an implementation class of Set interface.
- HashSet can be used for storing individual objects.
- HashSet does not allow duplicates.
- HashSet allows null value.
- HashSet is not synchronized.
- HashSet does not guarantee the order of insertion.

Creation of HashSet:

Syntax:

1) HashSet<E> hs = new HashSet<E>();
   The above syntax creates an empty set with the default initial capacity as 16.
2) HashSet<E> hs = new HashSet<E>(int initialCapacity);
   The above syntax creates an empty set with the specified capacity as the default initial capacity.
3) HashSet<E> hs = new HashSet<E>(Collection c);
   The above syntax creates a set with the elements that are available in the specified Collection

Here, E represents the element data type.

Methods of HashSet:

1) **boolean add(Element obj):** This method is used to place the specified element into the set.
2) **boolean remove(Element obj):** This method is used to remove the specified element from the set if available.
3) **boolean contains(Element obj):** This method returns true if the specified element is available in the set.
4) **boolean isEmpty():** This method returns true if the set is empty.
5) **int size():** This method returns the count of the number of elements available in the set.
6) **void clear():** This method is used to remove all the elements from the set.

Program 98:

```
import java.util.*;
public class HashSetDemo {
   public static void main(String[] args) {
      HashSet<Integer> hs = new HashSet<Integer>();
```

```
        hs.add(56);
        hs.add(23);
        hs.add(67);
        hs.add(45);
        hs.add(12);
        hs.add(34);
        System.out.println("Set : "+hs);
        hs.remove(67);
        System.out.println("Set : "+hs);
        Iterator it = hs.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

LinkedHashSet:

- LinkedHashSet is an implementation class of Set interface.
- LinkedHashSet can be used for storing individual objects.
- LinkedHashSet does not allow duplicates.
- LinkedHashSet allows null value.
- LinkedHashSet is not synchronized.
- LinkedHashSet guarantees the order of insertion.

Creation of LinkedHashSet:

Syntax:

1) LinkedHashSet<E> lhs = new LinkedHashSet<E>();
   The above syntax creates an empty set with the default initial capacity as 16.
2) LinkedHashSet<E> lhs = new LinkedHashSet<E>(int initialCapacity);
   The above syntax creates an empty set with the specified capacity as the default initial capacity.
3) LinkedHashSet<E> lhs = new LinkedHashSet<E>(Collection c);
   The above syntax creates a set with the elements that are available in the specified Collection.

Here, E represents the element data type.

Program 99:

```
import java.util.*;
class LinkedHashSetDemo {
    public static void main(String[] args) {
        LinkedHashSet<Integer>lhs  = new LinkedHashSet<Integer>();
        lhs.add(56);
        lhs.add(23);
        lhs.add(67);
        lhs.add(45);
        lhs.add(12);
        lhs.add(34);
        System.out.println("Set : "+lhs);
        lhs.remove(67);
        System.out.println("Set : "+lhs);
        Iterator it = lhs.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

TreeSet:

- TreeSet is an implementation class of Set interface.
- TreeSet can be used for storing individual objects.
- TreeSet does not allow duplicates.
- TreeSet is not synchronized.
- TreeSet sorts the elements in natural order(ascending order).
- TreeSet allows null value in java 1.6 version if it contains only one element and java 1.7 version does not allow null value even if the TreeSet contains one element.

Creation of TreeSet:

Syntax:

1) TreeSet<E> ts = new TreeSet<E>();
   The above syntax creates an empty set where the elements will be sorted in natural order.
2) TreeSet<E> ts = new TreeSet<E>(Comparator c);
   The above syntax creates an empty set where the elements will be sorted according to the specified comparator.

3) TreeSet<E> ts = new TreeSet<E>(Collection c);
The above syntax creates a set with the elements that are available in the specified Collection.

Here, E represents the element data type.

Program 100:

```
import java.util.*;
class TreeSetDemo {
   public static void main(String[] args) {
      TreeSet<Integer> ts = new TreeSet<Integer>();
      ts.add(56);
      ts.add(23);
      ts.add(67);
      ts.add(45);
      ts.add(12);
      ts.add(34);
      System.out.println("Set : "+ts);
      Iterator it = ts.iterator();
      while(it.hasNext()){
         System.out.println(it.next());
      }
   }
}
```

The TreeSet is a class which sorts the elements by default in natural order(ascending order). If we want to change the order of sorting, then we need to implement **Comparator** interface.

Comparator interface is available in java.util package and it contains 2 methods

1) public abstract int compare(Object o1, Object o2)
2) public abstract boolean equals(Object obj)

The default logic of compare() is to sort the elements in ascending order(natural order).

```
int compare(Object o1, Object o2) {
   if(o1 < o2)
      return –VE;
   else if(o1 > o2)
      return +VE;
   else
      return 0;
}
```


Program 101:

```
import java.util.*;
class TreeSetDemo {
   public static void main(String[] args) {
      TreeSet<Integer> ts = new TreeSet<Integer>(new Comparator<Integer>() {
         public int compare(Integer i1, Integer i2) {
            if(i1 < i2)
               return 12;
            else if(i1 > i2)
               return -12;
            else
               return 0;
         }
      });
      ts.add(56);
      ts.add(23);
      ts.add(67);
      ts.add(45);
      ts.add(12);
      ts.add(34);
      System.out.println("Set : "+ts);

      Iterator it = ts.iterator();
      while(it.hasNext()) {
         System.out.println(it.next());
      }
   }
}
```

TreeSet is a class which stores the elements in sorted order. To sort the elements in sorted order, we take the help of Comparator interface. The Comparator interface belongs to java.util package, which will decide whether the elements have to be sorted either in ascending order or descending order.

Two objects can be compared by the Comparator interface only if those objects are eligible for comparison. An object is said to be eligible for comparison when its corresponding class implements Comparable interface.

The Comparable interface is available in java.lang package and it contains only one method.

1) public int compare(Object obj)

Program 102:

```
import java.util.*;
class Student implements Comparable {
    int rollNo;
    Student(int rollNo) {
        this.rollNo = rollNo;
    }
    public int compareTo(Object obj) {
        Student s = (Student)obj;
        if(this.rollNo < s.rollNo)
            return -12;
        else if(this.rollNo>s.rollNo)
            return 12;
        else
            return 0;
    }
    public String toString() {
        return "" + rollNo;
    }
}
public class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet<Student> ts = new TreeSet<Student>();
        ts.add(new Student(14));
        ts.add(new Student(11));
        ts.add(new Student(15));
        ts.add(new Student(13));
        ts.add(new Student(16));
        ts.add(new Student(12));
        System.out.println("Set : "+ts);
    }
}
```


Differences between the Set implementation classes

	HashSet	LinkedHashSet	TreeSet
Ordered	Unordered	Ordered by Insertion	Sorted Order
Duplicates	Not Allowed	Not Allowed	Not Allowed
Null value	Allowed	Allowed	Allowed
Synchronized	Not Synchronized	Not Synchronized	Not Synchronized
Data Structure	Hashtable	Hashtable + double linked list	Balanced Tree
Initial Capacity	16	16	----

Map Category: Map category can be used for storing a group of objects in the form of key-value pairs. Map category does not allow keys to be duplicated whereas values can be duplicated.

HashMap:

- HashMap is an implementation class of Map interface.
- HashMap can be used for storing the elements in the form of key-value pairs, where the keys cannot be duplicated and the values can be duplicated.
- HashMap allows null in both keys and values.
- HashMap is not synchronized.
- HashMap does not guarantee the order of insertion.

Creation of HashMap:

Syntax:

1) HashMap<K,V> hm = new HashMap<K,V>();
   The above syntax creates an empty map with the default initial capacity as 16.
2) HashMap<K,V> hm = new HashMap<K,V>(int initialCapacity);
   The above syntax creates an empty map with the specified capacity as the default initial capacity.
3) HashMap<K,V> hm = new HashMap<K,V>(Map m);
   The above syntax creates an empty map with the elements that are available in the specified Map.

Here, K represents the type of the key and V represents the type of the value.

Methods of HashMap:

1) **value put(Object key, Object value):** This method is used to place a key and a value as a pair into the map.
2) **value remove(Object key):** This method is used remove the specified key and it's corresponding value.
3) **value get(Object key):** This method will return the value of the key that is specified.
4) **Set keySet():** This method returns all the keys available in the map in the form of a set.
5) **Collection values():** This method returns all the values available in the map in the form of a collection.
6) **void clear():** This method is used to remove all the key-value pairs from the map.
7) **int size():** This method will return the count of the number of key-value pairs available in the map.
8) **boolean containsKey(Object key):** This method returns true if the specified key is available in the map.
9) **boolean containsValue(Object value):** This method returns true if the specified value is available in the map.
10) **boolean isEmpty():** This method returns true if the map is empty.

Program 103:

```
import java.util.*;
public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, Integer> hm = new HashMap<String, Integer>();
        hm.put("mnop",40);
        hm.put("abcd",80);
        hm.put("pqrs",70);
        hm.put("qwer",60);
        hm.put("stuv",50);
        hm.put("ghij",30);
        System.out.println("Elements : "+hm);

        hm.remove("stuv");
        System.out.println("Elements : "+hm);

        Set s = hm.keySet();
        System.out.println("keys : "+s);

        Iterator it = s.iterator();
```



```
        while(it.hasNext()) {
            Object obj = it.next();
            String str = (String) obj;
            System.out.println(str+" "+hm.get(str));
        }

        Collection<Integer> c = hm.values();
        System.out.println("values : "+c);
        System.out.println(hm.containsKey("abcd"));
        System.out.println(hm.containsValue(50));
    }
}
```

LinkedHashMap:

- LinkedHashMap is an implementation class of Map interface.
- LinkedHashMap can be used for storing the elements in the form of key-value pairs, where the keys cannot be duplicated and the values can be duplicated.
- LinkedHashMap allows null in both keys and values.
- LinkedHashMap is not synchronized.
- LinkedHashMap guarantees the order of insertion.

Creation of LinkedHashMap:

Syntax:

1) LinkedHashMap<K,V> lhm = new LinkedHashMap<K,V>();
   The above syntax creates an empty map with the default initial capacity as 16.
2) LinkedHashMap<K,V> lhm = new LinkedHashMap<K,V>(int initialCapacity);
   The above syntax creates an empty map with the specified capacity as the default initial capacity.
3) LinkedHashMap<K,V> lhm = new LinkedHashMap<K,V>(Map m);
   The above syntax creates an empty map with the elements that are available in the specified Map.

Here, K represents the type of the key and V represents the type of the value.

Program 104:

```
import java.util.*;
public class LinkedHashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> lhm = new LinkedHashMap<String, Integer>();
        lhm.put("mnop",40);
        lhm.put("abcd",80);
        lhm.put("pqrs",70);
        lhm.put("qwer",60);
        lhm.put("stuv",50);
        lhm.put("ghij",30);
        System.out.println("Elements : "+lhm);

        lhm.remove("stuv");

        Set s = lhm.keySet();
        System.out.println("keys : "+s);

        Iterator it = s.iterator();
        while(it.hasNext()) {
            Object obj = it.next();
            String str = (String) obj;
            System.out.println(str+" "+lhm.get(str));
        }

        Collection<Integer> c = lhm.values();
        System.out.println("values : "+c);

        System.out.println(lhm.containsKey("abcd"));
        System.out.println(lhm.containsValue(50));
    }
}
```


TreeMap:

- TreeMap is an implementation class of Map interface.
- TreeMap can be used for storing the elements in the form of key-value pairs, where the keys cannot be duplicated and the values can be duplicated.
- TreeMap is not synchronized.
- Map does not guarantee the order of insertion.
- TreeMap sorts the keys in natural order(ascending order).
- TreeMap allows null into keys in java 1.6 version, if it contains only one element and java 1.7 version does not allow null value even if the TreeSet contains one element. TreeMap allows null into value.

Creation of TreeMap:

Syntax:

1) TreeMap<K,V> tm = new TreeMap<K,V>();
   The above syntax creates an empty map where the elements will be sorted in natural order.
2) TreeMap<K,V> tm = new TreeMap<K,V>(Comparator c);
   The above syntax creates an empty map where the elements will be sorted according to the specified comparator.
3) TreeMap<K,V> tm = new TreeMap<K,V>(Map m);
   The above syntax creates a map with the elements that are available in the specified Collection.

Here, K represents the type of the key and V represents the type of the value.

Program 105:

```
import java.util.*;
public class TreeMapDemo {
  public static void main(String[] args) {
  TreeMap<String, Integer> tm = new TreeMap<String, Integer>();
      tm.put("mnop",40);
      tm.put("abcd",80);
      tm.put("pqrs",70);
      tm.put("qwer",60);
      tm.put("stuv",50);
      tm.put("ghij",30);
      System.out.println("Elements : "+tm);
  }
}
```

Hashtable:

- Hashtable is an implementation class of Map interface.
- Hashtable can be used for storing the elements in the form of key-value pairs, where the keys cannot be duplicated and the values can be duplicated.
- Hashtable does not allow null in both keys and values.
- Hashtable is synchronized.
- Hashtable does not guarantee the order of insertion.

Creation of Hashtable:

Syntax:

1) Hashtable<K,V> ht = new Hashtable<K,V>();
   The above syntax creates an empty map with the default initial capacity as 11.
2) Hashtable<K,V> ht = new Hashtable<K,V>(int initialCapacity);
   The above syntax creates an empty map with the specified capacity as the default initial capacity.
3) Hashtable<K,V> ht = new Hashtable<K,V>(Map m);
   The above syntax creates an empty map with the elements that are available in the specified Map.

Here, K represents the type of the key and V represents the type of the value.

Program 106:

```
import java.util.*;
public class HashtableDemo {
    public static void main(String[] args) {
        Hashtable<String, Integer> ht = new Hashtable<String, Integer>();
        ht.put("mnop",40);
        ht.put("abcd",80);
        ht.put("pqrs",70);
        ht.put("qwer",60);
        ht.put("stuv",50);
        ht.put("ghij",30);
        System.out.println("Elements : "+ht);

        Enumeration e = ht.keys();
        while(e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
    }
}
```


Differences between the Map implementation classes

	HashMap	LinkedHashMap	TreeMap	Hashtable
Ordered	Unordered	Ordered by Insertion	Sorted Order	Unordered
Duplicates keys values	Not Allowed Allowed	Not Allowed Allowed	Not Allowed Allowed	Not Allowed Allowed
Null value	Allowed	Allowed	Allowed	Aot Allowed
Synchronized	Not Synchronized	Not Synchronized	Not Synchronized	Synchronized
Data Structure	Hashtable	Hashtable + Double Linked list	Red-Black Tree	Hashtable
Initial Capacity	16	16	-----	11

Cursors of collection framework: The cursors available in java.util package can be used for accessing the elements one by one and perform some other operations. There are 3 cursors and they are:

1) Iterator
2) ListIterator
3) Enumeration

Iterator:

- Iterator is an interface.
- This cursor can be used for accessing the elements one by one.
- This cursor can be applied to all the classes that implement Collection interface.
- Iterator can be used for accessing the elements in forward direction only.
- Iterator can be used for performing an additional task like removing the elements.

Iterator interface contains 3 methods and they are:

1. boolean hasNext()
2. Object next()
3. void remove()

Program 107:

```
import java.util.*;
public class IteratorDemo {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<Integer>();
```

```
        al.add(45);
        al.add(23);
        al.add(67);
        al.add(12);
        al.add(56);
        al.add(34);
        al.add(78);

        Iterator it = al.iterator();
        while(it.hasNext()) {
            Object obj = it.next();
            Integer i = (Integer) obj;
            if(i==34 | i==78)
                it.remove();
        }
        System.out.println(al);
    }
}
```

ListIterator:

- ListIterator is an interface.
- This cursor can be used for accessing the elements one by one.
- This cursor can be applied to all the classes that implement List interface. ListIterator can be used for accessing the elements in both forward and backward directions.
- ListIterator can be used for performing an additional tasks like removing, adding and replacing the elements.

ListIterator interface contains 9 methods and they are:

1. boolean hasNext()
2. Object next()
3. boolean hasPrevious()
4. Object previous()
5. int nextIndex()
6. int previousIndex()
7. void remove()
8. void set(Object)
9. void add(Object)

program 108:

```
import java.util.*;
public class ListIteratorDemo {
    public static void main(String[] args) {
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.add(45);
        ll.add(23);
        ll.add(67);
        ll.add(12);
        ll.add(56);
        ll.add(34);
        ll.add(78);
        ListIterator<Integer> lit = ll.listIterator();
        while(lit.hasNext()) {
            Object obj = lit.next();
            Integer i = (Integer) obj;
            if(i==56)
                lit.remove();
            if(i==45)
                lit.set(99);
            if(i==12)
                lit.add(22);
        }
        while(lit.hasPrevious()) {
            System.out.println(lit.previous());
        }
    }
}
```

Enumeration:

- Enumeration is an interface.
- This cursor can be used for accessing the elements one by one.
- This cursor can be applied to all the legacy classes.
- Enumeration can be used for accessing the elements in forward direction only.

Enumeration interface contains 2 methods and they are:

1. boolean hasMoreElements()
2. Object nextElement()

Program 109:

```
import java.util.*;
public class EnumerationDemo {
    public static void main(String[] args) {
        Vector<Integer> v = new Vector<Integer>();
        v.add(45);
        v.add(23);
        v.add(67);
        v.add(12);
        v.add(56);
        v.add(34);
        v.add(78);

        Enumeration e = v.elements();
        while(e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
    }
}
```

Differences between the Cursors

	Iterator	ListIterator	Enumeration
Applicable to	Classes implementing Collection interface	Classes implementing List interface	Legacy classes
Accessibility	Forward direction	Forward and backward	Forward direction
Operations performed	Accessing Removing	Accessing, Removing Replacing, Adding	Accessing
Method to access	iterator()	listIterator()	elements()
Number of methods	3	9	2

Collections: This class belongs to collection framework which consists exclusively of static methods that operate on collections.

Methods of Collections class:

1) **void sort(List):** This method is used to sort the elements available in the List class.
2) **int binarySearch(List, Object):** This method is used to perform binary search operation on the elements of the List.
3) **void reverse(List):** This method is used to reverse the elements available in the List.
4) **void swap(List, int index1, int index2):** This method is used to swap the elements available in the specified index positions.
5) **void copy(List, List):** This method is used to copy the elements of one List to another List.
6) **Object min(Collection):** This method will return the smallest element available in the specified Collection.
7) **Object max(Collection):** This method will return the biggest element available in the specified Collection.
8) **List synchronizedList(List):** This method is used to convert an unsynchronized list to synchronized list.
9) **Set synchronizedSet(Set):** This method is used to convert an unsynchronized set to synchronized set.
10) **Map synchronizedMap(Map):** This method is used to convert an unsynchronized map to synchronized map.

Program 110:

```
import java.util.*;
public class CollectionsDemo {
   public static void main(String[] ar) {
       ArrayList<Integer> al = new ArrayList<Integer>();
       al.add(56);
       al.add(23);
       al.add(67);
       al.add(89);
       al.add(34);
       al.add(78);
       al.add(12);
       al.add(45);
       System.out.println("List : "+al);
       Collections.reverse(al);
       System.out.println("List : "+al);
```

```
        Collections.sort(al);
        System.out.println("List : "+al);
        Collections.swap(al,1,4);
        System.out.println("List : "+al);
        System.out.println(Collections.min(al));
        System.out.println(Collections.max(al));
        List l = Collections.synchronizedList(al);
        System.out.println("List : "+l);
    }
}
```

Arrays: This class is a special class available in java.util package, using which we can perform some operations on the Arrays.

Methods of Arrays:

1) **void sort(Object[]):** This method can be used to sort the elements in the specified array in natural order.
2) **void sort(Object[], int index, int offset):** This method can be used to sort the elements in the specified range of the array in natural order.
3) **int binarySearch(Object[], Object):** This method can be used to search the specified array of objects for the specified value using the binary search algorithm.
4) **List asList(Object[]):** This method can be used to convert the elements of an array into a List.

Program 111:

```
import java.util.*;
public class ArraysDemo {
    public static void main(String[] ar) {
        Integer[] arr = {3,6,1,7,2,4,8,5};
        for(Integer x : arr) {
            System.out.println(x);
        }
        Arrays.sort(arr);
        for(Integer x : arr) {
            System.out.println(x);
        }
        List l = Arrays.asList(arr);
        System.out.println(l);
    }
}
```


NavigableSet: This interface is introduced in java 1.6 version and it provides some special methods, which are designed for accessing the elements from the TreeSet class.

Methods of NavigableSet:

1) **Object lower(Object element):** This method will return the highest element which is smaller than the specified element.
2) **Object floor(Object element):** This method will return the highest element which is smaller than or equal to the specified element.
3) **Object higher(Object element):** This method will return the smallest element which is bigger than the specified element.
4) **Object ceiling(Object element):** This method will return the smallest element which is bigger than or equal to the specified element.
5) **Object pollFirst():** This method will delete the first element from the set.
6) **Object pollLast():** This method will delete the last element from the set.
7) **NavigableSet descendingSet():** This method will sort the elements in descending order.
8) **SortedSet subSet(Object begin, Object end):** This method will return a group of elements starting from the specified begin element upto the specified end element. The begin element will be included and the end element will be excluded.
9) **SortedSet headSet(Object end):** This method will return a group of elements starting from the first element upto the specified end element. The specified end element will be excluded.
10) **SortedSet tailSet(Object begin):** This method will return a group of elements starting from the specified begin element upto the last element. The specified begin element will be included.

Program 112:

```
import java.util.*;
public class NavigableSetDemo {
  public static void main(String[] args) {
      TreeSet<Integer> ts = new TreeSet<Integer>();
      ts.add(56);
      ts.add(89);
      ts.add(67);
      ts.add(45);
      ts.add(23);
      ts.add(78);
      ts.add(12);
      ts.add(34);
      System.out.println("set : "+ts);
```

```
        System.out.println(ts.lower(56));
        System.out.println(ts.floor(56));
        System.out.println(ts.higher(56));
        System.out.println(ts.ceiling(56));
        System.out.println(ts.subSet(45,78));
        System.out.println(ts.headSet(56));
        System.out.println(ts.tailSet(56));
        System.out.println(ts.descendingSet());
    }
}
```

NavigableMap: This interface is introduced in java 1.6 version and it provides some special methods, which are designed for accessing the elements from the TreeMap class.

Methods of NavigableMap:

1) **Entry lowerEntry(Object key):** This method will return the highest key and its corresponding value which is smaller than the specified element.
2) **Entry floorEntry(Object key):** This method will return the highest key and its corresponding value which is smaller than or equal to the specified element.
3) **Entry higherEntry(Object key):** This method will return the smallest key and its corresponding value which is bigger than the specified element.
4) **Entry ceilingEntry(Object key):** This method will return the smallest key and its corresponding value which is bigger than or equal to the specified element.
5) **Object lowerKey(Object key):** This method will return the highest key which is smaller than the specified element.
6) **Object floorKey(Object key):** This method will return the highest key which is smaller than or equal to the specified element.
7) **Object higherKey(Object key):** This method will return the smallest key which is bigger than the specified element.
8) **Object ceilingKey(Object key):** This method will return the smallest key which is bigger than or equal to the specified element.
9) **Entry firstEntry():** This method will return the first entry available in the map.
10) **Entry lastEntry():** This method will return the last entry available in the map.
11) **Entry pollFirstEntry():** This method will delete the first entry available in the map.
12) **Entry pollLastEntry():**This method will delete the last entry available in the map.
13) **Entry descendingMap():**This method will sort the elements in descending order.
14) **Entry subMap(Object begin, Object end):** This method will return a group of elements starting from the specified begin element upto the specified end element. The begin element will be included and the end element will be excluded.

15) **Entry headMap(Object end):** This method will return a group of elements starting from the first element upto the specified end element. The specified end element will be excluded.
16) **Entry tailMap(Object begin):** This method will return a group of elements starting from the specified begin element upto the last element. The specified begin element will be included.

Program 113:

```
import java.util.*;
public class NavigableMapDemo {
  public static void main(String[] args){
     TreeMap<Integer,Integer> tm = new TreeMap<Integer,Integer>();
     tm.put(56,40);
     tm.put(89,10);
     tm.put(67,50);
     tm.put(45,80);
     tm.put(23,70);
     tm.put(78,30);
     tm.put(12,60);
     tm.put(34,20);
     System.out.println("map : "+tm);

     System.out.println(tm.lowerEntry(56));
     System.out.println(tm.floorEntry(56));
     System.out.println(tm.higherEntry(56));
     System.out.println(tm.ceilingEntry(56));
     System.out.println(tm.lowerKey(56));
     System.out.println(tm.floorKey(56));
     System.out.println(tm.higherKey(56));
     System.out.println(tm.ceilingKey(56));

     System.out.println(tm.firstEntry());
     System.out.println(tm.pollLastEntry());

     System.out.println(tm.subMap(45,78));
     System.out.println(tm.headMap(56));
     System.out.println(tm.tailMap(56));
     System.out.println(tm.descendingMap());
  }
}
```

StringTokenizer: This class is used to break a string into tokens (pieces).
Syntax: StringTokenizer st = new StringTokenizer(String, delimiter);

Program 114:

```
import java.util.*;
public class StringTokenizerDemo {
    public static void main(String[] args) {
        String str = "oneatwoathreeafour8five8six8seven#eight#nine#ten";
        StringTokenizer  st = new StringTokenizer(str,"a8#");
        System.out.println(st.countTokens());
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

Date: The Date class represents a specific instance of time.
Calendar: The Calendar class is an abstract class. It provides some methods for converting an instance of a time to day, month, year, second, minute, hour etc.
We cannot create an object of Calendar class directly. We can get the object of Calendar class by using a static method called getInstance() of Calendar class.

Program 115:

```
import java.util.*;
public class DateDemo {
    public static void main(String[] args) {
        Date d = new Date();
        System.out.println("Date: "+d);
        Calendar c = Calendar.getInstance();
        int date = c.get(Calendar.DATE);
        int month = c.get(Calendar.MONTH);
        int year = c.get(Calendar.YEAR);
        System.out.println("Date: "+date+ "/" +(++month)+ "/" +year);
        int hour = c.get(Calendar.HOUR);
        int minute = c.get(Calendar.MINUTE);
        int second = c.get(Calendar.SECOND);
        System.out.println("Time = "+hour+":"+minute+":"+second);
    }
}
```


Generics

Generics is a concept introduced in the java 1.5 version. Generics are called as **parameterized types.**
The generics are represented by a pair of angular brackets (<>), called as diamond operator.
Generics are designed to provide compile time **type safety**, which will reduce the need for typecasting.
Generics are said to be type **erasures**, which means the generic type information will be available only up to the compilation time, once the compilation is done the generic type information will be erased.

Procedure to create a generic method: To create a generic method, we need to specify the generic type parameter before the return type of the method.

Syntax:

```
<E> returntype methodName() {
}
```

E represents the generic type parameter.

Program 116:

```
public class GenericMethod {
   public <T> void display(T[] temp) {
      for(T x : temp) {
         System.out.println(x);
      }
   }

   public static void main(String[] args) {
      GenericMethod gm = new GenericMethod();
      Integer[] iarr = {1,2,3,4,5};
      gm.display(iarr);
      Double[] darr = {1.2,2.3,3.4,4.5,5.6};
      gm.display(darr);
      String[] sarr = {"abc","def","ghi","stu","xyz"};
      gm.display(sarr);
   }
}
```

Procedure to create Generic class: Declare the Generic type parameter after the class declaration.

Syntax: class ClassName<E> {
}

Program 117:

```
class MyClass<T> {
    T obj;
    MyClass(T obj) {
        this.obj = obj;
    }
    T getValue(){
        return obj;
    }
    public void showType() {
        System.out.println("type : " +obj.getClass().getName());
    }
}
public class GenericDemo{
    public static void main(String[] args){
        Integer iobj = new Integer(123);
        MyClass<Integer> mc1 = new MyClass<Integer>(iobj);
        System.out.println("value : " +mc1.getValue());
        mc1.showType();
        Double dobj = new Double(34.5);
        MyClass<Double> mc2 = new MyClass<Double>(dobj);
        System.out.println("value : " +mc2.getValue());
        mc2.showType();
        String sobj = new String("java");
        MyClass<String> mc3 = new MyClass<String>(sobj);
        System.out.println("value : " +mc3.getValue());
        mc3.showType();
    }
}
```


Procedure to create Generic interface: Declare the Generic type parameter after the interface declaration.

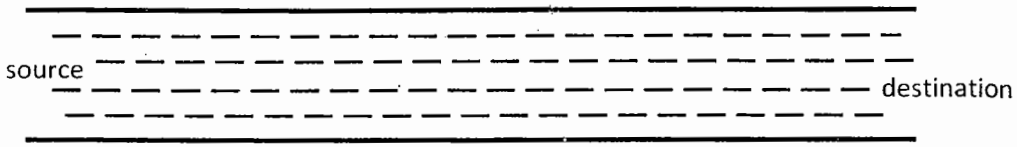
Syntax: interface InterfaceName<E> {
}

Program 118:

```
interface MyInterface<A,B> {
    void add(A a, B b);
}
public class GenericInterface1 implements MyInterface<String, String> {
    public void add(String s1, String s2) {
        System.out.println(s1+s2);
    }
    public static void main(String[] args) {
        GenericInterface1 gi1 = new GenericInterface1();
        gi1.add("hello","friends");
    }
}
public class GenericInterface2 implements MyInterface<Integer, Integer> {
    public void add(Integer i1, Integer i2) {
        System.out.println(i1+i2);
    }
    public static void main(String[] args){
        GenericInterface2 gi2 = new GenericInterface2();
        gi2.add(12,34);
    }
}
```

IOStreams

A stream represents a sequential flow of data from one location to another location.



The streams that are used for performing input and output operations are called as IOStreams. The streams are classified into two types and they are

1. Byte Streams
2. Character Streams

Byte Stream: The streams which perform the operations byte by byte are called as byte streams. These streams can handle any kind of data like text, audio, video, images etc. The byte streams are further classified into two categories.

1) **InputStream:** The input streams are used for performing reading operation from any resource. Using these streams we can read data into the application in the form of byte by byte.
   Example: FileInputStream
2) **OutputStream:** The output streams are used for performing writing operation into any resource. Using these streams we can write data from the application in the form of byte by byte.
   Example: FileOutputStream

Character Stream: The streams which perform the operations character by character are called as character streams. These streams can handle only text. They are also called as text streams. The character streams are faster than byte streams. The character streams are further classified into two categories.

1) **Reader:** The readers are similar to input streams performing reading operations from any resource. Using the reader we can read data into the application in the form of character by character.
   Example: FileReader
2) **Writer:** The writers are similar to output streams performing writing operations into any resource. Using the writer we can write data from the application in the form of character by character.
   Example: FileWriter

All the stream related classes are available in java.io.package.

DataInputStream: The DataInputStream is used to perform reading operation from any resource. The DataInputStream allows the application to read primitive data types.

Syntax:

```
DataInputStream dis = new DataInputStream(InputStream);
```

Program 119:

```
import java.io.*;
public class ReadingData {
   public static void main(String[] ar) throws IOException {
      DataInputStream dis = new DataInputStream(System.in);
      int ch;

      while((ch = dis.read()) != '$') {
         System.out.print((char) ch);
      }
   }
}
```

FileInputStream: The FileInputStream is used for reading the contents from a file.

Syntax:

```
FileInputStream fis = new FileInputStream(String);
FileInputStream fis = new FileInputStream(File);
```

The above two syntax can be used for reading the contents from the specified file. If the specified file is not available then we get an exception called FileNotFoundException.

Program 120:

```
import java.io.*;
public class FileRead {
   public static void main(String[] ar) throws IOException {
      FileInputStream fis = new FileInputStream("input.txt");
      BufferedInputStream bis = new BufferedInputStream(fis);
      int ch;

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

FileOutputStream: The FileOutputStream is used for writing the contents into a file.

Syntax:

1) FileOutputStream fos = new FileOutputStream(String);
2) FileOutputStream fos = new FileOutputStream(File);

The above two syntax can be used for writing the contents into the specified file. If the specified file is not available then it will create a new file and then write the contents but if the specified file is already available then it will overwrite the contents.

3) FileOutputStream fos = new FileOutputStream(String, boolean);
4) FileOutputStream fos = new FileOutputStream(File, boolean);

The above two syntax can be used for writing the contents into the specified file. If the specified file is not available then it will create a new file and then write the contents but if the specified file is already available then we can either append the contents or overwrite the contents.

Program 121:

```
import java.io.*;
public class FileCopy {
    public static void main(String[] ar) throws FileNotFoundException, IOException {
        FileInputStream fis = new FileInputStream("input.txt");
        FileOutputStream fos = new FileOutputStream("output.txt", true);

        int ch;
        while((ch = fis.read()) != -1) {
            fos.write(ch);
        }
        fis.close();
        fos.close();
    }
}
```

Note: If the specified output file is not available then we do not get FileNotFoundException instead a file with the specified name will be created automatically.

File: File is a predefined class using which we can refer to either a file or a directory.

Syntax:

1) File f = new File(String path);
   The above syntax will create a new file using the specified path.
2) File f = new File(String parent, String child);
   The above syntax will create a new file using the specified parent path and child path.

3) File f = new File(File parent, String child);
   The above syntax will create a new file using the specified parent path and child path.

FileReader: This class is used to Read the content from a file char by char

Syntax:

FileReader fr = new FileReader(String);
FileReader fr = new FileReader(File);

The above two syntax can be used for reading the contents from the specified file. If the specified file is not available then we get an exception called FileNotFoundException.

Program 122:

```
import java.io.*;
public class FileReaderDemo {
    public static void main(String[] ar) throws FileNotFoundException, IOException {
        File f = new File("D:/test/input.txt");
        FileReader fr = new FileReader(f);
        int ch;
        while((ch = fr.read()) != -1) {
            System.out.print((char)ch);
        }
        fr.close();
    }
}
```

FileWriter: The FileWriter is used for writing the contents into a file.

Syntax:

1) FileWriter fw = new FileWriter(String);
2) FileWriter fw = new FileWriter(File);

The above two syntax can be used for writing the contents into the specified file. If the specified file is not available then it will create a new file and then write the contents but if the specified file is already available then it will overwrite the contents.

3) FileWriter fw = new FileWriter(String,boolean);
4) FileWriter fw = new FileWriter(File, boolean);

The above two syntax can be used for writing the contents into the specified file. If the specified file is not available then it will create a new file and then write the contents but if the specified file is already available then we can either append the contents or overwrite the contents.

The FileWriter class provides various methods like

1) write(int)
2) write(String)
3) write(String, int, int)
4) write(char[])
5) write(char[], int , int)
6) flush() : This method can be used to clear the data available in the buffer.
7) close() : This method can be used to release the resource.

Every stream will be associated with a buffer and it will be cleared in the following situations:

1) When the buffer is completely filled, it will be cleared automatically.
2) When the streams are released.
3) When we invoke flush()

Program 123:

```
import java.io.*;
public class FileWriterDemo {
    public static void main(String[] ar) throws FileNotFoundException, IOException {
        File f = new File("D:/test/output.txt");
        FileWriter fw = new FileWriter(f);

        fw.write(100);

        String str = "abcdefghijklmnopqrstuvwxyz";
        fw.write(str);

        char[] ch = str.toCharArray();
        fw.write(ch);

        fw.close();
    }
}
```


ByteArrayInputStream: The ByteArrayInputStream is used to read data from the byte array.

Syntax:

```
ByteArrayInputStream bais = new ByteArrayInputStream(byte[]);
```

Program 124:

```
import java.io.*;
public class ByteArrayInputStreamDemo {
    public static void main(String[] args) throws IOException {
        String str = "abcdefghijklmnopqrstuvwxyz";
        byte[] buf = str.getBytes();

        ByteArrayInputStream bais = new ByteArrayInputStream(buf);

        int ch;
        while((ch = bais.read())!=-1) {
            System.out.println((char)ch);
        }
    }
}
```

ByteArrayOutputStream: The ByteArrayOutputStream is used to write data into the byte array.

Syntax:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

Program 125:

```
import java.io.*;
public class ByteArrayOutputStreamDemo {
    public static void main(String args[]) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();

        String str = "this is a byte array demo";
        byte buf[] = str.getBytes();
        baos.write(buf);

        byte b[] = baos.toByteArray();
        for (int i=0; i<b.length; i++) {
            System.out.print((char) b[i]);
        }
    }
}
```

DeflaterStreams: These streams are used for compressing the data. There are two deflater streams.

1) **DeflaterInputStream:** DeflaterInputStream is used for compressing the data in deflate format during the reading time.
   DeflaterInputStream dis = new DeflaterInputStream(InputStream);
2) **DeflaterOutputStream:** DeflaterOutputStream is used for compressing the data in deflate format during the writing time.
   DeflaterOutputStream dos = new DeflaterOutputStream(OutputStream);

Program 126:

```
import java.io.*;
import java.util.zip.*;
public class Compression {
    public static void main(String[] ar) throws IOException {
        FileInputStream fis = new FileInputStream("input.txt");
        FileOutputStream fos = new FileOutputStream("temp.txt");
        DeflaterOutputStream dos = new DeflaterOutputStream(fos);

        int ch;

        while((ch = fis.read()) != -1) {
            dos.write(ch);
        }
        fis.close();
        dos.close();
    }
}
```

InflaterStreams: These streams are used for uncompressing the data. There are two inflater streams.

1) **InflaterInputStream:** InflaterInputStream is used for uncompressing the data in deflate format during the reading time.
   Syntax:
   InflaterInputStream iis = new InflaterInputStream(InputStream);
2) **InflaterOutputStream:** InflaterOutputStream is used for uncompressing the data in deflate format during the writing time.
   Syntax:
   InflaterOutputStream ios = new InflaterOutputStream(OutputStream);

program 127:

```
import java.io.*;
import java.util.zip.*;
public class UnCompression {
    public static void main(String[] ar) throws IOException {
        FileInputStream fis = new FileInputStream("temp.txt");
        InflaterInputStream iis = new InflaterInputStream(fis);
        FileOutputStream fos = new FileOutputStream("output.txt");
        int ch;
        while((ch = iis.read()) != -1) {
            fos.write(ch);
        }
        iis.close();
        fos.close();
    }
}
```

ObjectInputStream: ObjectInputStream is used for reading an object.
Syntax:
ObjectInputStream ois = new ObjectInputStream(InputStream);

ObjectOutputStream: ObjectOutputStream is used for writing an object.
Syntax:
ObjectOutputStream oos = new ObjectOutputStream(OutputStream);

In an application, if we are creating an object of any class, then that object will be stored in heap memory. Since the heap memory is part of the JVM and the JVM is part of RAM memory, we cannot guarantee the existence of the object in the heap memory.

If we want to retain the object or use the same object in future, then we need to make the object persistent(permanent). In order to make the object persistent, we need to copy the object into a file.

By copying the object into a file, we can make the object persistent and we can transfer the object from one location to another location with the help of files.

An object can be made persistent only if the object is serialized. An object is said to be serialized when its corresponding class is implementing Serializable interface.

The Serializable interface is called as **marked interface** or **tagged interface**. An interface is said to marked or tagged, if it is empty i.e. it has no methods. The Serializable interface provides some special instructions to the JVM so that the JVM creates an object in special way so that the object can be broken into pieces.

Serialization: It is a process of converting an object into a stream of bytes.

Deserialization: It is a process of converting a stream of bytes into an object.

We can perform serialization and deserialization only when the object is serialized, otherwise a runtime error called NotSerializableException.

Program 128:

```
import java.io.*;
public class Customer implements Serializable {
    int custId = 1234;
    int pinNo = 5678;
    void getCustomerDetails() {
        System.out.println(custId+" : "+pinNo);
    }
}
```

```
import java.io.*;
public class StoreObject {
    public static void main(String[] ar) throws IOException {
        FileOutputStream fos = new FileOutputStream("mydata.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        Customer c = new Customer();
        oos.writeObject(c);
        oos.close();
    }
}
```

```
import java.io.*;
public class ReadObject {
    public static void main(String[] ar) throws IOException, ClassNotFoundException {
        FileInputStream fis = new FileInputStream("mydata.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Object obj = ois.readObject();
        Customer c = (Customer) obj;
        c.getCustomerDetails();
        ois.close();
    }
}
```


Transient: If a serialized object is transferred from one location to another location, then all the data available in that object will be transferred from one location to another location.

If we do not want to transfer any data(variable) from the serialized object, then declare the data(variable) as transient.

If a serialized object contains transient variable then it will hide the actual value of that variable and instead it will transfer the default value of that variable.

```
transient int pinNo = 5678;
```

transient is a modifier which can be applied to only variables. The transient keyword will have its effect only on serialized objects.

The transient keyword should be applied to only instance variables which are not final.

A transient keyword will not have any effect on static variable because the static variable will not participate in serialization as static variables are not part of the objects.

A transient keyword will not have any effect on final variable because the final variable will participate in serialization directly.

PrintWriter: PrintWriter is the most efficient class for performing the writing operations. PrintWriter class prints formatted representations of objects to a text stream. PrintWriter is a character based class which makes the Java program easy for Internationalization.

Syntax:

```
PrintWriter pw = new PrintWriter(File);
PrintWriter pw = new PrintWriter(String);
PrintWriter pw = new PrintWriter(Writer);
PrintWriter pw = new PrintWriter(Writer, boolean);
PrintWriter pw = new PrintWriter(OutputStream);
PrintWriter pw = new PrintWriter(OutputStream, boolean);
```

Methods of PrintWriter:

1) write(int)
2) write(char[])
3) write(char[], int offset, int length)
4) write(String)
5) write(String, int offset, int length)
6) print(xxxx)
7) println(xxxx)
8) flush()
9) close()

Program 129:

```
import java.io.*;
public class PrintWriterDemo {
    public static void main(String[] ar) throws IOException {
        FileWriter fw = new FileWriter("data.txt");
        PrintWriter pw = new PrintWriter(fw);
        pw.write(100);
        pw.print("hello");
        pw.println("good morning");
        pw.print(100);
        pw.flush();
    }
}
```

PrintStream: PrintStream is the most efficient class for writing the data into various resources. The PrintStream class should be used in situations that require writing characters rather than bytes. The PrintStream never throws IOException and it will automatically invoke flush().

Syntax:

```
PrintStream pw = new PrintStream(File);
PrintStream pw = new PrintStream(String);
PrintStream pw = new PrintStream(OutputStream);
PrintStream pw = new PrintStream(OutputStream, boolean);
```

Program 130:

```
import java.io.*;
public class PrintStreamDemo {
    public static void main(String[] ar) throws IOException {
        FileOutputStream fos = new FileOutputStream("data.txt");
        PrintStream ps = new PrintStream(fos);
        //PrintStream ps = new PrintStream(System.out);
        ps.write(100);
        ps.print("hello");
        ps.println("good morning");
        ps.print(100);
        ps.flush();
    }
}
```


Program 131 to read a single character from a keyboard without io package

```
public class Reading {
    public static void main(String[] ar) throws Exception {
        int ch = System.in.read();
        System.out.println((char)ch);
    }
}
```

Program 132 to read a multiple characters from a keyboard without io package

```
public class Reading {
    public static void main(String[] ar) throws Exception {
        int ch;
        while((ch = System.in.read()) != '@') {
            System.out.print((char)ch);
        }
    }
}
```

InputStreamReader: An InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters.

Program 133 to read a single line from a keyboard

```
import java.io.*;
public class Reading {
    public static void main(String[] ar) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str = br.readLine();
        System.out.print(str);
    }
}
```

The System class contains three predefined reference variables for performing reading and writing operations and they are

1) in: in is a reference variable of InputStream which is by default connected to standard input device(keyword).
2) out: out is a reference variable of PrintStream which is by default connected to standard output device(monitor) used for displaying normal messages.
3) err: err is a reference variable of PrintStream which is by default connected to standard output device(monitor) used for displaying error messages.

System.out.println:

System is a predefined class available in java.lang package. out is a reference variable of PrintStream class referring to the object of PrintStream class declared as static inside System class. println() is a predefined instance method of PrintStream class. PrintStream is a predefined class available in java.io package.

Since out is declared as static in System class, it can be accessed directly by using class name(and therefore we specify System.out). System.out will give the object of PrintStream class, with that object we can invoke println() and therefore we specify System.out.println().

Program 134 to read the data from a file by using System.in.read()

```
import java.io.*;
public class Reading {
    public static void main(String[] ar) throws IOException {
        FileInputStream fis = new FileInputStream("data.txt");
        System.setIn(fis);
        int ch;
        while((ch = System.in.read()) != -1) {
            System.out.print((char)ch);
        }
    }
}
```

Program 135 to divert the contents into a file by using System.out.println()

```
import java.io.*;
public class Writing {
    public static void main(String[] ar) throws IOException {
        FileOutputStream fos1 = new FileOutputStream("file1.txt");
        PrintStream ps1 = new PrintStream(fos1);
        System.setOut(ps1);
        System.out.println("hai friends");
        FileOutputStream fos2 = new FileOutputStream("file2.txt");
        PrintStream ps2 = new PrintStream(fos2);
        System.setErr(ps2);
        System.err.println("bye friends");
    }
}
```


Scanner: Scanner class is introduced in java 1.5 version and it is available in java.util package. It can be used for reading the contents from any resource. The Scanner class provides methods using which we can read any type of data without performing parsing.

Syntax:

```
Scanner sc = new Scanner(File);
Scanner sc = new Scanner(String);
Scanner sc = new Scanner(InputStream);
```

Program 136:

```
import java.util.*;
public class ScannerDemo {
    public static void main(String[] ar) {
        Scanner sc = new Scanner(System.in);
        int age = sc.nextInt();
        System.out.println(age);
        double marks = sc.nextDouble();
        System.out.println(marks);
    }
}
```

Console: Console class is introduced in java 1.6 version and it is available in java.io package. This class can be used to read the contents from the keyboard. Console class is available in java.io package. We cannot create an object of Console class directly, it can be created by using console() available in System class.

Program 137:

```
import java.io.*;
public class ConsoleDemo {
    public static void main(String[] ar) throws IOException {
        Console c = System.console();
        String name = c.readLine("Enter your name : ");
        System.out.println(name);
        char[] pwd = c.readPassword("Enter your password : ");
        System.out.println(pwd);
    }
}
```

Multithreading

Single tasking: The process of executing a single task at a time is called as single tasking. In single tasking much of the processor time is wasted.
Ex: DOS

Multi tasking: The process of executing multiple tasks at the same time is called as multitasking. In multitasking the processor time is utilized in an optimum way.
Ex: Windows

The process of loading and unloading the process into the memory is called as **context switching**. In multitasking, the processor time is divided among the tasks that are executed. The small amount of processor time that is given to a particular task for execution is called as **time slice**.

Advantage of multitasking: Using multitasking we reduce the waiting time and improve the response time and thereby improving the performance of the application.

Multitasking is of two types. They are:

1. Process based multitasking
2. Thread based multitasking

Process based multitasking: The process of executing different processes simultaneously at the same time is called as process based multitasking. Every process is independent of each other. Every process contains its own set of resources. Process based multitasking is an operating system approach.
Ex: writing java program, down loading s/w, listening music, copying s/w etc.

Thread based multitasking: The process of executing different parts of the same process simultaneously at the same time is called as thread based multitasking. The different parts may be dependent or independent of each other but they share the same set of resources. Thread based multitasking is a programmatic approach.
Ex: games, web applications etc.

Multithreading: The process of executing multiple threads at the same time is called as multi threading or thread based multitasking. If the application contains multiple threads then the application is said to be multithreaded.

Thread: A thread is a separate piece of code which will be executed separately.

Program 138 to get currently executing thread information:

```
public class ThreadInfo {
   public static void main(String[] ar) {
      Thread t = Thread.currentThread();
      System.out.println("Thread Info : " + t);
   }
}
```

```
Thread Info : Thread[main,5,main]
```

Every java program will by default contain one thread called as main thread, which is used for executing a java program and we can get the information of the currently executing thread by using currentThread().

currentThread() is a static method available in Thread class and the Thread class is available in java.lang package. The currentThread() provides the information of the currently executing thread like thread name, thread priority and the thread group name.

A java program contains a main thread by default. If we want our program to contain multiple threads then we can create our own threads called user defined threads. The threads that are created by the user or the programmer are called as user defined threads. We can create any number of user defined threads.

The user defined thread can be created in two ways.

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

Procedure to create a user defined thread by extending Thread class:

1. Create a class as sub class to Thread class.
   class MyClass extends Thread
2. Write the functionally of user thread within the run method.
   public void run()
3. Create the object of the class that is extending Thread class
   Myclass mc = new Myclass();
4. Attach the above created object to the Thread class
   Thread t = new Thread(mc);
5. Execute the user thread by invoking start();
   t.start();

Program 139:

```
public class ThreadDemo extends Thread {
    public void run() {
        for(int i=1; i<=10; i++) {
            System.out.println("user Thread : " + i);
        }
    }
    public static void main(String[] ar) {
        ThreadDemo td = new ThreadDemo();
        Thread t = new Thread(td);
        t.start();
    }
}
```

Procedure to create a user defined thread by extending Runnable interface:

1. Create a class implementing Runnable interface.
   class Myclass implements Runnable
2. Write the functionality of user thread within the run method
   public void run()
3. Create the object of class that is implementing Runnable interface.
   MyClass mc = new MyClass();
4. Attach the above created object to the Thread class
   Thread t = new Thread(mc);
5. Execute the user thread by invoking start method
   t.start();

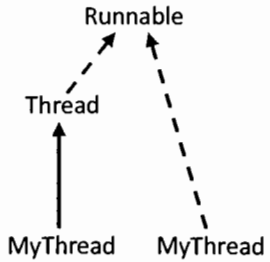
Program 140:

```
public class RunnableDemo implements Runnable {
    public void run() {
        for(int i=1; i<=10; i++) {
            System.out.println("user Thread : " + i);
        }
    }
    public static void main(String[] args) {
        RunnableDemo rd = new RunnableDemo();
        Thread t = new Thread(rd);
        t.start();
    }
}
```


Difference between extends Thread and implements Runnable:

When we create a thread by extending Thread class, then we do not have a chance to extend from another class whereas when we create a thread by implementing Runnable interface, then we have a chance to extend from another class.

Note: It is recommended to implement Runnable interface to create a user defined thread.



Program 141 creating multiple user defined threads acting on multiple objects

```
class MultiThread implements Runnable {
    String name;
    MultiThread(String name) {
        this.name = name;
    }
    public void run() {
        for(int i=1; i<=10; i++) {
            System.out.println(name+" : " + i );
        }
    }
}
public class MultiThreadDemo {
    public static void main(String[] args) {
        MultiThread mt1 = new MultiThread("Thread1");
        MultiThread mt2 = new MultiThread("Thread2");
        Thread t1 = new Thread(mt1);
        Thread t2 = new Thread(mt2);
        t1.start();
        t2.start();
        for(int i=1; i<=10; i++) {
            System.out.println("main : " + i);
        }
    }
}
```

If a program contains multiple threads then we cannot guarantee the order of thread execution. The execution of the threads in an application will be decided by **Thread Scheduler**, which is part of the JVM.

Program 142 creating multiple user defined threads acting on same object

```
class College implements Runnable {
    int seats;
    College(int seats) {
        this.seats = seats;
    }
    public void run() {
        Thread t = Thread.currentThread();
        String name = t.getName();
        System.out.println(name +" no of seats before allotment : " + seats);
        if(seats > 0) {
            try {
            Thread.sleep(2000);
                System.out.println("seat allotted to : " + name);
                seats = seats -1;
            }
            catch(InterruptedException ie) {
                ie.printStackTrace();
            }
        }
        else {
            System.out.println("seat not allotted to : " + name);
        }
        System.out.println(name +" no of seats after allotment : " +seats);
    }
}
public class Allotment {
    public static void main(String[] ar) {
        College c = new College(60);
        Thread t1 = new Thread(c);
        Thread t2 = new Thread(c);
        t1.setName("student1");
```



```
        t2.setName("student2");
        t1.start();
        t2.start();
    }
}
```

When multiple threads are acting on the same object then, there is a chance of data inconsistency problem occurring in the application.

Data inconsistency problem occurs when one of the threads is updating the value and the other thread is trying to read the value at the same time.

To avoid the data inconsistency problem we have to synchronize the threads that are the acting on the same object.

Thread Synchronization: When multiple threads wants to access the same object at the same time and giving access to only one of the thread is called as thread synchronization.

Thread synchronization can be done into two ways.

1. Synchronized Block
2. Synchronized Method

Synchronized Block: If we want to synchronize a group of statements or part of a code then we use synchronized block.

Syntax:
```
synchronized(object) {
        statements;
    }
```

Synchronized Method: If we want to synchronize all the statements in a method then we use synchronized method.

Syntax:
```
synchronized returntype methodName(list of parameters) {
        statements;
    }
```

Note: In the previous program multiple threads are acting on the same object at the same time leading to data inconsistency problem. To avoid the data inconsistency problem, we have to synchronize the threads acting on the same object.

```
public synchronized void run() {
    same code from the previous program;
}
```

When multiple threads are acting on synchronized object then there is chance of other problems like deadlock occurring in the application.

Deadlock: When a thread holds a resource and waits for another resource to be released by second thread, the second thread holding a resource and waiting for a resource to be realized by first thread, then in such case both the threads will be waiting indefinitely and they never execute. This situation is called as Deadlock.

In java language there is no mechanism to avoid deadlock situation. It is the responsibility of the programmer to write proper logic to avoid the deadlock situation.

Creation of a Thread:

1) Thread t = new Thread();
   The above syntax will create a thread having a default name.
2) Thread t = new Thread(String name);
   The above syntax will create a thread with the specified name.
3) Thread t = new Thread(Runnable obj);
   The above syntax will create a thread which is attached to the specified object.
4) Thread t = new Thread(Runnable obj, String name);
   The above syntax will create a thread with the specified name and attach the specified object.
5) Thread t = new Thread(ThreadGroup tg, Runnable obj);
   The above syntax will create a thread which is attached to the specified object and place the thread in the specified thread group.
6) Thread t = new Thread(ThreadGroup tg, String name);
   The above syntax will create a thread with the specified name and place the thread in the specified thread group.
7) Thread t = new Thread(ThreadGroup tg, Runnable obj, String name);
   The above syntax will create a thread with the specified name, attach the specified object and place the thread in the specified thread group.

Methods of Thread class:

1. **currentThread():** This method is used to provide the information of currently executing thread. The currentThread() will provide 3 values and they are thread name, thread priority and thread group name.

   Program 143:

```
class ThreadInfo extends Thread {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("user thread : "+t);
    }
    public static void main(String[] ar) {
        ThreadInfo ti = new ThreadInfo();
        ti.start();
        Thread t = Thread.currentThread();
        System.out.println("main thread : "+t);
    }
}
```

Output:

```
main thread : Thread[main,5,main]
user thread : Thread[Thread-0,5,main]
```

2. **start():** This method is used to execute a user defined thread. The functionality of user defined thread is available in run() method.
   When we invoke start(), the start() method will internally invoke run() method, but before invoking run() method, it will perform some low level activities.
   Low Level Activities are registering the user defined thread with **Thread Scheduler**, once the registration is done the thread scheduler will allocate some resources (like memory, processor etc) to the registered user defined thread, then the registered user defined thread will now invoke run() method.
   We can execute the logic available in run() method by invoking run() method directly in such case, the logic of run() method will be executed by the main thread and the run() method will be considered as a normal method.

   We cannot call start() method two times or multiple times on the same thread because, we cannot register the same thread multiple times. If we call start() method multiple times on the same thread then we will get run time error IllegalThreadStateException.

3. **sleep(long milliseconds):** This method is used to suspend the execution of a thread for a specified amount of time. This method throws InterruptedException, which must be handled.
4. **getName():** This method can be used for retrieving the name of the thread.
5. **setName(String name):** This method can be used to assign a name to a thread. The default names of the user defined threads will be Thread-0, Thread-1, Thread-2, ...
   Note: We can change the name of any thread including the main thread.

Program 144:

```
class ThreadDemo extends Thread {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("User Thread : " + t.getName());
        t.setName("student");
        System.out.println("User Thread : " + t.getName());
    }
    public static void main(String[] args) {
        ThreadDemo td=new ThreadDemo();
        td.start();
        Thread t = Thread.currentThread();
        System.out.println("Main Thread : " + t.getName());
        t.setName("inetsolv");
        System.out.println("Main Thread : " + t.getName());
    }
}
```

6. **getPriority():** This method can be used to access the priority of a thread.
7. **setPriority(int priority):** This method is used to change the priority of a thread.

The priority of a thread will indicate the amount of resources to be allocated to a thread. Allocation of resources to a thread will be based on thread priority i.e., high priority thread will get high resources and lower priority thread will get less resources.

When we want to change the thread priority it is always recommended to take the support of the constants declared in the Thread class.

MIN_PRIORITY
NORM_PRIORITY
MAX_PRIORITY

Example: t.setPriority(8); valid but not recommended
t.setPriority(Thread.MAX_PRIORITY-2); valid and recommended

We are recommended to use the constants when we are changing the priority of a thread because, the range of thread priority differs from JVM to JVM. If we are specifying a priority value outside the range, then we get a run time error called IllegalArgumentException.
When a thread is created, the child thread gets a priority same as that of the parent thread priority, which can be later on either increased or decreased.

Note: Priority of a thread will not decide the sequence of thread execution.

Program 145:

```
class ThreadDemo extends Thread {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("user Thread : " + t.getPriority());
        t.setPriority(Thread.MAX_PRIORITY-2);
        System.out.println("user Thread : " + t.getPriority());
    }
    public static void main(String[] ar) {
        Thread t = Thread.currentThread();
        System.out.println("main Thread : " + t.getPriority());
        t.setPriority(Thread.NORM_PRIORITY+1);
        System.out.println("main Thread : " + t.getPriority());
        ThreadDemo td = new ThreadDemo();
        td.start();
    }
}
```

8. **getThreadGroup():** This method can be used to retrieve the information of the group to which the thread belongs.

The advantage of thread group is that we can communicate with multiple threads at the same time. By default the main thread and all the user defined threads belong to main group. We can create our own group. Thread group can be created by using ThreadGroup class. Thread group has to be created at the time of thread creation and we can place any number of user defined threads in the created group. We can place the user defined thread in any group, but the main thread will belong to main group only.

Program 146:

```
class ThreadDemo extends Thread {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("user : " + t.getThreadGroup().getName());
    }
    public static void main(String[] ar) {
        Thread t = Thread.currentThread();
        System.out.println("main : " + t.getThreadGroup().getName());
        ThreadDemo td = new ThreadDemo();
        ThreadGroup tg = new ThreadGroup("inetsolv");
```

```
        Thread t1 = new Thread(tg,td);
        t1.start();
    }
}
```

9. **isAlive():** This method can be used to check whether a thread is alive or not. A user defined thread is said to be alive when is executing run() method.
10. **isDaemon():** This method can be used to check whether a thread is a daemon thread or not.
11. **setDeamon():** This method can be used for converting a thread from non-daemon to daemon and from non-daemon to daemon.

    In every application the execution begins from main thread and terminates with the main thread. In between the beginning and termination of main thread we can execute any number of user defined threads.
    When a thread is created, it will be either daemon or non-daemon depending upon the parent thread that is if the parent thread is a non-daemon thread then the child thread will also be non-daemon thread and if the parent thread is a daemon thread then the child thread will also be daemon thread.
    The main thread in java program is by default a non-daemon thread and any thread created under the main thread will also be non-daemon and those threads can be later if required can be converted to a daemon thread.
    If the main thread has finished its task then it has to be terminated. Before the main thread is terminated, the thread scheduler will verify whether any other non-daemon threads are still executing or not, if executing then the main thread will not be terminated until the other non-daemon does not finish their execution. Once all the other non-daemon threads finish their execution, then the main thread will be terminated. The main thread will not wait for the execution of the daemon thread. Once the main thread is terminated, then the daemon threads will also be terminated even if they are executing.

12. **join():** This method can be used to suspend the execution of a thread until another thread dies to which the thread is joined. This method throws InterruptedException, which must be handled.

Program 147:

```
class ThreadDemo extends Thread {
    public void run() {
        for(int i=1;i<=100;i++) {
```



```
            System.out.println("user : " + i);
        }
    }
    public static void main(String[] ar) throws InterruptedException {
        ThreadDemo td = new ThreadDemo();
        Thread t = new Thread(td);
        System.out.println(t.isDaemon());
        t.setDaemon(true);
        System.out.println(t.isDaemon());
        t.start();
        t.join();
    }
}
```

Methods related to thread available in Object class:

1) **wait():** This method is used to suspend the execution of a thread until it receives a notification.
   wait(long milliseconds): This method is used to suspend the execution of a thread until it receives a notification or a specified amount of time has elapsed.
   wait(long milliseconds, int nanoseconds): This method is used to suspend the execution of a thread until it receives a notification or a specified amount of time has elapsed.
2) **notify():** This method is used to send a notification to one of the waiting threads.
3) **notifyAll():** This method is used to send a notification to all the waiting threads.

Note: The above three methods are used for inter thread communication. The wait(), the notify() and notifyAll() must be called with in a synchronized block otherwise we get a run time error called IllegalMonitorStateException.

The wait(), notify() and notifyAll() are available in Object class, so that we can use those methods directly in our logic to make communication without the reference of Thread class.

Program 148:

```
public class MyThread extends Thread {
    static int total = 0;
    public synchronized void run() {
        System.out.println("user thread started calculation");
        for(int i=1; i<=10; i++) {
            total = total + i;
        }
```

```
        System.out.println("user thread sending notification");
        notifyAll();
        System.out.println("user total = " + total);
    }
    public static void main(String[] ar) throws InterruptedException {
        MyThread mt = new MyThread();
        Thread t = new Thread(mt);
        System.out.println("main thread calling user thread");
        t.start();
        synchronized(mt) {
            mt.wait();
        }
        System.out.println("main thread got notification");
        System.out.println("main Total = " + mt.total);
    }
}
```

Program 149: for inter thread communication

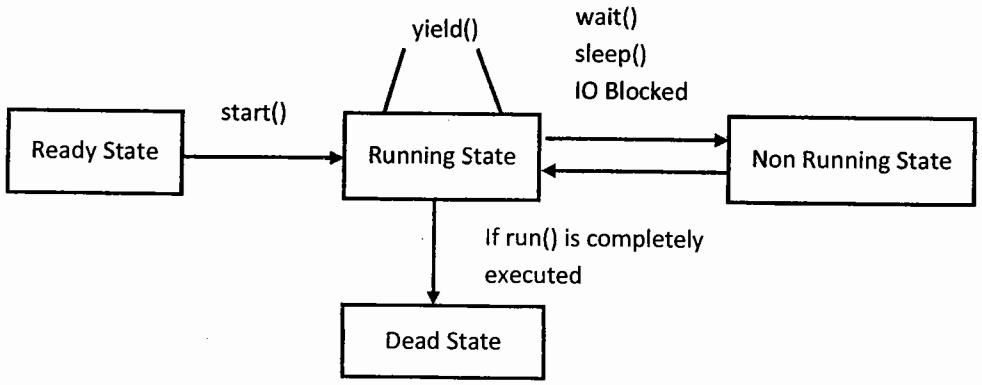
```
class ShowRoom {
    int value;
    boolean flag = true;
    public synchronized void produce(int i) {
        if(flag == true) {
            value = i;
            System.out.println("Produced value: " +i);
            notify();
            flag = false;
        }
        try {
            wait();
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
    }
    public synchronized int consume() {
        if(flag == true) {
```



```
            try{
                wait();
            }
            catch(InterruptedException ie) {
                ie.printStackTrace();
            }
        }
        notify();
        flag = true;
        return value;
    }
}
class Producer extends Thread {
    ShowRoom s;
    Producer(ShowRoom s) {
        this.s = s;
    }
    public void run() {
        int i = 1;
        while(true) {
            s.produce(i);
            i = i + 1;
            try {
                Thread.sleep(2000);
            }
            catch(InterruptedException ie) {
                System.out.println(ie);
            }
        }
    }
}
class Consumer extends Thread {
    ShowRoom s;
    Consumer(ShowRoom s) {
        this.s = s;
    }
```

```
    public void run() {
        while(true) {
            int x = s.consume();
            System.out.println("Consumed value: " + x);
            try {
                Thread.sleep(2000);
            }
            catch(InterruptedException ie) {
                System.out.println(ie);
            }
        }
    }
}
public class ProducerConsumer {
    public static void main(String[] args) {
        ShowRoom s = new ShowRoom();
        Producer p = new Producer(s);
        Consumer c = new Consumer(s);
        Thread t1 = new Thread(p);
        Thread t2 = new Thread(c);
        t1.start();
        t2.start();
    }
}
```

THREAD LIFE CYCLE



Inner Classes

The classes that are declared inside another class are called as inner classes or nested classes.

The **advantages** of inner classes are:

1) Inner classes have a special type of relationship that is it can access all the members (data members and methods) of outer class including private.
2) Inner classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
3) An inner class requires less code to write.
4) Nesting small classes within top-level classes places the code closer to where it is used.
5) If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.

Based on the location of the inner class or based on the keyword specified to the inner class, they are classified into 4 types.

1) Regular Inner class
2) Method Local Inner class
3) Anonymous Inner class
4) Static Inner class

Regular Inner class: If a class is declared inside a class directly without any keyword, then it is called as regular inner class.

Syntax:

```
class Outer {
    class Inner  {
    }
}
```

When we compile the above program, we get two .class files and they are

1) Outer.class
2) Outer$Inner.class

Program 150:

```
class Outer {
   class Inner  {
       public static void main(String[] ar) {
           System.out.println("inner class main method");
       }
   }
   public static void main(String[] ar) {
       System.out.println("outer class main method");
   }
```

ıe above code is invalid because the regular inner class cannot have any static declarations.
ıe regular inner class can be declared with final, abstract, public, private, protected and rictfp.

ccessing regular inner class inside the instance methods of outer class.

·ogram 151:

```
ass Outer {
  class Inner  {
      void innerMethod() {
          System.out.println("inner class method");
      }
  }
  void outerMethod() {
      System.out.println("outer class method");
      Inner i = new Inner();
      i.innerMethod();
  }
  public static void main(String[] ar) {
      System.out.println("outer class main method");
      Outer o = new Outer();
      o.outerMethod();
  }
```


Accessing regular inner class outside the instance methods of outer class(i.e. inside the static methods of outer class, inside instance and static methods of other class).

Program 152:

```
class Outer {
    class Inner  {
        void innerMethod() {
            System.out.println("inner class method");
        }
    }
    public static void main(String[] ar) {
        System.out.println("outer class main method");
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();
                (OR)
        Outer.Inner i = new Outer().new Inner();
        i.innerMethod();
    }
}
```

Method Local Inner class: If a class is declared inside a method(either instance or static), then it is called as method local inner class.
Method local inner class can be instantiated only in that method where it is declared and after the declaration.
Method local inner class cannot use the local variables declared in that method.
Method local inner class can use the local variable if it is declared as final.

Program 153:

```
class Outer {
    void outerMethod() {
        class Inner {
            void innerMethod() {
                System.out.println("inner class method");
            }
        }
        Inner i = new Inner();
        i.innerMethod();
    }
```

```
    public static void main(String[] ar) {
        System.out.println("outer class main method");
        Outer o = new Outer();
        o.outerMethod();
    }
```

nonymous Inner class: If an inner class is declared without a class name then it is called as nonymous inner class.

nonymous inner classes will be instantiated at the time of declaration only.

rogram 154:

```
ass Parent {
  void msg() {
      System.out.println("hai friends");
  }


ass Sample {
  public static void main(String[] ar) {
      Parent p1 = new Parent();
      p1.msg();
      Parent p2 = new Parent() {
          void msg() {
              System.out.println("bye friends");
          }
      };
      p2.msg();
}
```


Anonymous inner classes can be used to override a method available in a class or implement a n interface.

Program 155:

```
class Sample {
    public static void main(String[] ar) {
        Thread t = new Thread() {
            public void run() {
                System.out.println("hello friends");
            }
        };
        t.start();
    }
}
```

Program 156:

```
class Sample {
  public static void main(String[] ar) {
    Thread t = new Thread(new Runnable() {
      public void run() {
        System.out.println("hai friends");
      }
    });
    t.start();
  }
}
```

ic Inner class: If a class is declared inside a class with static keyword, then it is called as static er class.

atic nested class does not have access to the instance variables and methods of the outer s.

ic inner class can be accessed without instantiating the outer class, using other static nbers.

gram 157:

```
s Outer {
atic class Inner {
oid innerMethod() {
  System.out.println("inner method");


blic static void main(String[] ar) {
ystem.out.println("outer class main method");
nner i = new Inner();
.innerMethod();
```

ic inner class can have static declarations and they can be executed independent of the er class.

gram 158

```
s Outer {
atic class Inner {
ublic static void main(String[] ar) {
  System.out.println("inner class main method");


blic static void main(String[] ar) {
ystem.out.println("outer class main method");
```