Unit-I
Introduction to Java

Unit	Topic Name	Main Topic	Subtopics	Marks
1	Introduction to Java	Introduction	History, OOPS Features, Reusability of Code, Modularity of Code	14
		Features	Simple, Compiled and Interpreted, Platform independent and portable, Object oriented Distributed, Multithreaded and interactive, High performance, Secure.	
		Data Types	Data Types, Constant, Symbolic Constant, Scope of variable, Type casting, Standard default values, Java Literals.	
		Operators & Expressions	Arithmetic Operators, Bit wise Operator, Relational Operators, Boolean Logical Operators, Assignment Operator Increment and Decrement Operator, Conditional Operator, Special Operator, Operator Precedence.	
		Control Statements	If, Switch, For, While, Do-While, Break, Continue etc. and Math class functions.	

A. Features of OOPS:

1. CLASS: Class is a **collection of objects of similar type**. Once a class has been defined, we can create any number of objects belonging to that class. It is a logical entity. Example: Fruit; NOTE: **Classes** are **user defined data types**.

2. OBJECT: Object is a **basic runtime entity**. Objects are **reference variables of type class** and take up space in the memory. Objects are instances of classes. Any entity that has state and behavior is known as an object. It can be physical and logical. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

3. DATA ABSTRACTION: Hiding the complexity of program is called **Abstraction** and only essential features are represented. In short we can say that internal working is hidden. **Hiding internal details and showing functionality** is known as abstraction. In java, we use abstract class and interface to achieve abstraction.

4. ENCAPSULATION: Combining data and functions into a single unit called class and the process is known as **Encapsulation**. Data is not accessible from the outside world. **Binding (or wrapping) code and data together into a single unit is known as encapsulation**. A java **class** is the example of encapsulation.

5. INHERITANCE: it is the process by which **object of one class acquire the properties or features of objects of another class**. The concept of inheritance provide the idea of reusability means we can add additional features to an existing class without Modifying it.

6. POLYMORPHISM: A Greek term means **ability to take more than one form**. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. In java, we use **method overloading** and **method overriding** to achieve polymorphism.

7. DYNAMIC BINDING: Refers to **linking of function call** with function definition is called binding and when it is take place at **run time** called dynamic binding.

8. MESSAGE PASSING: The process by which **one object can interact with other object** is called message passing.

9. RESILIENCE TO CHANGE: Resilience is the **process of adapting** well in the face of adversity. It is the ability to recover from adversity. Needless to say, developing **resilience** is a highly desirable quality in today's ever-**changing** world.

10. REUSABILITY OF CODE: Code reuse, also called software reuse, is the use of existing software, or software knowledge, to build new software, following the **reusability** principles.

11. MODULARITY OF CODE: Modularity is the degree to which a system's components may be separated and recombined. **Modularity** is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable **modules**, such that each contains everything necessary to execute only one aspect of the desired functionality.

B. Benefits of OOPS:

1. **Reusability**: In OOP's programs functions and modules that are written by a user can be reused by other users without any modification.
2. **Inheritance**: Through this we can eliminate redundant code and extend the use of existing classes.
3. **Data Hiding**: The programmer can hide the data and functions in a class from other classes. It helps the programmer to build the secure programs.
4. **Reduced complexity of a problem**: The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.
5. **Easy to Maintain and Upgrade**: OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
6. **Message Passing**: The technique of message communication between objects makes the interface with external systems easier.
7. **Modifiability**: it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.
8. **Security:** OOPS is more secure than procedure oriented languages

C. Comparisons:

1. Procedure Oriented Vs. Object Oriented

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.

Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	overloading is possible in the form of Function and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

2. C Vs. Java

	C Programming	**Java Programming**
1	Procedure Oriented	Object Oriented
2	Function is a basic programming unit	Class is a basic programming unit (ADT)
3	Platform Dependant	Platform independent and portable
4	One step compilation process	Two step process i.e. Compilation and Interpretation
5	Concept of pointer is used	No pointers used in java
6	Source code directly converted to machine code	Source code is converted into byte code then it is interpreted by JVM to machine code
7	Manual management of memory using malloc() and free()	Automatic management of memory using garbage collector
8	Less secure	More secure
9	Provides preprocessors	Doesn't provides preprocessors
10	Does not supports for function overloading	Supports method overloading

3. C++ Vs. Java

	C++ Programming	**Java Programming**
1	Extension of C with object oriented behavior but not complete object oriented	Purely Object Oriented
2	Provides template classes	Does not provide template classes
3	Global variables can be used	Does not support global variables
4	One step compilation process	Two step process i.e. Compilation and Interpretation
5	Concept of pointer is used	No pointers used in java
6	Source code directly converted to machine code	Source code is converted into byte code then it is interpreted by JVM to machine code
7	Manual management of memory using malloc() and free()	Automatic management of memory using garbage collector

8	Less secure	More secure
9	Provides preprocessors	Doesn't provides preprocessors
10	Uses header files	Uses packages and class library
11	Support multiple inheritance	Multiple inheritance is supported using interfaces
12	Focuses on execution efficiency	Focuses on developers productivity
13	powerful capabilities of language	feature-rich, easy to use standard library
14	Uses operator overloading	Does not support operator overloading

D. History of Java:

1. James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
2. Originally designed for small, embedded systems in electronic appliances like set-top boxes.
3. Firstly, it was called "Greentalk" by James Gosling and file extension was .gt.
4. After that, it was called Oak and was developed as a part of the Green project.
5. **Why Oak?** Oak is a symbol of strength and choosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.
6. In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.
7. **Why they choose java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.
8. According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so unique, most of the team members preferred java.
9. Java is an island of Indonesia where first coffee was produced (called java coffee).
10. Notice that Java is just a name not an acronym.
11. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
12. In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.
13. JDK 1.0 released in(January 23, 1996)
14. There are many java versions that has been released. Current stable release of Java is Java SE 8.

E. Types of Java Applications:

1) Standalone Application
It is also known as desktop application or window-based application. AWT and Swing are used in java for creating standalone applications.

2) Web Application
An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) Enterprise Application
An application that is distributed in nature, such as banking applications etc. In java, EJB is used for creating enterprise applications.

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

F. Features of Java:

Simple

According to Sun, Java language is simple because: Syntax is based on C++ (so easier for programmers to learn it after C++). removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.

Object-oriented

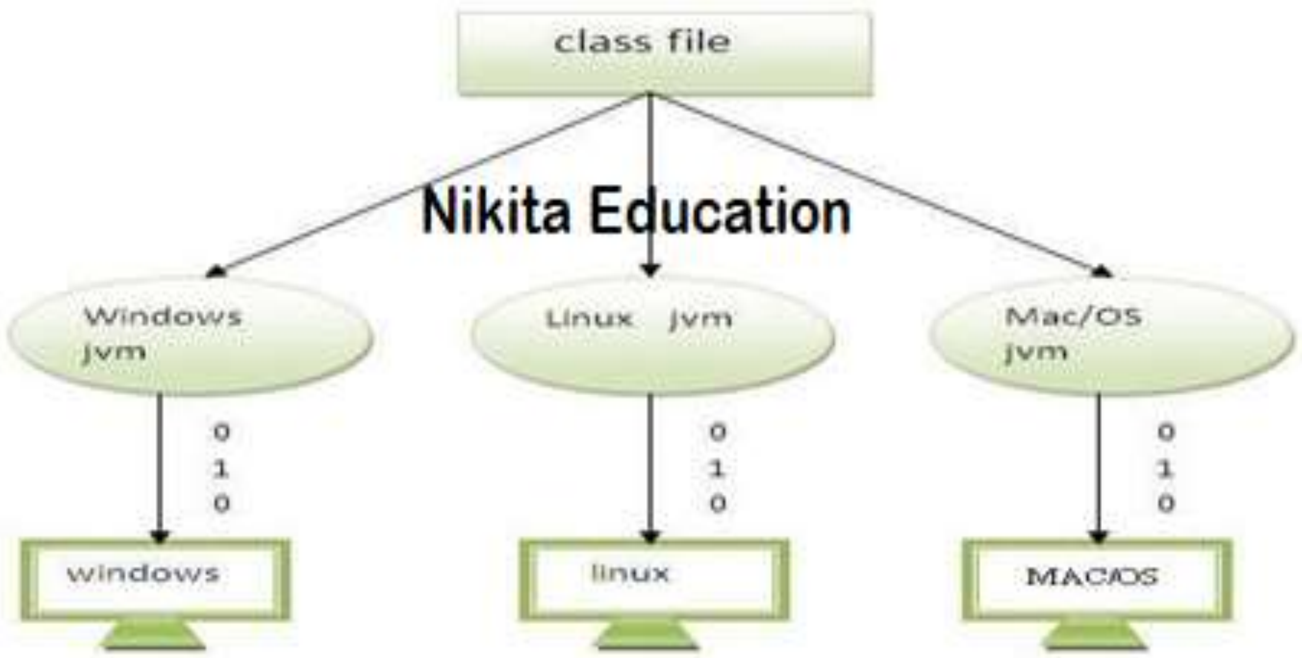
Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior. Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules. Basic concepts of OOPs are: *Object, Class, Polymorphism, Abstraction, Inheritance, Encapsulation*

Platform Independent

A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform. The Java platform differs from most other platforms in the sense that it's a software-based platform that runs on top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

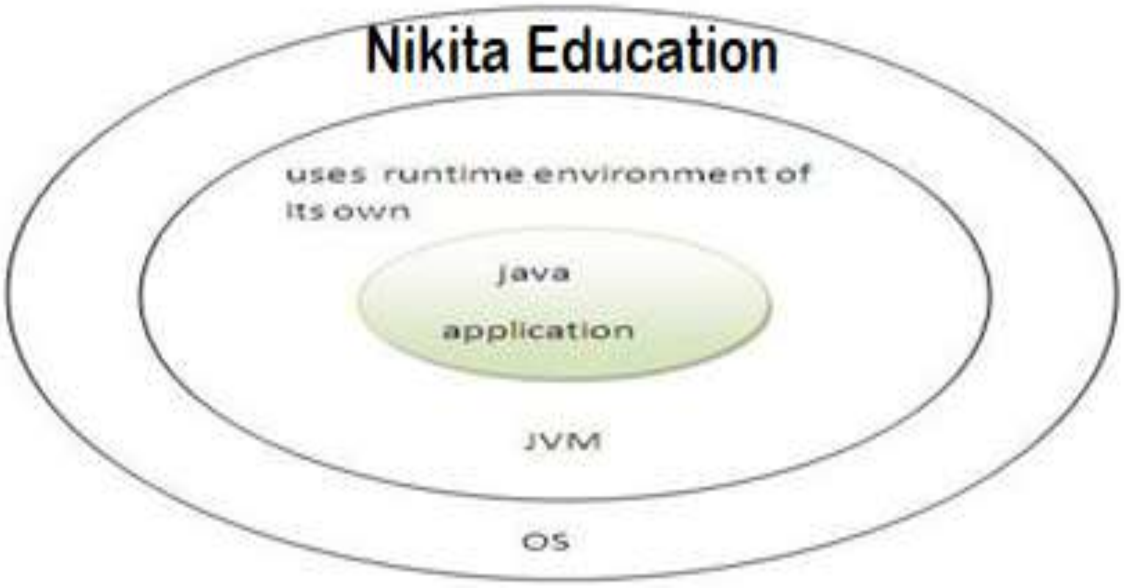


Secured

Java is secured because:

- No explicit pointer
- Programs run inside virtual machine sandbox.

- **Class loader-** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier-** checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager-** determines what resources a class can access such as reading and writing to the local disk.



Robust

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral & Portable

There is no implementation dependent features e.g. size of primitive types is set. We may carry the java bytecode to any platform.

High-performance

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

G. Components of JAVA:

1. JDK (Java Development Kit)

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools. It is used for developing, debugging, testing, and executing java programs. It exist in "C:\Program Files\Java\" folder. It contains utility tools to develop and test java applications.

javac (java compiler)	A java program can be typed using any text editor and save file with **.java** extension. Then **Java Compiler,** reads Java class and interface definitions and compiles them into **bytecode** i.e. it converts **.java** file into **.class** file. **Syntax:** C:\Program Files\Java\jdk1.6.0_23\bin>javac filename.java
java (java interpreter)	The **java** command starts a Java application. It does this by starting the Java Runtime Environment (**JRE**), loading the specified **.class**, and calling that class's **main()** method. It **interprets .class** file into **machine specific code** with the help of **JVM** and execute it**.** **Syntax:** C:\Program Files\Java\jdk1.6.0_23\bin>java filename[.class] (# extension not required)
javap (java disassembler)	Java disassembler is used to convert **bytecode** into **source code.**
javah (java header)	The **javah** command generates **C header** and source files that are needed to implement **native methods**. The generated header and source files are used by C programs to reference an object's instance variables from native code.
jdb (java debugger)	The Java Debugger (JDB) is a simple command-line debugger for Java classes. The jdb command provides inspection and debugging of a local or remote Java Virtual Machine (JVM).
jar (java archive)	The jar command is a general-purpose archiving and compression tool, based on ZIP and the ZLIB compression format. However, the jar command was designed mainly to package Java applets or applications into a single archive.

2. JRE (Java Runtime Environment)

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

a) JVM:

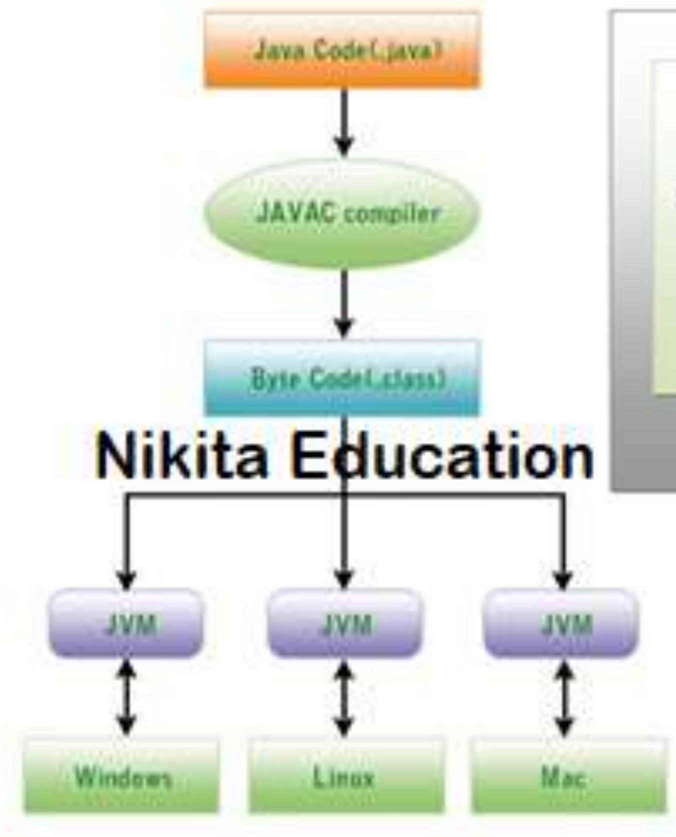
JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java **bytecode** can be executed. JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, **Java Application** is platform independent.

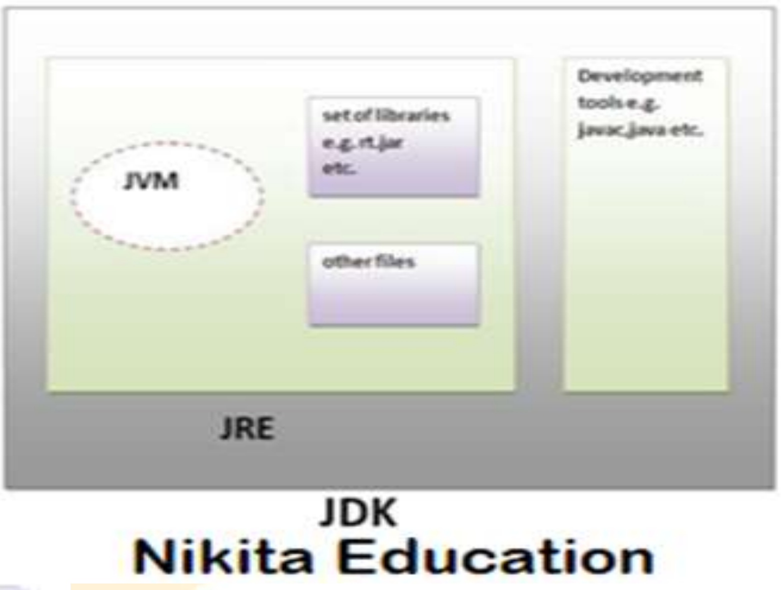
The JVM performs following main tasks:

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

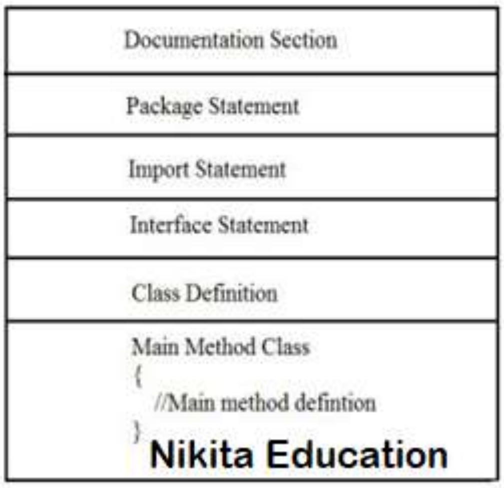
b) Bytecode:

Bytecode is nothing but the intermediate representation of Java source code which is produced by the Java compiler by compiling that source code. This byte code is an machine independent code. It is not an completely a compiled code but it is an intermediate code somewhere in the middle which is later interpreted and executed by JVM. It is stored in .class file which is created after compiling the source code. ***Bytecode files generally have a .class extension.***




JVM
set of libraries e.g. rt.jar etc.
other files
Development tools e.g. javac,java etc.
JRE
JDK

H. Java Program Structure:



Documentation Section	/* suggested: [title, author, dates, descriptions etc.] */
Package Statement	/* optional: [way to group classes into single unit] */
Import Statement	/* optional: [way to include other classes into our class] */
Interface Statement	/* optional: [interface definitions] */
Class Definition	/* optional: other class definitions into single file */
Main Method Class {	/* essential: definition of main class */
//Main method defintion }	/* essential: definition of main method to run class */

I. Simple Java Program:

For executing any java program, you need to

- install the JDK if you don't have installed it, download the JDK and install it.
- set path of the jdk/bin directory. (refer: http://www.javatpoint.com/how-to-set-path-in-java)
- create the java program
- compile and run the java program

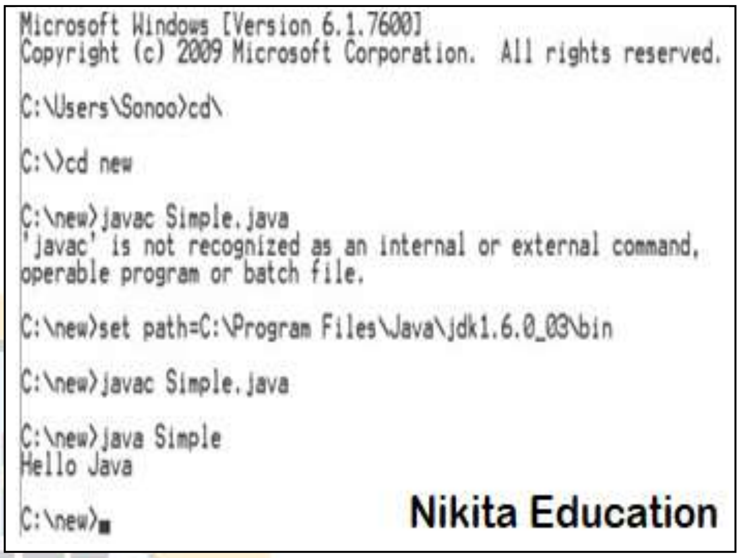
Open any **text editor** and write following program:

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

To compile: javac Simple.java

To execute: java Simple

Output is: Hello Java

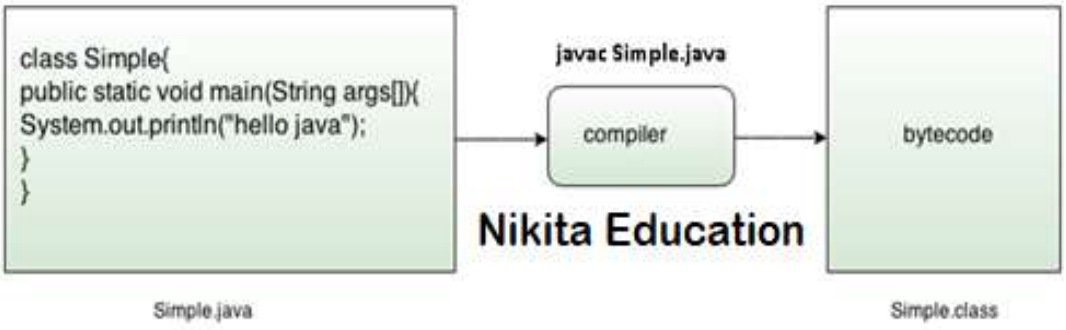


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\new>set path=C:\Program Files\Java\jdk1.6.0_03\bin

C:\new>javac Simple.java

C:\new>java Simple
Hello Java

C:\new>
```

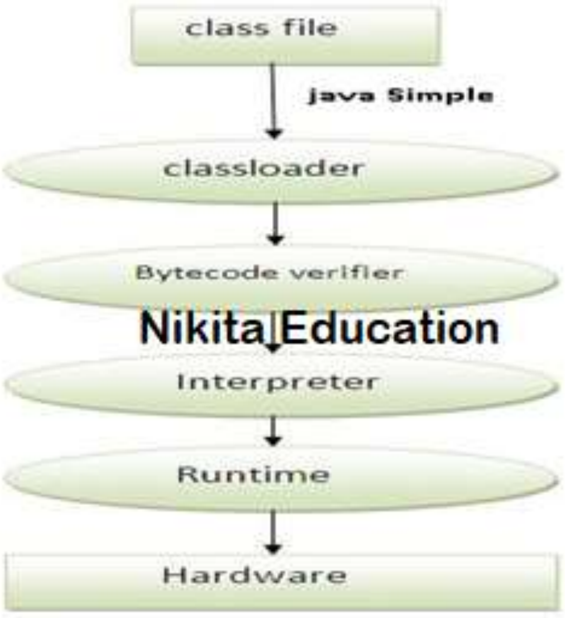
Let's see what is the meaning of **class, public, static, void, main, String[], System.out.println().**

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

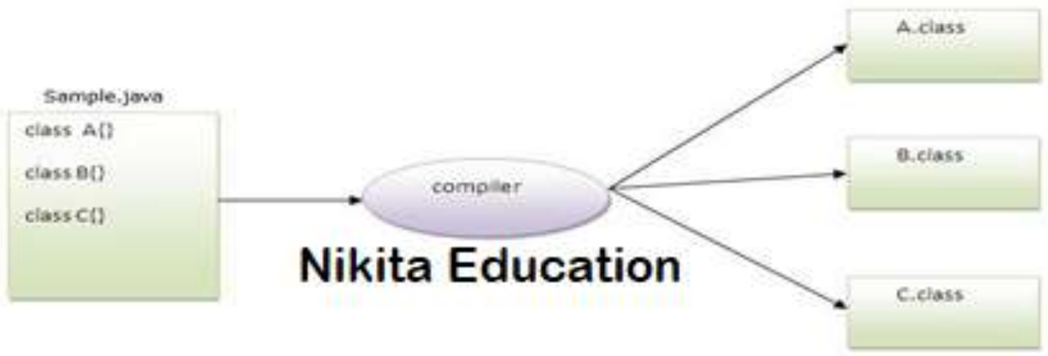
What happens at compile time? [i.e. javac Simple.java]: At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



What happens at runtime? [i.e. java Simple]: At runtime, following steps are performed:

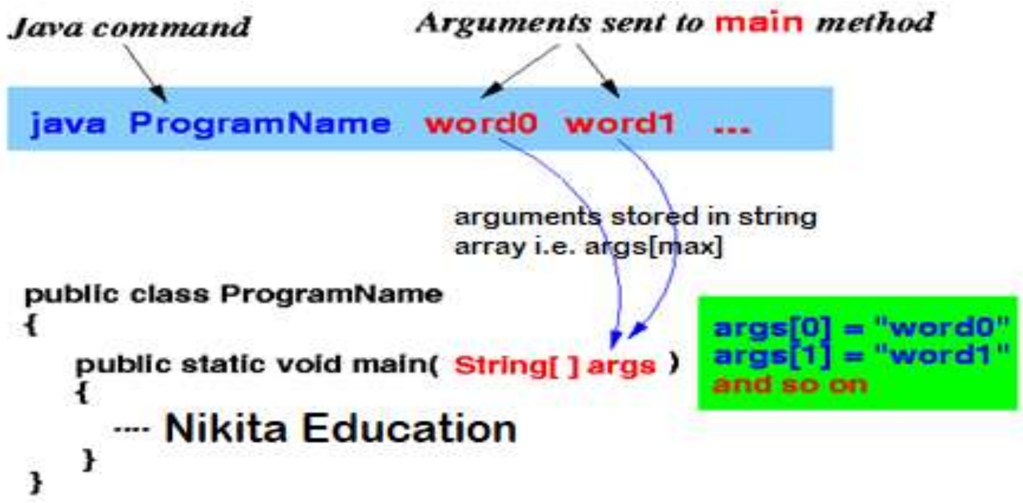


Multiple classes in single java source file: Yes, you can write multiple java classes in single java source file.



J. Command Line Arguments:

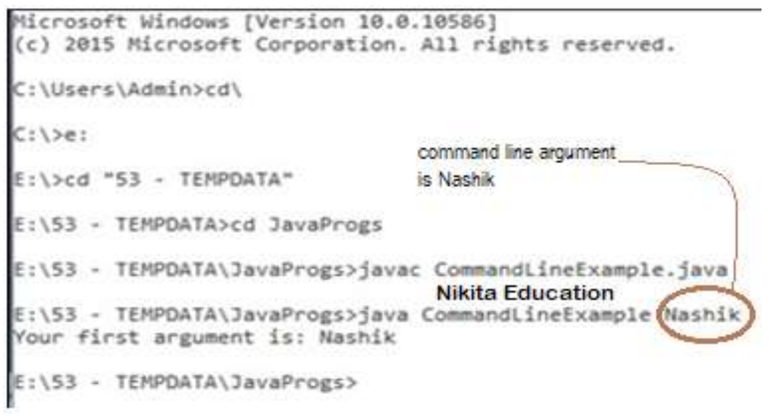
The java command-line argument is an argument i.e. passed at the time of running the java program. The arguments passed from the console (i.e. command prompt) can be received in the java program and it can be used as an input. You can pass N (1,2,3....... and so on) numbers of arguments from the command prompt. **All the arguments passed at command line are stored in *__" args[] "__* of main method in String format.**


Java command
Arguments sent to main method
java ProgramName word0 word1 ...
arguments stored in string array i.e. args[max]
args[0] = "word0"
args[1] = "word1"
and so on
Nikita Education

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

Example:

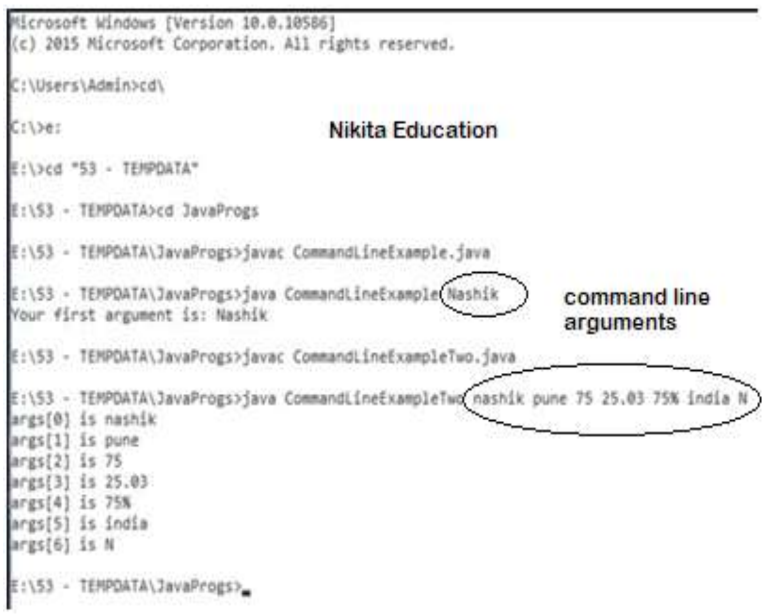
```
class CommandLineExample {
        public static void main(String args[])
       {
            String s=args[0];
            System.out.println("args"+s);
       }
}
```



```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd\

C:\>e:

E:\>cd "53 - TEMPDATA"

E:\53 - TEMPDATA>cd JavaProgs

E:\53 - TEMPDATA\JavaProgs>javac CommandLineExample.java

E:\53 - TEMPDATA\JavaProgs>java CommandLineExample Nashik
Your first argument is: Nashik

E:\53 - TEMPDATA\JavaProgs>
```

Example:

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

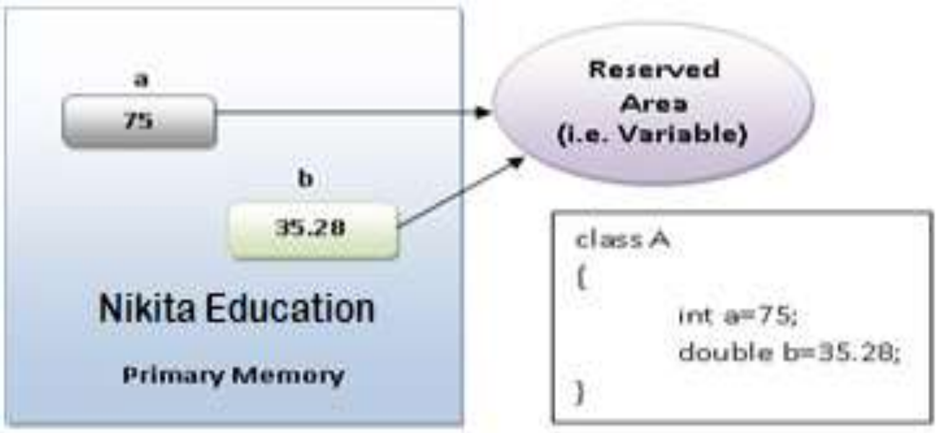


```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd\

C:\>e:

E:\>cd "53 - TEMPDATA"

E:\53 - TEMPDATA>cd JavaProgs

E:\53 - TEMPDATA\JavaProgs>javac CommandLineExample.java

E:\53 - TEMPDATA\JavaProgs>java CommandLineExample Nashik
Your first argument is: Nashik

E:\53 - TEMPDATA\JavaProgs>javac CommandLineExampleTwo.java

E:\53 - TEMPDATA\JavaProgs>java CommandLineExampleTwo nashik pune 75 25.03 75% india N
args[0] is nashik
args[1] is pune
args[2] is 75
args[3] is 25.03
args[4] is 75%
args[5] is india
args[6] is N

E:\53 - TEMPDATA\JavaProgs>
```

K. Variables and Data Types:

1. Variable

Variable is a name of reserved area allocated in memory**.** Variables represent memory area where the information is stored. A variable has a name, a value and a data type.

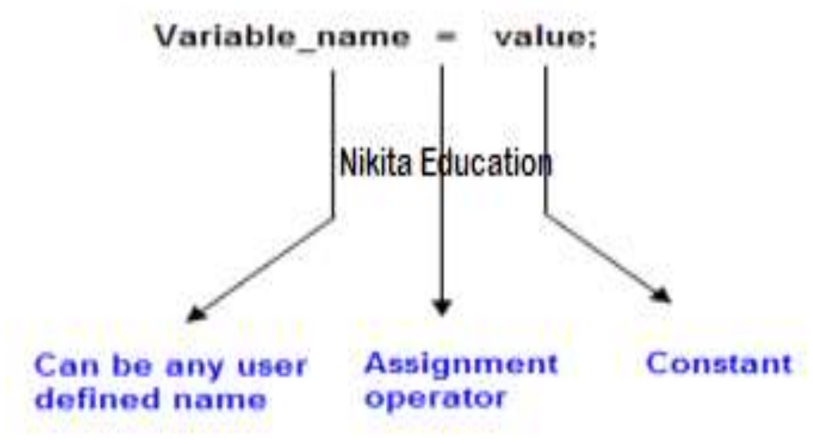


Rules for declaring variables:

1. Variable names are <u>case-sensitive</u>.
2. A variable's name <u>can be any legal identifier</u>.
3. It can contain Unicode <u>letter, Digits and Two Special Characters</u> such as Underscore and dollar Sign.

4. Length of Variable name can be any number.
5. It's necessary to use Alphabet at the start (however we can use underscore , but do not use it)
6. Some auto generated variables may contain '$' sign. But try to avoid using Dollar Sign.
7. White space is not permitted, Special Characters are not allowed.
8. Digit at start is not allowed.
9. Subsequent characters may be letters, digits, dollar signs, or underscore characters.
10. Variable name must not be a keyword or reserved word.

Initialization of variable:



2. Types of variable:

a. Local Variable

A variable that is declared inside the method is called local variable.

b. Instance Variable

A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static. Separate copy of instance variable is assigned to each object hence each object have their own copies of instance variables.

c. Static variable / Class variable

A variable that is declared inside the class and as static is called static variable or class variable. Static variables are shared by all objects of class hence these variables are common variables for all objects.

3. Scope of Variable:

The scope of a variable defines the section of the code in which the variable is visible. As a rule, variables that are defined within a block are not accessible outside that block. The lifetime of a variable refers to how long the variable exists before it is destroyed. Destroying variables refers to de-allocating the memory that was allotted to the variables when declaring it.

a. Local Level Scope

Local variables are created when the method, constructor or block is entered, and the variable will be destroyed once it exits the method, constructor or block. Local variables are visible only within the declared method, constructor or block.

b. Instance Level Scope

Instance variables are declared inside a class, but outside a method, constructor or any block. Instance variables are created when an object is created with the use of 'new' keyword and destroyed when the object is destroyed. The instance variables are visible for all methods, constructors and block in the class.

c. Class Level Scope

Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block. Static variables are created when the program starts and destroyed when the program stops.

Example:

```
class A{
        int data=50;            // instance variable
        static int m=100;       // static variable or class variable
        void method(){
                int n=90;       // local variable
        }//end of method
}//end of class
```

4. Data Types:

A **data type** or simply **type** is a classification identifying one of various types of data, such as real, integer or boolean, that determines the possible values for that type, the operations that can be done on values of that type, the meaning of the data, and the way values of that type can be stored. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

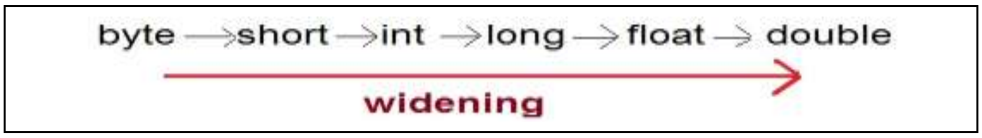
Data Type	Standard Default Value	Default size
boolean	false	1 byte
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

1. Type Casting or Type Conversion

The process of converting one data type to another is called **casting.** Assigning a value of one type to a variable of another type is known as **Type Casting**. In Java, type casting is classified into two types:

- **Widening Casting(Implicit / Automatic)**

  A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as automatic type conversion. Automatic Type casting take place when,
  - the two types are compatible
  - the target type is larger than the source type



Example :

```
int x = 10;                    // occupies 4 bytes
double y = x;                  // occupies 8 bytes
System.out.println(y);         // prints 10.0
```

- **Narrowing Casting(Explicit / Manual)**

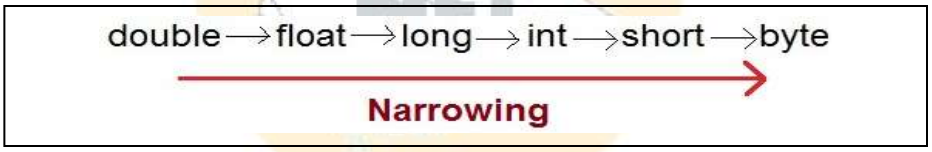
  When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting. A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires **explicit casting**; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

  ```
  double x = 10.5;         // 8 bytes
  int y = x;               // 4 bytes ;  raises compilation error
  ```

  In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.

  ```
  double x = 10.5;
  int y = (int) x;
  ```

  The double **x** is explicitly converted to int **y**. The thumb rule is, on both sides, the same data type should exist.



Example:

```
public class Test{
    public static void main(String[] args)    {
        double d = 100.04;
        long l = (long)d;                    //explicit type casting required
        int i = (int)l;                      //explicit type casting required
        System.out.println("Double value "+d);
        System.out.println("Long value "+l);
        System.out.println("Int value "+i);
    }
}
```

2. Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation. Literals can be assigned to any primitive type variable. For example: **byte a = 68; char a = 'A'**. String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are: **"Hello World", "two \n lines".**

L. Java Operators:

Java provides a rich set of operators environment. Java operators can be divided into following categories:

• Arithmetic operators	• Relation operators	• Logical operators
• Bitwise operators	• Assignment operators	• Conditional operators
• Special purpose operators		

Arithmetic operators

Arithmetic operators are used in mathematical expression in the same way that are used in algebra.

operator	description
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denominator
%	remainder of division
++	Increment operator increases integer value by one
--	Decrement operator decreases integer value by one

Relational operators

operator	description
= =	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Logical operators

Java supports following 3 logical operator. Suppose a=1 and b=0;

operator	description	example
&&	Logical AND	(a && b) is false
||	Logical OR	(a || b) is true
!	Logical NOT	(!a) is false

Bitwise operators

Java defines several bitwise operators that can be applied to the integer types long, int, short, char and byte

operator	description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift

Now let's see truth table for bitwise &, | and ^

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Assignment Operators

operator	description	example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

Conditional operator

It is also known as ternary operator and used to evaluate Boolean expression
epr1 ? expr2 : expr3
If **epr1**Condition is true? Then value **expr2** : Otherwise value **expr3**

instanceOf operator (Special Purpose Operator)

This operator is used for object reference variables. The operator checks whether the object is of particular type (class type or interface type)

Precedence of Java Operators (Priority):

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left

M. Decision Making & Branching Statements:

1. Java - Decision Making

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Statement	Description
if	An if statement consists of a boolean expression followed by one or more statements.
if...else	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
nested if	You can use one if or else if statement inside another if or else if statement(s).
switch	A switch statement allows a variable to be tested for equality against a list of values.

1. IF

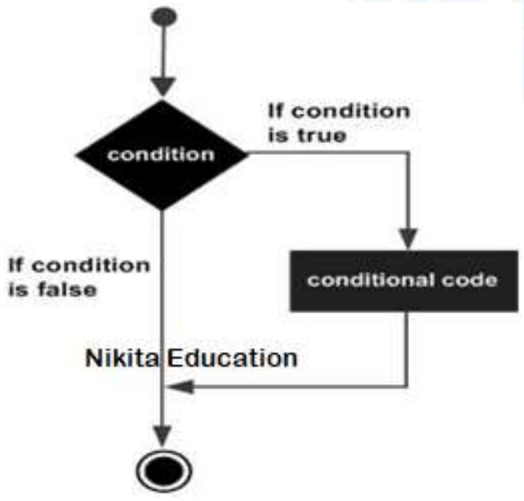
An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax:

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

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example:

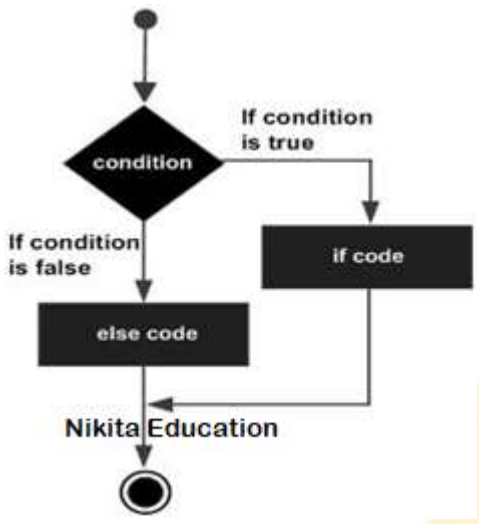
```
public class Test {
        public static void main(String args[]){
                int x = 10;
                if( x < 20 ){
                        System.out.print("This is if statement");
                }
        }
}
```

2. IF....ELSE

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax:
```
if(Boolean_expression){
        //Executes when the Boolean expression is true
}else{
        //Executes when the Boolean expression is false
}
```
If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



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

3. ELSE IF Ladder:

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement. When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:
```
if(Boolean_expression 1){
        //expression 1 is true
}else if(Boolean_expression 2){
        //expression 2 is true
}else if(Boolean_expression 3){
        //expression 3 is true
}else {
        //none of the above true.
}
```

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

4. Nested IF

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax:

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

You can nest **else if...else** in the similar way as we have nested *if* statement.

Example:

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

5. SWITCH

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

Syntax:

```
switch(expression){
        case value :
                        //Statements
        break;          //optional
        case value :
                        //Statements
        break;          //You can have any number of case statements.
        default :       //Optional
}
```

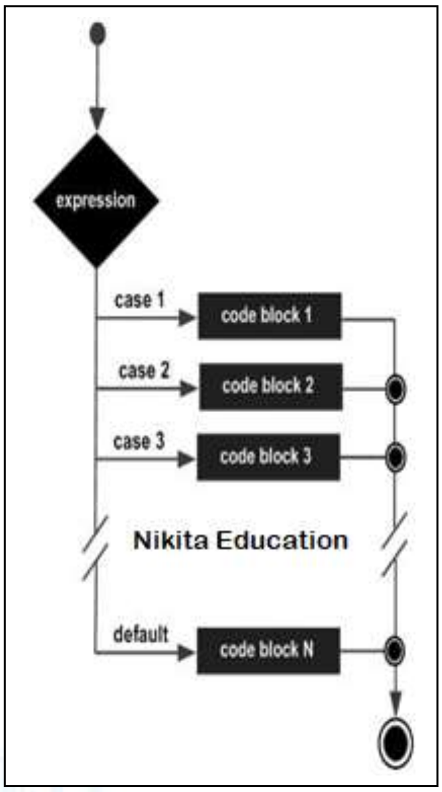
The following rules apply to a **switch** statement:

- The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.

- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {
        public static void main(String args[]){
                //char grade = args[0].charAt(0);
                char grade = 'C';
                switch(grade){
                        case 'A' :
                                System.out.println("Excellent!");
                                break;
                        case 'B' :
                        case 'C' :
                                System.out.println("Well done");
                                break;
                        case 'D' :
                                System.out.println("You passed");
                        break;
                        default :
                                System.out.println("Invalid grade");
                }
                System.out.println("Your grade is " + grade);
        }
}
```



Compile and run above program using various command line arguments.

6. The ? : Operator: (Conditional Operator)

We have covered **conditional operator ? :** in previous chapter which can be used to replace **if...else** statements. It has the following general form:

Exp1 ? Exp2 : Exp3;

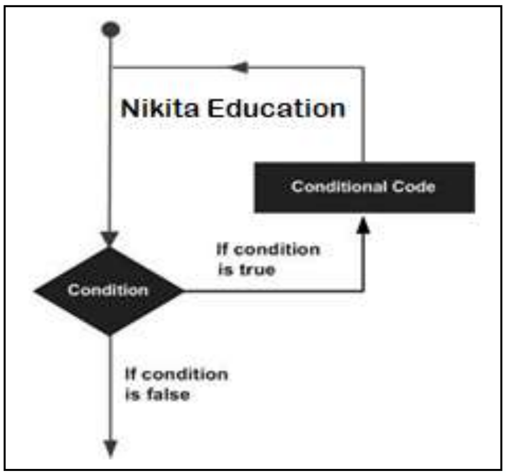
Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

To determine the value of whole expression, initially exp1 is evaluated

- If the value of exp1 is true, then the value of Exp2 will be the value of the whole expression.
- If the value of exp1 is false, then Exp3 is evaluated and its value becomes the value of the entire expression.

N. Loop Statements

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. A **loop** statement allows us to execute a statement or group of statements multiple times.



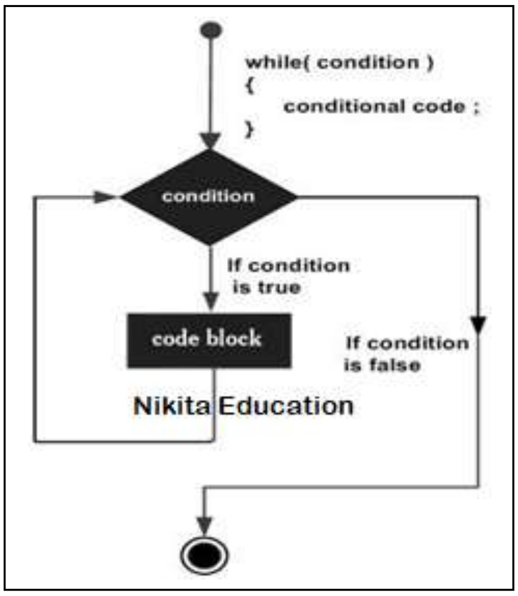
Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body

1. WHILE

A **while** loop statement in java programming language repeatedly executes a target statement as long as a given condition is true.

Syntax: **while(Boolean_expression){**
//Statements
}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value. When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true. When the condition becomes false, program control passes to the line immediately following the loop. Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.



Example:

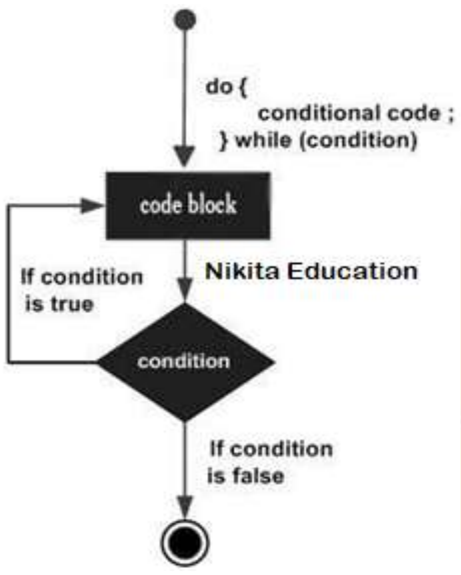
```
public class Test {
   public static void main(String args[]) {
      int x = 10;
      while( x < 20 ) {
         System.out.println("value of x : " + x );
         x++;
      }
   }
}
```

2. DO WHILE

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

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

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested. If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.



Example:

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

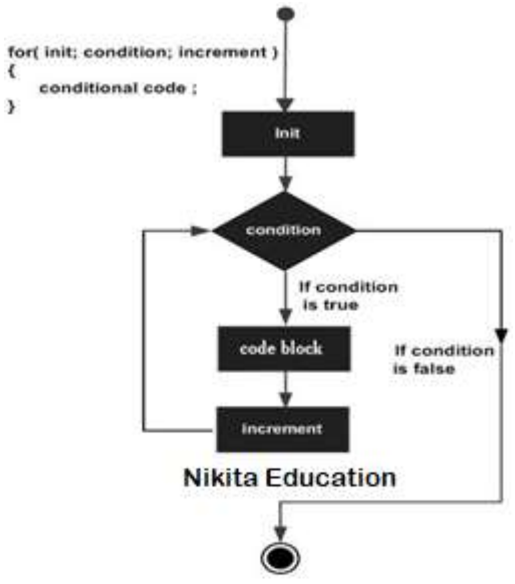
3. FOR

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. A for loop is useful when you know how many times a task is to be repeated.

Syntax:
```
for(initialization; Boolean_expression; update)
{
        //Statements
}
```

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. and this step ends with a semi colon (;)
- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
- After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Flow Diagram



Example:

Below given is the example code of the for loop in java

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

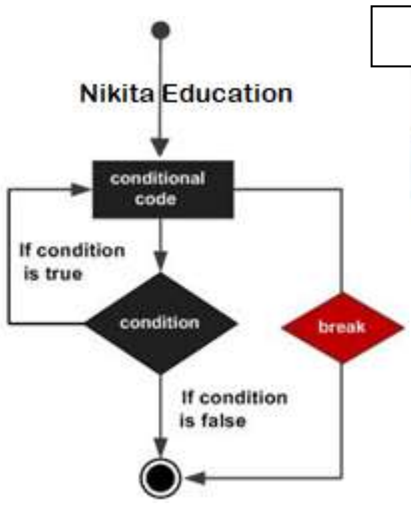
A. Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Java supports the following control statements.

1. BREAK

The **break** statement in Java programming language has the following two usages:

1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the **switch** statement.



```
Syntax:          break;
```

Example:

```
public class Test {
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};
      for(int x : numbers ) {
         if( x == 30 ) {
            break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

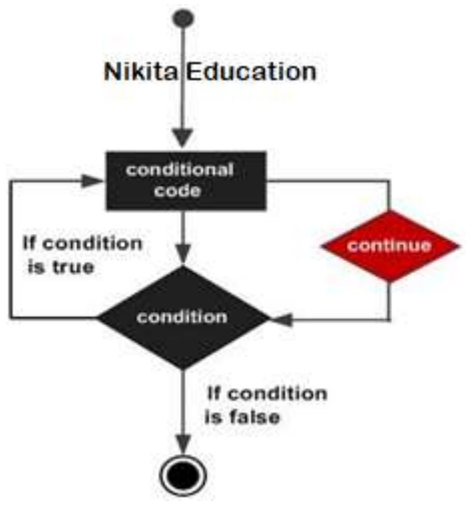
2. CONTINUE

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

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

```
Syntax:          continue;
```

Flow Diagram



Example:

```
public class Test {
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};
      for(int x : numbers ) {
         if( x == 30 ) {
            continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

B. Enhanced for loop in Java:

Syntax:

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

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

Example:

```
public class Test {
      public static void main(String args[]){
            int [] numbers = {10, 20, 30, 40, 50};
            for(int x : numbers ){
                  System.out.print( x );
                  System.out.print(",");
            }
      }
}
```

Output:

```
10,20,30,40,50,
```

O. Math Class Functions

The **java.lang.Math** class contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. It contains various **static** methods to perform numeric operations. Following are some important methods of **Math** class.

static double abs(double a) This method returns the absolute value of a double value.
static float abs(float a) This method returns the absolute value of a float value.
static int abs(int a) This method returns the absolute value of an int value.
static long abs(long a) This method returns the absolute value of a long value.

Example

```
        public class MathDemo {

        public static void main(String[] args) {

         // get some integers to find their absolute values
         int x = 175;
         int y = -184;

         // get and print their absolute values
         System.out.println("Math.abs(" + x + ")=" + Math.abs(x));
         System.out.println("Math.abs(" + y + ")=" + Math.abs(y));
         System.out.println("Math.abs(-0)=" + Math.abs(-0));
      }
   }
```

Output:

```
        Math.abs(175)=175
        Math.abs(-184)=184
        Math.abs(-0)=0
```

static double exp(double a) This method returns Euler's number e raised to the power of a double value.

```
        public class MathDemo {

        public static void main(String[] args) {

         // get two double numbers
         double x = 5;
         double y = 0.5;

         // print e raised at x and y
         System.out.println("Math.exp(" + x + ")=" + Math.exp(x));
         System.out.println("Math.exp(" + y + ")=" + Math.exp(y));
      }
   }
   Math.exp(5)=148.4131591025766
   Math.exp(0.5)=1.6487212707001282
```

static double max(double a, double b)	This method returns the greater of two double values.
static float max(float a, float b)	This method returns the greater of two float values.
static int max(int a, int b)	This method returns the greater of two int values.
static long max(long a, long b)	This method returns the greater of two long values.
static double min(double a, double b)	This method returns the smaller of two double values.
static float min(float a, float b)	This method returns the smaller of two float values.
static int min(int a, int b)	This method returns the smaller of two int values.
static long min(long a, long b)	This method returns the smaller of two long values.

```
public class MathDemo {
public static void main(String[] args) {
     float x = 60984.1f;
     float y = 1000f;
     System.out.println("Math.max(" + x + "," + y + ")="
                                          + Math.max(x, y));
     double x = 9875.875;
     double y = 154.134;
     System.out.println("Math.min(" + x + "," + y + ")="
                                          + Math.min(x, y));
}}
Output: Math.max(60984.1f, 1000f)=60984.1
         Math.min(9875.875, 154.134)=154.134
```

static double pow(double a, double b)	This method returns the value of the first argument raised to the power of the second argument.
static double random()	This method returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
static long round(double a)	This method returns the closest long to the argument.
static int round(float a)	This method returns the closest int to the argument.
static double sqrt(double a)	This method returns the correctly rounded positive square root of a double value.
static double pow(double a, double b)	This method returns the value of the first argument raised to the power of the second argument.

Example:

```
      class MathFunctions
      {
           public static void main(String[] args)
           {
                int i1,i2;
                double d1;
                double a1,b1;
                double e1;
                float f1;
                float f2;

                i1=35;
                i2=58;
                int minimum=Math.min(i1,i2);
                int maximum=Math.max(i1,i2);

                d1=64.00;
                double squareroot=Math.sqrt(d1);

                a1=2;
                b1=4;
                double power=Math.pow(a1,b1);

                e1=175;
                double exponential=Math.exp(e1);

                f1=75.20f;
                int roundval=Math.round(f1);

                f2=-50.26f;
                float absval=Math.abs(f2);

                System.out.println("min() = "+minimum);
                System.out.println("max() = "+maximum);
                System.out.println("sqrt() = "+squareroot);
                System.out.println("pow() = "+power);
                System.out.println("exp() = "+exponential);
                System.out.println("round() = "+roundval);
                System.out.println("abs() = "+absval);
           }
      }
```

Unit-II
Classes & Objects

Unit	Topic Name	Main Topic	Subtopics	Marks
2	Classes, Objects, and Methods	Classes & Objects	class Fundamentals, Declaring and Creating object, Accessing class members and methods, Constructor, Methods Overloading, Static Member	12
		Inheritance	Defining a subclass Constructor, Multilevel inheritance, Hybrid Inheritance, Hierarchical inheritance, Overriding Methods, Final variable and Methods, Final Classes, Abstract method and Classes	
		Visibility Control	Public access, default access, Protected access, Private access.	
		Array & String	Arrays, One Dimensional array, Creating an array, Two Dimensional array, String & StringBuffer	
		Vector & Wrapper Classes	Vector class, Wrapper Classes	

A. Class Fundamental

1. Class in Java

In Java everything is encapsulated under classes. Class is the core of Java language. Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. Once defined this new type can be used to create object of that type. A class is declared using **class** keyword. A class contain both data and code that operate on that data. A class is a group of objects that has common properties. A class in java can contain: *data member, method, constructor, block, class and interface.*

2. Syntax:

```
class    <class_name>
{
        Data members;
        Member method;
        Main method;
}
```

```
public class Dog{
        int ageC;                  //data members
        String color;

        void hungry(){ }           //member methods
        void sleeping(){ }
        public static void main(String args[]){
                //main method
        }
}
```

3. Rules for Java Class

- A class can have only public or default(no modifier) access specifier.
- It can be either abstract, final or concrete (normal class).
- It must have the class keyword, and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces {}.
- Each **.java** source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

B. Object Fundamental

1. Object in Java

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID.

Object is an instance of a class. Class is a template or blueprint from which objects are created.

Note: Object is the physical as well as logical entity whereas class is the logical entity only.

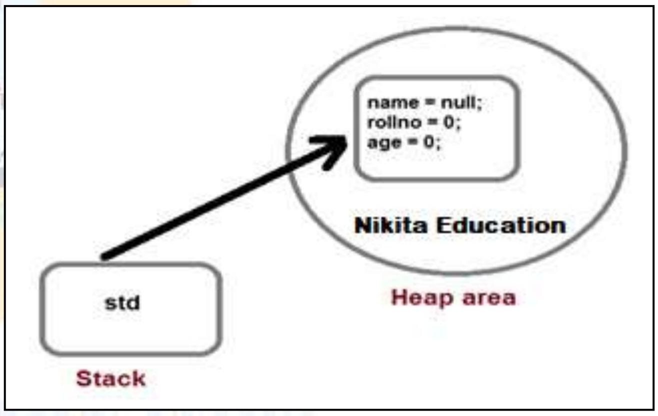
2. Creating an Object:

As mentioned previously, a class provides the blueprints for objects. So basically an object is created from a class. In Java, the new keyword is used to create new objects. There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' keyword is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

3. A simple class example

```
class Student.
{
        String name;
        int rollno;
        int age;
}
```



4. Object Example:

Student std=new Student();

Note: The new operator dynamically allocates memory for an object.

5. Accessing Instance Variables and Methods:

```
/* First create an object */
ClassName ObjectReference = new ClassConstructor();
```

```
/* Now call a variable as follows */
ObjectReference.variableName;
```

```
/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

6. Simple Example of Object and Class

Example 1:

```
class Student{
        int id;                 //data member (also instance variable)
        String name;            //data member (also instance variable)
        public static void main(String args[]){
                Student s1=new Student();
                //creating an object of Student
```

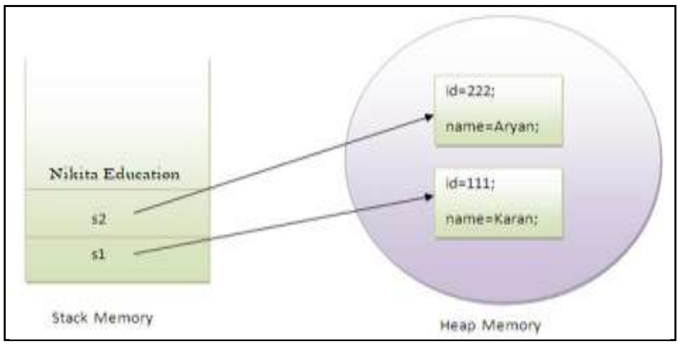
```
                        System.out.println(s1.id);
                        System.out.println(s1.name);
                }
        }
```

Example 2: public class Puppy{

```
        int puppyAge;
        public Puppy(String name){                      //constructor
                System.out.println("Name chosen is :" + name );
        }
        public void setAge( int age ){          //method
                puppyAge = age;
        }
        public int getAge( ){                   //method
                System.out.println("Puppy's age is :" + puppyAge );
                return puppyAge;
        }
        public static void main(String []args){
                Puppy myPuppy = new Puppy( "tommy" );
                myPuppy.setAge( 2 );
                int age=myPuppy.getAge( );
                System.out.println("Age is :" +age );
                System.out.println("Variable Value :" + myPuppy.puppyAge );
        }
}
```

Example 3: class Student2{

```
        int rollno;
        String name;
        void insertRecord(int r, String n){  //method
                rollno=r;
                name=n;
        }
        void displayInformation(){
                System.out.println(rollno+" "+name);
        }
public static void main(String args[]){
        Student2 s1=new Student2();
        Student2 s2=new Student2();
        s1.insertRecord(111,"Karan");
        s2.insertRecord(222,"Aryan");
        s1.displayInformation();
        s2.displayInformation();
}
}
```



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

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

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

8. Array of Objects

An array of objects means arrays are capable of storing objects.

For Example: **class Student { int marks; }**

An array of objects is created just like an array of primitive type data items in the following way.

```
Student[] studentArray = new Student[7];
```

The Student objects have to be instantiated using the constructor of the Student class and their references should be assigned to the array elements in the following way.

```
for ( int i=0; i<studentArray.length; i++) {
        studentArray[i]=new Student();
}
```

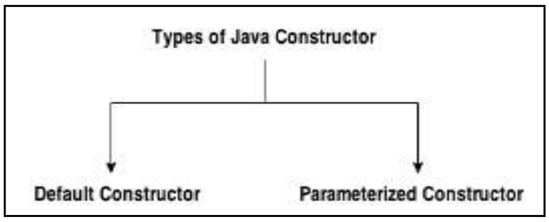
The above for loop creates seven Student objects and assigns their reference to the array elements.

Example:
```
class Person {
        private String lastName;
        private String firstName;
        private int age;
        public Person(String last, String first, int a) {
                lastName = last;
                firstName = first;
                age = a;
        }
        public String toString() {
                return "Last name: " + lastName + " First name: " + firstName +
                        " Age: " + age;
        }
}
public class MainClass {
        public static void main(String[] args) {
                Person[] persons = new Person[10];
                for(int i=0;i<persons.length;i++){
                        persons[i] = new Person("Patil","Sushmita",5);
                }
                for(Person p: persons){
                        System.out.println(p.toString());

                }
        }
}
```

C. Constructors

A constructor is a special method that is used to initialize an object. Every class has a constructor, if we don't explicitly declare a constructor for any java class the compiler builds a default constructor for that class. A constructor does not have any return type. A constructor has same name as the class in which it resides. Constructor in Java cannot be abstract, static, final or synchronized. These modifiers are not allowed for constructor. Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor:



There are basically two rules defined for the constructor.

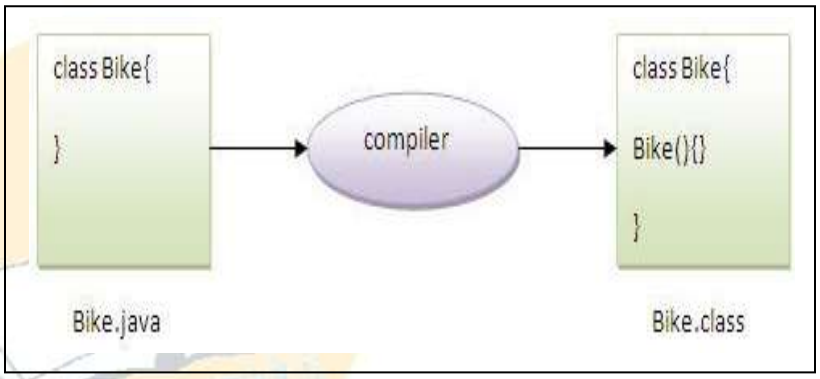
1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

1. Default Constructor

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

Syntax of default constructor:

```
<class_name>(){
        //initialization
}
```



Example:

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

Rule: If there is no constructor in a class, compiler automatically creates a default constructor. Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example:

```
class Student3{
        int id;
        String name;
        void display(){
                System.out.println(id+" "+name);
        }
        public static void main(String args[]){
                Student3 s1=new Student3();
                Student3 s2=new Student3();
                s1.display();
                s2.display();
        }
}
```

2. Parameterized Constructor

A constructor that have parameters is known as parameterized constructor. **Why use parameterized constructor?** Parameterized constructor is used to provide different values to the distinct objects.

Example:
```
class Student4{
        int id;
        String name;
                Student4(int i,String n){
                id = i;
                name = n;
        }
        void display(){
                System.out.println(id+" "+name);
        }
                public static void main(String args[]){
                Student4 s1 = new Student4(111,"Karan");
                Student4 s2 = new Student4(222,"Aryan");
                s1.display();
                s2.display();
        }
}
```

3. Constructor Overloading

Like methods, a constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters. Constructor overloading is not much different than method overloading. In case of method overloading you have multiple methods with same name but different signature, whereas in Constructor overloading you have multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java. Constructor overloading is done to construct object in different ways.

Example:
```
class Student5{
        int id;
        String name;
        int age;
        Student5(int i,String n){
                id = i;
                name = n;
        }
        Student5(int i,String n,int a){
                id = i;
                name = n;
                age=a;
        }
        void display(){System.out.println(id+" "+name+" "+age);}
                public static void main(String args[]){
                Student5 s1 = new Student5(111,"Karan");
                Student5 s2 = new Student5(222,"Aryan",25);
                s1.display();
                s2.display();
        }
}
```

Difference between constructor and method in java

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

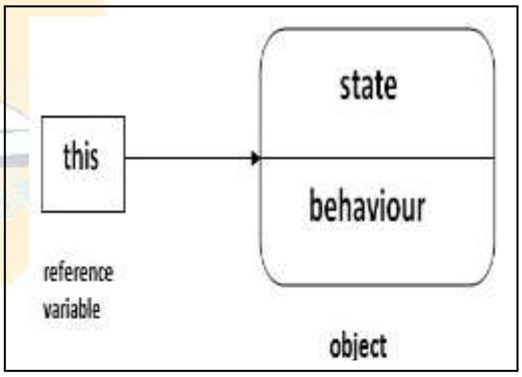
Java Constructor	**Java Method**
Constructor is used to initialize the state of an object.	Method is used to expose behavior of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a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

4. this keyword

- this keyword is used to refer to current object.
- this is always a reference to the object on which method was invoked.
- this can be used to invoke current class constructor.
- this can be passed as an argument to another method.

Example **:**

```
class Box {
Double width, weight, dept;

Box (double w, double h, double d){

        this.width = w;
        this.height = h;
        this.depth = d;

}
}
```



a. Usage of java this keyword

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

b. Understanding the problem without this keyword

```
class Student10{
        int id;
        String name;
        Student10(int id,String name){
                id = id;
                name = name;
        }

        void display(){
                System.out.println(id+" "+name);
        }
        public static void main(String args[]){
                Student10 s1 = new Student10(111,"Karan");
                s1.display();
                Student10 s2 = new Student10(321,"Aryan");
                s2.display();
        }
    }
```

Output:
```
0 null
0 null
```

c. Solution of the above problem by this keyword

```
class Student11{
    int id;
    String name;
        Student11(int id,String name){
        this.id = id;
        this.name = name;
    }

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

    public static void main(String args[]){
        Student11 s1 = new Student11(111,"Karan");
        Student11 s2 = new Student11(222,"Aryan");
        s1.display();
        s2.display();
    }
}
```

Output:
```
111 Karan
222 Aryan
```

d. The this is used to call overloaded constructor in java

```
    class Car{
        private String name;
        public Car(){
            this("BMW");    //oveloaded constructor is called.
        }
        public Car(String n){
            this.name=n;   //member is initialized using this.
        }
    }
```

e. The this is also used to call Method of that class.

```
    public void getName(){
        System.out.println("Studytonight");
    }
    public void display(){
        this.getName();
        System.out.println();
    }
```

f. this is used to return current Object

```
public Car getCar(){
        return this;
}
```

D. Garbage Collection

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

Advantage of Garbage Collection

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

How can an object be unreferenced?

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

By nulling a reference:
Employee e=new Employee();
e=null;
	By assigning a reference to another:
	Employee e1=new Employee();
	Employee e2=new Employee();
	e1=e2;
		By anonymous object:
		new Employee();

1. The finalize() method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation **finalize()** method is used. **finalize()** method is called by garbage collection thread before collecting object. It's the last chance for any object to perform cleanup utility. The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class.

Syntax: **protected void finalize(){** //finalize-code **}**

2. Simple Example of garbage collection in java

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

Output:

```
object is garbage collected
object is garbage collected
```

3. Some Important Points to Remember

1. finalize() method is defined in **java.lang.Object** class, therefore it is available to all the classes.
2. finalize() method is declare as **proctected** inside Object class.
3. finalize() method gets called only once by GC threads.

4. gc() Method

gc() method is used to call garbage collector explicitly. However **gc()** method does not guarantee that JVM will perform the garbage collection. It only request the JVM for garbage collection. This method is present in **System** and **Runtime** class. **Example:** **System.gc();**

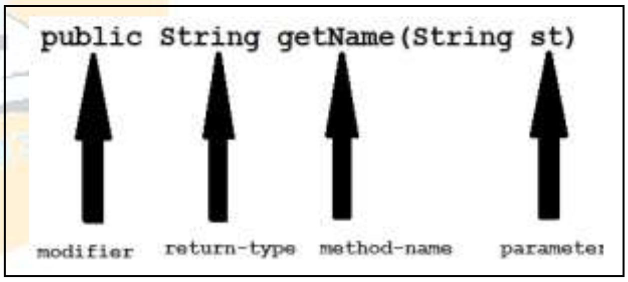
E. Method Overloading

1. Java - Methods

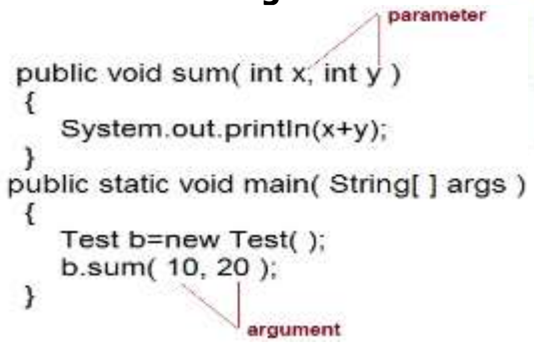
A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.**println()** method, for example, the system actually executes several statements in order to display a message on the console.

Syntax : **return-type methodName(parameter-list){** //body of method
}

Example:
```
public String getName(String st){
        String name="StudyTonight";
        name=name+st;
        return name;
}
```



a. Parameter Vs. Argument


```
public void sum( int x, int y )
 {
    System.out.println(x+y);
 }
public static void main( String[ ] args )
 {
    Test b=new Test( );
    b.sum( 10, 20 );
 }
```

b. call-by-value and call-by-reference

There are two ways to pass an argument to a method

1. **call-by-value :** In this approach copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.
2. **call-by-reference :** In this reference of an argument is pass to a method. Any changes made inside the method will affect the argument value.

NOTE : In Java, when you pass a primitive type to a method it is passed by value whereas when you pass an object of any type to a method it is passed as reference.

Example of call-by-value:

```
public class Test{
public void callByValue(int x){  x=100;  }
public static void main(String[] args){
        int x=50;
        Test t = new Test();
        t.callByValue(x);  //function call
        System.out.println(x);
}
}
```

```
Output :   50
```

Example of call-by-reference:

```
public class Test{
        int x=10, y=20;

        public void callByReference(Test t){
                t.x=100;
                t.y=50;
        }
        public static void main(String[] args){
                Test ts = new Test();
                System.out.println("Before "+ts.x+" "+ts.y);
                ts.callByReference(ts);
                System.out.println("After "+ts.x+" "+ts.y);
        }
}
```

```
Output :
Before 10 20
After   100 50
```

2. Method Overloading in Java

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**. If we have to perform only one operation, having same name of the methods increases the readability of the program.

- **Advantage of method overloading?**
  Method overloading **increases the readability of the program**.

- **Different ways to overload the method**
  1. By changing number of arguments
  2. By changing the data type

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

a. Changing data type of arguments

```
class Calculate{

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

    void sum (float a, float b){
            System.out.println("sum is"+(a+b));
    }
    public static void main (String[] args){
            Calculate  cal = new Calculate();
```

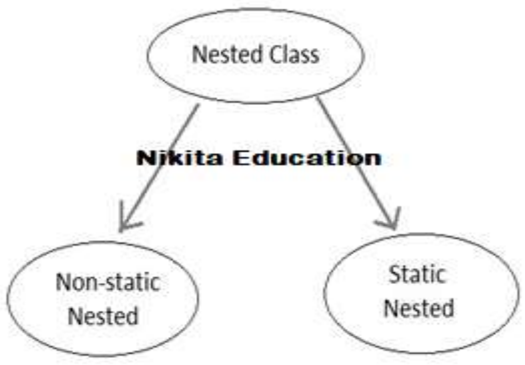
```
                cal.sum (8,5);       //sum(int a, int b) is method is called.
                cal.sum (4.6, 3.8);  //sum(float a, float b) is called.
        }
    }
```

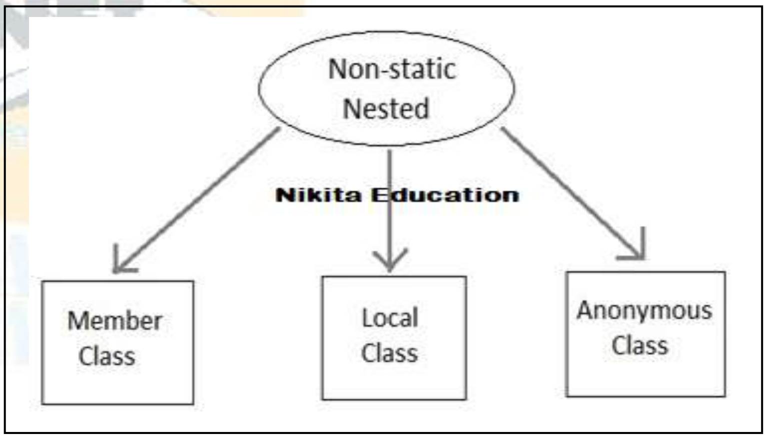
b. Changing no. of argument.

```
    class Area{
        void find(int l, int b){
                System.out.println("Area is"+(l*b)) ;
        }
        void find(int l, int b,int h){
                System.out.println("Area is"+(l*b*h));
        }
        public static void main (String[] args){
                Area  ar = new Area();
                ar.find(8,5);          //find(int l, int b) is method is called.
                ar.find(4,6,2);        //find(int l, int b,int h) is called.
        }
    }
```

F. Nested Classes

A class within another class is known as Nested class. The scope of the nested is bounded by the scope of its enclosing class.





a. Static Nested Class

A static nested class is the one that has **static** modifier applied. Because it is static it cannot refer to non-static members of its enclosing class directly.

b. Non-static Nested classes

Non-static Nested class is most important type of nested class. It is also known as **Inner** class. It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is, **Outer** class cannot directly access members of **Inner** class. One more important thing to notice about an **Inner** class is that it can be created only within the scope of **Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner** class.

i. Example of Inner class:

```
class Outer{
        public void display(){
                Inner in=new Inner();
                in.show();
        }

        class Inner{

                public void show(){
                        System.out.println("Inside inner");
                }

        }

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

ii. Example of Inner class inside a method:

```
class Outer{
        int count;

        public void display(){

                for(int i=0;i<5;i++){

                        class Inner{      //Inner class defined inside for loop

                                public void show(){
                                        System.out.println("Inside inner "+(count++));
                                }

                        }

                        Inner in=new Inner();
                        in.show();
                }
        }
        public static void main(String[] args){
                Outer ot=new Outer();
                ot.display();
        }
}
```

Output :

```
Inside inner 0
Inside inner 1
Inside inner 2
Inside inner 3
Inside inner 4
```

iii. Example of Inner class instantiated outside Outer class

```
class Outer{

        int count;
        public void display(){
                Inner in=new Inner();
                in.show();
        }

        class Inner{
                public void show(){
                        System.out.println("Inside inner "+(++count));
                }
        }
}
class Test{
        public static void main(String[] args){
                Outer ot=new Outer();
                Outer.Inner in= ot.new Inner();
                in.show();
        }
}
```

Output :

```
Inside inner 1
```

iv. **Anonymous class:** A class without any name is called Anonymous class.

```
interface Animal{
        void type();
}

public class ATest {

        public static void main(String args[]){

                Animal an = new Animal(){        //Annonymous class created
                        public void type(){
                                System.out.println("Annonymous animal");
                        }
                };

                an.type();
        }
}
```

Output :

```
Annonymous animal
```

G. Static Members

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

1. Java static variable

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

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

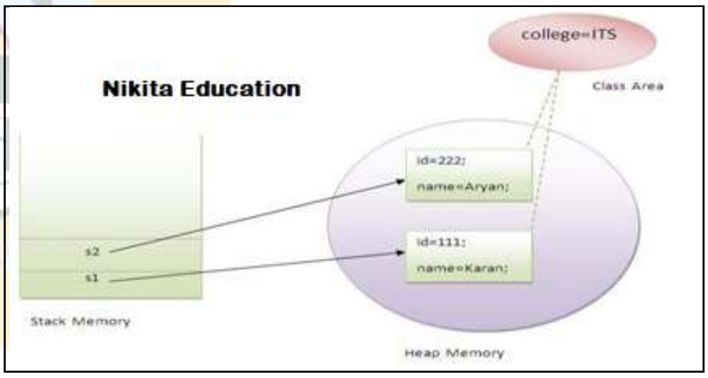
Advantage of static variable
It makes your program **memory efficient** (i.e it saves memory).

Understanding problem without static variable

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

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

Example:



```
class Student8{
        int rollno;
        String name;
        static String college ="ITS";
        Student8(int r,String n){
                rollno = r;
                name = n;
        }

        void display (){
                System.out.println(rollno+" "+name+" "+college);
        }

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

Output:

```
111 Karan ITS
222 Aryan ITS
```

Program of counter without static variable

```
    class Counter{

        int count=0;        //will get memory when instance is created
        Counter(){
                count++;
                System.out.println(count);
        }
        public static void main(String args[]){
                Counter c1=new Counter();
                Counter c2=new Counter();
                Counter c3=new Counter();
        }
}
```

```
Output:
        1
        1
        1
```

Program of counter by static variable

```
class Counter2{

    static int count=0;        //will get memory only once and retain its value

    Counter2(){
        count++;
        System.out.println(count);
    }

    public static void main(String args[]){
        Counter2 c1=new Counter2();
        Counter2 c2=new Counter2();
        Counter2 c3=new Counter2();
    }
}
```

```
Output:
        1
        2
        3
```

Static variable vs. Instance Variable

Static variable	**Instance Variable**
Represent common property	Represent unique property
Accessed using class name	Accessed using object
get memory only once	get new memory each time a new object is created

2) Java static method

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

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

Example:

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

        Student9(int r, String n){
                rollno = r;
                name = n;
        }

        static void change(){
                college = "GPN";
    }
```

```
                void display (){
                        System.out.println(rollno+" "+name+" "+college);
                }
                public static void main(String args[]){
                        Student9.change();
                        Student9 s1 = new Student9 (111,"Karan");
                        Student9 s2 = new Student9 (222,"Aryan");
                        Student9 s3 = new Student9 (333,"Sonoo");
                                s1.display();
                        s2.display();
                        s3.display();
                }
        }
```

Output:

```
111 Karan GPN
222 Aryan GPN
333 Sonoo GPN
```

3) Java static block

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

Example:

```
        class ST_Employee{
                int eid;
                String name;
                static String company_name;
                static {                //static block invoked before main() method
                        company_name ="StudyTonight";
                }
                public void show(){
                        System.out.println(eid+" "+name+" "+company_name);
                }
                public static void main( String[] args ){
                        ST_Employee se1 = new ST_Employee();
                        se1.eid = 104;
                        se1.name = "Abhijit";
                        se1.show();
                }
        }
```

Output :

```
104 Abhijit StudyTonight
```

4. Why a non-static variable cannot be referenced from a static context ?

When you try to access a non-static variable from a static context like main method, java compiler throws a message like *"a non-static variable cannot be referenced from a static context"*. This is because non-static variables are related with instance of class(object) and they get created when instance of a class is created by using **new** operator. So if you try to access a non-static variable without any instance compiler will complain because those variables are not yet created and they don't have any existence until an instance is created and associated with it.

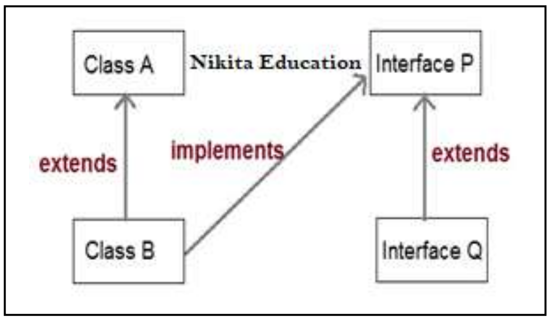
H. Inheritance Basics

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order. The class which inherits the properties of other is known as **Subclass** (derived class, child class) and the class whose properties are inherited is known as **Super Class** (base class, parent class). **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of parent object. Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship. **extends and implements** keywords are used to describe inheritance in Java.

Why use inheritance in java:

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

1. **extends Keyword:** inherit the properties of a class.



Syntax:

```
class Super{
}

class Sub extends Super{
}
```

Example:

```
class Vehicle{
  ......
}

class Car extends Vehicle{
 .......   //extends the property of vehicle class.
}
```

2. **Example:**

```
        class Parent{
        public void p1(){
                System.out.println("Parent method");
        }
}

public class Child extends Parent {
        public void c1(){
                System.out.println("Child method");
        }
        public static void main(String[] args){
                Child cobj = new Child();
                cobj.c1();   //method of Child class
                cobj.p1();   //method of Parent class
        }
}
```

```
Output :
Child method
Parent method
```

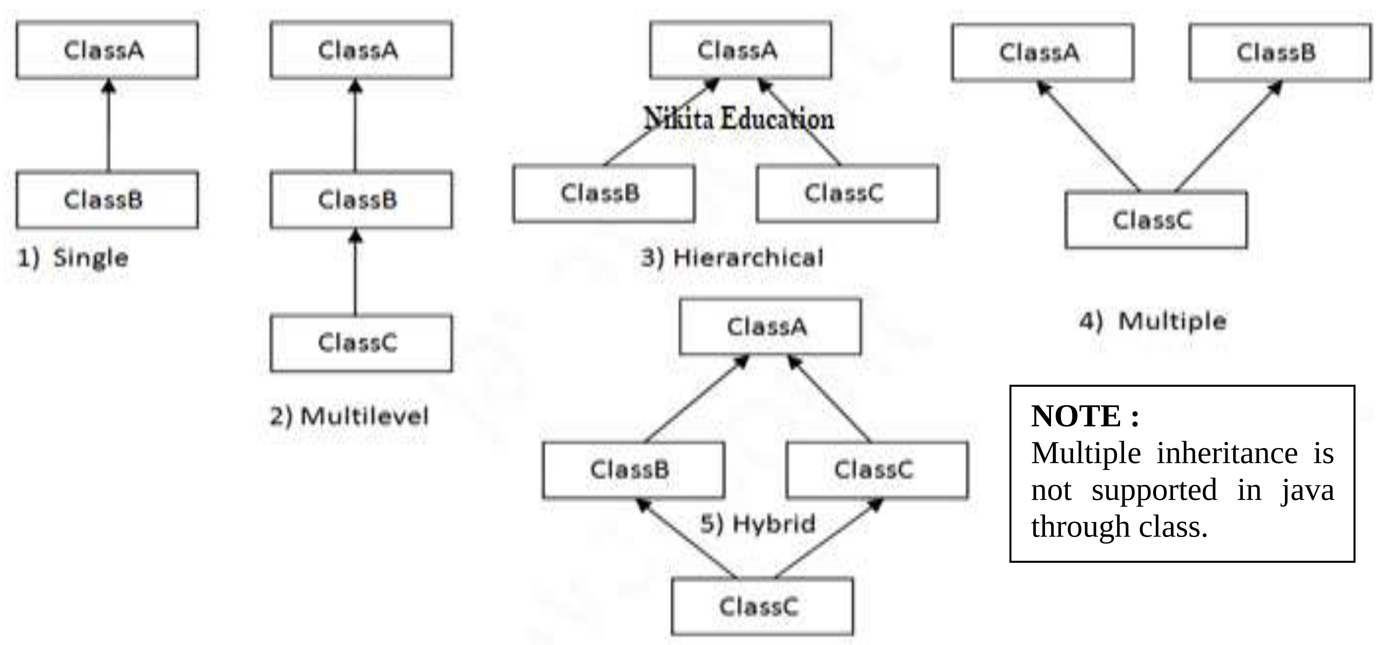
3. **Example:**

```
        class Vehicle{          String vehicleType;          }

public class Car extends Vehicle {
        String modelType;

        public void showDetail(){
                vehicleType = "Car";        //accessing Vehicle class member
                modelType = "sports";
                System.out.println(modelType+" "+vehicleType);
        }
        public static void main(String[] args){
                Car car =new Car();
                car.showDetail();
        }
}
```

Output :
sports Car

4. Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance



NOTE :
Multiple inheritance is not supported in java through class.

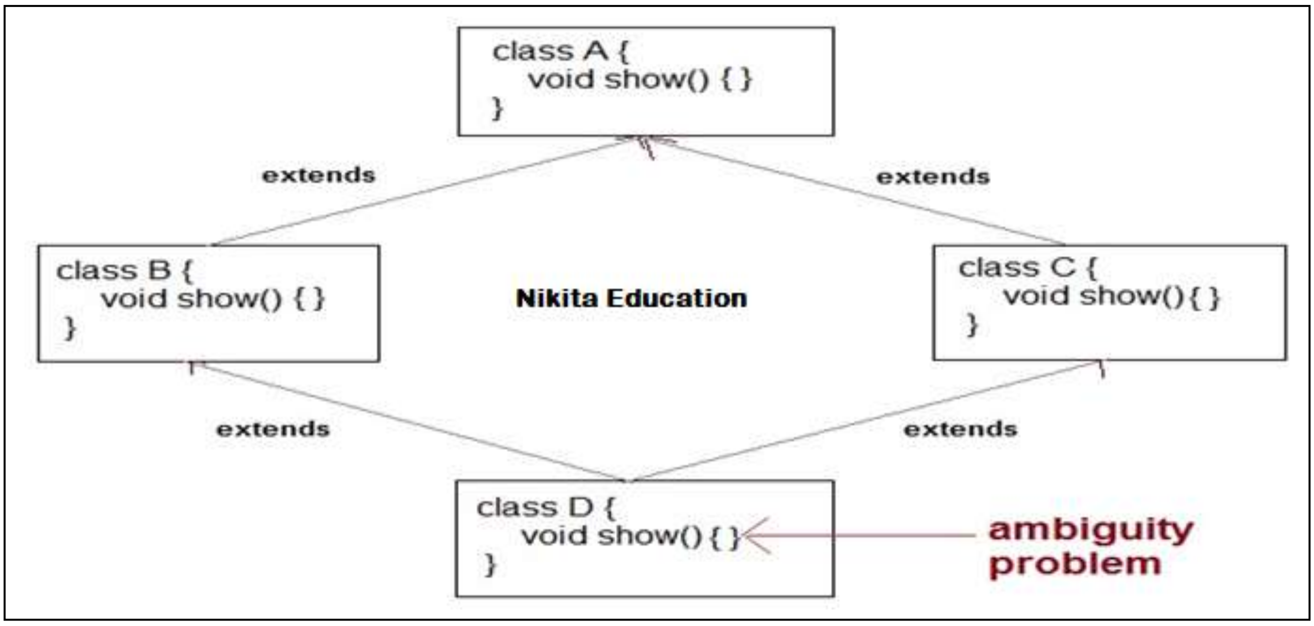
5. Why multiple inheritance is not supported in java?

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

Example:

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

        class B{
                void msg(){System.out.println("Welcome");}
        }
        //suppose if it were
class C extends A,B{
        public static void main(String args[]){
                C obj=new C();
                obj.msg();
                //Now which msg() method would be invoked?
        }
}
```

Compile Time Error



I. Super Keyword

The **super** keyword is similar to **this** keyword following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of super class from the members of subclass, if they have same names.
- It is used to **invoke the super class** constructor from subclass.

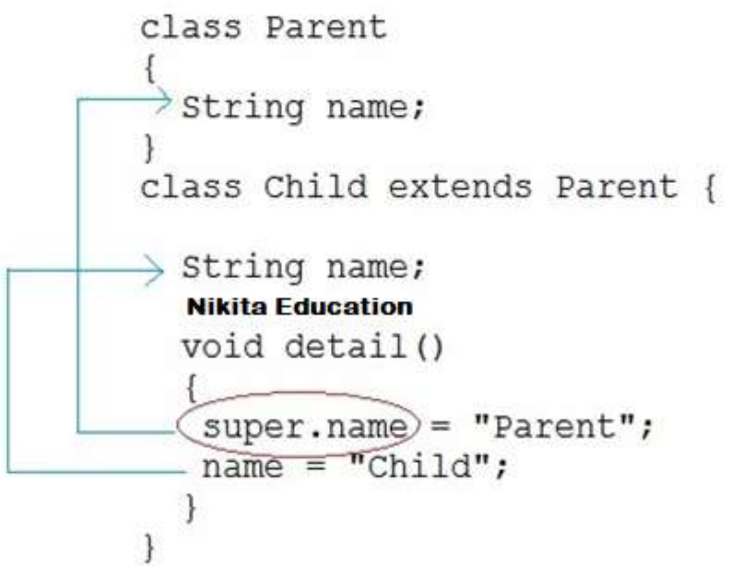
a. Differentiating the members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

super.variable; **super.method();**

b. Invoking Super class constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the super class. But if you want to call a parameterized constructor of the super class, you need to use the super keyword as shown below. **super(values);**



```
class Parent
{
   String name;
}
class Child extends Parent {

   String name;
   void detail()
   {
     super.name = "Parent";
     name = "Child";
   }
}
```

1. Example of Child class referring Parent class property using super keyword

```
        class Parent{
                String name;
        }

        public class Child extends Parent {
                String name;
                public void details(){
                        super.name = "Parent";        //refers to parent class member
                        name = "Child";
                        System.out.println(super.name+" and "+name);
                }
                public static void main(String[] args){
                        Child cobj = new Child();
                        cobj.details();
                }
}
```

Output :
Parent and Child

2. Example of Child class referring Parent class methods using super keyword

```
        class Parent{
                String name;
                public void details(){
                        name = "Parent";    System.out.println(name);
                }
        }
        public class Child extends Parent {
                String name;
                public void details(){
                        super.details();        //calling Parent class details() method
                        name = "Child";
                        System.out.println(name);
                }
                public static void main(String[] args){
                        Child cobj = new Child();
                        cobj.details();
                }
        }
```

Output :
Parent
Child

3. Example of Child class calling Parent class constructor using super keyword

```
        class Parent{
                String name;
                public Parent(String n){
                        name = n;
                }
        }
        public class Child extends Parent {
                String name;
                public Child(String n1, String n2){
                        super(n1);        //passing argument to parent class constructor
                        this.name = n2;
                }
                public void details(){
                        System.out.println(super.name+" and "+name);
                }
                public static void main(String[] args){
                        Child cobj = new Child("Parent","Child");
                        cobj.details();
                }
        }
```

Output :

```
Parent and Child
```

4. Super class reference pointing to Sub class object.

In context to above example where Class B extends class A. **A a=new B();** is legal syntax because of IS-A relationship is there between class A and Class B.

5. Can you use both this() and super() in a Constructor?

NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.

J. Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**. In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

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

When a method in a sub class has same name and type signature as a method in its super class, then the method is known as overridden method.

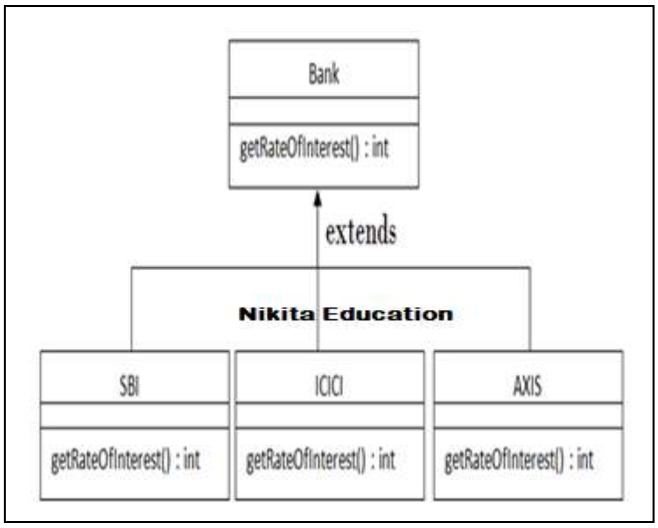
1. Example:

```
class Animal{
        public void eat(){
                System.out.println("Generic Animal eating");
        }
}
class Dog extends Animal{
        public void eat(){   //eat() method overriden by Dog class.
                System.out.println("Dog eat meat");
        }
}
```

NOTE : Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

2. Real example of Java Method Overriding

```
        class Bank{
                int getRateOfInterest(){return 0;}
        }

        class SBI extends Bank{
                int getRateOfInterest(){return 8;}
        }

        class ICICI extends Bank{
                int getRateOfInterest(){return 7;}
        }

        class AXIS extends Bank{
                int getRateOfInterest(){return 9;}
        }

class Test2{
        public static void main(String args[]){
                SBI s=new SBI();
                ICICI i=new ICICI();
                AXIS a=new AXIS();
                System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
                System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
                System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}}
```


Bank
getRateOfInterest() : int
extends
Nikita Education
SBI
getRateOfInterest() : int
ICICI
getRateOfInterest() : int
AXIS
getRateOfInterest() : int

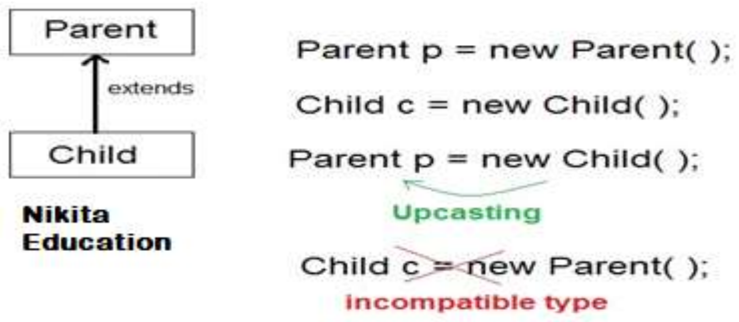
Output:

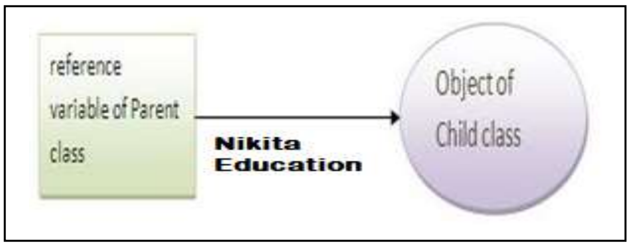
```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	*parameter must be different*.	*parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

K. Dynamic Dispatch Methods

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.





a. Up-casting

When **Parent** class reference variable refers to **Child** class object, it is known as **Up-casting**

b. Example:

```
class Game{
        public void type(){
                System.out.println("Indoor & outdoor");
        }
}
Class Cricket extends Game{
        public void type() {
                System.out.println("outdoor game");
        }

        public static void main(String[] args){
                Game gm = new Game();
                Cricket ck = new Cricket();
                gm.type();
                ck.type();
                gm=ck;      //gm refers to Cricket object
                gm.type();  //calls Cricket's version of type
        }
}
```

Notice the last output. This is because of **gm = ck**; Now gm.type() will call Cricket version of type method. Because here gm refers to cricket object.

Output :

```
Indoor & outdoor
Outdoor game
Outdoor game
```

c. Example:

```
class Animal{
        void eat(){System.out.println("eating");}
}
class Dog extends Animal{
        void eat(){ System.out.println("eating fruits"); }
}
class BabyDog extends Dog{
        void eat(){System.out.println("drinking milk");}
}
public class Test{
        public static void main(String args[]){
                Animal a1,a2,a3;
                a1=new Animal();
                a2=new Dog();
                a3=new BabyDog();
                a1.eat();
                a2.eat();
                a3.eat();
        }
}
```

Output:

```
eating
eating fruits
drinking Milk
```

d. Difference between Static binding and Dynamic binding in java?

Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

L. Final Keyword

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be: **variable, method, or class.** The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1. final Variables:

If you make any variable as final, you cannot change the value of final variable(It will be constant). A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object. With variables, the *final* modifier often is used with *static* to make the constant a class variable.

Example:

```
public class Test{

        final int value = 10;
        // The following are examples of declaring constants:
        public static final int BOXWIDTH = 6;
        static final String TITLE = "Manager";

        public void changeValue(){
                value = 12;                    //will give an error
                BOXWIDTH = 8;
        }
}
```

Output: Compile Time Error

2. final Methods:

A final method cannot be overridden by any subclasses. As mentioned previously the final modifier prevents a method from being modified in a subclass. The main intention of making a method final would be that the content of the method should not be changed by any outsider. If you make any method as final, you cannot override it.

Example:

```
public class Test{
        public final void changeName(){
                // body of method
        }
}
```

Example:

```
class Bike{
        final void run(){System.out.println("running");}
}
```

```
class Honda extends Bike{

        void run(){
                System.out.println("running safely with 100kmph");
        }

                public static void main(String args[]){
                Honda honda= new Honda();
                honda.run();
        }
}
```

Output: Compile Time Error

3. **final Classes:**

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class. If you make any class as final, you cannot extend it.

Example:

```
public final class Test {
        // body of class
}
```

Example:

```
final class Bike{
}

class Honda1 extends Bike{
        void run(){
                System.out.println("running safely with 100kmph");
        }
                public static void main(String args[]){
                Honda1 honda= new Honda();
                honda.run();
        }
}
```

Output: Compile Time Error

4. **Can we declare a constructor final?**
   No, because constructor is never inherited.

M. Abstract Classes & Methods

1. **Abstraction in Java**

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only important things to the user and hides the internal details.

Ways to achieve Abstraction in java: Abstract class (0 to 100%), Interface (100%)

2. **Abstract class**

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.
Example: abstract class A{}

3. Abstract method

A method that is declared as abstract and does not have implementation is known as abstract method. **Example:** abstract void printStatus(); //no body and abstract

4. Example:

```
abstract class Bike{

        abstract void run();

}

class Honda4 extends Bike{
        void run(){
                System.out.println("running safely..");
        }
        public static void main(String args[]){
                Bike obj = new Honda4();
                obj.run();
        }
}
```

Output: running safely..

5. Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**. A **factory method** is the method that returns the instance of the class.

Example:

```
abstract class Shape{
        abstract void draw();
}
class Rectangle extends Shape{
        void draw(){
                System.out.println("drawing rectangle");
        }
}

class Circle1 extends Shape{
        void draw(){
                System.out.println("drawing circle");
        }
}
class TestAbstraction1{
        public static void main(String args[]){
                Shape s=new Circle1();
                s.draw();
        }
}
```

Output: drawing circle

6. Abstract class with concrete(normal) method.

Abstract classes can also have normal methods with definitions, along with abstract methods.

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

class B extends A{
        void callme() {
                System.out.println("this is callme.");
        }

        public static void main(String[] args) {
                B b=new B();
                b.callme();
                b.normal();
        }
}
```

Output :

```
this is callme.
this is concrete method.
```

7. Points to Remember

1. Abstract classes are not Interfaces. They are different, we will study this when we will study Interfaces.
2. An abstract class must have an abstract method.
3. Abstract classes can have Constructors, Member variables and Normal methods.
4. Abstract classes are never instantiated.
5. When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

Rule: If there is any abstract method in a class, that class must be abstract.

8. Abstract Class Vs. Concrete (Normal) Class

A partial implemented class is called an abstract class , Where as fully implemented class is commonly known as concrete or normal class.

Abstract class:
- It is must to declare a class with an abstract access modifier .
- May or may not contain abstract methods.
- It is not possible to instantiate a abstract class .
- Variables are not final by default. We can able to reassign values.
- The abstract methods should implement in the derived classes. If not , the derived class also become an abstract.
- Interface implementation is possible.

Concrete class:
- Should not declare a concrete class with an abstract access modifier.
- Should not contain abstract methods.
- Instantiation is possible for a concrete class.
- Variables are not final by default.
- There is no abstract methods in any level to implement.
- Interface implementation is possible.

N. Visibility Controls

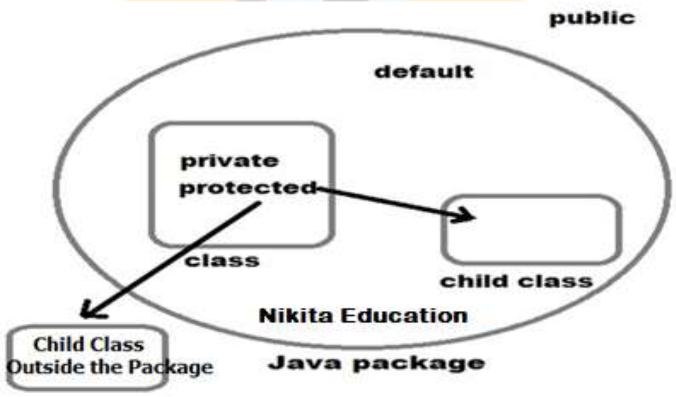
Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package. the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**. The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class. There are 4 types of java access modifiers:

- **Default :** Default has scope only inside the same package
- **Public :** Public scope is visible everywhere
- **Protected :** Protected has scope within the package and all sub classes
- **Private :** Private has scope only within the classes

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



1. Private

The private access modifier is accessible only within class.

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

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

Role of Private Constructor

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

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

2. Default

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

Example:

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

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

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

3. Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only. The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example:

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

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

4. Public

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

Example:

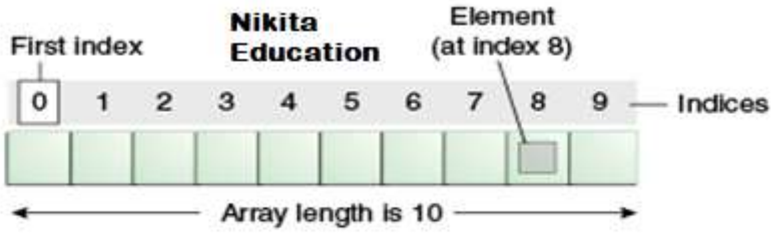
```
package pack;
public class A{
        public void msg(){
                System.out.println("Hello");
        }
}

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

5. Understanding all java access modifiers

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

O. Arrays

Normally, array is a collection of similar type of elements that have contiguous memory location. **Java array** is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array. Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

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

Disadvantage of Java Array

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

Types of Array in java: Single Dimensional Array, Two Dimensional Array, Multidimensional Array.

Array Declaration: **datatype[] identifier;**
or
datatype identifier[];

Example :

```
int[] arr;
char[] arr;
short[] arr;
long[] arr;
int[][] arr;  //two dimensional array.
```

Initialization of Array:
new operator is used to initialize an array.

Example :

```
int[] arr = new int[10];   //10 is the size of array.
        or
int[] arr = {10,20,30,40,50};
```

Example:

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

Multidimensional array in java

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

Syntax to Declare Multidimensional Array in java

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

Example to instantiate Multidimensional Array in java

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

Example to initialize Multidimensional Array in java

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

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

P. Strings

Java String provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc. In java, string is basically an object that represents sequence of char values. An array of characters works same as java string.

For example:

1. char[] ch={'n','i','k','i','t','a','e','d','u','c','a','t','i','o','n'};
1. String s=new String(ch); is same as: String s="nikitaeducation";

The java String is immutable i.e. it cannot be changed but a new instance is created. For mutable class, you can use StringBuffer and StringBuilder class.

How to create String object?
There are two ways to create String object:
1. By string literal **2.** By new keyword

1) By String Literal:

Java String literal is created by using double quotes.
For Example: **String s="welcome";**

2) By new keyword:

1. **String s1=new String("Welcome");**
2. **String str=new String(s);**

Example:

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

        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
    }
}
```

Output:

```
java
strings
example
```

What is an Immutable object?

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper class's objects are immutable.

Java String class methods

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

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns formatted string
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5	String substring(int beginIndex)	returns substring for given begin index
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10	boolean equals(Object another)	checks the equality of string with object
11	boolean isEmpty()	checks if string is empty
12	String concat(String str)	concatinates specified string
13	String replace(char old, char new)	replaces all occurrences of specified char value
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15	String trim()	returns trimmed string omitting leading and trailing spaces
16	String split(String regex)	returns splitted string matching regex
17	String split(String regex, int limit)	returns splitted string matching regex and limit
18	String intern()	returns interned string
19	int indexOf(int ch)	returns specified char value index
20	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21	int indexOf(String substring)	returns specified substring index
22	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23	String toLowerCase()	returns string in lowercase.
24	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25	String toUpperCase()	returns string in uppercase.

1. StringBuffer class:

StringBuffer class is used to create a **mutable** string object. It represents growable and writable character sequence. As we know that String objects are immutable, so if we do a lot of changes with **String** objects, we will end up with a lot of memory leak. So **StringBuffer** class is used when we have to make lot of modifications to our string. StringBuffer defines 3 constructors. They are,

a. Important Constructors of StringBuffer class

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
2. **StringBuffer(String str):** creates a string buffer with the specified string.
3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

b. Important methods of StringBuffer class

1. **public synchronized StringBuffer append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

2. **public synchronized StringBuffer insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

3. **public synchronized StringBuffer replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.

4. **public synchronized StringBuffer delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.

5. **public synchronized StringBuffer reverse():** is used to reverse the string.

6. **public int capacity():** is used to return the current capacity.

7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.

8. **public char charAt(int index):** is used to return the character at the specified position.

9. **public int length():** is used to return the length of the string i.e. total number of characters.

10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.

11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

c. Difference between String and StringBuffer

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Q. Vector class

Vector is a class and it implements a dynamic array. It is similar to ArrayList, but with two differences:

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program. Constructors provided by the vector class.

SR	Constructor and Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size:
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward
4	**Vector(Collection c)** **creates a vector that contains the elements of collection c**

Vector defines several legacy method.

Method	Description
addElement()	add element to the Vector
elementAt()	return the element at specified index
elements	return an enumeration of element in vector
firstElement()	return first element in the Vector
lastElement()	return last element in the Vector
removeAllElement()	remove all element of the Vector

Example

```
import java.util.*;
public class VectorDemo {
        public static void main(String args[]) {
                Vector v = new Vector(3, 2);
                System.out.println("Initial size: " + v.size());
                System.out.println("Initial capacity: " + v.capacity());
```

```
                    v.addElement(new Integer(1));
                    v.addElement(new Integer(2));
                    v.addElement(new Integer(3));
                    v.addElement(new Integer(4));
                    System.out.println("Capacity after four additions: " + v.capacity());
                    v.addElement(new Double(5.45));
                    System.out.println("Current capacity: " +v.capacity());
                    v.addElement(new Double(6.08));
                    v.addElement(new Integer(7));
                    System.out.println("Current capacity: " +v.capacity());
                    v.addElement(new Float(9.4));
                    v.addElement(new Integer(10));
                    System.out.println("Current capacity: " + v.capacity());
                    v.addElement(new Integer(11));
                    v.addElement(new Integer(12));
                    System.out.println("First element: " + (Integer)v.firstElement());
                    System.out.println("Last element: " +(Integer)v.lastElement());
                    if(v.contains(new Integer(3)))
                    System.out.println("Vector contains 3.");
                    Enumeration vEnum = v.elements();
                    System.out.println("\nElements in vector:");
                    while(vEnum.hasMoreElements())
                            System.out.print(vEnum.nextElement() + " ");
                    System.out.println();
            }
    }
```

ArrayList	**Vector**
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.
3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
5) ArrayList uses **Iterator** interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.

R. Wrapper Classes

Wrapper class in java provides the mechanism *to convert primitive into object and object into primitive*. Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known and autoboxing and vice-versa unboxing. One of the eight classes of *java.lang* package are known as wrapper class in java. The list of eight wrapper classes are given below:

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

Wrapper class Example: Primitive to Wrapper

```
public class WrapperExample1{

public static void main(String args[]){

int a=20;

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

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

Wrapper class Example: Wrapper to Primitive

```
public class WrapperExample2{
    public static void main(String args[]){             //Converting Integer to int
            Integer a=new Integer(3);
            int i=a.intValue();                         //converting Integer to int
            int j=a;                                    //unboxing,
             //now compiler will write a.intValue() internally
            System.out.println(a+" "+i+" "+j);
    }
}
```

Output: 3 3 3

Unit-III
Packages & Interfaces

Unit	Topic Name	Main Topic	Subtopics	Marks
3	Packages & Interfaces	Packages	Define a Package, Using system Package, Naming Convention, Creating Package, Accessing a package, Import a package, adding a class to a package	12
		Interfaces	Defining interfaces, Extending interfaces, Implementing interfaces, Accessing Interface variable, Applying Interfaces	

A. Packages

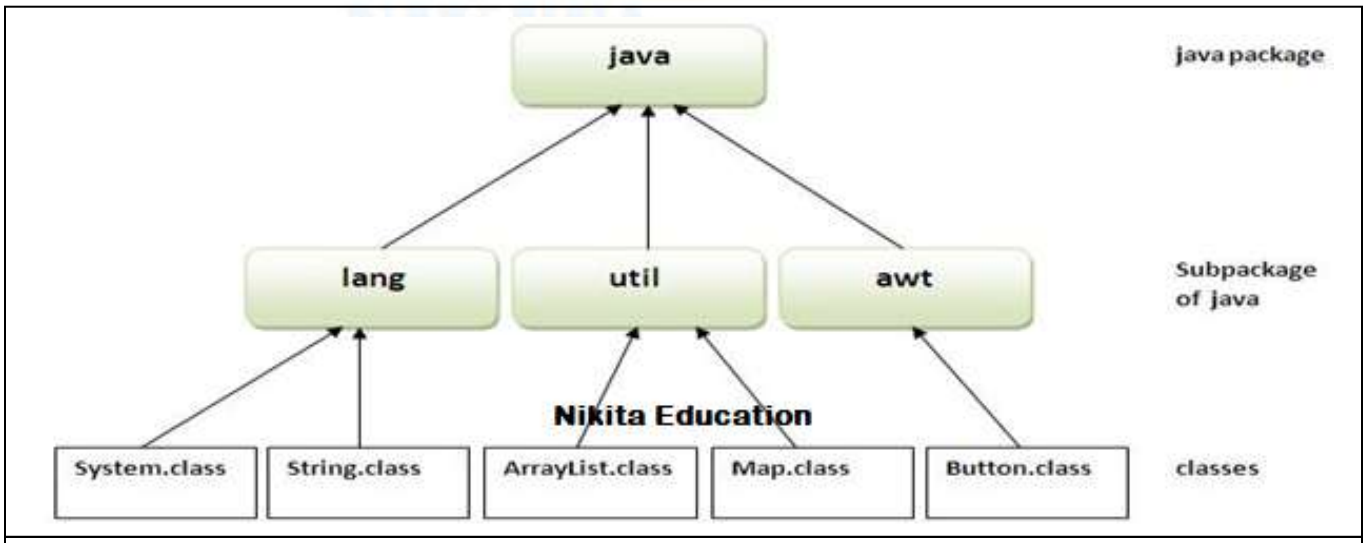
a) Package Definition:

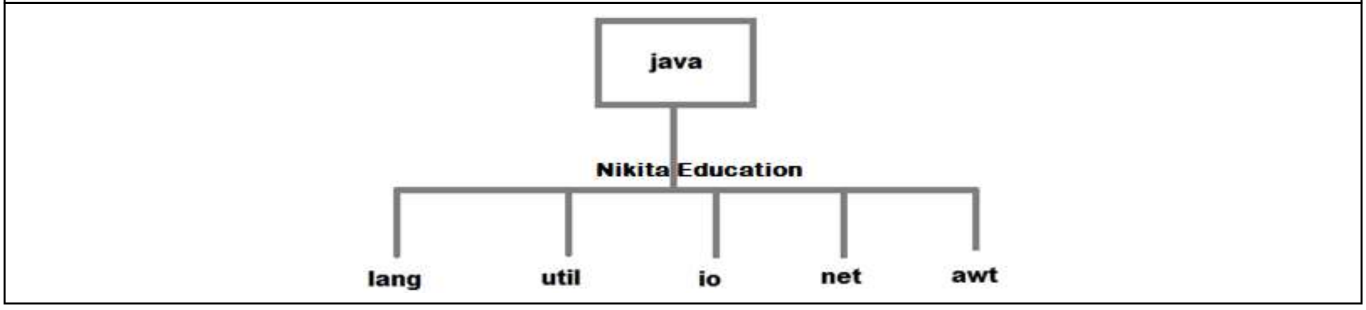
A package can be defined as a group of similar types of classes, interfaces, enumeration and sub-packages. Package in java can be categorized in two form, **built-in package** and **user-defined package**. There are many built-in packages such as **java, lang, awt, javax, swing, net, io, util, sql** etc. Package are used in Java, in-order to avoid name conflicts and to control access of class, interface and enumeration etc. Using package it becomes easier to locate the related classes.

Advantage of Java Package:

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

System Packages:





Package are categorized into two forms:

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

Uses of java package

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

b) Creating Package:

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

Syntax:
```
package packageName;
        Or
package pack1.pack2.pack3.pack4.pack5;
```

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

The above statement create a package called **mypack**. Java uses file system directory to store package.

Example: The **package keyword** is used to create a package in java.

```
package mypack;
public class Simple{
        public static void main(String args[]){
                System.out.println("Welcome to package");
        }
}
```

Example:
```
package mypack
class Book{
        String bookname;
        String author;
        Book(String b, String c){
                this.bookname = b;
                this.author = c;
        }
        public void show(){
                System.out.println(bookname+" "+ author);
        }
}
```

```
class test{
        public static void main(String[] args){
                Book bk = new Book("java","Herbert");
                bk.show();
        }
}
```

How to compile java package: There are two ways used to compile java packages.

1. By Creating Directory:

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

2. By Using -d attribute

If you are not using any IDE, you need to follow the **syntax** given below:

Syntax: javac -d directory javafilename

Example: c:\javaprog>javac -d **.** Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use **.** (dot).

How to run java package program: You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

c) Adding a class to a Package:

We can add a class to an existing package by using the package name at the top of the program as shown above and saving the **.java** file under the package directory. We need a new **java** file if we want to define a **public** class, otherwise we can add the new class to an existing **java** file and recompile it. Similar to classes, we can also define **interfaces** as a part of a package and import and implement it as and when required. Moreover, we can extend a class inside a package to create subclasses.

d) Accessing Package:

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

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

1) Using packagename.*:

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

Example of package that import the packagename.*:

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

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

Output: Hello

2) Using packagename.classname:

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

Example of package by import package.classname:

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

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

Output: Hello

3) Using fully qualified name:

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface. It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

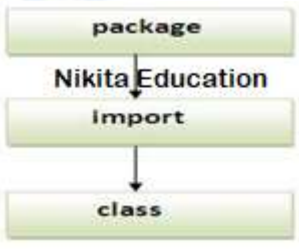
Example of package by import fully qualified name:

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

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

```
Output: Hello
```

Note: If you import a package, subpackages will not be imported. If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



e) Subpackage:

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**. Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on. The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
package com.javatpoint.core;
class Simple{
        public static void main(String args[]){
                System.out.println("Hello subpackage");
        }
}
```

```
To Compile:     javac -d . Simple.java
To Run:         java com.javatpoint.core.Simple
Output:         Hello subpackage
```

e) System Packages:

All the classes and interfaces that come with the installation of JDK are put together are known as Java **API (Application Programming Interface)**. All the Java API packages are prefixed with java or javax. Following table gives some important packages, a few prominent classes and their functionality.

PACKAGE NAME	CLASSES INSIDE PACKAGE	FUNCTIONALITY (PURPOSE)
java.lang	System, String, Object, Thread, Exception etc.	These classes are indispensable for every Java program. For this reason, even if this package is not imported, JVM automatically imports.
java.util	Random, Date, GregorianCalendar, Stack, Vector, LinkedList, HashMap etc.	These are called as utility (service) classes and are used very frequently in coding.
java.io	FileInputStream, FileOutputStream, FileReader, FileWriter, RandomAccessFile, BufferedReader, BufferedWriter etc.	These classes are used in all I/O operations including keyboard input.
java.net	URL, ServerSocket, Socket, DatagramPacket, DatagramSocket etc.	Useful for writing socket programming (LAN communication).
java.applet	AppletContext, Applet, AudioStub, AudioClip etc	Required for developing applets that participate on client-side in Internet (Web) programming.
java.awt	Button, Choice, TextField, Frame, List, Checkbox etc.	Essential for developing GUI applications.
java.awt.event	MouseListener, ActionListener, ActionEvent, WindowAdapter etc.	Without these classes, it is impossible to handle events generated by GUI components
java.sql	DriverManager, Statement, Connection, ResultSet etc	Required for database access in JDBC applications.

1. Accessing classes in a package:

```
1 ) import java.util.Vector; // import the Vector class from util package

2 ) import java.util.*; // import all the classes from util package
```

First statement imports **Vector** class from **util** package which is contained inside **java** package. Second statement imports all the classes from **util** package. Packages have an advantage over header files of C-lang. A package allows importing a single class also instead of importing all. C-lang does not have this ease of getting one function from a header file.

```
import java.net.*;                      // imports all the classes and interfaces
import java.awt.evnet.*;                // imports all the classes and interfaces
import java.net.Socket;                 // imports only Socket class
import java.awt.event.WindowEvent;      // imports only WindowEvent class
```

Note: While importing a single class, asterisk (*) should not be used.

f) Naming Conventions:

Like identifiers have conventions, the packages come with their own **naming conventions**. Like keywords and protocols, packages are also of lowercase letters. In a single project, a number of programmers may be involved assigned with different modules and tasks. To avoid namespace problems in storing their work, naming conventions are followed. While creating packages, they may follow company name, project name or personal name etc. to precede their package. Following are a few examples.

1. **jyothi.solutions:** solutions is the package (folder) preceded by individual name, jyothi.
2. **forecastingtool.jyothi.solutions:** Preceded by the project name(forecastingtool) and individual name.
3. **mindspace.forecastingtool.jyothi.solutions:** Preceded by company name (mindspace), project name and individual name.

This package naming convention never clashes with the others work in a big project involving many programmers. The same convention is also followed in working on a domain also.

g) Import keyword:

import keyword is used to import built-in and user-defined packages into your java source file. So that your class can refer to a class that is in another package by directly using its name. There are 3 different ways to refer to class that is present in different package

1. Using fully qualified name (But this is not a good practice.)

Example :
```
class MyDate extends java.util.Date {     //statements;          }
```

2. import the only class you want to use

Example :
```
import java.util.Date;
class MyDate extends Date{
        //statement.
}
```

3. import all the classes from the particular package

Example :
```
import java.util.*;
class MyDate extends Date{
        //statement;
}
```

4. import statement is kept after the package statement.

Example :
```
package mypack;
import java.util.*;
```

But if you are not creating any package then import statement will be the first statement of your java source file.

h) Static import:

static import is a feature that expands the capabilities of **import** keyword. It is used to import **static** member of a class. We all know that static member are referred in association with its class name outside the class. Using **static import**, it is possible to refer to the static member directly without its class name. There are two general form of static import statement.

1. The first form of **static import** statement, import only a single static member of a class

Syntax: import static *package.class-name.static-member-name;*

Example: import static java.lang.Math.sqrt; //importing static method **sqrt** of **Math** class

2. The second form of **static import** statement, imports all the static member of a class

Syntax: import static *package.class-type-name.*;*

Example: **i**mport static java.lang.Math.*; //importing all static member of **Math** class

3. Example without using static import

```
public class Test{
    public static void main(String[] args){
            System.out.println(Math.sqrt(144));
    }
}
```

Output: 12

4. Example using static import

```
import static java.lang.Math.*;
public class Test{
    public static void main(String[] args){
            System.out.println(sqrt(144));
    }
}
```

Output: 12

B. Interfaces

a) Interface Definition:

An **interface** is like a class containing methods and variables but the difference is that an interface can define only **abstract methods** and **final fields**. Interfaces doesn't contain constructors and they cannot be instantiated. There is no method definition inside the interface. It is the responsibility of the class that implements the interface to define those methods. If the class that implements an interface doesn't define all the methods of the interface, then that class becomes an abstract class and cannot be instantiated. An **interface in java** is a blueprint of a class. It has static constants and abstract methods only. The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java. Java Interface also **represents IS-A relationship**. It cannot be instantiated just like abstract class.

Syntax : interface interface_name { }

Example:

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

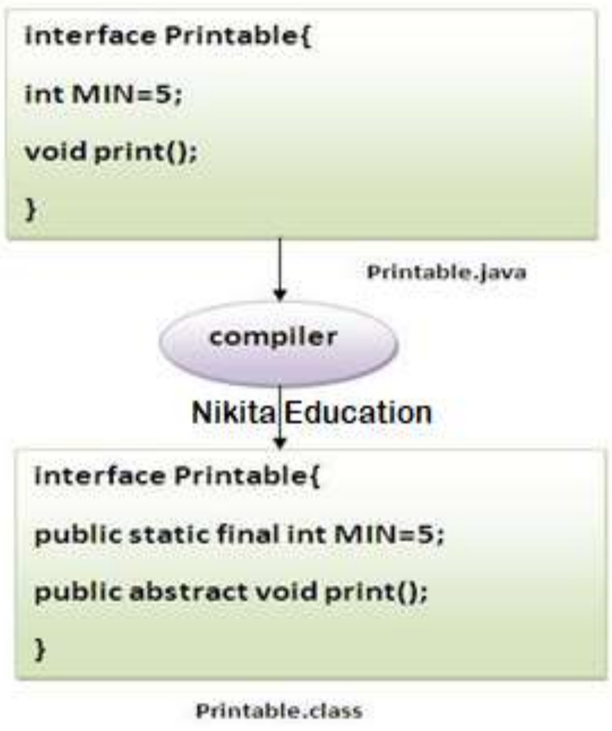
b) Why use Java interface?

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

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

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members automatically.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.

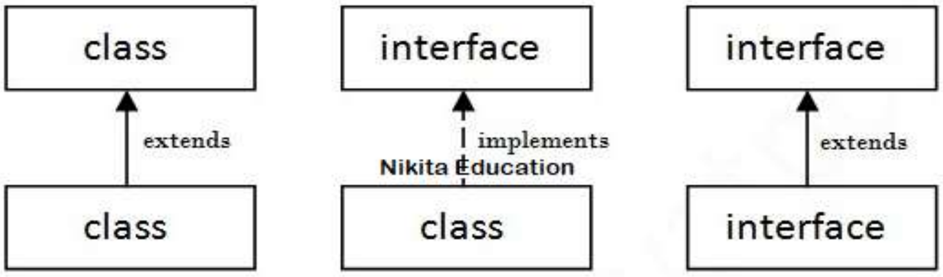


```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

c) Understanding relationship between classes and interfaces

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



Simple example of Java interface

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

```
Output: Hello
```

d) Features of Interfaces

Interfaces are derived from abstract classes with a special additional rule that interfaces should contain only abstract methods. Let us see some more properties.

1. Interfaces support multiple inheritance which is not possible with concrete and abstract classes.
2. "implements" keyword is used in place of "extends". "extends" comes with concrete and abstract classes.
3. Interfaces must contain abstract methods only.
4. As in abstract classes, all the abstract methods of the interfaces should be overridden by the subclass.
5. As with abstract classes, with interfaces also, objects cannot be created but reference variables can be created.
6. Interface reference variable can be assigned with concrete subclass objects. Once assigned, the interface reference variable works like an object. This feature applies to abstract classes also.
7. As all methods must be abstract and public, these two keyword can be omitted in method declaration. If omitted, JVM takes by default.
8. All interface variables must be public, static and final. If omitted, they are taken by default.
9. All interface methods should be overridden with public only as overridden method cannot have a weaker access specifier.

e) Rules for using Interface

- Methods inside Interface must not be static, final, native or strictfp.
- All variables declared inside interface are implicitly public static final variables(constants).
- All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.
- Interface can extend one or more other interface.
- Interface cannot implement a class.
- Interface can be nested inside another interface.

f) Example of Interface implementation

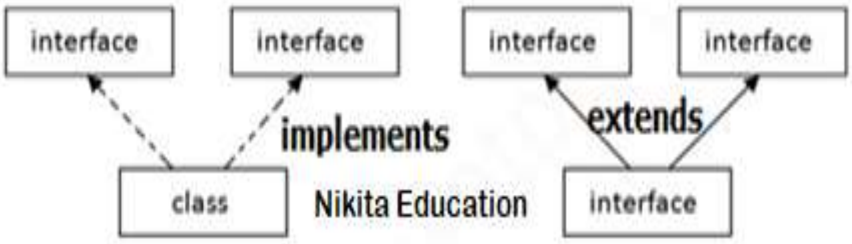
```
interface Moveable{
        int AVG-SPEED = 40;
        void move();
}

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

Output :

```
Average speed is 40.
```

g) Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Example:

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

interface Showable{
        void show();
}

class A7 implements Printable,Showable{
        public void print(){
                System.out.println("Hello");
        }
        public void show(){
                System.out.println("Welcome");
        }
        public static void main(String args[]){
                A7 obj = new A7();
                obj.print();
                obj.show();
        }
}
```

Output:

```
Hello
Welcome
```

h) Interface inheritance

Classes implements interfaces, but an interface extends other interface.

Example:

```
interface NewsPaper{
        void news();
}

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

Example:

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

interface Showable extends Printable{
        void show();
}

class Testinterface2 implements Showable{
        public void print(){
                System.out.println("Hello");
        }
        public void show(){
                System.out.println("Welcome");
        }
        public static void main(String args[]){
                Testinterface2 obj = new Testinterface2();
                obj.print();
                obj.show();
        }
}
```

Output:

```
Hello
Welcome
```

i) Abstract class Vs. Interface

ABSTRACT CLASS	INTERFACE
1. Multiple inheritance is not supported	Multiple inheritance is supported
2. To inherit "extends" keyword is used	To inherit "implements" keyword is used
3. May include concrete methods also	All must be abstract methods
4. Methods may be of any specifier except private. That is, may be public, protected or default	Methods must be public only
5. Access specifier should be written	If omitted, taken by default (as public)
6. Access modifier abstract should be written	If omitted, taken by default (as abstract)
7. Variables can be any access specifier including private	Variables should be public, static and final. If omitted, taken by default.
8. Constructors can be included in code	No constructors
9. main() can be included in code	main() method cannot be included

Unit-IV
Multithreading & Exception Handling

Unit	Topic Name	Main Topic	Subtopics	Marks
4	Multithreading & Exception Handling	Multithreading	The Java Thread Model, The Main Thread, Creating Thread, Extending a thread class, Stopping and Blocking a thread, Life cycle of thread, Using thread method, Thread exceptions, Thread priority, Synchronization, Implementing Runnable Interface, Thread class methods.	16
		Errors & Exceptions	Exception handling fundaments, Types of errors, Exception, Using Try and Catch, Multiple catch statement, using finally statement, Custom Exceptions, Exception Chain	

Multi-Threading:

A. Introduction to Multithreading

Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc.

Advantage of Java Multithreading

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
2. You **can perform many operations together so it saves time**.
3. Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

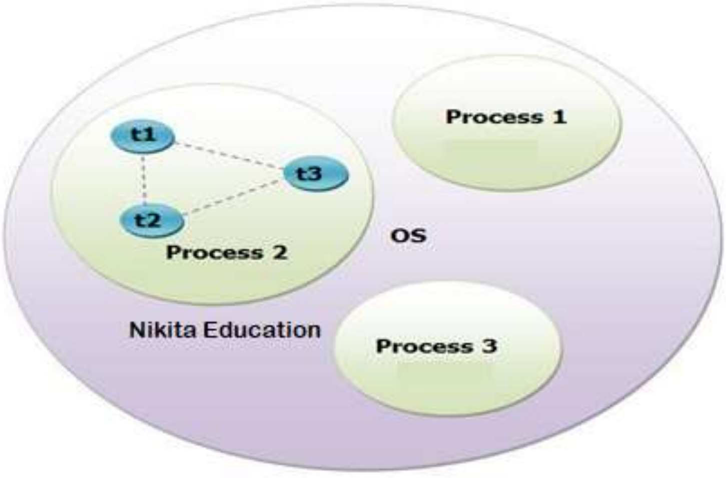
- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight & Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight & Cost of communication between the thread is low.

What is Thread in java?

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution. Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



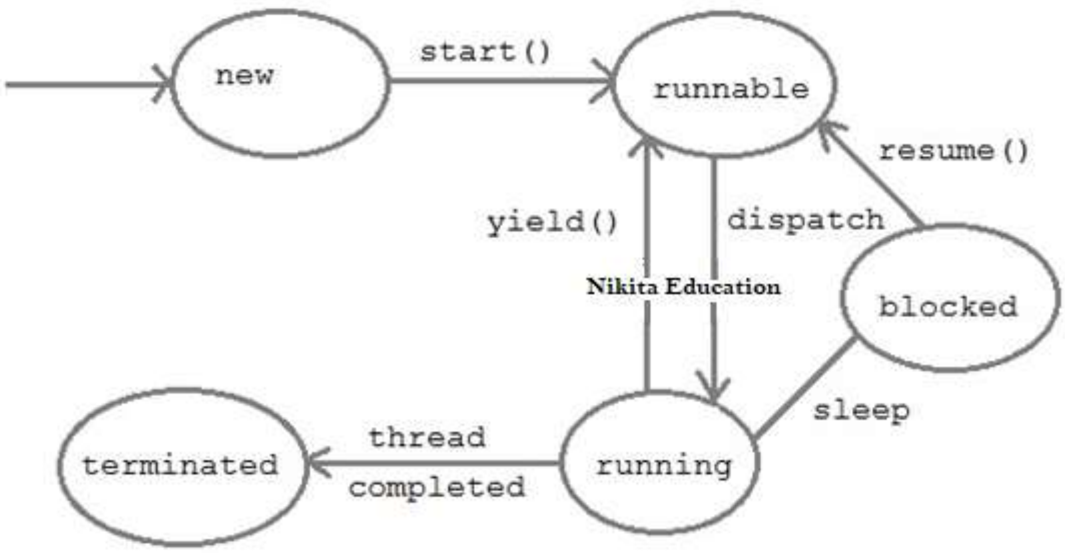
As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time one thread is executed only.

B. Life Cycle of Thread

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state. But for better understanding the threads, we are explaining it in the 5 states. The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New (born)
2. Runnable
3. Running
4. Non-Runnable (Blocked/waiting)
5. Terminated (dead)



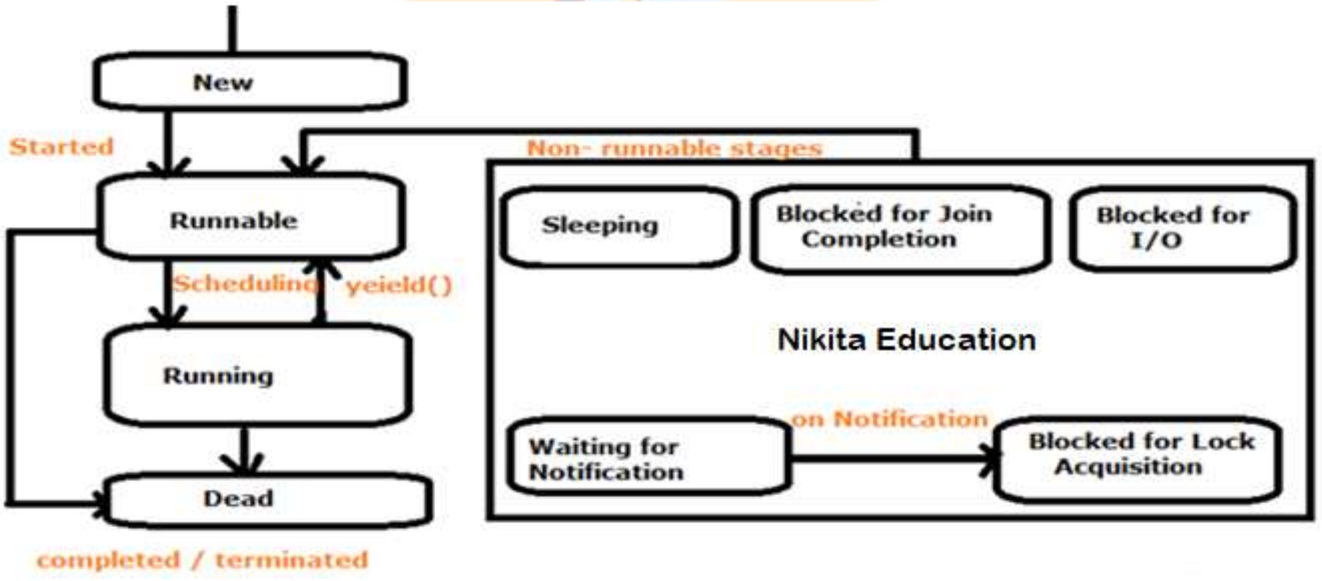
1. **New/born :** A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.

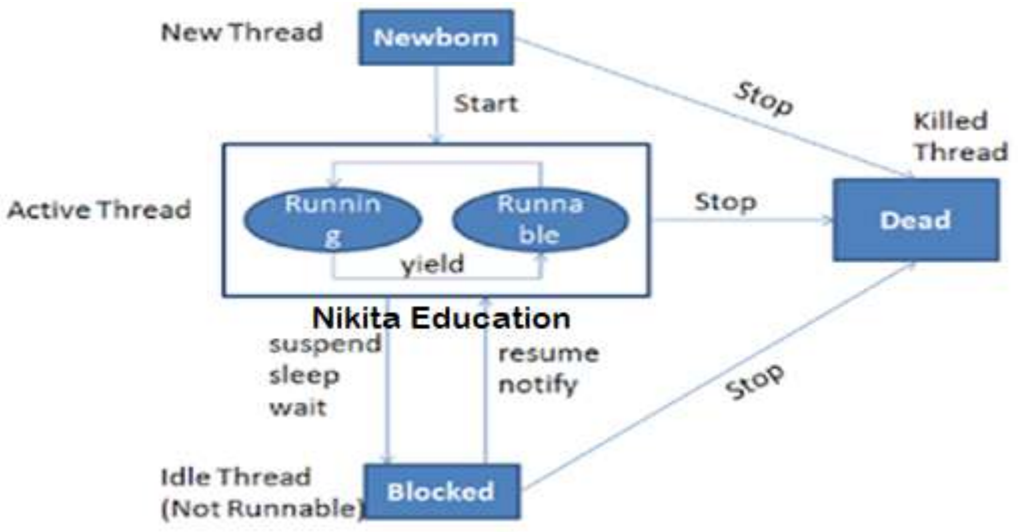
2. **Runnable :** After invocation of start() method on new thread, the thread becomes runnable.

3. **Running :** A method is in running thread if the thread scheduler has selected it.

4. **Waiting/blocked :** A thread is waiting for another thread to perform a task. In this stage the thread is still alive.
   - A Running Thread transit to one of the non runnable states, depending upon the circumstances.
   - A Thread remains non runnable until a special transition occurs.
   - A Thread does not go directly to the running state from non runnable state. But transits first to runnable state.
     1. Sleeping: The Threas sleeps for specified amount of time.
     2. Blocked for I/O: The Thread waits for a blocking operation to complete.
     3. Blocked for join completion: The Thread waits for completion of another Thread.
     4. Waiting for notification: The Thread waits for notification another Thread.
     5. Blocked for lock acquisition: The Thread waits to acquire the lock of an object.
   - JVM executes the Thread based on their priority and scheduling.

5. **Terminated/dead :** A thread enter the terminated state when it complete its task.





C. The 'main()' Thread

Even if you don't create any thread in your program, a thread called main thread is still created. Although the main thread is automatically created, you can control it by obtaining a reference to it by calling currentThread() method. Two important things to know about main thread are,

- It is the thread from which other threads will be produced.
- main thread must be always the last thread to finish execution.

Example:

```
class MainThread{
        public static void main(String[] args){
                Thread t=Thread.currentThread();
                t.setName("MainThread");
                System.out.println("Name of thread is "+t);
        }
}
```

Output :

```
Name of thread is
Thread[MainThread,5,main]
```

D. Creating Threads

There are two ways to create a thread:

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

1. Thread class:

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

Constructors of Thread class

1. Thread ( )
2. Thread (*String str*)
3. Thread (*Runnable r*)
4. Thread (*Runnable r*, *String str*)
5.

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.

13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

2. Runnable interface:

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

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

3. Starting a thread:

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

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

4. By extending Thread class:

Syntax:
```
import java.lang.*;
public class Counter extends Thread {
        public void run(){
                .... //thread task goes here
        }
}
```

Example:

```
class Multi extends Thread{
    public void run(){
            System.out.println("thread is running...");
    }
    public static void main(String args[]){
            Multi t1=new Multi();
            t1.start();
    }
}
```

Output: thread is running...

5. Who makes your class object as thread object?

Thread class constructor allocates a new thread object. When you create object of Multi class, your class constructor is invoked(provided by Compiler) from where Thread class constructor is invoked(by super() as first statement).So your Multi class object is thread object now.

Example:
```
class MyThread extends Thread{
        public void run(){
                System.out.println("Concurrent thread started running..");
        }
}

Class MyThreadDemo{
        public static void main( String args[] ){
                MyThread mt = new  MyThread();
                mt.start();
        }
}
```

Output :
```
concurrent thread started running..
```

6. By implementing the Runnable interface:

Syntax:
```
import java.lang.*;
public class Counter implements Runnable{
        Thread T;
        public void run(){
                ....    //thread task goes here
        }
}
```

Example:
```
        class Multi3 implements Runnable{
        public void run(){
                System.out.println("thread is running...");
        }
        public static void main(String args[]){
                Multi3 m1=new Multi3();
                Thread t1 =new Thread(m1);
                t1.start();
        }
}
```

Output:
```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute. The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface , the class needs to implement the **run()** method, which is of form,

public *void* **run**()

- run() method introduces a concurrent thread into your program. This thread will end when run() returns.
- You must specify the code for your thread inside run() method.
- run() method can call other methods, can use other classes and declare variables just like any other normal method.

Example:
```
class MyThread implements Runnable{
        public void run(){
                System.out.println("concurrent thread started running..");
        }
}
```

```
class MyThreadDemo{
        public static void main( String args[] ){
                MyThread mt = new MyThread();
                Thread t = new Thread(mt);
                t.start();
        }
}
```

Output :

```
concurrent thread started running..
```

To call the **run()** method, **start()** method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

7. Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run. There is no guarantee that which runnable thread will be chosen to run by the thread scheduler. Only one thread at a time can run in a single process. The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Difference between preemptive scheduling and time slicing
Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

8. Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.
Syntax: The Thread class provides two methods for sleeping a thread:

- public static void **sleep(long miliseconds)**throws InterruptedException
- public static void **sleep(long miliseconds, int nanos)**throws InterruptedException

Example:

```
class TestSleepMethod1 extends Thread{
        public void run(){
                for(int i=1;i<5;i++){
                        try{
                                Thread.sleep(500);
                        }catch(InterruptedException e){
                                System.out.println(e);
                        }
                        System.out.println(i);
                }
        }
        public static void main(String args[]){
                TestSleepMethod1 t1=new TestSleepMethod1();
                TestSleepMethod1 t2=new TestSleepMethod1();
                t1.start();
                t2.start();
        }
}
```

Output:

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

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

9. Naming Thread and Current Thread:

a) Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

Example:

```
class TestMultiNaming1 extends Thread{
        public void run(){
                System.out.println("running...");
        }
        public static void main(String args[]){
                TestMultiNaming1 t1=new TestMultiNaming1();
                TestMultiNaming1 t2=new TestMultiNaming1();
                System.out.println("Name of t1:"+t1.getName());
                System.out.println("Name of t2:"+t2.getName());
                t1.start();
                t2.start();
                t1.setName("Sonoo Jaiswal");
                System.out.println("After changing name of t1:"+t1.getName());
        }
}
```

Output:

```
Name of t1:Thread-0
Name of t2:Thread-1
id of t1:8
running...
After changing name of t1:
Sonoo Jaiswal
running...
```

b. Current Thread

The currentThread() method returns a reference of currently executing thread.

Syntax: **public static Thread currentThread()**

Example:

```
class TestMultiNaming2 extends Thread{
        public void run(){
                System.out.println(Thread.currentThread().getName());
        }
        public static void main(String args[]){
                TestMultiNaming2 t1=new TestMultiNaming2();
                TestMultiNaming2 t2=new TestMultiNaming2();
                t1.start();
                t2.start();
        }
}
```

Output:

```
Thread-0
Thread-1
```

c. How to perform single task by multiple threads?

```
class TestMultitasking1 extends Thread{
    public void run(){
            System.out.println("task one");
    }
    public static void main(String args[]){
            TestMultitasking1 t1=new TestMultitasking1();
            TestMultitasking1 t2=new TestMultitasking1();
            TestMultitasking1 t3=new TestMultitasking1();
            t1.start();
            t2.start();
            t3.start();
    }
}
```

Output:

```
task one
task one
task one
```

d. How to perform single task by multiple threads using Runnable interface?

```
class TestMultitasking2 implements Runnable{
    public void run(){
            System.out.println("task one");
    }
    public static void main(String args[]){
            Thread t1 =new Thread(new TestMultitasking2());
            //passing annonymous object of TestMultitasking2 class
            Thread t2 =new Thread(new TestMultitasking2());
            t1.start();
            t2.start();
    }
}
```

Output:

```
task one
task one
```

Note: Each thread run in a separate callstack.

e. How to perform multiple tasks by multiple threads (multitasking in multithreading)?

```
class Simple1 extends Thread{
    public void run(){
            System.out.println("task one");
    }
}

class Simple2 extends Thread{
    public void run(){
            System.out.println("task two");
    }
}

class TestMultitasking3{
    public static void main(String args[]){
            Simple1 t1=new Simple1();
            Simple2 t2=new Simple2();
            t1.start();
            t2.start();
    }
}
```

Output:

```
task one
task two
```

E. Synchronization

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to
1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization
1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.
1. Mutual Exclusive
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:
1. by synchronized method
2. by synchronized block
3. by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them. From Java 5 the package java.util.concurrent.locks contains several lock implementations.

a. Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
class Table{
    void printTable(int n){                         //method not synchronized
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){
                System.out.println(e);
            }
        }
    }
}
```

```
    class MyThread1 extends Thread{
        Table t;
        MyThread1(Table t){
                this.t=t;
        }
        public void run(){
                t.printTable(5);
        }
    }
    class MyThread2 extends Thread{
        Table t;
        MyThread2(Table t){
                this.t=t;
        }
        public void run(){
                t.printTable(100);
        }
    }
    class TestSynchronization1{
        public static void main(String args[]){
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
        }
    }
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

b. Java synchronized method

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Synchronized Keyword

To synchronize above program, we must *serialize* access to the shared **display()** method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display() method.

Syntax: **synchronized** *void* **display** (String msg)

Example:

```
class Table{
        synchronized void printTable(int n){        //synchronized method
                for(int i=1;i<=5;i++){
                                System.out.println(n*i);
                                try{
                                Thread.sleep(400);
                                }catch(Exception e){
                                System.out.println(e);
                        }
                }
        }
}
```

```
class MyThread1 extends Thread{
    Table t;
    MyThread1(Table t){
            this.t=t;
    }
    public void run(){
            t.printTable(5);
    }
}
class MyThread2 extends Thread{
    Table t;
    MyThread2(Table t){
            this.t=t;
    }
    public void run(){
            t.printTable(100);
    }
}
public class TestSynchronization2{
    public static void main(String args[]){
            Table obj = new Table();        //only one object
            MyThread1 t1=new MyThread1(obj);
            MyThread2 t2=new MyThread2(obj);
            t1.start();
            t2.start();
    }
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

c. Synchronized block in java

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method. **Points to remember for Synchronized block:**

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax: **synchronized (object reference expression) { //code block }**

Example:

```
class Table{
        void printTable(int n){
                synchronized(this){                    //synchronized block
                        for(int i=1;i<=5;i++){
                                System.out.println(n*i);
                                try{
                                Thread.sleep(400);
                                }catch(Exception e){
                                System.out.println(e);
                        }
                }
        }
    }
}
```

```
class MyThread1 extends Thread{
        Table t;
        MyThread1(Table t){
                this.t=t;
        }
        public void run(){
                t.printTable(5);
        }
}

class MyThread2 extends Thread{
        Table t;
        MyThread2(Table t){
                this.t=t;
        }
        public void run(){
                t.printTable(100);
        }
}
public class TestSynchronizedBlock1{
        public static void main(String args[]){
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
        }
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

d. Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object. **Problem without static synchronization:** Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem. **Example:** In this example we are applying **synchronized** keyword on the **static method** to perform static synchronization.

Example:

```
class Table{
        synchronized static void printTable(int n){
                for(int i=1;i<=10;i++){
                        System.out.println(n*i);
                        try{
                                Thread.sleep(400);
                        }catch(Exception e){
                                System.out.println(e);
                        }
                }
        }
}
```

```
class MyThread1 extends Thread{
        public void run(){
                Table.printTable(1);
        }
}

class MyThread2 extends Thread{
        public void run(){
                Table.printTable(10);
        }
}

class MyThread3 extends Thread{
        public void run(){
                Table.printTable(100);
        }
}

class MyThread4 extends Thread{
        public void run(){
                Table.printTable(1000);
        }
}

public class TestSynchronization4{
        public static void main(String t[]){
                MyThread1 t1=new MyThread1();
                MyThread2 t2=new MyThread2();
                MyThread3 t3=new MyThread3();
                MyThread4 t4=new MyThread4();
                t1.start();
                t2.start();
                t3.start();
                t4.start();
        }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

F. Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behavior and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

public void interrupt()	public static boolean interrupted()	public boolean isInterrupted()

Example:

```
class TestInterruptingThread1 extends Thread{
        public void run(){
                try{
                        Thread.sleep(1000);
                        System.out.println("task");
                }catch(InterruptedException e){
                        throw new RuntimeException("Thread interrupted..."+e);
                }
        }
        public static void main(String args[]){
                TestInterruptingThread1 t1=new TestInterruptingThread1();
                t1.start();
                try{
                        t1.interrupt();
                }catch(Exception e){
                        System.out.println("Exception handled "+e);
                }
        }
}
```

Output:

```
Exception in thread-0
java.lang.RuntimeException: Thread interrupted...
java.lang.InterruptedException: sleep interrupted
at A.run(A.java:7)
```

Example of interrupting a thread that doesn't stop working:

```
class TestInterruptingThread2 extends Thread{
        public void run(){
                try{
                        Thread.sleep(1000);
                        System.out.println("task");
                }catch(InterruptedException e){
                        System.out.println("Exception handled "+e);
                }
                System.out.println("thread is running...");
        }
        public static void main(String args[]){
                TestInterruptingThread2 t1=new TestInterruptingThread2();
                t1.start();
                t1.interrupt();
        }
}
```

Output:

```
Exception handled
java.lang.InterruptedException: sleep interrupted
thread is running...
```

Example of interrupting thread that behaves normally

```
class TestInterruptingThread3 extends Thread{
        public void run(){
                for(int i=1;i<=5;i++)
                        System.out.println(i);
        }
        public static void main(String args[]){
                TestInterruptingThread3 t1=new TestInterruptingThread3();
                t1.start();
                t1.interrupt();
        }
}
```

Output:

```
1
2
3
4
5
```

G. Stopping Thread:

This noncompliant code example shows a thread that fills a vector with pseudorandom numbers. The thread is forcefully stopped after a given amount of time.

Example:

```
public final class Container implements Runnable {
        private final Vector<Integer> vector = new Vector<Integer>(1000);
        public Vector<Integer> getVector() {
                return vector;
        }
        @Override public synchronized void run() {
                Random number = new Random(123L);
                int i = vector.capacity();
                while (i > 0) {
                        vector.add(number.nextInt(100));
                        i--;
                }
        }
        public static void main(String[] args) throws InterruptedException {
                Thread thread = new Thread(new Container());
                thread.start();
                Thread.sleep(5000);
                thread.stop();
        }
}
```

H. Thread Exceptions

- **SecurityException** - if the current thread cannot create a thread in the specified thread group
- **IllegalArgumentException** - if the value of millis is negative
- **InterruptedException** - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.
- **IllegalThreadStateException** - if the thread was already started.
- **NullPointerException** - if obj is null

I. Thread Priority

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Three constants defined in Thread class:

public static int MIN_PRIORITY	**public static int NORM_PRIORITY**	**public static int MAX_PRIORITY**

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example:

```
class TestMultiPriority1 extends Thread{
        public void run(){
             System.out.println("running thread name is:"+Thread.currentThread().getName());

System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
        }
        public static void main(String args[]){
                  TestMultiPriority1 m1=new TestMultiPriority1();
                  TestMultiPriority1 m2=new TestMultiPriority1();
                  m1.setPriority(Thread.MIN_PRIORITY);
                  m2.setPriority(Thread.MAX_PRIORITY);
                  m1.start();
                  m2.start();
         }
}
```

Output:

```
running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

J. Daemon Thread in Java

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically. There are many java daemon threads running automatically e.g. gc, finalizer etc. You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.
Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

No.	Method	Description
1)	**public void setDaemon(boolean status)**	is used to mark the current thread as daemon thread or user thread.
2)	**public boolean isDaemon()**	is used to check that current is daemon.

Example:

```
public class TestDaemonThread1 extends Thread{
        public void run(){
                if(Thread.currentThread().isDaemon()){
                        //checking for daemon thread
                        System.out.println("daemon thread work");
                }
                else{
                        System.out.println("user thread work");
                }
        }
        public static void main(String[] args){
                //creating thread
                TestDaemonThread1 t1=new TestDaemonThread1();
                TestDaemonThread1 t2=new TestDaemonThread1();
                TestDaemonThread1 t3=new TestDaemonThread1();
                t1.setDaemon(true);             //now t1 is daemon thread
                t1.start();             //starting threads
                t2.start();
                t3.start();
        }
}
```

Output
```
daemon thread work
user thread work
user thread work
```

Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

```
class TestDaemonThread2 extends Thread{
        public void run(){
                System.out.println("Name: "+Thread.currentThread().getName());
                System.out.println("Daemon: "+Thread.currentThread().isDaemon());
        }
        public static void main(String[] args){
                TestDaemonThread2 t1=new TestDaemonThread2();
                TestDaemonThread2 t2=new TestDaemonThread2();
                t1.start();
                t1.setDaemon(true);     //will throw exception here
                t2.start();
        }
}
```

Output:
```
exception in thread main:
java.lang.
IllegalThreadStateException
```

Exception-Handling:

A. Introduction

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained. In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions. **What is exception?: Dictionary Meaning:** Exception is an abnormal condition. In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. **What is exception handling?:** Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

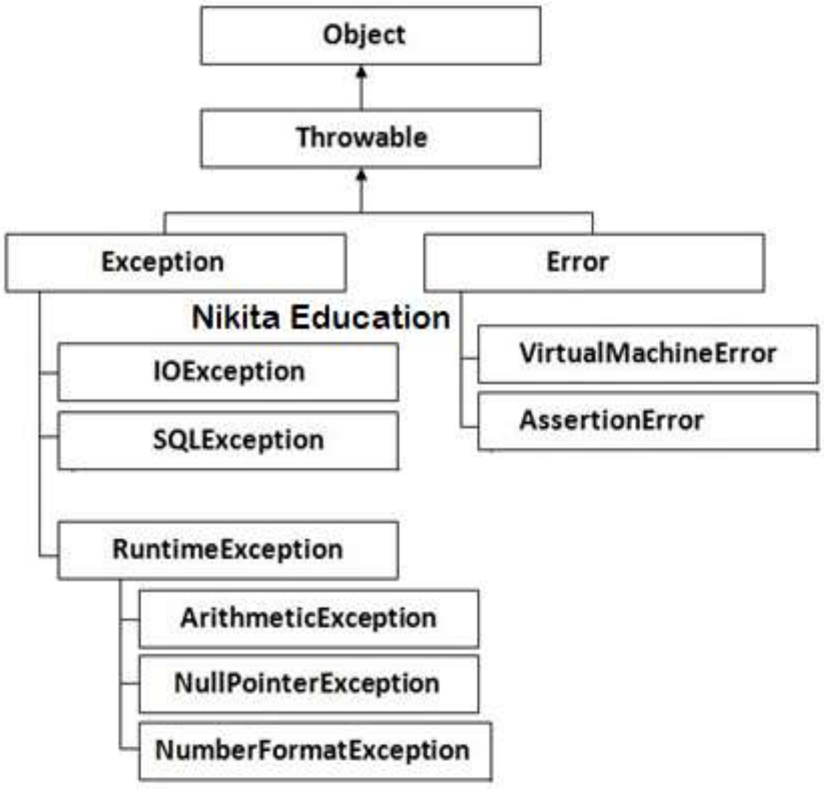
Advantage of Exception Handling:

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

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

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

Hierarchy of Java Exception classes



B. Types of Exception

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

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

C. Common scenarios where exceptions may occur

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

1) Scenario where ArithmeticException occurs

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

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

2) Scenario where NullPointerException occurs

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

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

3) Scenario where NumberFormatException occurs

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

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

4) Scenario where ArrayIndexOutOfBoundsException occurs

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

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

D. Java Exception Handling Keywords / Exception Handling Mechanism

There are 5 keywords used in java exception handling.

1. try
2. catch

3. finally
4. throw
5. throws

E. Java try-catch

1. Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

Syntax:

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

Syntax:

```
try{
        //code that may throw exception
}finally{
        //important code which must be execute
}
```

2. Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try.

i. Problem without exception handling

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

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

Output:

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

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed). There can be 100 lines of code after exception. So all the code after exception will not be executed.

ii. Solution by exception handling

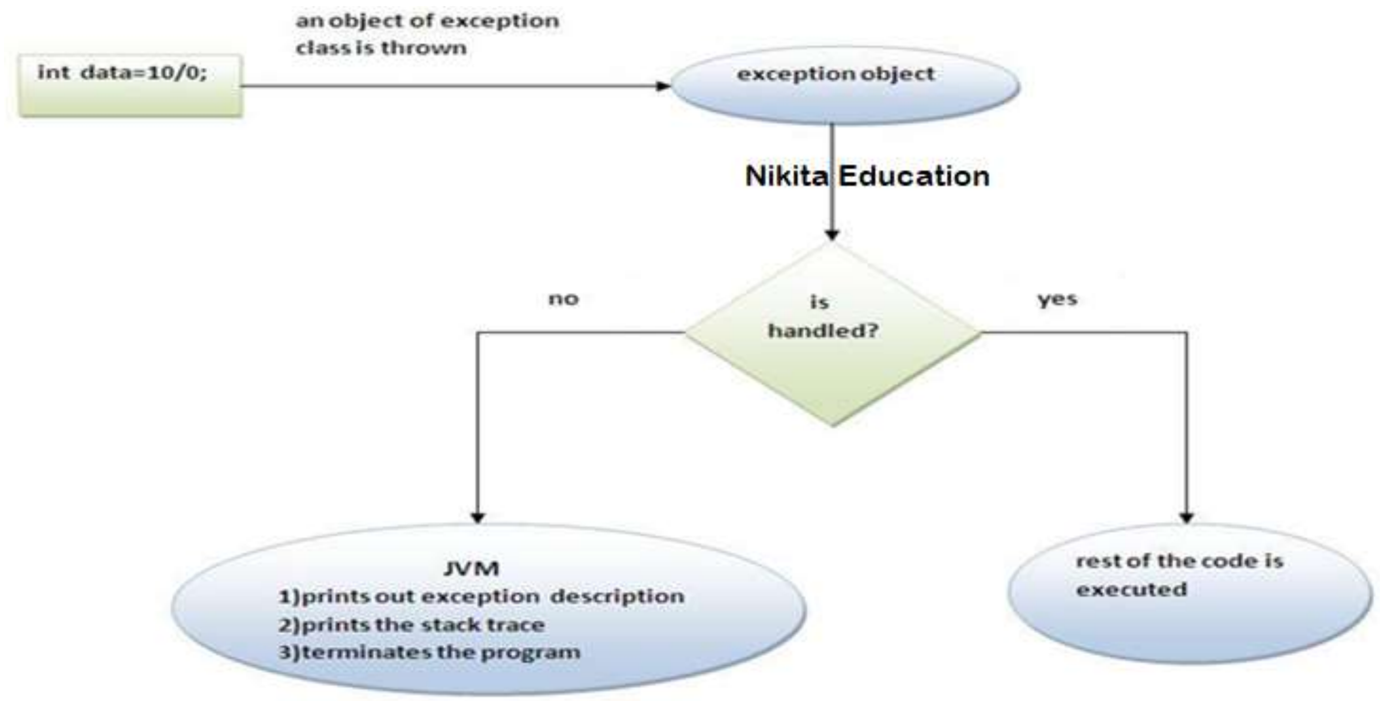
```
public class Testtrycatch2{
        public static void main(String args[]){
                try{
                        int data=50/0;
                }catch(ArithmeticException e){
                        System.out.println(e);
                }
```

Output:

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

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

iii. Internal working of java try-catch block



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

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

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

2.1. Java multiple catch / Java Multi catch block

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

Example:

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

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

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

Rule: At a time only one Exception is occurred and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception.

Ex**ample:**

```
class TestMultipleCatchBlock1{
	public static void main(String args[]){
		try{
			int a[]=new int[5];
			a[5]=30/0;
		}
		catch(Exception e){
			System.out.println("common task completed");
		}
		catch(ArithmeticException e){
			System.out.println("task1 is completed");
		}
		catch(ArrayIndexOutOfBoundsException e){
			System.out.println("task 2 completed");
		}
		System.out.println("rest of the code...");
	}
}
```

Output:

```
Compile-time error
```

2.2. Java Nested try block

The try block within a try block is known as nested try block in java. **Why use nested try block:** Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

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

Example:
```
class Excep6{
        public static void main(String args[]){
                try{
                        try{
                                System.out.println("going to divide");
                                int b =39/0;
                        }catch(ArithmeticException e){
                                System.out.println(e);
                        }
                        try{
                                int a[]=new int[5];
                                a[5]=4;
                        }catch(ArrayIndexOutOfBoundsException e){
                                System.out.println(e);
                        }
                        System.out.println("other statement);
                }catch(Exception e){
                        System.out.println("handeled");
                }
                System.out.println("normal flow..");
        }
}
```

3. Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.
Java finally block is always executed whether exception is handled or not. Java finally block must be followed by try or catch block.

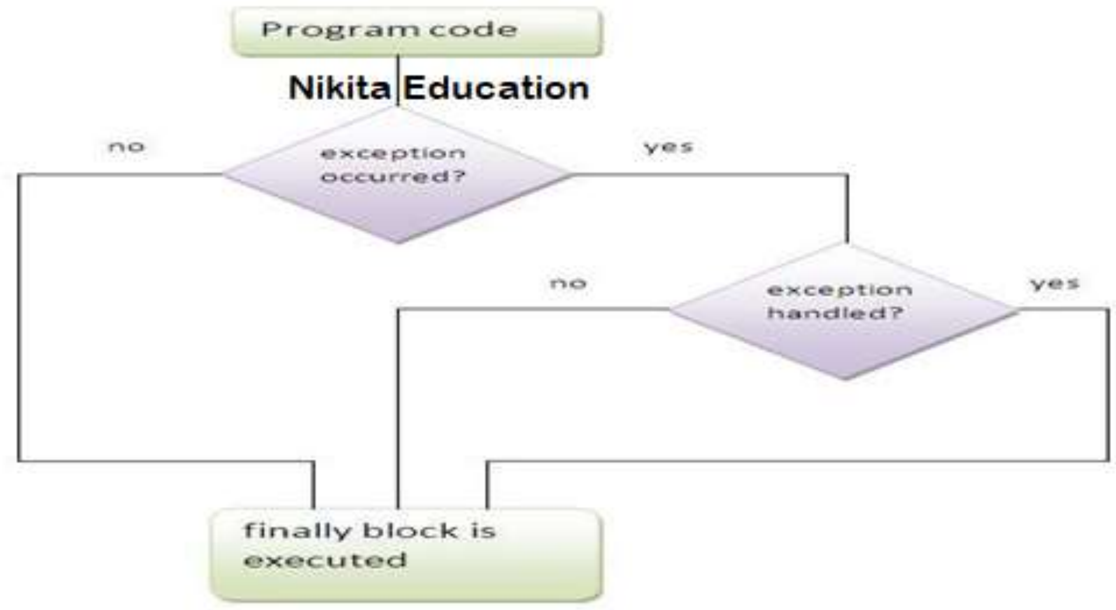
Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

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



Case 1: Let's see the java finally example where **exception doesn't occur**.

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

Output:
```
5
finally block is always executed
rest of the code...
```

Case 2: Let's see the java finally example where **exception occurs and not handled**.

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

Output:
```
finally block is always executed
Exception in thread main
java.lang.ArithmeticException:/ by zero
```

Case 3: Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2{
        public static void main(String args[]){
                try{
                        int data=25/0;
                        System.out.println(data);
                }catch(ArithmeticException e){
                        System.out.println(e);
                }finally{
                        System.out.println("finally block is always executed");
                }
                System.out.println("rest of the code...");
        }
}
```

Output:
```
Exception in thread main
java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

F. Java throw keyword

The Java throw keyword is used to explicitly (manually) throw an exception. We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

Syntax: **throw** exception; **e.g. :** **throw new IOException**("sorry device error);

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

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

G. Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained. Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax: **return_type method_name() throws exception_class_name{**
```
return_type method_name() throws exception_class_name{
        //method code
}
```

Which exception should be declared with throws?

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

Example:
```
import java.io.IOException;
class Testthrows1{
        void m()throws IOException{
                throw new IOException("device error");  //checked exception
        }
        void n()throws IOException{
                m();
        }
        void p(){
                try{
                        n();
                }catch(Exception e){
                        System.out.println("exception handled");
                }
        }
        public static void main(String args[]){
                Testthrows1 obj=new Testthrows1();
                obj.p();
                System.out.println("normal flow...");
        }
}
```

Output:
```
exception handled
normal flow...
```

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

Case1: You handle the exception

```
import java.io.*;
class M{
        void method()throws IOException{
                throw new IOException("device error");
        }
}
public class Testthrows2{
        public static void main(String args[]){
                try{
                        M m=new M();
                        m.method();
                }catch(Exception e){
                        System.out.println("exception handled");
                }
                System.out.println("normal flow...");
        }
}
```

```
Output:
exception handled
normal flow...
```

Case2: You declare the exception

In case you declare the exception, if exception does not occur, the code will be executed fine.
In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

a. if exception not occurs

```
import java.io.*;
class M{
        void method()throws IOException{
                System.out.println("device operation performed");
        }
}
class Testthrows3{
        public static void main(String args[])throws IOException{//declare exception
                M m=new M();
                m.method();
                System.out.println("normal flow...");
        }
}
```

```
Output:
device operation performed
normal flow...
```

b. if exception occurs

```
import java.io.*;
class M{
        void method()throws IOException{
                throw new IOException("device error");
        }
}
class Testthrows4{
        public static void main(String args[])throws IOException{//declare exception
                M m=new M();
                m.method();
                System.out.println("normal flow...");
        }
}
```

```
Output:
Runtime Exception
```

H. Difference between throw and throws in Java

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

1. Java throw example:

```
void m(){
        throw new ArithmeticException("sorry");
}
```

2. Java throws example:

```
        void m()throws ArithmeticException{
        //method code
}
```

3. Java throw and throws example:

```
        void m()throws ArithmeticException{
        throw new ArithmeticException("sorry");
}
```

I. Difference between final, finally and finalize

final	finally{}	finalize()
Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
Final is a keyword.	Finally is a block.	Finalize is a method.
Java final example:	**Java finally example:**	**Java finalize example:**

class FinalExample{
public static void main(String[] args){
final int x=100;
	class FinallyExample{
	public static void main(String[] args){
	try{
		class FinalizeExample{
		public void finalize(){
		System.out.println("finalize called");

x=200;
//Compile Time Error
}
}
	int x=300;
	}catch(Exception e){
	System.out.println(e);
	}**finally{**
	System.out.println("finally block i s executed");
	}
	}
	}
		}
		public static void main(String[] args){
		FinalizeExample f1=new Finaliz eExample();
		FinalizeExample f2=new Finaliz eExample();
		f1=null;
		f2=null;
		System.gc();.
		}}

J. Java Custom Exception / User defined exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need. By the help of custom exception, you can have your own exception and message.

Example:

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

class TestCustomException1{
        static void validate(int age)throws InvalidAgeException{
                if(age<18)
                        throw new InvalidAgeException("not valid");
                else
                        System.out.println("welcome to vote");
        }
        public static void main(String args[]){
                try{
                        validate(13);
                }catch(Exception m){
                        System.out.println("Exception occured: "+m);
                }
                System.out.println("rest of the code...");
        }
}
```

Output:

```
Exception occured:
InvalidAgeException:not valid
rest of the code...
```

Unit-V
Graphics & Internet Programming

Unit	Topic Name	Main Topic	Subtopics	Marks
5	Graphics & Internet Programming	Applet	Applet Class, Applet Architecture, Local and remote applets, How applet differ from application, Preparing to write applets, Building applet code, Applet life cycle, Applet tag, Adding Applet to HTML file, Running the Applet, Passing parameter to applet.	14
		Graphics	The Graphics Class, Lines and rectangle, Circle and Ellipse, Drawing Arcs, Drawing Polygons, Line Graphs, working with Color, Color methods, working with Fonts, Font Metrics, Determining available Fonts.	

Internet Programming (Applet)

A. Java Applet:

Applet is a special type of java program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

- Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as apart of a web document. Any applet in Java is a class that extends the **java.applet.Applet** class.
- An Applet class does not have any main() method.
- It is viewed using JVM. The JVM can use either a plug-in of the Web browser or a separate runtime environment to run an applet application.
- JVM creates an instance of the applet class and invokes **init()** method to initialize an Applet.

Advantage of Applet

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plug-in is required at client browser to execute applet.

B. The Applet Class:

Every applet is an extension of the ***java.applet.Applet*** *class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context. These include methods that do the following:

• Get applet parameters	• Fetch an image	• Fetch an audio clip
• Play an audio clip	• Resize the applet	• Print a status message in the browser
• Get the network location of the HTML file that contains the applet		
• Get the network location of the applet class directory		

java.applet.Applet:

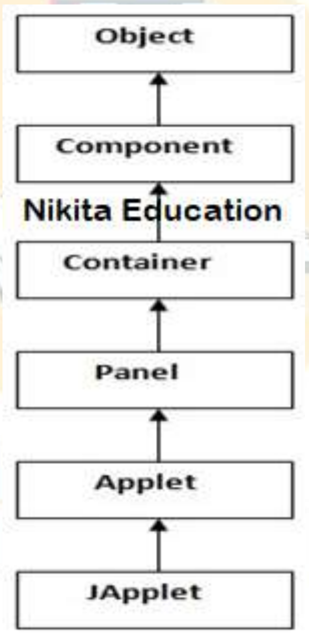
The **java.applet.Applet** class has **4 life cycle methods** and **java.awt.Component** class provides **1 life cycle methods** for an applet. For creating any applet java.applet.Applet class must be inherited.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

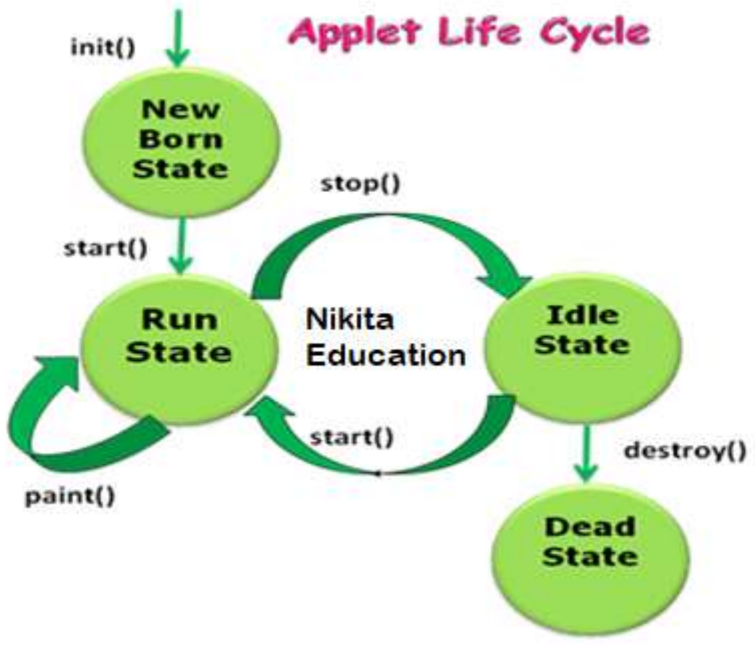
1. **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.



C. Applet Life Cycle

Applet runs in the browser and its lifecycle method are called by JVM at its birth, its death and when it is momentarily away. Every Applet can be said to be any of the following state

1. New Born state
2. Running state
3. Idle state (may or may not)
4. Dead state



1. New Born State

- The life cycle of an applet is begin on that time when the applet is first loaded into the browser and called the init() method.
- The init() method is called only one time in the life cycle on an applet.
- The init() method is basically called to read the PARAM tag in the html file.
- The init () method retrieve the passed parameter through the PARAM tag of html file using get Parameter() method .
- All the initialization such as initialization of variables and the objects like image, sound file are loaded in the init () method .
- After the initialization of the init() method user can interact with the Applet

Syntax:

```
public void init(){

        //Statements

}
```

2. Running State

- After initialization, this state will automatically occur by invoking the start method of applet class which again calls the run method and which calls the paint method.
- The running state also occurs from idle state when the applet is reloaded.
- This method may be called multiples time when the Applet needs to be started or restarted.
- For Example if the user wants to return to the Applet, in this situation the start Method() of an Applet will be called by the web browser and the user will be back on the applet.
- In the start method user can interact within the applet.

Syntax:

```
public void start(){

        //Statements

}
```

3. Idle State

- The idle state will make the execution of the applet to be halted temporarily.
- Applet moves to this state when the currently executed applet is minimized or when the user switches over to another page.
- At this point the stop method is invoked.
- From the idle state the applet can move to the running state.
- The stop() method can be called multiple times in the life cycle of applet Or called at least one time.
- For example the stop() method is called by the web browser on that time When the user leaves one applet to go another applet

Syntax:

```
public void stop(){

        //Statements

}
```

4. Dead State

- When the applet programs terminate, the destroy function is invoked which makes an applet to be in dead state.
- The destroy() method is called only one time in the life cycle of Applet like init() method.

Syntax:

```
public void destroy(){

        //Statements

}
```

5. Display State

- The applet is said to be in display state when the paint method is called.
- This method can be used when we want to display output in the screen.
- This method can be called any number of times.
- paint() method is must in all applets when we want to draw something on the applet window.
- paint() method takes Graphics object as argument

Syntax:

```
public void paint(Graphics g){

        //Statements

}
```

Who is responsible to manage the life cycle of an applet?
Java Plug-in software.

D. A Simple Applet

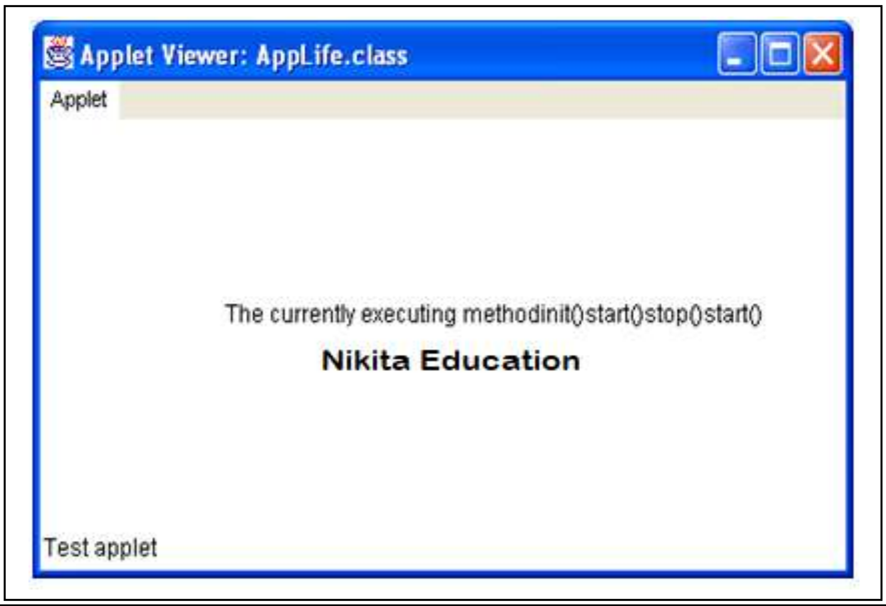
Example:

```
import java.awt.*;
import java.applet.*;
public class Simple extends Applet{
        public void paint(Graphics g){
                g.drawString("A simple Applet", 20, 20);
        }
}
```

Example:

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

/* <applet code="AppLife.class" width="200" height="200">
   </applet> */

public class AppLife extends Applet{

        String msg="The currently executing method";

        public void init(){
                msg+="init()";
        }

        public void start(){
                msg+="start()";
        }

        public void stop(){
                msg+="stop()";
        }

        public void paint(Graphics g){
                g.drawString(msg,100,100);
                showStatus("Test applet");
        }
}
```

Output:



```
The currently executing methodinit()start()stop()start()
```

```
Test applet
```

E. How to run an Applet Program

An Applet program is compiled in the same way as you have been compiling your console programs. However there are two ways to run an applet.

- Executing the Applet within Java-compatible web browser.
- Using an Applet viewer, such as the standard tool, appletviewer. An applet viewer executes your applet in a window

For executing an Applet in an web browser, create short HTML file in the same directory. Inside body tag of the file, include the following code. (applet tag loads the Applet class)

< applet code = "MyApplet.class" width=400 height=400 >< /applet >

a. There are two ways to run an applet

1. By html file.
2. By appletviewer tool (for testing purpose).

HTML <applet> Tag

```
<applet code="Bubbles.class" width="350" height="350">
        Java applet that draws animated bubbles.
</applet>
```

HTML <param> Tag

```
<applet code="Bubbles.class" width="350" height="350">
        <param name="nameofparameter" value="valueofparameter">
</applet>
```

1. Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
        public void paint(Graphics g){
                g.drawString("welcome",150,150);
        }
}
```

Note: class must be public because its object is created by Java Plugin software that resides on the browser.

myapplet.html:

```
<html>
        <body>
                <applet code="First.class" width="300" height="300">
                </applet>
        </body>
</html>
```

2. Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

First.java:

```
import java.applet.Applet;
import java.awt.Graphics;

/* <applet code="First.class" width="300" height="300">
</applet> */

public class First extends Applet{
        public void paint(Graphics g){
                g.drawString("welcome to applet",150,150);
        }
}
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java

c:\>appletviewer First.java
```

F. Applet Vs. Application

Applet	Application
Small Program	Large Program
Used to run a program on client Browser	Can be executed on standalone computer system
Applet is portable and can be executed by any JAVA supported browser.	Need JDK, JRE, JVM installed on client machine.
Applet applications are executed in a Restricted Environment	Application can access all the resources of the computer
Applets are created by extending the java.applet.Applet	Applications are created by writing public static void main(String[] s) method.
Applet application has 5 methods which will be automatically invoked on occurrence of specific event	Application has a single start point which is main method

Example:
import java.awt.*;
import java.applet.*;
public class Myclass **extends Applet**
{
   public void init() { }
   public void start() { }
   public void stop() {}
   public void destroy() {}
   public void paint(Graphics g) {}
}
	public class MyClass
	{
	        public static void main(String args[]) {
	        }
	}

1. Advantages of Applets in Applets Vs Applications

1. Execution of applets is easy in a Web browser and does not require any installation or deployment procedure in real-time programming (where as servlets require).
2. Writing and displaying (just opening in a browser) graphics and animations is easier than applications.
3. In GUI development, constructor, size of frame, window closing code etc. are not required (but are required in applications).

2. Restrictions of Applets in Applets Vs Applications

1. Applets are required separate compilation before opening in a browser.
2. In real-time environment, the bytecode of applet is to be downloaded from the server to the client machine.
3. Applets are treated as un-trusted (as they were developed by unknown people and placed on unknown servers whose trustworthiness is not guaranteed) and for this reason they are not allowed, as a security measure, to access any system resources like file system etc. available on the client system.
4. Extra Code is required to communicate between applets using AppletContext.

G. Types of Applet

1. Local Applets

Local applets are applet types that are developed and stored in local system. The web page will search the local system directories, find the local applet and execute it. Execution of local applet does not require internet connection. Specifying a Local Applet:

<applet codebase="path" code="NewApplet.class" width=120 height=120 >
</apple>

In the above listing , the codebase attribute specifies a path name on your system for the local applet, whereas the code attribute specifies the name of the byte-code file that contains the applet's code. The path specified in the codebase attribute is relative to the folder containing the HTML document that references the applet.

2. Remote Applets

Remote applets are applet types that are developed and stored in remote system. The web page requires internet connection to locate and load the remote applet from remote computer. Specifying a Remote Applet

<applet codebase="http://www.gpn.com/applets/" code="NewApplet.class" width=120 height=120 >
</applet>

The only difference between *Local Applet* and *Remote Applet* is the value of the codebase attribute. In the first case, codebase specifies a local folder, and in the second case, it specifies the URL at which the applet is located.

H. Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

Syntax:

```
public String getParameter(String parameterName)
```

Example:

```
import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet{
        public void paint(Graphics g){
                String str=getParameter("msg");
                g.drawString(str,50, 50);
        }
}
```

myapplet.html:

```
<html>
<body>
        <applet code="UseParam.class" width="300" height="300">
                <param name="msg" value="Welcome to GPN">
        </applet>
</body>
</html>
```

Graphics Programming

A. Graphics Class:

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a graphics context. A ***graphics context*** is encapsulated by the **Graphics** class and is obtained in two ways:

- It is passed to an applet when one of its various methods, such as **paint()** or **update(),** is called.
- It is returned by the **getGraphics()** method of Component.

1. Commonly used methods of **java.awt.Graphics** class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public abstract void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

2. Example:

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo extends Applet{
        public void paint(Graphics g){
                g.setColor(Color.red);
                g.drawString("Welcome",50, 50);
                g.drawLine(20,30,20,300);
                g.drawRect(70,100,30,30);
                g.fillRect(170,100,30,30);
                g.drawOval(70,200,30,30);
                g.setColor(Color.pink);
                g.fillOval(170,200,30,30);
                g.drawArc(90,150,30,30,30,270);
                g.fillArc(270,150,30,30,0,180);
        }
}
```

myapplet.html:

```
<html>
<body>
        <applet code="GraphicsDemo.class" width="300" height="300">
        </applet>
</body>
</html>
```

3. Example:

```
import java.awt.*;
import java.awt.event.*;
public class GraphicsMethods extends Frame{
        public GraphicsMethods(){
                setTitle("Frame Window");
                setSize(600,600); setVisible(true);
                addWindowListener(new WindowAdapter(){
                        public void windowClosing(WindowEvent we){
                                System.exit(0);
                        }
                });
        }
        public void paint(Graphics g)       {
                g.drawString("Graphics Objects",50,50);
                g.drawLine(50, 60, 550, 60);
                g.drawRect(50, 80, 100, 100);
                g.fillRect(50, 200, 100, 100);
                g.drawRoundRect(50, 320, 100, 100, 20, 20);
                g.fillRoundRect(350, 80, 100, 100, 20, 20);
                g.drawOval(350, 200, 100, 100);
                g.fillOval(350, 320, 100, 100);
                g.drawArc(170, 80, 300, 300, 90, 45);
                g.fillArc(170, 200, 200, 200, 90, 45);
                int [] x={500,550,530,580,480};
                int [] y={300,250,330,210,150};
                g.drawPolygon(x,y,5);
                int [] y1={500,550,530,580,480};
                int [] x1={300,250,330,210,150};
                g.fillPolygon(x1,y1,5);
        }
        public static void main(String [] args){
                GraphicsMethods newFrame=new GraphicsMethods();
        }
}
```

Introduction to AWT Classes: Color, Font, FontMetrics

1. Color

Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet. Color is encapsulated by the **Color** class.

D.1. Constants of Color class:

- Color.black, Color.red, Color.green, Color.blue, Color.pink, Color.orange, Color.yellow

D.2. Important Constructors:

▪ Color(int, int, int)	// int value in between 0 – 255
▪ Color(int rgbValue)	// combined value of RGB
▪ Color(float, float, float)	// float value in between 0.0 – 1.0

D.3. Important Methods:

- Color brighter() - Creates a new Color that is a brighter version of this Color.
- Color darker() - Creates a new Color that is a darker version of this Color.
- Int getBlue() - Returns the blue component in the range 0-255 in the default sRGB space.
- Int getGreen() - Returns the green component in the range 0-255 in the default sRGB space.
- Int getRed() - Returns the red component in the range 0-255 in the default sRGB space.
- Int getRGB() - Returns the RGB value representing the color in the default sRGB ColorModel.

D.4. Setting the current Graphics color

- void setColor(Color *newColor*)
- Color getColor( )

D.5. Setting the foreground and background of Frame or Component

- Color getBackground() - Gets the background color of this component.
- Color getForeground() - Gets the foreground color of this component.
- Void setBackground(Color c) - Sets the background color of this component.
- Void setForeground(Color c) - Sets the foreground color of this component.

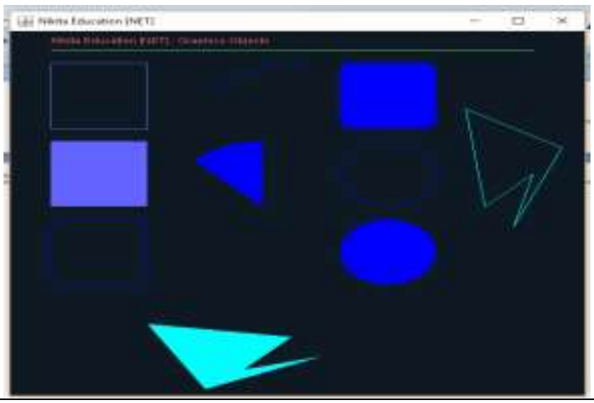
D.6. Example:

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

/* <applet code="ColorDemo.class" width="300" height="300">
</applet> */
```

```
public class ColorDemo extends Applet
{
     public void init()
     {
          Color cl1=new Color(10,50,223);
          Color cl2=new Color(15,25,35);
          setForeground(cl1);
          setBackground(cl2);
     }

     public void paint(Graphics g)
     {
          Color c1 = new Color(255, 100, 100);
          Color c2 = new Color(100, 255, 100);
          Color c3 = new Color(100, 100, 255);

          g.setColor(c1);
          g.drawLine(10, 10, 100, 10);
          g.drawLine(10, 15, 100, 15);

          g.setColor(c2);
          g.drawLine(10, 20, 100, 20);
          g.drawLine(10, 25, 100, 25);

          g.setColor(c3);
          g.drawLine(10, 30, 100, 30);
          g.drawLine(10, 35, 100, 35);

          g.setColor(Color.red);
          g.drawOval(10, 50, 100, 100);
          g.fillOval(120, 50, 100, 100);

          g.setColor(Color.blue);
          g.drawOval(10, 160, 100, 80);
          g.drawRect(120, 160, 100, 80);

          g.setColor(Color.cyan);
          g.fillRect(10, 270, 100, 100);
          g.drawRoundRect(190, 10, 60, 50, 15, 15);
     }
}
```

Output:

```
cmd>javac ColorDemo.java
cmd>appletviewer ColorDemo.java
```



2. Font

The AWT supports **multiple type fonts.** The AWT provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts. Fonts have a **family name**, a **logical font name**, and a **face name**. Fonts are encapsulated by the **Font** class

1. The *family name* is the **general name** of the font, such as Courier.
2. The *logical name* specifies **a category of f**ont, such as Monospaced.
3. The *face name* specifies **a specific font**, such as Courier Italic.

E.1. Important Constructor

Font(String name, int style, int size) - Creates a new Font from the specified name, style and point size.

E.2. Important Methods

Modifier and Type	**Method and Description**
String	**getFamily**() Returns the family name of this Font.
String	**getFontName**() Returns the font face name of this Font.
String	**getName**() Returns the logical name of this Font.
int	**getSize**() Returns the point size of this Font, rounded to an integer.
int	**getStyle**() Returns the style of this Font.
boolean	**isBold**() Indicates whether or not this Font object's style is BOLD.
boolean	**isItalic**() Indicates whether or not this Font object's style is ITALIC.
boolean	**isPlain**() Indicates whether or not this Font object's style is PLAIN.

E.3. Setting or Obtaining Font

Font getFont() - Gets the font of this component.
Void setFont(Font f) - Sets the font of this component.

E.4. Determining Available Fonts:

```
import java.awt.*;
class AvailableFonts{
     public static void main(String[] args){

          GraphicsEnvironment ge=
                    GraphicsEnvironment.getLocalGraphicsEnvironment();

          String [] list = ge.getAvailableFontFamilyNames();

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

E.5. Example:

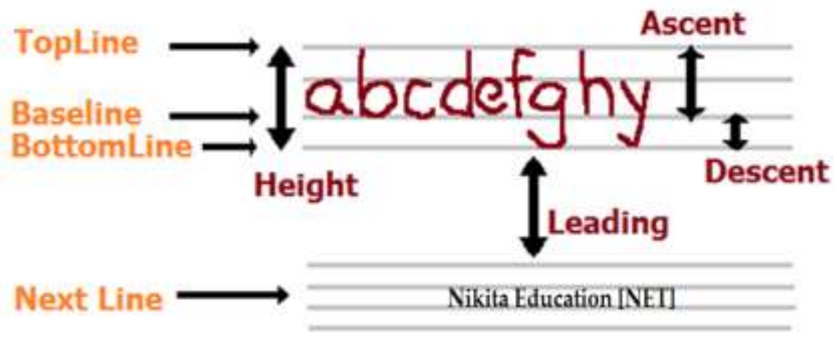
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/* <applet code="FontDemo.class" width="300" height="300">
</applet> */

public class FontDemo extends Applet
{
      Font F;
      public void init()
      {
            F=new Font("Dialog",Font.BOLD, 17);
            setFont(F);

            Label l1=new Label("This is a Dialog Font");
            add(l1);
      }
      public void paint(Graphics g)
      {
            String msg="This is a Serif Font";
            Font f1=new Font("Serif",Font.BOLD | Font.ITALIC,28);
            g.setFont(f1);
            g.drawString(msg,100,300);
      }
}
```

3. FontMetrics

For most fonts, characters are not all the same **dimension**—most fonts are proportional. Also, the height of each character, the length of ***descenders*** (the hanging parts of letters, such as *y*), And the amount of **space** between **horizontal lines** vary from font to font. Further, the **point size** of a font can be **changed.** Hence we need to obtain font information for managing text outputs. To fulfill above need AWT provides us **FontMetrics** class. **FontMetrics** class encapsulates information about a font.

F.1. The common terminology used when describing fonts:

1. **Height** The top-to-bottom size of a line of text
2. **Baseline** The line that the bottoms of characters are aligned to (not counting descent)
3. **Ascent** The distance from the baseline to the top of a character
4. **Descent** The distance from the baseline to the bottom of a character
5. **Leading** The distance between the bottom of one line of text and the top of the next



To get above information **FontMetrics** class is used

F.2. Obtaining FontMetrics object:

FontMetrics doesn't provide visible constructor hence we need to obtain its object using following way.

- FontMetrics FM=getFontMetrics(Font F); OR
- FontMetrics FMS=g.getFontMetrics();

F.3. Methods of FontMetrics

a. int getAscent( ) - Determines the *font ascent* of the Font described by this FontMetrics object.
b. int getDescent( ) - Determines the *font descent* of the Font described by this FontMetrics object.
c. int getHeight( ) - Gets the standard height of a line of text in this font.
d. int getLeading( ) - Determines the *standard leading* of the Font described by this FontMetrics object.
e. int getMaxAscent( ) - Determines the maximum ascent of the Font described by this FontMetrics object.
f. int getMaxDescent() - Determines the maximum descent of the Font described by this FontMetrics object.

F.4. Example:

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

/* <applet code="FontMetricsDemo.class" width="300" height="300">
</applet> */

class FontMetricsDemo extends Applet
{
     public void init()
     {
          Font f=new Font("Courier New", Font.BOLD, 20);
          setFont(f);
     }
     public void paint(Graphics g)
     {
          FontMetrics fm = g.getFontMetrics();
          int asc  = fm.getAscent();
          int des = fm.getDescent();
          int led = fm.getLeading();
          int hgt  = fm.getHeight();

          g.drawString("FontMetrics for Courier New Font",50,70);
          g.drawString("ascent : "+Integer.toString(asc),100,100);
          g.drawString("descent : "
                         + Integer.toString(des),100,140);
          g.drawString("leading : "
                         + Integer.toString(led),100,180);
          g.drawString("height : "+Integer.toString(hgt),100,220);
     }
}
```

Unit-VI
File I/O & Collection Framework

Unit	Topic Name	Main Topic	Subtopics	Marks
6	File I/O & Collection Framework	File I/O	Stream Classes - Character Stream, Byte Stream. Using Stream I/O, Serialization.	12
		Collection Framework	Introduction to collections framework, Array List, LinkedList, HashSet class, using Iterator, Map class. Date, Calendar, Random.	

File I/O

A. Java I/O and Streams:

Java I/O (Input and Output) is used to process the input and produce the output based on the input. Java uses the concept of stream to make I/O operation fast. The **java.io** package contains all the classes required for input and output operations. We can perform **file handling in java** by java IO API. Java performs I/O through **Streams**. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output.

- **Stream**

  A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow. In java, 3 streams are created for us automatically. All these streams are attached with console.

  1) System.out: standard output stream
  2) System.in: standard input stream
  3) System.err: standard error stream

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

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

Let's see the code to get **input** from console.

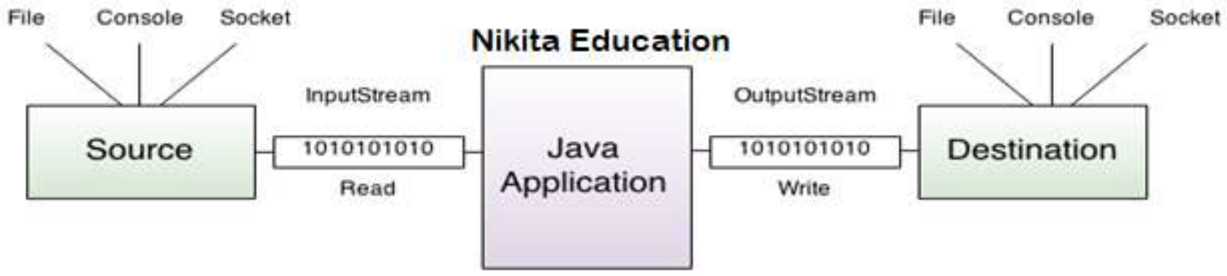
1. int i=System.in.read();//returns ASCII code of 1st character
2. System.out.println((char)i);//will print the character

1. OutputStream

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

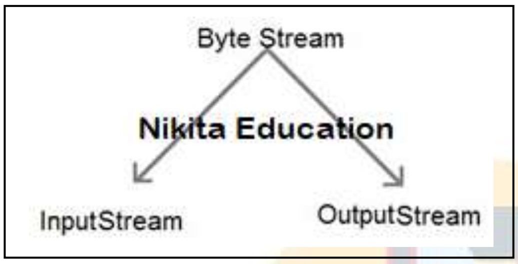
2. InputStream

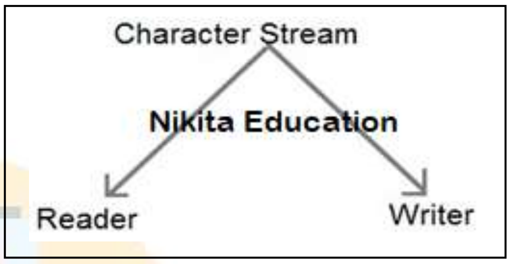
Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.



Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

1. **Byte Stream :** It provides a convenient means for handling input and output of byte.
2. **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.





1. Byte Stream Classes

Byte stream is defined by using two abstract class at the top of hierarchy, they are **InputStream** and **OutputStream**. These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc. **Some important Byte stream classes.**

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

These classes define several key methods. Two most important are

1. **read()** : reads byte of data.
2. **write()** : Writes byte of data.

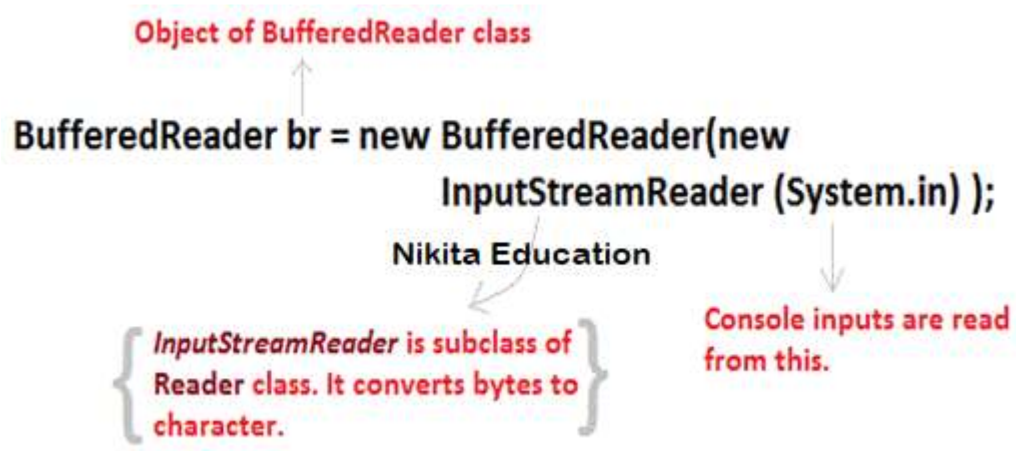
2. Character Stream Classes

Character stream is also defined by using two abstract class at the top of hierarchy, they are **Reader** and **Writer**. These two abstract classes have several concrete classes that handle unicode character. **Some important Character stream classes.**

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

3. Reading Console Input:

We use the object of BufferedReader class to take inputs from the keyboard.


Object of BufferedReader class
BufferedReader br = new BufferedReader(new
InputStreamReader (System.in));
Nikita Education
InputStreamReader is subclass of Reader class. It converts bytes to character.
Console inputs are read from this.

3.1. Reading Characters:

read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

Syntax: *int* **read()** throws **IOException**

Example:

```
class CharRead{
        public static void main( String args[]){
                BufferedReader br = new Bufferedreader(new InputStreamReader(System.in));
                //Reading character
                char c = (char)br.read();
        }
}
```

3.2. Reading Strings:

To read string we have to use **readLine()** function with BufferedReader class's object.

Syntax: *String* **readLine()** throws **IOException**

Example:
```
import java.io.*;
class MyInput{
        public static void main(String[] args){
                String text;
                InputStreamReader isr =
                                new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr);
                text = br.readLine();        //Reading String
                System.out.println(text);
        }
}
```

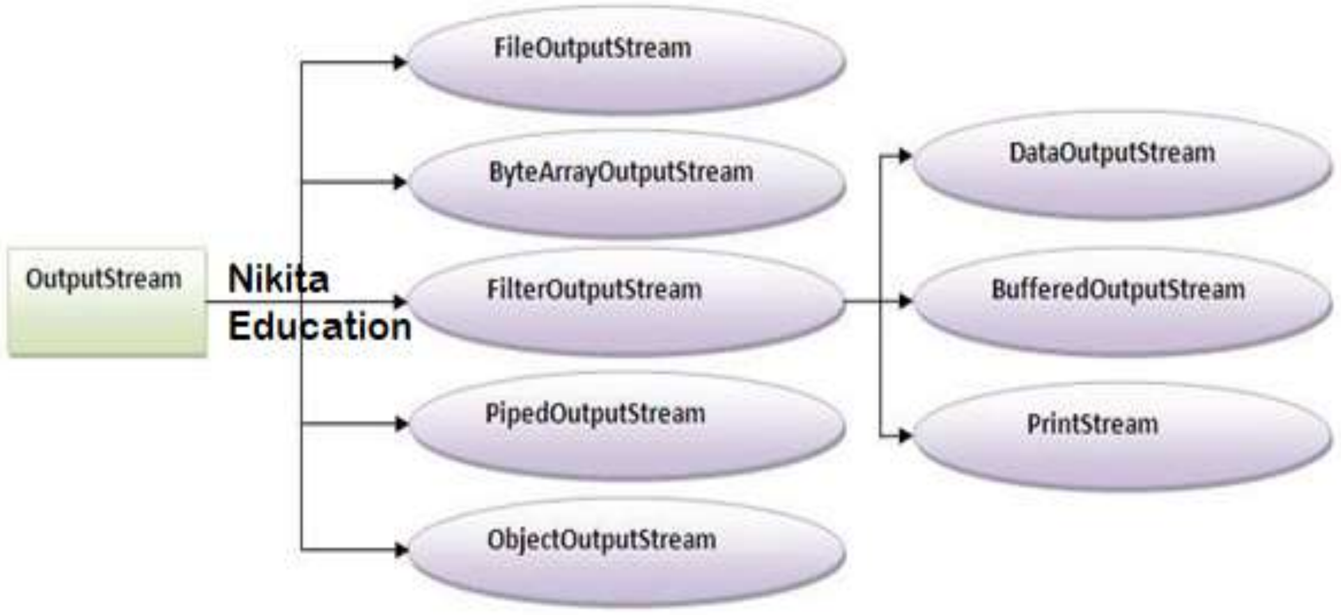
B. Byte Stream Classes:

1. OutputStream and InputStream (Basic)

1.1. OutputStream

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

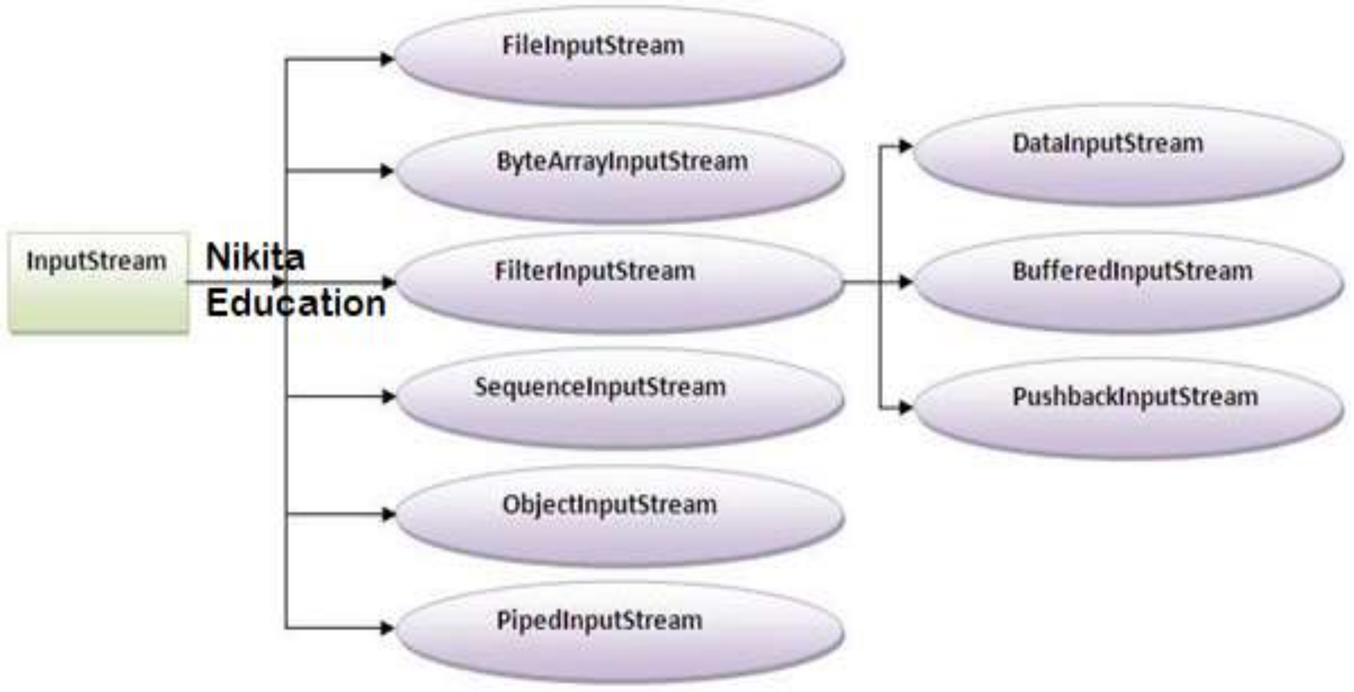
Method	**Description**
public void write(int)throws IOException:	is used to write a byte to the current output stream.
public void write(byte[])throws IOException:	is used to write an array of byte to the current output stream.
public void flush()throws IOException:	flushes the current output stream.
public void close()throws IOException:	is used to close the current output stream.



1.2. InputStream

InputStream class is an abstract class.It is the superclass of all classes representing an input stream of bytes.

Method	**Description**
public abstract int read()throws IOException:	reads the next byte of data from the input stream.It returns -1 at the end of file.
public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
public void close()throws IOException:	is used to close the current input stream.



2. FileInputStream and FileOutputStream (File Handling)

In Java, FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

2.1. FileOutputStream

Java FileOutputStream is an output stream for writing data to a file. If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter. But you can write byte-oriented as well as character-oriented data.

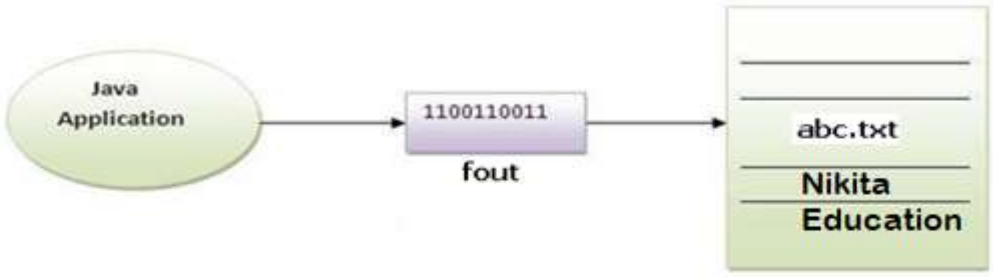
Example:

```
import java.io.*;
class Test{
        public static void main(String args[]){
                try{
                        FileOutputstream fout=new FileOutputStream("abc.txt");
                        String s="Sachin Tendulkar is my favourite player";
                        //converting string into byte array
```

```
                    byte b[]=s.getBytes();
                    fout.write(b);
                    fout.close();
                    System.out.println("success...");
            }catch(Exception e){
                    system.out.println(e);
            }
    }
}
```

Output:

```
success...
```



2.2. FileInputStream

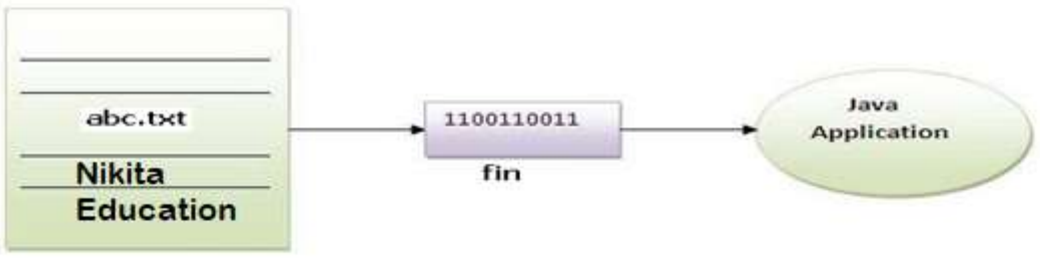
Java FileInputStream class obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader. It should be used to read byte-oriented data for example to read image, audio, video etc.

Example:

```
import java.io.*;
class SimpleRead{
        public static void main(String args[]){
                try{
                        FileInputStream fin=new FileInputStream("abc.txt");
                        int i=0;
                        while((i=fin.read())!=-1){
                                System.out.println((char)i);
                        }
                        fin.close();
                }catch(Exception e){
                        system.out.println(e);
                }
        }
}
```

Output:

```
Sachin is my favourite player.
```



2.3. Example of Reading the data of current java file and writing it into another file

```
import java.io.*;
class C{
        public static void main(String args[])throws Exception{
                FileInputStream fin=new FileInputStream("C.java");
```

```
                    FileOutputStream fout=new FileOutputStream("M.java");
                    int i=0;
                    while((i=fin.read())!=-1){
                            fout.write((byte)i);
                    }
                    fin.close();
            }
    }
```

3. DataOutputStream and DataInputStream (Primitive Type Handling)

3.1. DataOutputStream

The DataOutputStream stream let you write the primitives to an output source. Following is the constructor to create a DataOutputStream: **public DataOutputStream(OutputStream out);** Once you have *DataOutputStream* object in hand, then there is a list of helper methods, which can be used to write the stream or to do other operations on the stream.

Methods with Description
public final void write(byte[] w, int off, int len)throws IOException: Writes len bytes from the specified byte array starting at point off , to the underlying stream.
Public final int write(byte [] b)throws IOException: Writes the current number of bytes written to this data output stream. Returns the total number of bytes write into the buffer.
(a) public final void writeBooolean()throws IOException, **(b) public final void writeByte()throws IOException,** **(c) public final void writeShort()throws IOException** **(d) public final void writeInt()throws IOException** These methods will write the specific primitive type data into the output stream as bytes.
Public void flush()throws IOException: Flushes the data output stream.
public final void writeBytes(String s) throws IOException: Writes out the string to the underlying output stream as a sequence of bytes. Each character in the string is written out, in sequence, by discarding its high eight bits.

Example:

```
import java.io.*;

public class DataInput_Stream{

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

                //writing string to a file encoded as modified UTF-8
                DataOutputStream dataOut;
                dataOut = new DataOutputStream(new FileOutputStream("E:\\file.txt"));
                dataOut.writeUTF("hello");
                //Reading data from the same file
                DataInputStream dataIn;
                dataIn = new DataInputStream(new FileInputStream("E:\\file.txt"));
                while(dataIn.available()>0){
                    String k = dataIn.readUTF();
                    System.out.print(k+" ");
            }
    }
}
```

```
Output:
THIS IS TEST 1  ,
THIS IS TEST 2  ,
THIS IS TEST 3  ,
THIS IS TEST 4  ,
THIS IS TEST 5  ,
```

3.2. DataInputStream

The DataInputStream is used in the context of DataOutputStream and can be used to read primitives. Following is the constructor to create an InputStream: **public DataInputStream(InputStream in);** Once you have *DataInputStream* object in hand, then there is a list of helper methods, which can be used to read the stream or to do other operations on the stream.

Methods with Description
public final int read(byte[] r, int off, int len)throws IOException: Reads up to len bytes of data from the input stream into an array of bytes. Returns the total number of bytes read into the buffer otherwise -1 if it is end of file.
Public final int read(byte [] b)throws IOException: Reads some bytes from the inputstream an stores in to the byte array. Returns the total number of bytes read into the buffer otherwise -1 if it is end of file.
(a) public final boolean readBoolean()throws IOException **(b) public final byte readByte()throws IOException** **(c) public final short readShort()throws IOException** **(d) public final Int readInt()throws IOException** These methods will read the bytes from the contained InputStream. Returns the next two bytes of the InputStream as the specific primitive type.
public String readLine() throws IOException: Reads the next line of text from the input stream. It reads successive bytes, converting each byte separately into a character, until it encounters a line terminator or end of file; the characters read are then returned as a String.

Example:

```
import java.io.*;
public class DataInput_Stream{
        public static void main(String args[])throws IOException{
                //writing string to a file encoded as modified UTF-8
                DataOutputStream dataOut;
                dataOut = new DataOutputStream(new FileOutputStream("E:\\file.txt"));
                dataOut.writeUTF("hello");
                //Reading data from the same file
                DataInputStream dataIn;
                dataIn = new DataInputStream(new FileInputStream("E:\\file.txt"));
                while(dataIn.available()>0){
                        String k = dataIn.readUTF();
                        System.out.print(k+" ");
                }
        }
}
```

```
Output:
hello
```

4. BufferedOutputStream and BufferedInputStream (Using Buffer)

4.1. BufferedOutputStream

Java BufferedOutputStream class uses an internal buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Example:
```
import java.io.*;
class Test{
        public static void main(String args[])throws Exception{
                FileOutputStream fout=new FileOutputStream("f1.txt");
                BufferedOutputStream bout=new BufferedOutputStream(fout);
                String s="Sachin is my favourite player";
                byte b[]=s.getBytes();
                bout.write(b);
                bout.flush();
                bout.close();
                fout.close();
                System.out.println("success");
        }
}
```

Output:
```
success...
```

4.2. BufferedInputStream

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

Example:
```
import java.io.*;
class SimpleRead{
        public static void main(String args[]){
                try{
                        FileInputStream fin=new FileInputStream("f1.txt");
                        BufferedInputStream bin=new BufferedInputStream(fin);
                        int i;
                        while((i=bin.read())!=-1){
                                        System.out.println((char)i);
                        }
                        bin.close();
                        fin.close();
                }catch(Exception e){
                        system.out.println(e);
                }
        }
}
```

Output:
```
Sachin is my favorite player.
```

5. PrintStream

The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

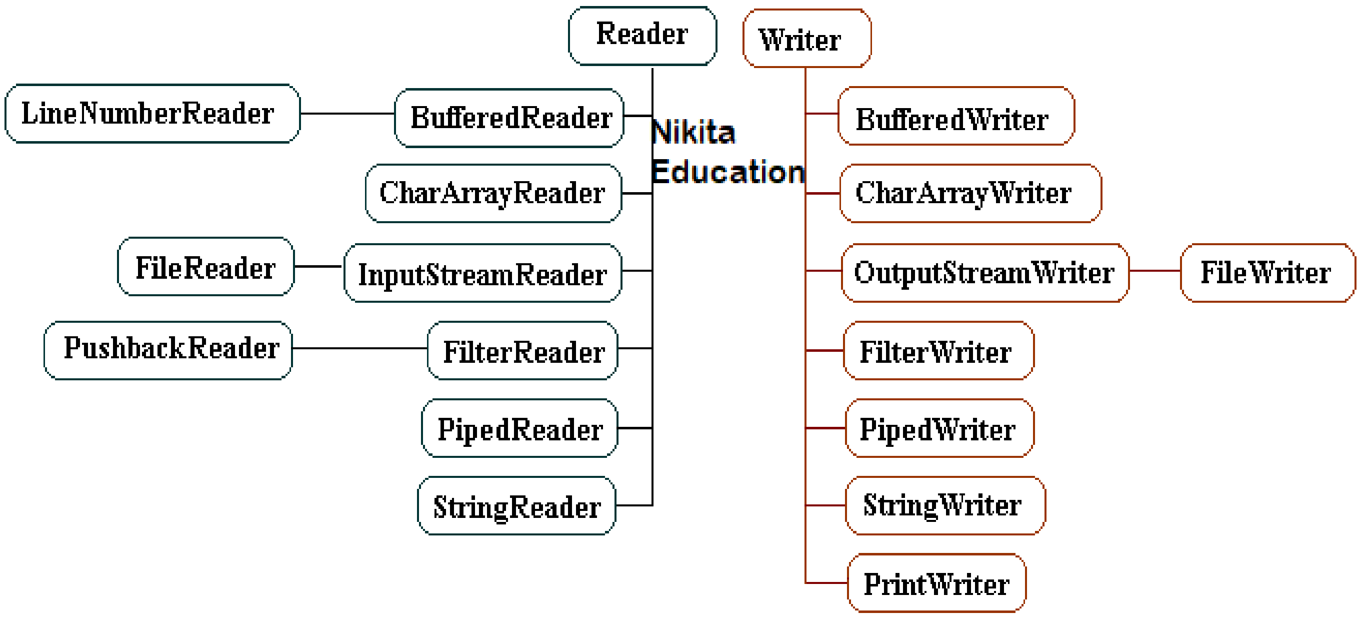
Commonly used methods of PrintStream class:

- **public void print(boolean b):** it prints the specified boolean value.
- **public void print(char c):** it prints the specified char value.
- **public void print(char[] c):** it prints the specified character array values.
- **public void print(int i):** it prints the specified int value.
- **public void print(long l):** it prints the specified long value.
- **public void print(float f):** it prints the specified float value.
- **public void print(double d):** it prints the specified double value.
- **public void print(String s):** it prints the specified string value.
- **public void print(Object obj):** it prints the specified object value.
- **public void println(boolean b):** it prints the specified boolean value and terminates the line.
- **public void println(char c):** it prints the specified char value and terminates the line.
- **public void println(char[] c):** it prints the specified character array values and terminates the line.
- **public void println(int i):** it prints the specified int value and terminates the line.
- **public void println(long l):** it prints the specified long value and terminates the line.
- **public void println(float f):** it prints the specified float value and terminates the line.
- **public void println(double d):** it prints the specified double value and terminates the line.
- **public void println(String s):** it prints the specified string value and terminates the line./li>
- **public void println(Object obj):** it prints the specified object value and terminates the line.
- **public void println():** it terminates the line only.
- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.
- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the formatted string to the current stream using specified format.

Example:

```
import java.io.*;
class PrintStreamTest{
        public static void main(String args[])throws Exception{
                FileOutputStream fout=new FileOutputStream("mfile.txt");
                PrintStream pout=new PrintStream(fout);
                pout.println(1900);
                pout.println("Hello Java");
                pout.println("Welcome to Java");
                pout.close();
                fout.close();
                }
        }
```

C. Character Stream Classes:



1. Reader and Writer (Basic)

1.1. Reader

The Java.io.Reader class is a abstract class for reading character streams.

Syntax: public abstract class Reader extends Object implements DataOutput, DataInput, Closeable

Constructors & Description
protected Reader(): This creates a new character-stream reader whose critical sections will synchronize on the reader itself.
protected Reader(Object lock): This creates a new character-stream reader whose critical sections will synchronize on the given object.

Methods & Description
abstract void close(): This method closes the stream and releases any system resources associated with it.
void mark(int readAheadLimit): This method marks the present position in the stream.
boolean markSupported(): This method tells whether this stream supports the mark() operation.
int read(): This method reads a single character.
int read(char[] cbuf): This method reads characters into an array.

abstract int read(char[] cbuf, int off, int len): This method reads characters into a portion of an array.
int read(CharBuffer target): This method attempts to read characters into the specified character buffer.
boolean ready(): This method tells whether this stream is ready to be read.
void reset(): This method resets the stream.
long skip(long n): This method skips characters.

1.2. Writer

The Java.io.Writer class is a abstract class for writing to character streams.

Syntax: public abstract class Writer extends Object implements Appendable, Closeable, Flushable

Constructor & Description
protected Writer(): This creates a new character-stream writer whose critical sections will synchronize on the writer itself.
protected Writer(Object lock): This creates a new character-stream writer whose critical sections will synchronize on the given object.

Method & Description
Writer append(char c): This method appends the specified character to this writer.
Writer append(CharSequence csq): This method appends the specified character sequence to this writer.
Writer append(CharSequence csq, int start, int end): This method appends a subsequence of the specified character sequence to this writer.
abstract void close(): This method loses the stream, flushing it first.
abstract void flush(): This method flushes the stream.
void write(char[] cbuf): This method writes an array of characters.
abstract void write(char[] cbuf, int off, int len): This method writes a portion of an array of characters.
void write(int c): This method writes a single character.
void write(String str): This method writes a string.
void write(String str, int off, int len): This method writes a portion of a string.

2. BufferedReader and BufferedWriter (Using Buffer)

2.1. BufferedReader

The Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. Following are the important points about BufferedReader:

- The buffer size may be specified, or the default size may be used.

- Each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream.

Syntax: public class BufferedReader extends Reader

Constructor & Description
BufferedReader(Reader in): This creates a buffering character-input stream that uses a default-sized input buffer.
BufferedReader(Reader in, int sz): This creates a buffering character-input stream that uses an input buffer of the specified size.

Method & Description
void close(): This method closes the stream and releases any system resources associated with it.
void mark(int readAheadLimit): This method marks the present position in the stream.
boolean markSupported(): This method tells whether this stream supports the mark() operation or not.
int read(): This method reads a single character.
int read(char[] cbuf, int off, int len): This method reads characters into a portion of an array.
String readLine(): This method reads a line of text.
boolean ready(): This method tells whether this stream is ready to be read.
void reset(): This method resets the stream.
long skip(long n): This method skips characters.

Example:

```
import java.io.*;
class G5{
        public static void main(String args[])throws Exception{
                InputStreamReader r=new InputStreamReader(System.in);
                BufferedReader br=new BufferedReader(r);
                System.out.println("Enter your name");
                String name=br.readLine();
                System.out.println("Welcome "+name);
        }
}
```

Output:

```
Enter your name
Nikita Education

Welcome
Nikita Education
```

2.2. BufferedWriter

The **Java.io.BufferedWriter** class writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.Following are the important points about BufferedWriter:

- The buffer size may be specified, or the default size may be used.
- A Writer sends its output immediately to the underlying character or byte stream.

Syntax: public class BufferedWriter extends Writer

Constructor & Description
BufferedWriter(Writer out): This creates a buffered character-output stream that uses a default-sized output buffer.
BufferedWriter(Writer out, int sz): This creates a new buffered character-output stream that uses an output buffer of the given size.

Method & Description
void close(): This method closes the stream, flushing it first.
void flush(): This method flushes the stream.
void newLine(): This method writes a line separator.
void write(char[] cbuf, int off, int len): This method writes a portion of an array of characters.
void write(int c): This method writes a single character.
void write(String s, int off, int len): This method writes a portion of a String.

Example:

```
package org.gpn.ifcm;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
public class WriteToFileExample {
        public static void main(String[] args) {
                try {
                        String content = "This is the content to write into file";
                        File file = new File("c:/Temp/filename.txt");
                        // if file doesnt exists, then create it
                        if (!file.exists()) {
                                file.createNewFile();
                        }
                        FileWriter fw = new FileWriter(file.getAbsoluteFile());
                        BufferedWriter bw = new BufferedWriter(fw);
                        bw.write(content);
                        bw.close();
                        System.out.println("Done");
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

3. FileWriter and FileReader (File Handling)

Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java. Java has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

3.1. FileWriter

Java FileWriter class is used to write character-oriented data to the file.

Constructor	Description
FileWriter(String file)	creates a new file. It gets file name in string.
FileWriter(File file)	creates a new file. It gets file name in File object.

Method	Description
public void write(String text)	writes the string into FileWriter.
public void write(char c)	writes the char into FileWriter.
public void write(char[] c)	writes char array into FileWriter.
public void flush()	flushes the data of FileWriter.
public void close()	closes FileWriter.

Example:

```
import java.io.*;
class Simple{
        public static void main(String args[]){
                try{
                        FileWriter fw=new FileWriter("abc.txt");
                         fw.write("my name is sachin");
                        fw.close();
                }catch(Exception e){
                        System.out.println(e);
                }
                System.out.println("success");
        }
}
```

Output:

```
success...
```

3.2. FileReader

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Method	Description
1) public int read()	returns a character in ASCII form. It returns -1 at the end of file.
2) public void close()	closes FileReader.

Example:
```
import java.io.*;
class Simple{
        public static void main(String args[])throws Exception{
                FileReader fr=new FileReader("abc.txt");
                int i;
                while((i=fr.read())!=-1)
                        System.out.println((char)i);
                        fr.close();
                }
        }
}
```

Output:
```
my name is sachin
```

4. InputStreamReader and OutputStreamWriter (Bridge between byte and character streams)

4.1. InputStreamReader

The Java.io.InputStreamReader class is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset.

Syntax: public class InputStreamReader extends Reader

Constructor & Description
InputStreamReader(InputStream in): This creates an InputStreamReader that uses the default charset.
InputStreamReader(InputStream in, Charset cs): This creates an InputStreamReader that uses the given charset.
InputStreamReader(InputStream in, CharsetDecoder dec): This creates an InputStreamReader that uses the given charset decoder.
InputStreamReader(InputStream in, String charsetName): This creates an InputStreamReader that uses the named charset.

Method & Description
void close(): This method closes the stream and releases any system resources associated with it.
String getEncoding(): This method returns the name of the character encoding being used by this stream.
int read(): This method reads a single character.
int read(char[] cbuf, int offset, int length): This method reads characters into a portion of an array.
boolean ready(): This method tells whether this stream is ready to be read.

Example:
```
InputStream inputStream = new FileInputStream("c:\\data\\input.txt");
Reader inputStreamReader = new InputStreamReader(inputStream);
int data = inputStreamReader.read();
while(data != -1){
        char theChar = (char) data;
        data = inputStreamReader.read();
}
inputStreamReader.close();
```

4.2. OutputStreamWriter

The Java.io.OutputStreamWriter class is a bridge from character streams to byte streams. Characters written to it are encoded into bytes using a specified charset.

Syntax: public class OutputStreamWriter extends Writer

Constructor & Description
OutputStreamWriter(OutputStream out): This creates an OutputStreamWriter that uses the default character encoding.
OutputStreamWriter(OutputStream out, Charset cs): This creates an OutputStreamWriter that uses the given charset.
OutputStreamWriter(OutputStream out, CharsetEncoder enc): This creates an OutputStreamWriter that uses the given charset encoder.
OutputStreamWriter(OutputStream out, String charsetName): This creates an OutputStreamWriter that uses the named charset.

Method & Description
void close(): This method closes the stream, flushing it first.
void flush(): This method flushes the stream.
String getEncoding(): This method returns the name of the character encoding being used by this stream.
void write(char[] cbuf, int off, int len): This method writes a portion of an array of characters.
void write(int c): This method writes a single character.
void write(String str, int off, int len): This method writes a portion of a string.

Example:

```
OutputStream outputStream = new FileOutputStream("c:\\data\\output.txt");
Writer outputStreamWriter = new OutputStreamWriter(outputStream);
outputStreamWriter.write("Hello World");
outputStreamWriter.close();
```

5. PrintWriter

The **Java.io.PrintWriter** class prints formatted representations of objects to a text-output stream.

Syntax: public class PrintWriter extends Writer

Field: protected Writer out -- This is the character-output stream of this PrintWriter.
protected Object lock -- This is the object used to synchronize operations on this stream.

Constructor & Description
PrintWriter(File file): This creates a new PrintWriter, without automatic line flushing, with the specified file.
PrintWriter(File file, String csn): This creates a new PrintWriter, without automatic line flushing, with the specified file and charset.
PrintWriter(OutputStream out): This creates a new PrintWriter, without automatic line flushing, from an existing OutputStream.
PrintWriter(OutputStream out, boolean autoFlush): This creates a new PrintWriter from an existing OutputStream.
PrintWriter(String fileName): This creates a new PrintWriter, without automatic line flushing, with the specified file name.
PrintWriter(String fileName, String csn): This creates a new PrintWriter, without automatic line flushing, with the specified file name and charset.
PrintWriter(Writer out): This creates a new PrintWriter, without automatic line flushing.
PrintWriter(Writer out, boolean autoFlush): This creates a new PrintWriter.

Method & Description
PrintWriter append(char c): This method appends the specified character to this writer.
PrintWriter append(CharSequence csq): This method appends the specified character sequence to this writer.
PrintWriter append(CharSequence csq, int start, int end): This method appends a subsequence of the specified character sequence to this writer.
boolean checkError(): This method flushes the stream if it's not closed and checks its error state.
protected void clearError(): This method Clears the error state of this stream.
void close(): This method Closes the stream and releases any system resources associated with it.
void flush(): This method Flushes the stream.
PrintWriter format(Locale l, String format, Object... args): This method writes a formatted string to this writer using the specified format string and arguments.
PrintWriter format(String format, Object... args): This method writes a formatted string to this writer using the specified format string and arguments.
void print(boolean b): This method prints a boolean value.
void print(char c): This method prints a character.
void print(char[] s): This method Prints an array of characters.
void print(double d): This method Prints a double-precision floating-point number.
void print(float f): This method prints a floating-point number.
void print(int i): This method prints an integer.
void print(long l): This method prints a long integer.

void print(Object obj): This method prints an object.
void print(String s): This method prints a string.
PrintWriter printf(Locale l, String format, Object... args): This is a convenience method to write a formatted string to this writer using the specified format string and arguments.
PrintWriter printf(String format, Object... args): This is a convenience method to write a formatted string to this writer using the specified format string and arguments.
void println(): This method terminates the current line by writing the line separator string.
void println(boolean x): This method prints a boolean value and then terminates the line.
void println(char x): This method prints a character and then terminates the line.
void println(char[] x): This method prints an array of characters and then terminates the line.
void println(double x): This method prints a double-precision floating-point number and then terminates the line.
void println(float x): This method prints a floating-point number and then terminates the line.
void println(int x): This method prints an integer and then terminates the line.
void println(long x): This method prints a long integer and then terminates the line.
void println(Object x): This method prints an Object and then terminates the line.
void println(String x): This method prints a String and then terminates the line.
protected void setError(): This method indicates that an error has occurred.
void write(char[] buf): This method writes an array of characters.
void write(char[] buf, int off, int len): This method writes a portion of an array of characters.
void write(int c): This methodWrites a single character.
void write(String s): This method writes a string.
void write(String s, int off, int len): This method writes a portion of a string.

Example:

```
import java.io.*;
public class PrintWriterDemo {
        public static void main(String args[]) {
                PrintWriter pw = new PrintWriter(System.out, true);
                pw.println("This is a string");
                int i = -7;
                pw.println(i);
                double d = 4.5f-7;
                pw.println(d);
        }
}
```

Example:

```
import java.io.IOException;
import java.io.PrintWriter;
public class MainClass {
        public static void main(String[] args) {
                try {
                PrintWriter pw = new PrintWriter("c:\\temp\\printWriterOutput.txt");
                pw.println("PrintWriter is easy to use.");
                pw.println(1234);
                pw.close();
                } catch (IOException e) {
                e.printStackTrace();
                }
        }
}
```

6. Scanner

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

Method	**Description**
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.
public short nextShort()	it scans the next token as a short value.
public int nextInt()	it scans the next token as an int value.
public long nextLong()	it scans the next token as a long value.
public float nextFloat()	it scans the next token as a float value.
public double nextDouble()	it scans the next token as a double value.

Example:

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

Output:

```
Enter your rollno
01603
Enter your name
Nitin
Enter your fee
45000
Rollno:01603 name:Nitin
fee:45000
```

D. Serialization and De-serialization

Serialization is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called **deserialization**. A class must implement **Serializable** interface present in **java.io** package in order to serialize its object successfully. **Serializable** is a **marker interface** that adds serializable behaviour to the class implementing it. Java provides **Serializable** API encapsulated under *java.io* package for serializing and deserializing objects which include,

java.io.serializable	*java.io.Externalizable*	*ObjectInputStream*	*ObjectOutputStream*

1. Marker interface

Marker Interface is a special interface in Java without any field and method. Marker interface is used to inform compiler that the class implementing it has some special behavior or meaning. Some example of Marker interface are,

java.io.Serializable	java.lang.Cloneable	java.rmi.Remote	java.util.RandomAccess

All these interfaces does not have any method and field. They only add special behavior to the classes implementing them. However marker interfaces have been deprecated since Java 5, they were replaced by **Annotations**. Annotations are used in place of Marker Interface that play the exact same role as marker interfaces did before.

2. Signature of writeObject() and readObject()

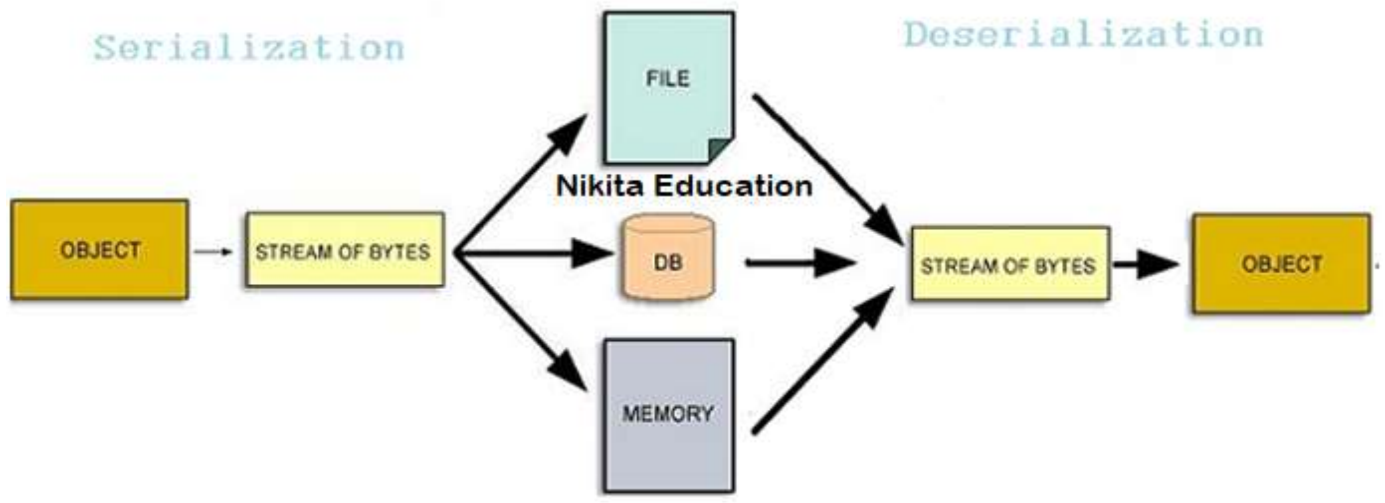
2.1. writeObject() method of *ObjectOutputStream* class serializes an object and send it to the output stream.

Syntax: public *final* void **writeObject**(*object x*) throws **IOException**

2.2. readObject() method of *ObjectInputStream* class references object out of stream and deserialize it.

Syntax: public final *Object* **readObject()** throws **IOException, ClassNotFoundException**

while serializing if you do not want any field to be part of object state then declare it either static or transient based on your need and it will not be included during java serialization process.



3. Serializing an Object

```
import java.io.*;
class Studentinfo implements Serializable{
        String name;
        int rid;
        static String contact;
        Studentinfo(string n, int r, string c){
                this.name = n;
                this.rid = r;
                contact = c;
        }
}

class Test{
        public static void main(String[] args){
                try{
                        Studentinfo si = new Studentinfo("Abhi", 104, "110044");
                        FileOutputStream fos = new FileOutputStream("student.ser");
                        Objectoutputstream oos = new ObjectOutputStream(fos);
                        oos.writeObject(si);
                        oos.close();
                        fos.close();
                }catch (Exception e){
                        e. printStackTrace();
                }
        }
}
```

Output:
Object of Studentinfo class is serialized using writeObject() method and written to **student.ser** file

4. Deserialization of Object

```
import java.io * ;
class DeserializationTest{
        public static void main(String[] args){
                studentinfo si=null ;
                try{
                        FileInputStream fis = new FileInputStream("student.ser");
                        ObjectOutputStream ois = new ObjectOutputStream(fis);
                        si = (studentinfo)ois.readObject();
                }catch (Exception e){
                        e.printStackTrace();
                }
                System.out.println(si.name);
                System.out. println(si.rid);
                System.out.println(si.contact);
        }
}
```

Output :

```
Abhi
104
null
```

Contact field is null because, it was marked as static and as we have discussed earlier static fields does not get serialized.

NOTE : Static members are never serialized because they are connected to class not object of class.

Collection Framework

B. Introduction:

A **Collection** is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit. The **Collection** interface (java.util.Collection) and **Map** interface (java.util.Map) are the two main "root" interfaces of Java collection classes.

Need for Collection Framework:

Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or HashTables. All of these collections had no common interface. Accessing elements of these Data Structures was a difficult task as each had a different method (and syntax) for accessing its members**.** Hence java introduced a collection framework.

What is Collection in Java?

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java?

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

What is Collection framework?

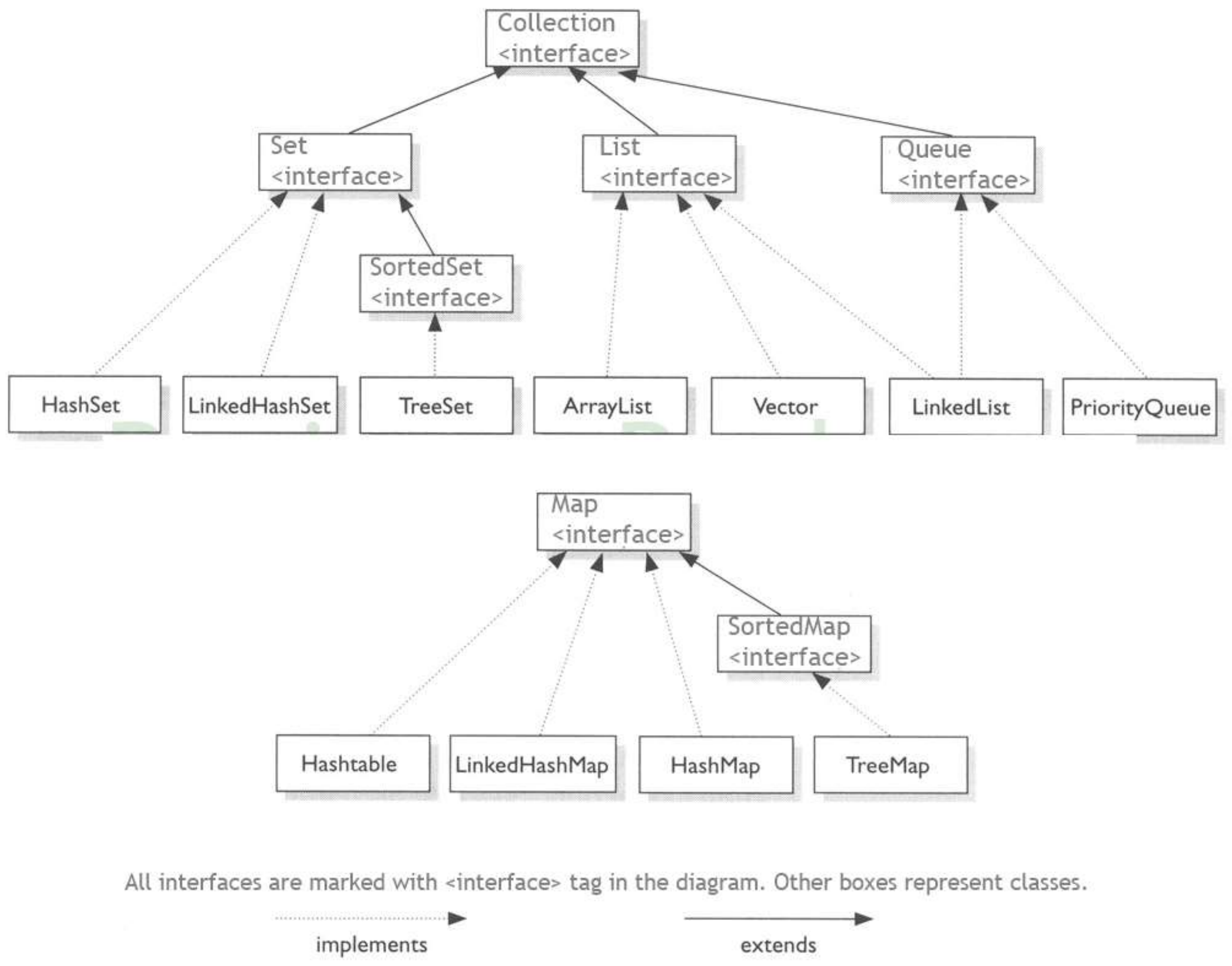
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its classes
2. Algorithms

Java Collections Classes:

ArrayList	LinkedList	Stack
HashSet	LinkedHashSet	TreeSet
HashMap	LinkedHashMap	TreeMap
Vector	PriorityQueue	HashTable

C. Collections Framework hierarchy



Iterators in Java

Iterators are used in Collection framework in Java to retrieve elements one by one. There are three types of iterators:

1. Enumeration
2. Iterator
3. ListIterator

Enumeration :

It is a interface used to get elements of legacy collections (Vector, Hashtable). Enumeration is the first iterator present from JDK 1.0.

```
// Here "v" is an Vector class object. e is of
// type Enumeration interface and refers to "v"
Enumeration e = v.elements();
```

Limitations of Enumeration:

- Enumeration is for **legacy** classes(Vector, Hashtable) only. Hence it is not a universal iterator.
- Remove operations can’t be performed using Enumeration.
- Only forward direction iterating is possible.

Iterator:

It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of remove-ability of a element. Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the **only** cursor available for entire collection framework.

```
// Here "c" is any Collection object. itr is of
// type Iterator interface and refers to "c"
Iterator itr = c.iterator();
```

Limitations of Iterator:

- Only forward direction iterating is possible.
- Replacement and addition of new element is not supported by Iterator.

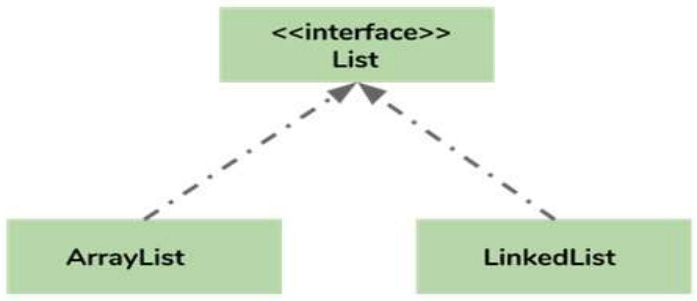
ListIterator:

It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration. ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

```
// Here "l" is any List object, ltr is of type
// ListIterator interface and refers to "l"
ListIterator ltr = l.listIterator();
```

Limitations of ListIterator:
It is the most powerful iterator but it is only applicable for List implemented classes, so it is not a universal iterator.

D. Java Collections – List



1. ArrayList

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by a size, however the size can increase if collection grows or shrunk if objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.

Constructors in Java ArrayList:

1. ArrayList()
2. ArrayList(Collection c)
3. ArrayList(int capacity)

Methods in Java ArrayList:

1. void clear()
2. void add(int index, Object element)
3. int indexOf(Object O)
4. int lastIndexOf(Object O)
5. Object clone()
6. Object[] toArray()
7. boolean addAll(Collection C)
8. boolean add(Object o)
9. boolean addAll(int index, Collection C)

Example:

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

class myarraylist
{
    public static void main(String[] args)
                        throws IOException
    {
        // size of ArrayList
        int n = 5;

        //declaring ArrayList with initial size n
        ArrayList<Integer> arrli = new ArrayList<Integer>(n);

        // Appending the new element at the end of the list
        for (int i=1; i<=n; i++)
            arrli.add(i);

        // Printing elements
        System.out.println(arrli);

        // Remove element at index 3
        arrli.remove(3);

        // Displaying ArrayList after deletion
        System.out.println(arrli);

        // Printing elements one by one
        for (int i=0; i<arrli.size(); i++)
            System.out.print(arrli.get(i)+" ");
    }
}
```

2. LinkedList

In java LinkedList class provides the functionality of doubly linked list concept used in data structure. LinkedList class implements the list interface. The LinkedList class also consists of various constructors and methods like other java collections.

Constructors for Java LinkedList:

1. LinkedList()
2. LinkedList(Collection C)

Methods for Java LinkedList:

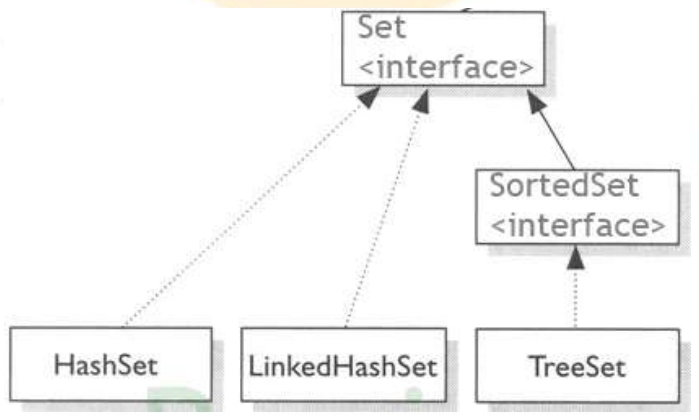
1. int size()
2. void clear()
3. Object set(int index, Object element)
4. boolean contains(Object element)
5. boolean add(Object element)
6. void add(int index, Object element)
7. boolean addAll(Collection C)
8. boolean addAll(int index, Collection C)
9. void addFirst(Object element)
10. void addLast(Object element)
11. Object get(int index)
12. Object getFirst()
13. Object getLast()
14. int indexOf(Object element)
15. Object remove()
16. Object remove(int index)
17. boolean remove(Object O)
18. Object removeFirst()
19. Object removeLast()

Example:

```
import java.util.*;

public class LinkedListTest
{
    public static void main(String args[])
    {
        // Creating object of class linked list
        LinkedList<String> object = new LinkedList<String>();

        // Adding elements to the linked list
        object.add("A");
        object.add("B");
        object.addLast("C");
        object.addFirst("D");
        object.add(2, "E");
        object.add("F");
        object.add("G");
        System.out.println("Linked list : " + object);
        // Removing elements from the linked list
        object.remove("B");
        object.remove(3);
        object.removeFirst();
        object.removeLast();
        System.out.println("Linked list after deletion: " + object);

        // Finding elements in the linked list
        boolean status = object.contains("E");

        if(status)
            System.out.println("List contains the element 'E' ");
        else
            System.out.println("List doesn't contain the element 'E'");

        // Number of elements in the linked list
        int size = object.size();
        System.out.println("Size of linked list = " + size);

        // Get and set elements from linked list
        Object element = object.get(2);
        System.out.println("Element returned by get() : " + element);
        object.set(2, "Y");
        System.out.println("Linked list after change : " + object);
    }
}
```

E. Java Collections – Set

Difference between List and Set: List can contain duplicate elements whereas Set contains unique elements only.



1. HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface. The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet does not maintain any order of elements.

Constructors in HashSet:

1. HashSet()
2. HashSet(Collection C)
3. HashSet(int capacity)

Methods in HashSet:

1. boolean add(E e)
2. void clear()
3. boolean contains(Object o)
4. boolean remove(Object o)
5. Iterator iterator()
6. boolean isEmpty()
7. int size()
8. Object clone()

Example:

```
import java.util.*;

class HashSetTest
{
    public static void main(String[]args)
    {
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet usind add()
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India");// adding duplicate elements

        // Displaying the HashSet
        System.out.println(h);
        System.out.println("List contains India or not:" +
                           h.contains("India"));

        // Removing items from HashSet using remove()
        h.remove("Australia");
        System.out.println("List after removing Australia:"+h);

        // Iterating over hash set items
        System.out.println("Iterating over list:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

2. TreeSet

1. It extends **AbstractSet** class and implements the **NavigableSet** interface.
2. It stores the elements in ascending order.
3. It uses a Tree structure to store elements.
4. It contains unique elements only like HashSet.
5. It's access and retrieval times are quite fast.

Constructors of TreeSet class:

1. TreeSet()
2. TreeSet(Collection *C*)
3. TreeSet(Comparator *comp*)
4. TreeSet(SortedSet *ss*)

Example:

```
import java.util.*;

class TreeSetDemo
{
    public static void main (String[] args)
    {
        TreeSet<String> ts1= new TreeSet<String>();

        // Elements are added using add() method
        ts1.add("A");
        ts1.add("B");
        ts1.add("C");

        // Duplicates will not get insert
        ts1.add("C");

        // Elements get stored in default natural
        // Sorting Order(Ascending)
        System.out.println(ts1);
    }
}
```

3. LinkedHashSet

1. LinkedHashSet class extends **HashSet** class
2. LinkedHashSet maintains a linked list of entries in the set.
3. LinkedHashSet stores elements in the order in which elements are inserted i.e it maintains the insertion order.

Example

```
import java.util.*;
class LinkedHashSetDemo
{
 public static void main(String args[])
 {
  LinkedHashSet< String> hs = new LinkedHashSet< String>();
  hs.add("B");
```

```
   hs.add("A");
   hs.add("D");
   hs.add("E");
   hs.add("C");
   hs.add("F");
   System.out.println(hs);
  }
 }
```

4. PriorityQueue

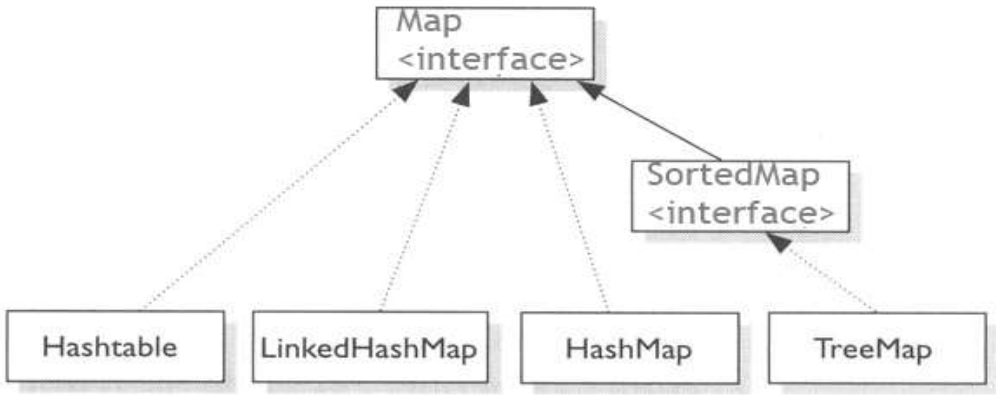
1. It extends the **AbstractQueue** class.
2. The PriorityQueue class provides the facility of using queue.
3. It does not orders the elements in FIFO manner.
4. PriorityQueue has six constructors. In all cases, the capacity grows automatically as elements are added.

Example:

```
import java.util.*;

class PriorityQueueDemo
{
public static void main(String args[])
{
  PriorityQueue<String> queue=new PriorityQueue<String>();
  queue.add("WE");
  queue.add("LOVE");
  queue.add("STUDY");
  queue.add("TONIGHT");
  System.out.println("At head of the queue:"+queue.element());
  System.out.println("At head of the queue:"+queue.peek());
  System.out.println("Iterating the queue elements:");
  Iterator itr=queue.iterator();
  while(itr.hasNext()){
    System.out.println(itr.next());
  }
  queue.remove();
  queue.poll();
  System.out.println("After removing two elements:");
  Iterator itr2=queue.iterator();
  while(itr2.hasNext()){
    System.out.println(itr2.next());
  }
}
}
```

F. Java Collections – Map

A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map is useful if you have to search, update or delete elements on the basis of key. The java.util.Map interface represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types. A Map stores data in key and value association. Both key and values are objects. The key must be unique but the values can be duplicate.



1. HashMap

1. HashMap class extends **AbstractMap** and implements **Map** interface.
2. It uses a **Hashtable** to store the map. This allows the execution time of get() and put() to remain same.
3. HashMap has four constructor.
    1. HashMap()
    2. HashMap(Map< ? extends k, ? extends V> m)
    3. HashMap(int capacity)
    4. HashMap(int capacity, float fillratio)
4. HashMap does not maintain order of its element.

Example:

```
import java.util.*;
class HashMapDemo
{
 public static void main(String args[])
 {
  HashMap< String,Integer> hm = new HashMap< String,Integer>();
  hm.put("a",new Integer(100));
  hm.put("b",new Integer(200));
  hm.put("c",new Integer(300));
  hm.put("d",new Integer(400));

  Set< Map.Entry< String,Integer> > st = hm.entrySet();
     //returns Set view
  for(Map.Entry< String,Integer> me:st)
  {
   System.out.print(me.getKey()+":");
   System.out.println(me.getValue());
  }
 }
}
```

2. TreeMap

1. TreeMap class extends **AbstractMap** and implements **NavigableMap** interface.
2. It creates Map, stored in a tree structure.
3. A **TreeMap** provides an efficient means of storing key/value pair in efficient order.
4. It provides key/value pairs in sorted order and allows rapid retrieval.

Example:

```
import java.util.*;
class TreeMapDemo
{
 public static void main(String args[])
 {
  TreeMap< String,Integer> tm = new TreeMap< String,Integer>();
  tm.put("a",new Integer(100));
  tm.put("b",new Integer(200));
  tm.put("c",new Integer(300));
  tm.put("d",new Integer(400));

  Set< Map.Entry< String,Integer> > st = tm.entrySet();
  for(Map.Entry me:st)
  {
   System.out.print(me.getKey()+":");
   System.out.println(me.getValue());
  }
 }
}
```

3. Hashtable

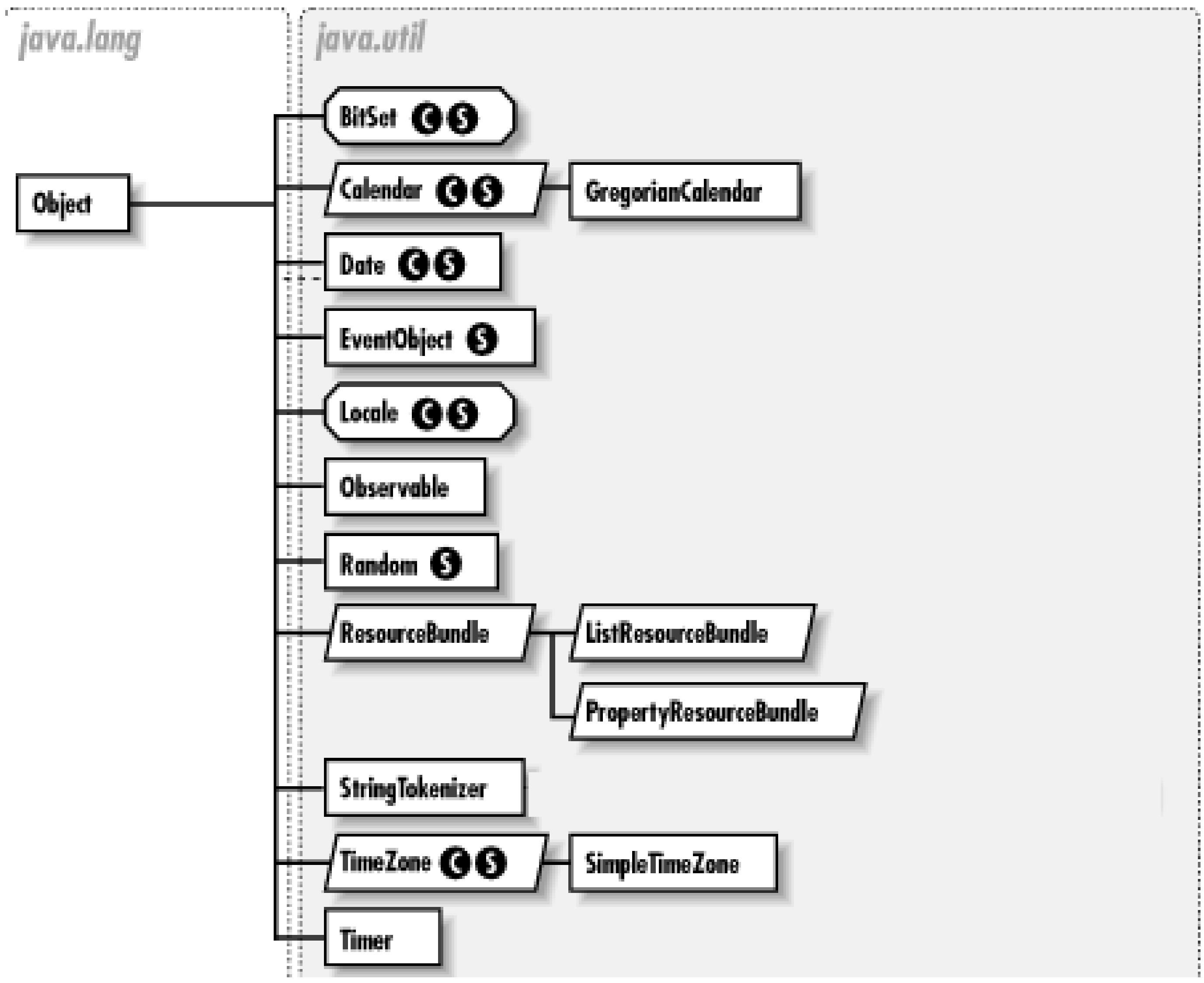
1. Like HashMap, Hashtable also stores key/value pair. However neither **keys** nor **values** can be **null**.
2. There is one more difference between **HashMap** and **Hashtable** that is Hashtable is synchronized while HashMap is not.
3. Hashtable has following four constructor
    1. Hashtable()
    2. Hashtable(int size)

Example:

```
import java.util.*;
class HashTableDemo
{
 public static void main(String args[])
 {
  Hashtable< String,Integer> ht = new Hashtable< String,Integer>();
  ht.put("a",new Integer(100));
  ht.put("b",new Integer(200));
  ht.put("c",new Integer(300));
  ht.put("d",new Integer(400));

  Set st = ht.entrySet();
  Iterator itr=st.iterator();
  while(itr.hasNext())
  {
   Map.Entry m=(Map.Entry)itr.next();
   System.out.println(itr.getKey()+" "+itr.getValue());
  }
 }
}
```

Utility Classes:

The java.util package contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes.



1. Date

The java.util.Date class represents date and time in java. It provides constructors and methods to deal with date and time in java. The java.util.Date class implements Serializable, Cloneable and Comparable<Date> interface. It is inherited by java.sql.Date, java.sql.Time and java.sql.Timestamp interfaces.

java.util.Date Constructors

No.	Constructor	Description
1)	Date()	Creates a date object representing current date and time.
2)	Date(long milliseconds)	Creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.

java.util.Date Methods

No.	Method	Description
1)	boolean after(Date date)	tests if current date is after the given date.
2)	boolean before(Date date)	tests if current date is before the given date.
3)	Object clone()	returns the clone object of current date.
4)	int compareTo(Date date)	compares current date with given date.
5)	boolean equals(Date date)	compares current date with given date for equality.
6)	static Date from(Instant instant)	returns an instance of Date object from Instant date.
7)	long getTime()	returns the time represented by this date object.
8)	int hashCode()	returns the hash code value for this date object.
9)	void setTime(long time)	changes the current date and time to given time.
10)	Instant toInstant()	converts current date into Instant object.
11)	String toString()	converts this date into Instant object.

Example:

Getting Current Date and Time

```
import java.util.Date;
public class DateDemo {

   public static void main(String args[]) {
      // Instantiate a Date object
      Date date = new Date();

      // display time and date using toString()
      System.out.println(date.toString());
   }
}
```

```
import java.util.*;

public class Main
{
    public static void main(String[] args)
    {
        Date d1 = new Date();
        System.out.println("Current date is " + d1);
        Date d2 = new Date(2323223232L);
        System.out.println("Date represented is "+ d2 );
    }
}
```

```
import java.util.*;

public class Main
{
    public static void main(String[] args)
    {
        // Creating date
        Date d1 = new Date(2000, 11, 21);
        Date d2 = new Date();  // Current date
        Date d3 = new Date(2010, 1, 3);

        boolean a = d3.after(d1);
        System.out.println("Date d3 comes after " +
                           "date d2: " + a);

        boolean b = d3.before(d2);
        System.out.println("Date d3 comes before "+
                           "date d2: " + b);

        int c = d1.compareTo(d2);
        System.out.println(c);

        System.out.println("Miliseconds from Jan 1 "+
                "1970 to date d1 is " + d1.getTime());

        System.out.println("Before setting "+d2);
        d2.setTime(204587433443L);
        System.out.println("After setting "+d2);
    }
}
```

Date Formatting Using SimpleDateFormat

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

public class DateDemo {

   public static void main(String args[]) {
      Date dNow = new Date( );
      SimpleDateFormat ft =
      new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

      System.out.println("Current Date: " + ft.format(dNow));
   }
}
```

Date Formatting Using printf

```
import java.util.Date;
public class DateDemo {

   public static void main(String args[]) {
      // Instantiate a Date object
      Date date = new Date();

      // display time and date
      String str = String.format("Current Date/Time : %tc", date );
      System.out.printf(str);
   }
}
```

```
import java.util.Date;
public class DateDemo {

   public static void main(String args[]) {
      // Instantiate a Date object
      Date date = new Date();

      // display time and date
      System.out.printf("%1$s %2$tB %2$td, %2$tY", "Due date:", date);
   }
}
```

```
import java.util.Date;
public class DateDemo {

   public static void main(String args[]) {
      // Instantiate a Date object
      Date date = new Date();

      // display formatted date
      System.out.printf("%s %tB %<te, %<tY", "Due date:", date);
   }
}
```

Parsing Strings into Dates

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

public class DateDemo {

   public static void main(String args[]) {
      SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");
      String input = args.length == 0 ? "1818-11-11" : args[0];

      System.out.print(input + " Parses as ");
      Date t;
      try {
         t = ft.parse(input);
         System.out.println(t);
      } catch (ParseException e) {
         System.out.println("Unparseable using " + ft);
      }
   }
}
```

2. Calendar

The **java.util.calendar** class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and for manipulating the calendar fields, such as getting the date of the next week.

- This class also provides additional fields and methods for implementing a concrete calendar system outside the package.
- Calendar defines the range of values returned by certain calendar fields.

Constructors:

Sr.No.	Constructor & Description
1	**protected Calendar()** This constructor constructs a Calendar with the default time zone and locale.
2	**protected Calendar(TimeZone zone, Locale aLocale)** This constructor constructs a calendar with the specified time zone and locale.

Methods:

Method	Description
public void add(int field, int amount)	Adds the specified (signed) amount of time to the given calendar field.
public boolean after (Object when)	The method Returns true if the time represented by this Calendar is after the time represented by when Object.
public boolean before(Object when)	The method Returns true if the time represented by this Calendar is before the time represented by when Object.
public int get(int field)	In get() method fields of the calendar are passed as the parameter, and this method Returns the value of fields passed as the parameter.
public static Set<String> getAvailableCalendarTypes()	Returns a set which contains string set of all available calendar type supported by Java Runtime Environment.
public String getCalendarType()	Returns in string all available calendar type supported by Java Runtime Environment.
public int getFirstDayOfWeek()	Returns the first day of the week in integer form.
public static Calendar getInstance()	This method is used with calendar object to get the instance of calendar according to current time zone set by java runtime environment
public int getMinimalDaysInFirstWeek()	Returns required minimum days in integer form.
public final Date getTime()	This method gets the time value of calendar object and Returns date.
public int getWeekYear()	This method gets the week year represented by current Calendar.
public void set(int field, int value)	Sets the specified calendar field by the specified value.
public void setWeekDate(int weekYear, int weekOfYear, int dayOfWeek)	Sets the current date with specified integer value as the parameter. These values are weekYear, weekOfYear and dayOfWeek.

Example:

```
import java.util.Calendar;
public class CalendarExample1 {
   public static void main(String[] args) {
   Calendar calendar = Calendar.getInstance();
   System.out.println("The current date is : " + calendar.getTime());
   calendar.add(Calendar.DATE, -15);
   System.out.println("15 days ago: " + calendar.getTime());
   calendar.add(Calendar.MONTH, 4);
   System.out.println("4 months later: " + calendar.getTime());
   calendar.add(Calendar.YEAR, 2);
   System.out.println("2 years later: " + calendar.getTime());
   }
}
```

```
import java.util.*;
public class CalendarExample2{
  public static void main(String[] args) {
   Calendar calendar = Calendar.getInstance();
   System.out.println("At present Calendar's Year: " +
                                  calendar.get(Calendar.YEAR));
   System.out.println("At present Calendar's Day: " +
                                  calendar.get(Calendar.DATE));
   }
}
```

```
import java.util.*;
public class CalendarExample4 {
   public static void main(String[] args) {
   Calendar calendar = Calendar.getInstance();
   int maximum = calendar.getMaximum(Calendar.DAY_OF_WEEK);
   System.out.println("Maximum number of days in week: " + maximum);
   maximum = calendar.getMaximum(Calendar.WEEK_OF_YEAR);
   System.out.println("Maximum number of weeks in year: " + maximum);
   }
}
```

```
import java.util.*;
public class CalendarExample5 {
   public static void main(String[] args) {
   Calendar cal = Calendar.getInstance();
   int maximum = cal.getMinimum(Calendar.DAY_OF_WEEK);
   System.out.println("Minimum number of days in week: " + maximum);
   maximum = cal.getMinimum(Calendar.WEEK_OF_YEAR);
   System.out.println("Minimum number of weeks in year: " + maximum);
   }
}
```

3. Random

Random class is used to generate pseudo random numbers in java. Instance of this class are thread safe. The instance of this class are however cryptographically insecure. This class provides various method call to generate different random data types such as float, double, int.

Constructors:

- **Random()**: Creates a new random number generator
- **Random(long seed)**: Creates a new random number generator using a single long seed

Methods:

Method & Description
protected int next(int bits) This method generates the next pseudorandom number.
boolean nextBoolean() This method returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence.

void nextBytes(byte[] bytes) This method generates random bytes and places them into a user-supplied byte array.
double nextDouble() This method returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.
float nextFloat() This method returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.
int nextInt() This method returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.
int nextInt(int n) This method returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
long nextLong() This method returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence.
void setSeed(long seed) This method sets the seed of this random number generator using a single long seed.

Example:

```
import java.util.Random;
public class RandomNumberExample {

     public static void main(String[] args) {

          //initialize random number generator
          Random random = new Random();

          //generates boolean value
          System.out.println(random.nextBoolean());

          //generates double value
          System.out.println(random.nextDouble());

          //generates float value
          System.out.println(random.nextFloat());

          //generates int value
          System.out.println(random.nextInt());

          //generates int value within specific limit
          System.out.println(random.nextInt(20));

     }

}
```

Best Of Luck

Nikita Education - 7350009884