What is Java?

Java is a internet software developed by Sun Microsystems for development of distributed applications.

What is software?

Is a set of programs meant for performing some specific operation in real world we have 3 types of software they are.

Types of java application

- **Standalone Application**
- **Web Application**
- **Enterprise Application**
- **Mobile Application**

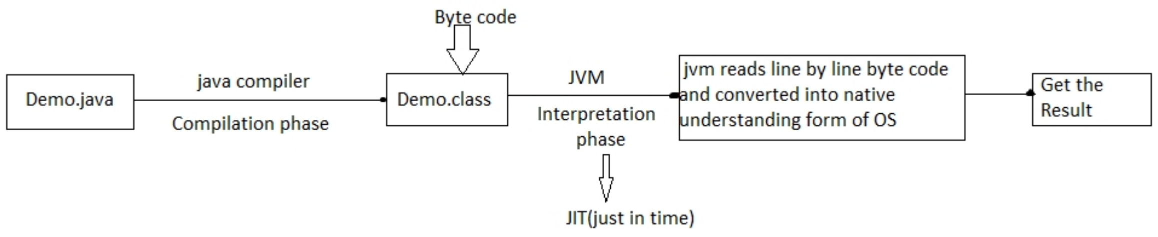
Features and Buzzwords of Java

Sun Microsystems has provided 13 features for java language

1. Simple
2. Platform Independent
3. Architectural neutral
4. Portable
5. Multithread
6. Networked Programming Language
7. Distributed Programming Language
8. High Performance
9. High Interpreted
10. Secured
11. Robust(Strong)
12. Dynamic
13. OOPL...................

    Features of a language are nothing but the services or facilities provided by language vendor to the industry programmer.

1) **Simple:**
Java is simple programming language because of the following factors.
   a) Java programming language eliminates a complex concept called pointers hence we are able to develop the application with less time and it will take less execution time because of magic of **Byte code.**
   Let us consider the following Diagram



Byte Code: Byte code is set of optimized instructions generated during compilation phase by the java compiler and nature of byte code is so powerful than ordinary pointer code.

JVM: JVM is set of programs developed by sun Microsystems and supplied as a part of JDK and its role is reading line by line of byte code and converting into native understanding form of OS.

Note: Byte code is OS independent and JVM is OS dependent.

JIT (Just in time compiler): JIT is a program developed by sun Microsystems added as a part of JVM to speed up the interpretation phase by reading the entire section of byte code.*"Java is one of the compiler and interpretation based prog lang"*

   b) Java programming environment contains inbuilt garbage collector program.
   Garbage Collector: Is a system background java program which is running along with our regular java program for collecting un-used/un-reference memory space and it improves the performance if java application.
   Garbage program is internally running for regular intervals of time.

2) **Platform Independent:** The language/technology related application are said to platform independent if and only if those application run on every OS without considering their providers / vendors.
In order to say a language / technology are platform independent, it has to satisfy the following properties.

   a) The data types of the language must take the same amount of memory space on all OS.
   b) The language / technology must contains some special powerful internal programs which will convert the native understanding form of one OS to the native understanding form of another OS.

3) **Architecture Neutral:** Java's byte code is not machine dependent. It can run on any machine with any processor and any operating system.
4) **Portable:** Java does not have implemented-dependent aspects. So the results will be same on all machine, portable means yielding same results on different machines.
5) **Multithreading:** The basic aim of multithreading is to achieve the concurrent execution. A flow of control in java is known as thread.

- The purpose of thread concept is to execute user define method which contains the logic of the java program.
- If any java program contains multiple flow of controls then that java program is known as multithreaded.
- The language like C, C++, Pascal, COBOL etc come under single threaded modelling language because their execution environment contains single flow of control.
- In general single threaded modelling language provides sequential execution but not concurrent execution and they not contain any library for development of multithreaded application.
- The language like Java and .Net are threaded as multithreaded because their execution environment provides and contains multiple flow of controls.
- In Java programming we have the following API for development of multithreading application
    - Java.lang.Thread(class).
    - Java.lang.Runnable(Interface).

Whenever we write a java program 2 types of threads exist

1) Fore-ground / child thread.
2) Back-ground / parent thread.

“Multitasking concept of java is one of the specialized form of multitasking concept of OS.

6) **Networked:** To share the data between multiple machines which are located either in same network or in different network, **Intranet** application will be developed by J2SE with network programming concept.
7) **Distributed:** According to real world project, java project are classified into 2 types
    i. Centralized Application.
    ii. Distributed Application.

i) **Centralized application:** Centralized application are those which runs in the context of single server and their result can be accessible across the globe and they are operated by authorized people only. (Banking , insurance ...)

ii) **Distributed application:** which runs in the context of multiple servers and result are accessible across the globe and they are operated by both authorized and unauthorized user.

Note: Centralized application will have private URL.
Distributed application will have public URL.

8) **High Performance:** Java is one of the high performance programming language because of the following factors
    i. Automatic memory management due to inbuilt garbage collector.
    ii. Java is free from pointers.
    iii. Due to JIT which enhances the speed of the execution.
9) **Robust:** The language like Java and .Net are threaded as Robust/Strong programming language because Runtime error are effectively addresses by the concept called Exception Handling. **OR** java programs will not crash because of its exception handling and its memory management features.

10) **Highly Interpreted:** In the older version of java (JDK 1.0) compilation phase was very faster and interpretation phase was very slow this is one the industry programmer complimented and compliant to sun Microsystems.

    Sun Microsystems has taken this task as challenging issue and they developed a program called JIT and added as part of JVM to speed up the interpretation phase by reading entire section of byte code at once.

11) **Secure:** Security is one of the principal in IT world for preventing / protecting the unauthorized modification on confidential data. The language like Java and .Net are treated has high secured programming language because their API contains readily available security programs in a package called javaX.security.*;

12) **Dynamic:** In any programming language, if we write a program and that program accepts input then the input of program is stored in the main memory by allocating sufficient amount of memory space.

    In any programming language memory allocation can be done in 2 ways

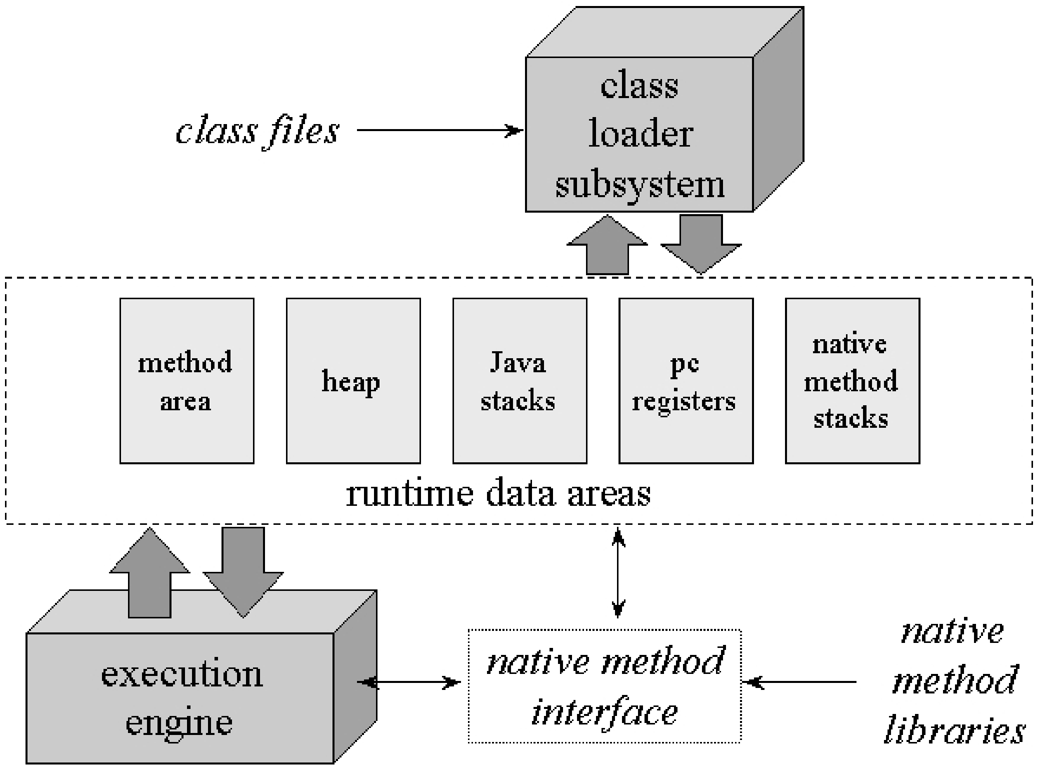
    i. Static Memory Allocation.
    ii. Dynamic Memory Allocation.

    i) **Static Memory Allocation** is one in which memory will be allocated for the impact of the program at compile time because of static Memory allocation we get the following drawbacks
        a. Waste of memory space.
        b. Loss of Data.
        c. Overlapping of Existing Data.

    ii) **Dynamic Memory Allocation** is one in which memory will be allocated for the input of the program at run time .
        - Java follows only dynamic memory allocation but not static memory allocation.
        - To allocate the memory space dynamically is java programming we use operator called new.

13) **Object Oriented Programming:** Main Feature we will discuss further

What is the architecture of JVM?



JVM has various sub components internally. You can see all of them from the above diagram.

1. Class loader sub system: JVM's class loader sub system performs 3 tasks
a. It loads **.class** file into memory.
b. It verifies byte code instructions.
c. It allots memory required for the program.

2. Run time data area: This is the memory resource used by JVM and it is divided into 5 parts
a. Method area: Method area stores class code and method code.
b. Heap: Objects are created on heap.
c. Java stacks: Java stacks are the places where the Java methods are executed. A Java stack contains frames. On each frame, a separate method is executed.
d. Program counter registers: The program counter registers store memory address of the instruction to be executed by the micro processor.
e. Native method stacks: The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.

3. Native method interface: Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.

4. Native method library: holds the native libraries information.

5. Execution engine: Execution engine contains interpreter and JIT compiler, which covert byte code into machine code. JVM uses optimization technique to decide which part to be interpreted and which part to be used with JIT compiler. The Hotspot represents the block of code executed by JIT compiler.

Java memory management

Java Heap Memory v/s Stack Memory

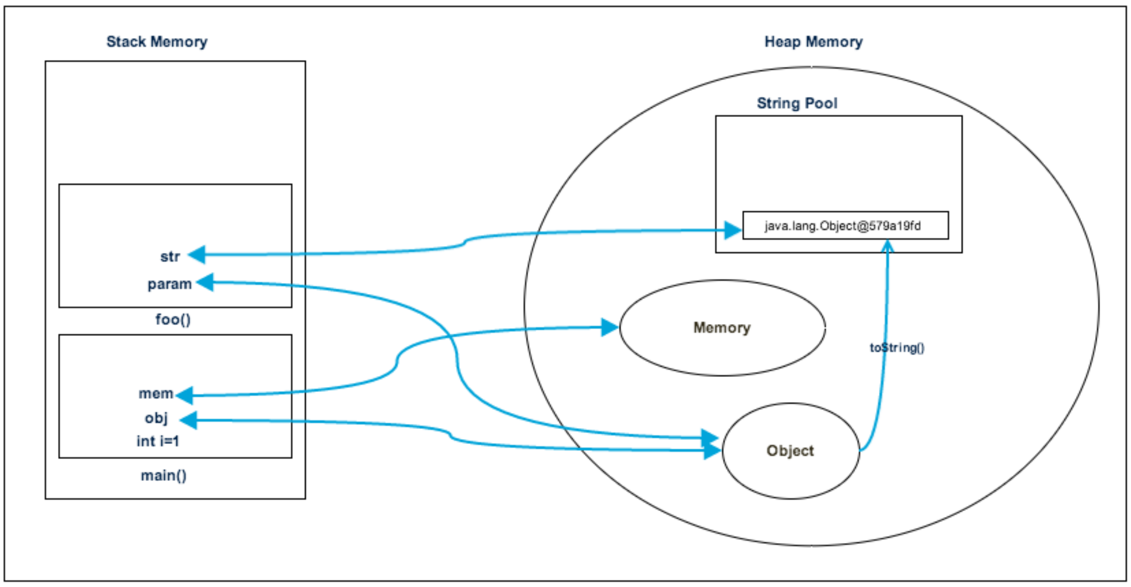
Java Heap Memory

This is the memory where the Objects and JRE classes are stored at the runtime. When the object is created it is stored in the heap. Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application. In multithreading the objects are available for all the threads, means all the threads share the same heap

Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method. Stack memory size is very less compared to Heap memory.

```
public class Memory {

    public static void main(String[] args) { // Line 1
        int i=1; // Line 2
        Object obj = new Object(); // Line 3
        Memory mem = new Memory(); // Line 4
        mem.foo(obj); // Line 5
    } // Line 9

    private void foo(Object param) { // Line 6
        String str = param.toString(); // Line 7
        System.out.println(str);
    } // Line 8

}
```



Java Runtime Memory

- When we run a program it loads all the runtime class into the heap space
- The main method is found at the **Line-1** now the memory for the main method is allocated in the stack region by the main thread.

- We are creating the local variables at **Line-2** it is going to be created in the stack region of main block.
- In **Line-3** we are creating the object the object is going to be created in the heap region and its reference is stored in the stack region, similarly in **Line-4** we are creating one more object.
- In **Line-5** we are calling the method foo( ) than the memory for the method is going to be allocated in the stack region (the block of memory is going to be allocated in the stack region). Since java is pass by value the one more reference is created at the **Line-6**
- From the foo( ) method we are creating one more String object it is going to be created in the heap region, the reference is present in the stack region inside the stack. (String objects are going to be created in the String pool in the heap)
- Has soon has the control reach **Line-8** the method looses the scope, than the memory that was allocated to the method foo( ) becomes free
- When the **Line-9** is executed than the main method also going to loose the scope, and the memory allocated to it also get free. Than java runtime will free al the memory allocated and the execution completes.

Difference between Heap v/s Stack

- Objects stored in the heap are globally accessible whereas stack memory is not accessed by other threads .
- Heap is the place where objects are stored, the stack region is the place where the local variables are stored and the reference to the objects that are stored in heap region.
- Stack memory is short lived where has heap memory is going to exists till the end of application execution.
- When stack memory is full it throws java.lang.StackOverFlowError , where has if heap memory is full java.lang.OutOfMemoryError
- Stack memory is very less when compared to Heap
- Stack is very fast when we compared to Heap

How to set a path of JDK?

```
Set path=C:\Program Files\Java\jdk1.7.0_45\bin
```

How to change the driver?

- Step1 C:\User\Vikas> cd..
- Step2 C:\>d:
- Step3 D:\>

How to create a folder?

- Step1 D:\> mkdir NewFolderName
- Step2 D:\NewFolderName>mkdir OtherFolder
- Step3 D:\NewFolderName\OtherFolder>

How to create a java file?

- Step1 D:\NewFolderName\OtherFolder> notepad FileName.java (if the notepad does not open than open notepad and save it inside a folder (OtherFolder) and save with .java extension

How to compile a java program?

- Step1 D:\NewFolderName\OtherFolder>javac FileName.java
- Step2 D:\NewFolderName\OtherFolder>java FileName

Data type

A **data type** is a classification of the type of data that a variable can hold in computer programming.
Data types in Java are classified into two types:

1. Primitive—which include integer, character, boolean, and floating Point type values.
2. Non-primitive—which include Classes, Interfaces, Object type and Arrays.

Primitive Data Types:

There are eight primitive data types supported by Java.

byte:

- data type is an 8-bit
- value from -128 to 127 (2^7 to 2^7-1)
- default value is 0

short:

- data type is an 16-bit
- value from -32,768 to 32,767 (2^{15} to 2^{15}-1)
- default value is 0

int:

- data type is an 32-bit
- value from - 2,147,483,648 to 2,147,483,647 (2^{31} to 2^{31}-1)
- default value is 0

long:

- data type is an 64-bit
- value from (2^{63} to 2^{63}-1)
- default value is 0L

float:

- data type is an 32-bit
- Float is mainly used to save memory in large arrays of floating point numbers
- default value is 0.0f

double:

- data type is an 64-bit
- This data type is generally used as the default data type for decimal values, generally the default choice
- default value is 0.0

boolean:

- it represents one bit of information
- it can contain only true or false (no 1 or 0 is allowed)
- default value is false

char:

- its sixe is 16 bit
- it is used to store only one character
- default value is space (we represent it by 0)

Variables

A variable is the one which holds some value which can be altered later in the program. Or it is the named storage which can be altered later in the program.

int i; //declaration

i = 20; // initialisation

Methods

- Methods contains group of statements
- A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name
- It increase the readability and reusability of code
- In java we call functions has methods, because hear the methods can be written only inside class but not outside the class has in **C++**
- Methods are used to tell the behaviour of the object
    - **Syntax**

```
<return type> <method Name>(arguments)
                    {
                              // Body of method
                              //return keyword with returning value
                    }
```

Example

```
int add()
          {
                    return 5+6;
          }
```

In the above example the add() method returns 11, which of type int

```
void drink()
          {
                    System.out.println("Drinking water.....!");
          }
```

In the above example the drink() method will not return anything so the return type is void.

Note:

- If we specify any return type while declaring than that method should return that type value.
- The return statement word should be present at the last. In the séance the return statement should be logically the last line in the method block
- If the return type is void means we need not to specify the return statement at the last. Compiler itself is going to write the return statement. If at all if you need to specify means do has below.(a empty return statement)
- void drink()

```
void drink()
          {
                    System.out.println("Drinking water.....!");
                    return;
          }
```

- the method can have empty return statement, if the return type of the method is void. But that return statement should not even return 0(zero) , ‘ ’(space) or null
- Advantage is code reusability
- There are two types of methods
  - **Static methods**
  - **Non-Static methods**
- **Static methods :** These methods a declared using the **static** key word
  - **Example**

```
void static method1()
        {
                System.out.println("In method1() its a static method.....!");
        }
```

- Static method can be accessed directly by just using the class name. From static methods we cant accesses non static members directly
  - **Example :**

```
class AddingMethod
        {
                static void add(int i , int j)
                {
                        System.out.println("Addition of two numbers......");
                        int sum=0;
                        sum = i + j;
                        System.out.println("Sum of "+i+" and "+j+" is = "+sum);
                }
                void sub(int m , int n)
                {
                        System.out.println("Addition of two numbers......");
                        int  diff = 0;
                        diff = m - n;
                        System.out.println("Difference  of "+n+" from "+m+" is = "+diff);
                }
}
class MethodMember
{
        public static void main(String args[])
                {
                        AddingMethod.add(50,60);        // static method is called by class name
                        AddingMethod.sub(150,60);       //Error sub is non-static
                }
}
```

- **Non-static methods :** These methods are declared without static keyword. Non-static methods can accesses static members directly
  - **Example**

```
        void method2()
        {
                System.out.println("In method1() its a non-static method.....!");
        }
```

- *Non-static members are called by creating the instance of the class or by using the object reference of the class*

  - **Example**

```
 class MethodMember
	{
	public static void main(String args[])
		{
		AddingMethod.add(50,60);	// static method is called by class name
		AddingMethod ref1 = new AddingMethod();
		ref1.sub(150,110); // non-static method can be called only using reference
		AddingMethod ref2 = new AddingMethod();
		ref2.add(80,20); // static method can be  also called by using reference
		}
	}
class AddingMethod
{
	static void add(int i,int j)
		{
			System.out.println("Addition of two numbers......");
			int sum=0;
			sum = i + j;
			System.out.println("Sum of "+i+" and "+j+" is = "+sum);
		}
	void sub(int m , int n)
		{
			System.out.println("Addition of two numbers......");
			int  diff = 0;
			diff = m - n;
			System.out.println("Difference  of "+n+" from "+m+" is = "+diff);
		}
}
```

Static members	**Non-Static members**
static members are also known has Class members	*non-static members are also known has Object members*
static members are loaded only once for entire program execution	*non-static members are loaded for every instance of class*
If static members are changed it gets changed for entire program	*if a non-static members of an instance is changed that change will not reflect in other instance of class*
static members can be used across any objects	*non-static members cannot be shared across the Object*
static members are loaded into heap by class loader	*non-static members are loaded into heap when JVM encounter Object creation statement*
all static members reside in common area known has static pool	*non-static member are loaded into Object memory after object creation*
static members are accessed through class name	*non-static members are accessed by using the reference variable*

Class

A *class* is the blueprint from which individual objects are created. Class is a one which contains states (variables) and behaviour (methods)

Syntax –

```
class <className>
{
        //code
}
```

```
 public class Lover
{

        String name;     //state
        int age;         //state
        char sex;        //state it takes only M or F
        double height;   //state

        void takeToPark()        //behaviour
        {
                System.out.println("Take in a R-15");
        }

        int spendMoney()         //behaviour
        {

                int money=5000;
                return money;
        }
}
```

Object

An object is an instance of a class. The term ‘object’, however, refers to an actual **instance** of a class. Every object must belong to a class.

Note → **Objects have a lifespan but classes do not**

Object of a calss can be created using the new keyword and name of the class.

new <className>;

```
public class Test
 {
        public static void main(String[] args)
        {
                Lover   l1      =       new Lover();
                Lover    l2     =       new Lover();
        }
}
```

OOPS CONCEPTS

Inheritance

It is one of the oops concept of java. Inheritance means, it’s a mechanism where the child class acquires the properties of the parent class or supper class.

Example :

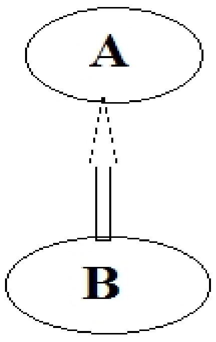
```
                    class Boy
                    {
                            String name;
                            int age;

                            void factory()
                            {
                                    System.out.println("factory() method of boy");
                            }

                            void banglo()
                            {
                                    System.out.println("banglo() method of boy");
                            }
                    }

            class Girl extends Boy
            {
                    double height;
                    String colour;

                    void job()
                    {
                            System.out.println("job() method of Girl");
                    }

                    void prepareFood()
                    {
                            System.out.println("prepareFood() method of Girl");
                    }

                    void details()
                    {
                            System.out.println("Name "+name+" age is "+age+" height is
"+height+" and colour is "+colour);
                    }
            }

class InheritanceProgram
```

```
{
        public static void main(String args[])
        {
                Girl ref = new Girl();
                ref.name="reka";
                ref.age=20;
                ref.factory();
                ref.banglo();

                ref. height =5.5;
                ref.colour="pink";

                ref.job();
                ref.prepareFood();
                ref. details();
        }
}
```

OutPut

```
C:\Users\Vikas\Desktop\java>javac InheritanceProgram.java

C:\Users\Vikas\Desktop\java>java InheritanceProgram

factory() method of boy
banglo() method of boy
job() method of Girl
prepareFood() method of Girl
Name reka age is 20 height is 5.5 and colour is pink
```

In the above example we see the single level inheritance, hear the Girl class inherit all the properties of Boy class . The name, age , factory(), bangle() of Boy class is acquired by the Girl class, so by creating the object of Girl class I can accesses both the Boy class and Girl class properties.



B inherit the properties of **A**
A is called has parent class and **B** is called has child class
A cant acquire the properties of **B** class

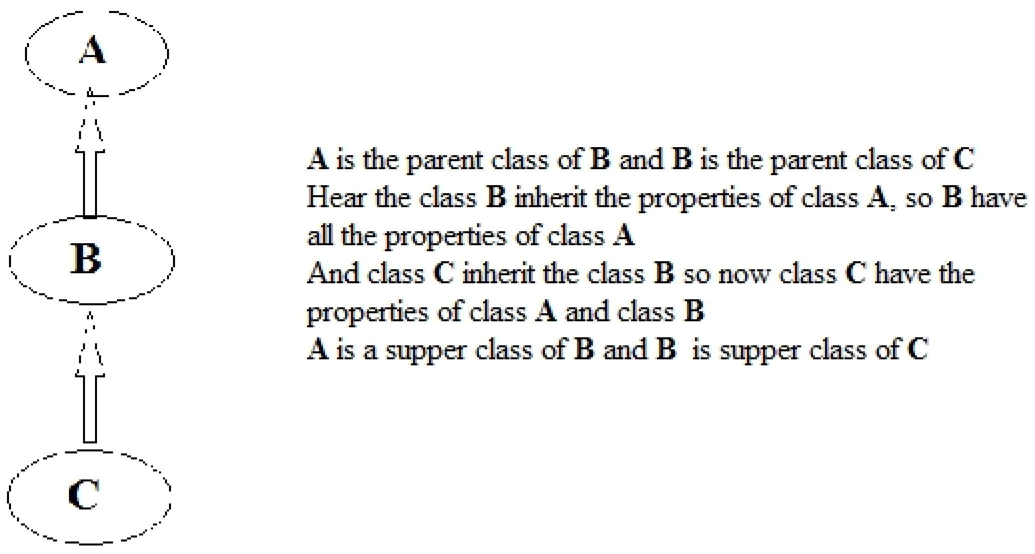
- *java supports inheritance there are many types of in heritance*
    - *single inheritance (above example)*
    - *multilevel inheritance*

- *hierarchical inheritance*

Example for multilevel inheritance

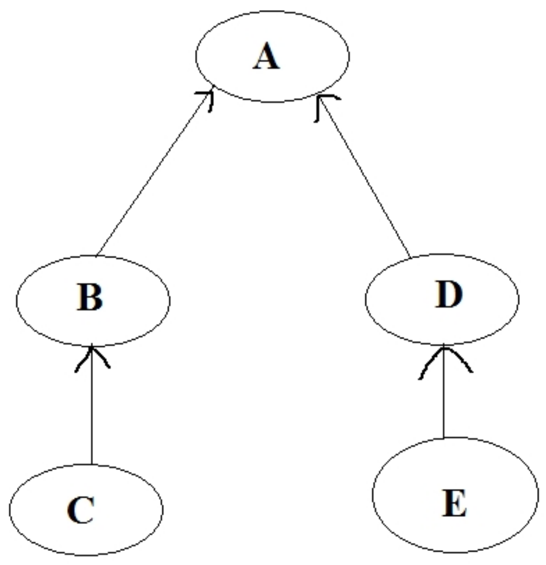
```
class Parent

{

}

class Boy extends Parent

{

}

class Girl extends Boy

{

}
```

In the above example the Boy class can acquire the properties of Parent class, mean while has Girl class extends Boy class, so a Girl class can acquire the properties of both the Parent class and the Boy class.



A is the parent class of **B** and **B** is the parent class of **C**
Hear the class **B** inherit the properties of class **A**, so **B** have all the properties of class **A**
And class **C** inherit the class **B** so now class **C** have the properties of class **A** and class **B**
A is a supper class of **B** and **B** is supper class of **C**

Example for hierarchical inheritance



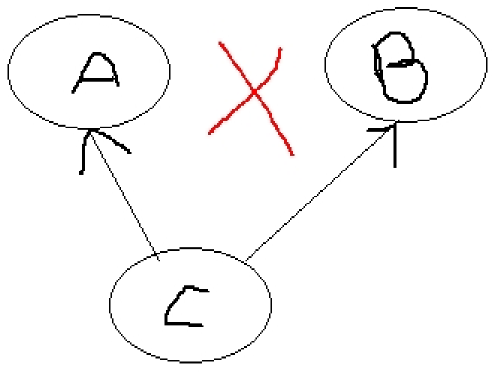
class B and class D extends the class A,
class C extends class B so class C have properties of class B and class A

class E extends class D so class E have properties of class D and class A

There is no relation between class C and class E
There is no relation between class C and class D

There is no relation between class B and class E
There is no relation between class B and class D

Note : *Java does not support the multiple inheritance where has it only supports multilevel inheritance.*



class C extends class A and class B

java does not support this. A single class cant extend multiple class at a time.

If class A and class B have same methods than the class C will be in confusion to take the method from which class so this concept was ruled out in java.

If you create the Object of class C than the constructor is called, hear that constructor will execute the super statement, here again there is a confusion which is parent class and whose constructor is need to be called

IS-A Relationship

- This refers to inheritance or implementation.
- Expressed using keyword “extends”.
- Main advantage is code reusability.

```
class Animal
{
	void walk()
	{
		System.out.println("Walk");
	}
	void eat()
	{
		System.out.println("eat food");
	}
}

class Cow extends Animal //Cow is-a Animal
{
	void giveMilk()
	{
		System.out.println("cow give milk");
	}
	void sound()
	{
		System.out.println("ambaa...ambaaa");
	}
}

class Dog extends Animal //Dog is-a Animal
{
	void sound()
	{
		System.out.println("bow...bow");
	}
}

public class Test {
	public static void main(String[] args) {
		Animal a = new Cow();
		a.walk();
		a.eat();

	a.giveMilk(); //not possible Parent reference cant accesses child member

	a.sound(); //not possible Parent reference cant accesses child member

	Dog d = new Animal(); //not possible because child reference cannot be give to Parent object

	Cow c = new Cow();
		c.walk();
		c.eat();
		c.sound();
		c.giveMilk();
	}
}
```

- *Whatever the parent class has, is by default available to the child. Hence by using child reference, we can call both parent and child class methods.*

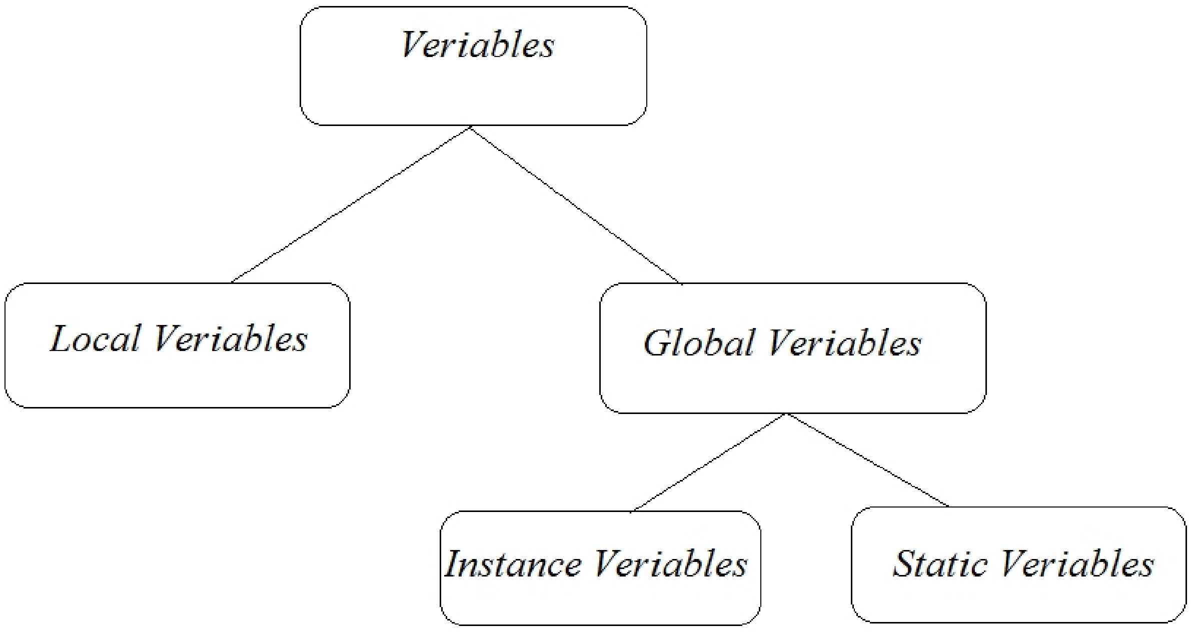
- *Whatever the child class has, by default is not available to parent, hence on the parent class reference we can only call parent class methods but not child specific methods.*
- *Parent class reference can be used to hold child class objects , but by using that reference we can call only parent class methods but not child specific methods.*
- *We cant use child class reference to hold parent class objects*

HAS-A Relationship

- Has-A means an instance of one class “has a” reference to an instance of another class or another instance of same class.
- It is also known as “composition” or “aggregation”.
- There is no specific keyword to implement HAS-A relationship but mostly we are depended upon “new” keyword.

```
class Mobile
{
        void giveCall()
        {
                //some code
        }
        void playGame()
        {
                //some code
        }
}
class Person
{
        Mobile m = new Mobile(); //So we can tell person has a Mobile
        void eatFood()
        {
                //some code
        }
        void drinkWater()
        {
                //some code
        }
}
public class Test
{
        public static void main(String[] args)
        {
                Person p = new Person();

                p.m.giveCall(); //Person has a mobile so accessing it using person object
                p.m.playGame(); //Person has a mobile so accessing it using person object
        }
}
```

Variables



Local Variables

Any variables declared inside a method or constructor is called has local variables. Local variables need to be initialised before they are used. Local variables do not have default value.

Global Variables

Any variables declared inside the class but outside the methods or constructors are called has global variables. Global variables will take a default if u have not initialised it.

Instance variables the global variables declared without static keyword is called has instance variables. These variables get initialised whenever the object is created.
Instance variables are one for the object.

Static variables the global variables declared with the static keyword are called has static variables. These variables will get initialized whenever the class loads into memory. These variables are one for the class.

An **argument** is an expression passed to a function or constructor, where a **parameter** is a reference declared in a function declaration or even constructor.

```
class Gift {

void buy(String  name , int  cost)          //name and cost r called has formal parameters
        {
                //some code hear
        }
void Gift(double  d , char  c)              //d and c are called has formal parameters
        {
                //some code
        }
public static void main(String[] args)
        {
                double   e  = 5.5;
                char   v  =  'M';
                Gift g = new Gift(e , v); //e and v are the actual arguments

                String  hesaru = "barby";
                int  money = 200;

                g.buy(hesaru , money);  //hesaru and money are the actual arguments
                }
    }
```

Constructor

- *Constructor is special kind of method which gets called when object is created*
- *Constructor don't have return type*
- *Constructor cannot be inherited*
- *Constructor name is same has the class name*
- *Every class should have a constructor, if we don't write a constructor compiler will write a default constructor.*
- *Compiler will not write a default constructor, if we are going to write any of the constructor explicitly*
- *Constructor with no parameters or a zero parameter constructor is called has default constructor*
- *Constructor with parameter is called has parameterized constructor. These constructor is used to initialise the instance variables at the time of creating object*
- ***Syntax***

```
        <accesses-specifier> ClassName ( )
                        {
                                //some code…
                        }
```

```
Eg :- class Cow
        {
                //zero parameter constructor of Cow Class
                public Cow()
                {
                        System.out.println("I am in Cow class constructor.....!");
                }
        }
```

Note : If you don't write any constructor inside a class by default there will be a default constructor inside that class, and that default constructor will be similar like a zero-parameterized constructor. Zero parameterized constructor is also called has default constructor.

```
class Cow
                {
                        //zero parameter constructor of Cow Class
                        public Cow()
                        {
                                System.out.println("I am in Cow class constructor.....!");
                        }
                }
                class Test1
                {
                        public static void main(String[] args)
                        {
System.out.println("Before creating object");
Cow ref = new Cow();    //when this line is executed the Cow() constructor is called
System.out.println("After creating 1st object");
new Cow();      // when this line is executed again the Cow() constructor is called
System.out.println("After creating 2nd object");
                        }
                }
```

Output :

```
                Before creating object
                I am in Cow class constructor.....!
                After creating 1st object
                I am in Cow class constructor.....!
                After creating 2nd object
```

Constructor Over Loading

A class can contain more than one constructor inside it, but they should vary with number of parameters or type of parameters. This concept is called has contractor overloading.

```
class Cow
{
        public Cow()
        {
                System.out.println("I am in Cow() constructor.....!");
        }

        public Cow(int leg)
        {
                System.out.println("I am in Cow(int leg) constructor.....!");
        }

        public Cow(double height)
        {
                System.out.println("I am in Cow(double hight) constructor.....!");
        }

        public Cow(int ears , int mouth)
        {
        System.out.println("I am in Cow(int ears , int mouth) constructor.....!");
        }
}
```

These constructors can be called by creating a object like

```
new Cow();              // Cow() constructor is called

new Cow(4.5);  // Cow(double height) constructor is called

new Cow(2,1);  // Cow(int ears , int mouth)constructor is called

new Cow(4);             // Cow(int leg) constructor is called
```

Note :

- The parameterized constructor is used to initialize the instance variables of the class
- If you don't define a constructor, the default constructor (which has no parameters) will be generated by default.
- If you have any user-defined constructor, the default constructor will not be generated at all.

this key word

- *this is a keyword in java which refers to the current class object*
- *if there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity*
- *this keyword refers to current class instance variables*

Example :

```
class SweetHeart
{
        String name;
        int  age , salry;
        double hight;

        SweetHeart(String name , double hight , int age , int salry)
        {
                this.name=name;
                this.hight=hight;
                this.age=age;
                this.salry=salry;
        }
        void detailsOfAge()
        {
                System.out.println("My sweetHeart name is "+name+" and the age is "+age);
        }
        void fullDetails()
        {
                System.out.println("My sweetHeart name is "+name+" and the age is "+age+" hight
                is "+hight+" and get a salry rs "+salry);
        }
}

class Test2
{
        public static void main(String[] args)
        {
                SweetHeart ref1=new SweetHeart("Geetha",4.9,20,15000);
                ref1.fullDetails();

                SweetHeart ref2=new SweetHeart("Ramesh",5.5,18,20000);
                ref2.fullDetails();

                //To print only name and age call detailsOfAge()

                ref1.detailsOfAge();
                ref2.detailsOfAge();

        }
} Try this program and write the output. Pleas practice more programs on all topics
```

Note :

```
class SweetHeart
{
        String name;
        int age,salry;
        double hight;

        SweetHeart(String name , double hight , int age , int salry)
                {
                        name=name;
                        hight=hight;
                        age=age;
                        salry=salry;
                }
}
```

Hear the local variables get more priority so the instance variables will not get a scope to get initialize,

name (local variable) is different than the name (instance variable). So we make use of this keyword to initialize the instance variables of the class, when both local and instance variables name is same.

➢
```
class Test
{
        Test()
        {
                return ;
        }
}
```

This example will compile and run. But return keyword should not return any value

Constructor chaining

- *In Java you can call one constructor from another and it's known as constructor chaining in Java.*
- *We can call the same class constructor using* ***this*** *key word*
- *this( ) must be the 1st line inside a constructor*

Example :

```
class Bus
{
        Bus()
        {
                this("Banglore");          //this("Banglore") will call Bus(String name)
                System.out.println("Bus is empty.....!");
        }

        Bus(String name)
        {
                this(10);          //this(10) will call Bus(int num)
                System.out.println("Bus is having a name.....!");
                }

                Bus(int num)

                {
                        System.out.println("Bus is having a NUMBER.....!");
                }

        }

class ConstructorChaining

        {

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

Output *:*

```
                Bus is having a NUMBER.....!
                Bus is having a name.....!
                Bus is empty.....!
```

- *We can call the parent (super) class constructor by using* ***super*** *key word*

super( ) is used to call the immediate parent class constructor. super( ) must be the 1st line in constructor ***Example :***

```
                         class Train
                        {
                                Train()
                                {
                                        this("Hassan"); //this("Hassan") will call Train(String
        name)
                                        System.out.println("Train is empty.....!");
                                }
                                Train(String name)
                                {
                                        this(55);           //this(10) will call Train(int num)
                                        System.out.println("Train is having a name.....!");
                                }
                                Train(int num)
                                {
                                        System.out.println("Train is having a NUMBER.....!");
                                }
                        }
                        class Bus extends Train
                        {
                                Bus()
                                {
                        this("Banavara");         //this("Banglore") will call Bus(String name)
```

```
                System.out.println("Bus is empty.....!");
        }
        Bus(String name)
        {
                this(10);       //this(10) will call Bus(int num)
                System.out.println("Bus is having a name.....!");
        }
        Bus(int num)
        {
        super();//super() will call Train() because Train is parent class
                System.out.println("Bus is having a NUMBER.....!");
        }
    }
class ConstructorChaining
{
    public static void main(String[] args)
    {
        new Bus();
    }
}
```

OutPut :

```
Train is having a NUMBER.....!
Train is having a name.....!
Train is empty.....!
Bus is having a NUMBER.....!
Bus is having a name.....!
Bus is empty.....!
```

Note :

Constructor chaing should not lead to recursion.

Example:

```
class Bus extends Train

{

    Bus()

    {

        this("Banavara");

        System.out.println("Bus is empty.....!");

    }

    Bus(String name)

    {
```

```
                    this(10);

                    System.out.println("Bus is having a name.....!");

                }

                Bus(int num)

                {

                this();   // Hear the error occurs because it is leading for recursion

                System.out.println("Bus is having a NUMBER.....!");

                }

            }
```

super( ) and this( ) cannot be present inside the same constructor. Any one can be present inside.

```
    Bus(int num)

        {

            this();

            super(); //Error because super( ) must be 1st line

            System.out.println("Bus is having a NUMBER.....!");

        }

    Bus(int num)

        {

            super();

            this(); //Error because this( ) must be 1st line

            System.out.println("Bus is having a NUMBER.....!");

        }
```

Instance-Init Block (non-static block)

- *Instance block will execute whenever a new object is created.*
- *If there are more than one instance block than it will execute in the sequence of the order they are written*
- *These are used to declare the instance variables of the class*

```
class Run1
{
	//instance block
	{
		System.out.println("In INIT block of Run1");
	}
}
class Run2
{
	//instance block
	{
		System.out.println("In INIT block of Run2");
	}
}
class Test1
{
	public static void main(String[] args)
	{
		new Run1();//Now the instance blcok of Run1 class will execute
		System.out.println("1st instance block is created.......!");
		new Run2();//Now the instance blcok of Run2 class will execute
	}
}
```

OutPut

```
D:\bin>javac Test1.java

D:\bin>java Test1

In INIT block of Run1
1st instance block is created.......!
In INIT block of Run2
```

Static Block

- *It's a block of code which is executed when the class gets loaded by a class loader. It is meant to do initialization of static members of the class.*
- *static block executes once in life cycle of any program another property of static block is that it executes before main method*
- *Static blocks are also called Static initialization blocks*

```
	class Test1
	{
		static
		{
			System.out.println("In Test1 static block");
		}
	}

	class Demo1
	{
		public static void main(String[] args)
		{
			new Test1();
		}
	}
```

OutPut

```
		D:\bin>javac Demo1.java
		D:\bin>java Demo1
		In Test1 static block
```

```
		class Test1
		{
			static
			{
				System.out.println("In Test1 first static block");
			}
			static
			{
				System.out.println("In Test1 second static block");
			}
		public static void main(String[] args)
			{
				new Test1(); } }
```

OutPut

```
		D:\bin>javac Demo1.java
		D:\bin>java Demo1
		In Test1 first static block
		In Test1 second static block
```

Note – If there are multiple static blocks than it will execute in the sequence has they present

Static Block	**Non-Static Block**
Gets executed before running the main method. Or it get executed at the time of class get loaded	*Gets executed at the time of Object creation. Or it get executed only after creating a object of class*
These blocks are used to initialize the static members (static variables) of class	*These blocks are used to initialize the non-static members (instance variables) of class*
These block get executed only once for entire program execution	*These block get executed when ever the new object is created*

Inside static block static members can be accessed directly where as non static members should be accessed by using the reference of the object
 	
class StaticBlock
{
int a=23;
static int b=15;
 	
static {
 System.out.println("Adding two number.");
 	
  int sum=0;
  sum = a + b; //Error because a is non-static. b can be accessed because b is static
 }
 	
}
	Inside the non-static block, both static and non-static members can be accessed directly
	
	class InstanceBlock
	{
	 int a=23;
	 static int b=15;
	
	 {
	  System.out.println("Addition of two numbers......");
	  int sum=0;
	  sum = a + b; // no error because its non-static block
	 }
	
	}

Static block is called has static initialization block ***(SIB)***	*Non-static block is called has instance initialization block* ***(IIB)***

Difference between constructor and method

Constructor	*Method*
Name of constructor should be same has the class name	*Name of method can be anything other than a class name*
Constructor cannot be inherited	*Method can be inherited*
Constructor cant return anything	*Method can return a value*
By default the compiler will have a default constructor	*There is nothing called has default method*
Constructor is used to initialize the instance variables of class	*Method is used to tell the behavior of class*

Method Overriding

Method overriding means having a different implementation of the same method in the inherited class. These two methods would have the same signature, but different implementation. One of these would exist in the base class and another in the derived class. These cannot exist in the same class.

A subclass inherits methods from a superclass. Sometimes, it is necessary for the subclass to modify the methods defined in the superclass. This is referred to as method overriding. The following example demonstrates method overriding.

```
class Father
{
	String name;
	double height;
	int age;

	void factory()
	{
		System.out.println("Parent factory()");
	}
	void home()
	{
		System.out.println("some home()");
	}
}

class Son extends Father
{

	void factory() //Son modifying Father factory()
	{
		System.out.println("modified Parent factory() by Son“);
	}
	void home() //Son modifying Father factory()

	{
		System.out.println("modified Parent home() by Son");
	}
}
public class Test {
public static void main(String[] args) {
	Father f = new Son();

f.factory(); //hear the Son's factory() is called, because Parent factory() is overridden by Son

f.home();  //hear the Son's home() is called, because Parent home() is overridden by Son
}
}
```

Rules for method overriding:

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: if the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Type Casting in JAVA

Java supports two types of castings – **primitive data type casting** and **reference type casting**. Reference type casting is nothing but assigning one Java object to another object. It comes with very strict rules and is explained clearly in Object Casting(next topic). Now let us go for data type casting.

a) data type casting

Java **data type** casting comes with 3 flavors.

1. **Implicit casting**
2. **Explicit casting**
3. **Boolean casting.**

1. Implicit casting (widening conversion)

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.

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

		int a = 10;
		double d = a;
		System.out.println(d); //prints 10.0

	}
}
```

2. Explicit casting (narrowing conversion)

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.

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

                double d = 10.89;
                int c = d ; //Error
                double d = 10.89;
                int c = d ; //Error
                int a = (int)d; //explicit casting

        }

}
```

3. Boolean casting

A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is incompatible for conversion.

NOTE :

byte –> short –> int –> long –> float –> double

In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is, byte can be assigned to short implicitly but short to byte requires explicit casting.

b) Reference type casting

- Subclass object can be assigned to a super class object and this casting is done implicitly. This is known as upcasting (upwards in the hierarchy from subclass to super class)
  Eg : **Father f = new Son();**//implicit casting

- Java does not permit to assign a super class object to a subclass object (implicitly) and still to do so, we need **explicit casting**. This is known as down casting (super class to subclass). Down casting requires explicit conversion. In down casting , first you need to do the upcasting after upcasting than only u can perform the down casting. Or else its a classCastException
  Eg : **Father f = new Son();** //upcasting
  Son s = (Son) f; //down casting

  Son s = (Son) new Father(); //compiles but at runtime give exception

```
class Father
{
	void factory()
	{
		System.out.println("Parent factory()");
	}
	void home()
	{
		System.out.println("some home()");
	}
}

class Son extends Father
{

	void factory() {
	System.out.println("modified Parent factory() by Son");
	}
	void home()
	{
		System.out.println("modified Parent home() by Son");
	}
	void car()
	{
		System.out.println("Son alone has car()");
	}

}
```

implicit casting (implicit UPCASTING)

```
public class Test {
public static void main(String[] args)
	{
		Father f = new Son();//implicit casting
		f.factory();
		f.home();
		f.car(); //Error cant accesses Son class methods using parent reference
	}
}
```

explicit casting (explicit DOWNCASTING)

```
public class Test {
public static void main(String[] args) {
	Father f =  new Son();// implicit casting
	Son s = (Son)f;
	s.factory();
	s.home();
	s.car(); //no error because the explicit casting is done
}
}
```

So by explicit casting we can accesses the child class members using the parent class reference

Abstract class

An *abstract class* is a class that is declared with the abstract key word, it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be sub classed. Cant create the object of abstract class

```
public abstract class Game
{
            //abstract class without any methods
}

public abstract class Cade
  {

   abstract void draw();

            //abstract class with abstract methods
  }

abstract class Plan
{
        void costEstimation()
        {
                //some code to tell about cost to build
        }
        abstract void baseMent();
        abstract void build();

} //abstract class with concreate and abstract methods
```

Whenever you inherit the abstract class you need to override the abstract methods of the abstract(PARENT) class in the inherited (CHILD) class. OR else you should declare the child class also has abstract class. See the example below

Example :

```
class Volvo
{
        void body()
        {
                //code
        }
        void ac();
        void music();
        void seats();
}

abstract class ElectricWork extends Volvo
{
        void ac()
        {
                //code
        }
        void music()
        {
                //code
        }
}
class KSRTC extends ElectricWork
{
        void seats()
        {
                //code
        }
}
```

- Hear ElectricWork class is also abstract because all the abstract methods of Volvo class has not been overridden in this class, i.e seats() method is not overridden.
  KSRTC class has overridden all the parents class abstract methods so it need not to be abstract class

Important Points about Abstract class

- abstract keyword is used to make a class abstract.
- Abstract class can't be instantiated.
- We can use abstract keyword to create an abstract method, an abstract method doesn't have body.
- If a class have abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.
- Its not necessary to have abstract classes to have abstract method.
- If abstract class doesn't have any method implementation, its better to use interface because java doesn't support multiple class inheritance.
- The subclass of abstract class must implement all the abstract methods unless the subclass is also an abstract class.
- Abstract classes can implement interfaces without even providing the implementation of interface methods.
- Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation.
- Abstract class can have a constructor but it can be called from the child class using supper statement
- We can run abstract class like any other class if it has main() method.
- In interview give some good examples like bank or google example (refer class notes)

Example1:

```
abstract class MediaPlayer

{
        void playSong(String songName)
        {
                System.out.println(songName+" is playing in the player");
        }
        abstract void pauseControle();
        abstract void screenControler();
}
```

```
class Windows extends MediaPlayer
{
        void pauseControle()
        {
                System.out.println("Press P to pause the song");
        }
        void screenControler()
        {
                System.out.println("Do changes using menu button");
        }
}

class VLC extends MediaPlayer
{
        void pauseControle()
        {
                System.out.println("Press space-bar to pause the song");
        }
        void screenControler()
        {
                System.out.println("Press A to change the screen size");
        }

}
```

Interface

- Interface in java is one of the way to achieve 100% abstraction in Java .
- Interface in java is declared using keyword interface. It is also just like normal class. For interface also the .class file is going to be generated.
- All methods inside interface are public by default
- All variables declared inside interface is implicitly public final variable or constants.

```
interface Office
{
        int employee;//Error because only final variables so they need to be initialised has
donoted in below example. No other types are allowed inside interface
}

interface Office
{
        int employee=100;
}
```

- All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword
- Cant create the Object of interface
- In Java its legal for an interface to extend multiple interface. for example following code will run without any compilation error

```
package com.vikas.jspiders;
interface Animal
{
	public void eat();
	public void walk();
}
interface Human
{
	public void talk();
	public void work();
}
interface Livingbeing extends Animal,Human // extend multiple interface
{
	public void enjoyWithLover();
}
```

Whenever we implement interface we need to provide implementation to all the abstract methods of it, or else we need to make the inherited class also has abstract.

```
interface Office
{
	void work();
	public void  enjoy();
}
abstract class Manager implements Office
{
	//implementation is not give for work() and enjoy() so class should be abstract
}
```

Example:-

```
interface Animal
{
	public void eat();
	public void walk();
}
interface Human
{
	public void talk();
	public void work();
}
interface Livingbeing extends Animal,Human
{
	public void enjoyWithLover();
}
public class Test implements Livingbeing{

	public void eat()
	{
		// TODO Auto-generated method stub
	}
	public void walk()
	{
		// TODO Auto-generated method stub
	}
	public void talk()
	{
		// TODO Auto-generated method stub
	}
	public void work()
	{
		// TODO Auto-generated method stub
	}
	public void enjoyWithLover()
	{
		// TODO Auto-generated method stub
	}

}
```

The Interface reference can be give to its child class object. It behave same has the inheritance. Using the interface reference we cant accesses the child class members. But we can accesses the overrided methods.

Example:

```
interface Animal
{
	public void eat();
	public void walk();
}

class Cow implements Animal
{

	public void eat()
	{
		System.out.println("Cow eat hullu");
	}

	public void walk()
	{
		System.out.println("Cow walk in four legs");
	}
	public void giveMilk()
	{
		System.out.println("Cow give white milk");
	}

}
class Test1
{
	public static void main(String[] args) {

		Animal a = new Cow();
		a.eat();
		a.walk();
		a.giveMilk();//Error because parent reference cant accesses child members
	}
}
```

	Abstract	**Interface**

Multiple Inheritance	A class can inherit only one Abstract Class	A class can implement multiple Interfaces
Concrete methods (Default Behavior)	In abstract class you can provide default behavior of a method, so that even if child class does not provide its own behavior, you do have a default behavior to work with	You cannot provide a default behavior in interfaces. Interfaces only allow you to provide signature of the method
Access Modifiers	You can provide access modifiers to methods in abstract classes (static , final, etc)	You cannot provide access modifiers methods in Interfaces.
Methods and variables	The methods or variables can be declared with any accesses specifiers and modifiers. (Note : but abstract methods should not be static , final, private)	All methods are public by default, all the variables are static final by default.

Abstraction

abstraction in Java is used to hide certain details and only show the essential features of the object.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

We are going to achieve abstraction using

Abstract class
Interface

Encapsulation

Encapsulation is the packing of data and functions into a single component. Encapsulation is the ability to package data, related behaviour in an object bundle and control/restrict access to them (both data and function) from other objects. It is all about packaging related stuff together and hide them from external elements.

Encapsulation can be achieved by declaring fields in a class as private, while providing access to these fields via public, typically, getter and setter methods. Or else we can tell that the encapsulation can be achieved by java beans.

```
package com.vikas.jspiders;

import java.util.Scanner;

class Register
{
	private String name,email,password;
	private int age;
	private long phone;
	private double hight;
	private char gender;
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public long getPhone() {
		return phone;
	}
	public void setPhone(long phone) {
		this.phone = phone;
	}
	public double getHight() {
		return hight;
	}
	public void setHight(double hight) {
		this.hight = hight;
	}
	public char getGender() {
		return gender;
	}
	public void setGender(char gender) {
		this.gender = gender;
	}

}
class RegisterDetails
{
	void insertDetails(Register r)
	{
		System.out.println("Name is "+r.getName());
		System.out.println("Email is "+r.getEmail());
		System.out.println("Password is "+r.getPassword());
		System.out.println("Age is "+r.getAge());
		System.out.println("Gender is "+r.getGender());
		System.out.println("Phone number "+r.getPhone());
		System.out.println("Height is "+r.getHight());

	}
```

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


        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the name");
        String name = scan.next();
        System.out.println("Enter the Email");
        String email = scan.next();
        System.out.println("Enter the password");
        String password = scan.next();
        System.out.println("Enter the age");
        int age=scan.nextInt();
        System.out.println("Enter phone number");
        long l = scan.nextLong();
        System.out.println("Enter height");
        double height = scan.nextDouble();
        System.out.println("Enter gender");
        char gender =(char) scan.next().charAt(0);

        Register r = new Register();
        r.setName(name);
        r.setEmail(email);
        r.setPassword(password);
        r.setHight(hight);
        r.setAge(age);
        r.setGender(gender);

        RegisterDetails rd = new RegisterDetails();
        rd.insertDetails(r);// hear the bean object is sent to the insertDetails method in RegisterDetails
class
}

}
```

Output

```
Enter the name
vikas
Enter the Email
vikasbanavara@gmail.com
Enter the password
12345
Enter the age
25
Enter phone number
8147214063
Enter height
5.5
Enter gender
M
```

Out put after taking input

Name is vikas
Email is vikasbanavara@gmail.com
Password is 12345
Age is 25
Gender is M
Phone number 8147214063
Height is 5.5

Polymorphism

Polymorphism is the ability by which, we can create functions or reference variables which behaves differently in different programmatic context. Polymorphism is a Greek word which means many(poly) forms(morphism).
Polymorphism is tightly coupled to inheritance and is one of the most powerful advantages to object-oriented technologies.

Polymorphism is essentially considered into two versions.

1. Compile time polymorphism (static binding or method overloading)
2. Runtime polymorphism (dynamic binding or method overriding)

Note : see class notes for overload and overriding examples

Static binding	Dynamic binding
When type of the object is determined at compiled time(by the compiler), it is known as static binding	When type of the object is determined at run-time, it is known as dynamic binding
If there is any private, final or static method in a class, there is static binding	Other methods than private , final , static
Static binding uses Type(Class in Java) information for binding	Dynamic binding uses Object to resolve binding.
Overloaded methods are bonded using static binding	Overridden methods are bonded using dynamic binding at runtime.
Compile time polymorphism	Runtime polymorphism

What Is a Package?
Packages are nothing but folder structure. It groups different class and files according to their relations which performs corresponding relation. (Eg : How you keep the photos separately in the folders, and also sometimes u group different photos into different folders)
Steps to create package –

- Create New Java Project
- Select src folder and right click on it and select the package option
- Than give the package name has you like but it is good practice to start the package name with com (eg : **com.vikas.jspider) com** means commercial
- **Or** you can right click on src and select the new and than u can also select folder name and type the name and create the package

Note – Package name should be the 1st line of the code

What is import?
We use import statement to indicate for our JVM to import the particular class from the package

Note –

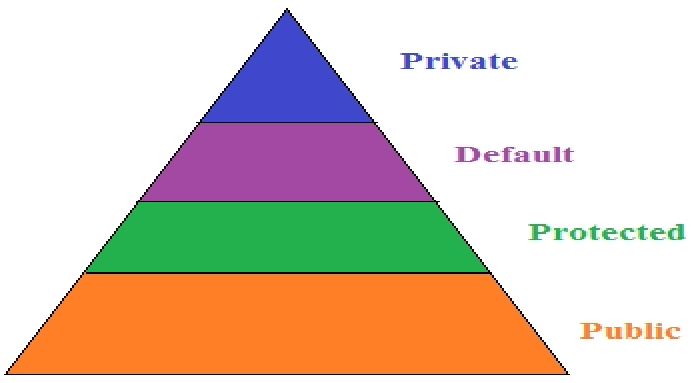
- Import must be immediate line after the package and before the class
- We cannot import different class which has the same name from multiple packages
- It imports class which r in same package but it does not import the class which are present in sub packages. So we need to import them
- The java.lang package is imported by the compiler, and we need not to do it explicitly. The java.lang package is the default package

Structure of code

```
PackageName


Import statements


class <className>
{
        //code
}
```

Accesses specifiers

- **Public**
- **Protected**
- **Default**
- **Private**
    - ➢ The accesses specifiers will define the scope of every member
    - ➢ In java accessibility of any member can be controlled by using accesses modifiers. In picture the accessibility can be given has



- **PUBLIC** is the high visible access specifier in java which can be accessed every were
- **PRIVATE** is the highly restricted access specifier in java that it can be accessed form only inside the class. Such that the members which are declared has private cannot be accessed from outside the class
- **DEFAULT** the scope of default members in only inside the package. They can be accessed within package but not from outside the package. If we don't specify any access specifier than the member would be taken has default access specifier
- **PROTECTED** the protected members can be accessed within the package, but they can be accessed outside the package also only in the case of inheritance

Access modifiers

We know access specifiers specify the access and access modifiers modify the access of variables, methods and classes. In general, an access modifier works between the classes of the same application or within the classes of same package.

Six modifiers exist that can be used with methods.

final Subclass cannot override
abstract No method body exists
native Java method takes the help of underlying OS
static Object is not necessary to call
synchronized Used to lock the source and renders a thread-safe operation
strictfp Guarantees the same precision for floating-point values in the class irrespective of the OS on which the program is executed

Four modifiers and four specifiers exist that can be applied to variables

static Also known as "class variable". No object is necessary to call
final cannot be reassigned
transient Not serialized
volatile Value is more liable to change

Three modifiers exist that can be applied to classes. Modifier comes just before the class name

abstract Objects cannot be created
final cannot be inherited
strictfp Guarantees the same precision for floating-point values in the class irrespective of the OS on which the program is executed

Points to note in method overriding with respect to access specifiers

- **If super class method is public, while overriding it in child class, it should be public, Since it is the weakest access specifier or least restrictive access specifier**
- **If super class method is protected, while overriding it in child class, it can be public or protected but not anything else**
- **If superclass method is default, then while overriding it in child class, it can be public, protected or no access, but cannot be private.**

- **If superclass method is private, then while overriding it in child class, it can be anything. Since private method cannot be overrided.**

Singleton

One instance allowed for a class. In the séance we can create only one object of that class and we cant create multiple object of that class.

```
public class SingletonObject {

        private SingletonObject()

        {

        }

        private static SingletonObject singleObject=new SingletonObject();

        public static SingletonObject giveObject()

        {

                return singleObject;

        }

}
```

Program to get the object of this SingletonObject class

```
public class TestForSingle {

public static void main(String[] args) {

        SingletonObject obj1=SingletonObject.giveObject();

        System.out.println(obj1);

        SingletonObject obj2=SingletonObject.giveObject();

        System.out.println(obj2);

        }

}
```

OutPut of the above program is it will print the reference of the Objects, has it is singleton only one object is created so both have same reference

```
        com.vikas.SingletonObject@19821f
        com.vikas.SingletonObject@19821f
```

Object Class

Object class is the super class of all the classes in java. The Object class reference can be given to any child object or any class object. Every class in the Java system is a descendent (direct or indirect) of the Object class

The object class methods are

- public int **hashCode()**
- public boolean **equals(Object obj)**
- public String **toString()**
- **protected** Object **clone()**
- **protected** void **finalise()**
- public final Class **getClass()**
- public final void **notify()**
- public final void **notifyAll()**
- public final void **wait(long time)**
- public final void **wait(long time, int nanoSec)**
- public final void **wait()**

hashCode() If you only override hash-code method nothing will happen. It will return the same hashcode for all the objects of that class. In the below example all the objects of cow class will have the same hash code 123.

If you don't override hash-code method nothing will happen Because it always return new hash-code for each object as an Object class. Overriding only hash code method will not have much effect, so whenever u over ride hash code method better to override the equals(Object) method also.

```
package com.vicky;
public class Cow
{
        String name;
        void giveMilk()
        {
                System.out.println("Cow give milk");
        }
        public int hashCode() //over riding hash code method
        {
                return 123;
        }
}

package com.vicky;
public class CheckHashCode
{
public static void main(String[] args)
{
        Cow c = new Cow();
        c.name="getha";
        System.out.println(c);
```

```
		Cow c1 = new Cow();
		c1.name="getha";
		System.out.println(c1);

		System.out.println("c.equals(c1) is "+c.equals(c1));
		}
}
```

OutPut

```
com.vicky.Cow@7b  (Note – 7b is a hexa decimal value of 123)
com.vicky.Cow@7b

c.equals(c1) is false
```

We generally override hash code method in HashSet , HashMap. Hear even though the both the objects hash code is same and also the content is same, c.equals(c1) is false. So over riding only hash code method will not give us the proper result we need, so whenever we over ride the hash code method we should also over ride the equals(Object) method also

equals(Object obj) This method is used to tell whether the objects are equal are not, it always checks for the content of the object

```
package com.vicky;
public class CheckHashCode
{
public static void main(String[] args)
{
		Cow c = new Cow();
		c.name="getha";
		c.age=24;
		System.out.println(c);

		Cow c1 = new Cow();
		c1.name="getha";
		c1.age=24;
		System.out.println(c1);

		System.out.println("c.equals(c1) is "+c.equals(c1));

		}
}

package com.vicky;
public class Cow
{
		String name;
		int age;
		void giveMilk()
		{
				System.out.println("Cow give milk");
		}
		public int hashCode()
		{
				int a = this.name.hashCode();
```

```
			return a;
		}

public boolean equals(Object obj) {
		if(obj instanceof Cow)
		{
			Cow c = (Cow) obj;
			if(this.name.hashCode()==c.name.hashCode())
			{
				if(this.age==c.age)
				{
					return true;
				}
				else
				{
					return false;
				}

				return true;
			}
			else
				return false;
		}
		else
			return false;
		}
}
```

OutPut

```
com.vicky.Cow@5db1ecf
com.vicky.Cow@5db1ecf

c.equals(c1) is true
```

equals method always check for the content of the object, so I have overridden my class equals method which also check the content of my class objects.

toString()

The toString() method in the Object class is used to display some information regarding any object. If any code needs some information of an object of a class, then it can get it by using this method. The toString() method of an object gets invoked automatically, when an object reference is passed in the System.out.println() method.

```
package com.vicky;
public class Cow
{
		String name;
		void giveMilk()
		{
			System.out.println("Cow give milk");
		}

		public String toString() {
			return "this is a Cow class";
		}
}
package com.vicky;
```

```
public class CheckHashCode
{
public static void main(String[] args)
{
            Cow c = new Cow();
            c.name="getha";
            System.out.println(c);

            Cow c1 = new Cow();
            c1.name="getha";
            System.out.println(c1);

            }
}
```

Output

```
this is a Cow class
this is a Cow class
```

clone()

The object cloning is a way to create exact copy of an object. In cloning the content of that object is also cloned(copied) into the new object. But both the objects are different from one another, change in one object after cloning will not affect the content of another object.

The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.

```
public class Cow implements Cloneable
{
            String name;
            void giveMilk()
            {
                        System.out.println("Cow give milk");
            }

            protected Object clone() throws CloneNotSupportedException {
                        return super.clone();
            }
}

public class CheckHashCode
{
public static void main(String[] args) throws CloneNotSupportedException
{
            Cow c = new Cow();
            c.name="getha";
            System.out.println(c);

            Cow newObj = (Cow) c.clone();
            }
}
```

Scanning input in java

In java we can scan the input in different ways

- BufferedReader and InputStreamReader classes
- DataInputStream class
- Console class
- Scanner class

An **InputStreamReader** is a bridge from byte streams to character streams, It reads bytes and decodes them into characters. The BufferedReader class provides buffering to your Reader's. Buffering can speed up IO quite a bit. Rather than read one character at a time from the network or disk, you read a larger block at a time. This is typically much faster, especially for disk access and larger data amounts.

The main difference between BufferedReader and BufferedInputStream is that Reader's work on characters (text), wheres InputStream's works on raw bytes.

void	**close**( ) Closes the stream and releases any system resources associated with it.
void	**mark**(int readAheadLimit) Marks the present position in the stream.
boolean	**markSupported**( ) Tells whether this stream supports the mark() operation, which it does.
int	**read**( ) Reads a single character.
int	**read**(char[ ] cbuf, int off, int len) Reads characters into a portion of an array.
String	**readLine**( ) Reads a line of text.
boolean	**ready**( ) Tells whether this stream is ready to be read.
void	**reset** ( ) Resets the stream to the most recent mark.
long	**skip**(long n) Skips characters.

```
package com.vikas.jspiders;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class BufferedReaderClass {
        public static void main(String[] args) throws IOException
          {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                System.out.println("Enter the carector");
                char c = (char)br.read(); // reads a single charector
                System.out.println("Enter the String");
                String name = br.readLine(); // reads the string set of charectors
                System.out.println("Charector is "+c);
                System.out.println("String is "+name);
          }
        }
```

A **DataInputStream** lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream. DataInputStream is not necessarily safe for multithreaded access. Thread safety is optional and is the responsibility of users of methods in this class. Refer File Handling for more example on this class

```
package com.vikas.jspiders;
import java.io.DataInputStream;
import java.io.IOException;

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

```
            {
                    DataInputStream d = new DataInputStream(System.in);
                    System.out.println("Enter string");
                    String name = d.~~readLine()~~;
                    System.out.println(name);
            }
}
```

Console class Actually this is a class in Java that used to input and output to the console window. There are many methods which are used to read the input and methods to print the output also. But this class does not work with your ID such has eclipse so please try this example with your command prompt.

```
package com.vikas.jspiders;
import java.io.Console;
import java.io.IOException;

public class ScannerClass {
        public static void main(String[] args) throws IOException
            {
                    Console c = System.console();
                    String s = c.readLine();
                    char sp[] = c.readPassword();
                    System.out.println(sp[1]);
            }
        }
```

Scanner is a class in java. Which is present in java.util package. Scanner is used to scan the input for the program. Scanner will help to scan the input from the keyboard and also we can read the content of the file has a input for the program.

```
Scanner scanFromKey = new Scanner(System.in);

Scanner scanFromFile = new Scanner(new FileReader("vikas.txt"));
```

Methods of scanner class

Method	Description
byte nextByte()	To scan the byte
short nextShort()	To scan the short
int nextInt( )	To scan the int, there are also some methods like nextBigInteger( )
double nextDouble( )	To scan double
float nextFloat( )	To scan the float
String next( )	To scan the string until the space in encountered
String nextLine( )	To scan the entire line until the new line carector is encountered

There are few more methods present in the scanner class, they are just used to find the next type of element stored in the input.

Method	*Returns*
boolean hasNextLine()	Returns true if the scanner has another line in its input; false otherwise.

boolean hasNextInt()	Returns true if the next token in the scanner can be interpreted as an int value.
boolean hasNextFloat()	Returns true if the next toke in the scanner can be interpreted as a float value.

```
package com.vikas.jspiders;
import java.util.Scanner;

public class ScannerClass {
        public static void main(String[] args)
          {
            double number;
            Scanner in = new Scanner(System.in);
            System.out.println("Enter your gross income: ");
            if (in.hasNextInt())
            {
              number = (double)in.nextInt();
              System.out.println("You entered " + number);
            }
            else if (in.hasNextFloat())
            {
              number = (double)in.nextFloat();
              System.out.println("You entered " + number);
            }
            else if (in.hasNextDouble())
            {
              number = in.nextDouble();
              System.out.println("You entered " + number);
            }
            else
              System.out.println("Token not an integer or a real value.");
          }
}
```

String class

Java String is a immutable object. For an immutable object you cannot modify any of its attribute's values. Once you have created a java String object it cannot be modified to some other object or a different String. String is one of the widely used java classes

- Strings are not null terminated in Java.
- Strings are immutable and final in Java
- Strings are maintained in String Pool
- Use equals methods for comparing String in Java
- Use indexOf() and lastIndexOf() or matches(String regex) method to search inside String
- Use SubString to get part of String in Java
- "+" is overloaded for String concatenation
- Use trim() to remove white spaces from String
- Use split() for splitting String using Regular expression

Difference between String , StringBuffer and StringBuilder

The most important difference between String and StringBuffer , StringBuilder in java is that String objects are **immutable** (can't change the content of String Object)

StringBuffer and StringBuilder objects are **mutable** (can change the content of StringBuffer and StringBuilder objects). StringBuilder and StringBuffer are almost the same.

The difference is that StringBuffer is **synchronized** and StringBuilder is not. Although, StringBuilder is faster than StringBuffer, the difference in performance is very little. StringBuilder is a SUN's replacement of StringBuffer. It just avoids synchronization from all the public methods. Rather than that, their functionality is the same.

We can create the objects of String , StringBuilder and StringBuffer has shown below. There are few methods in each class refer them.

String

```
String  name  =  “vikas”;
String  place  =  new String(“banavara”);
```

String buffer

```
StringBuffer sbf = new StringBuffer( );
        sbf.append(“Hassan”);
StringBuffer sbf = new StringBuffer(10);
        sbf.append(“Banglore”);
StringBuffer sbf = new StringBuffer(“Mysore”);
```

String builder

```
StringBuilder sbr = new StringBuilder ( );
        sbr.append(“Kannada”);
StringBuilder sbr = new StringBuilder (10);
        sbr.append(“Kaaveri”);
StringBuilder sbr = new StringBuilder (“Mysore”);
```

Assertion

Asseertion is a statement, it is used to detect the program errors. We can tell that assertion is used to rais the error, if anything goes wrong while running the program.

Syntax 1 : assert expression;

If the expression is not true than the AssertionError is going to be generated

```
import java.util.Scanner;
public class Program {
	public static void main(String[] args) {
		Scanner scan = new Scanner(System.in);
		System.out.println("Enter the age of voter");
		int voter = scan.nextInt();
		assert (voter>=18);
		System.out.println("Thank you for voting");
	}
	} Note : Run has     java  -ea Program (-ea means enable assertion)
```

Output

```
Enter the age of voter
21
Thank you for voting
```

Output

```
Enter the age of voter
15
Exception in thread "main" java.lang.AssertionError
	at com.vikas.jspiders.AssertionProgram.main(AssertionProgram.java:9)
```

Syntax 2 : assert expression : message;

If the expression is not true than the message is going to be printed along with the line number and class name;

```
import java.util.Scanner;
public class AssertionProgram {
	public static void main(String[] args) {
		Scanner scan = new Scanner(System.in);
		System.out.println("Enter the age of GIRL and BOY ");
		int girl = scan.nextInt();
		int boy = scan.nextInt();

		assert (girl>=18)  : "girl is too young";
		assert (boy>=21) : "Boy is too young";

		System.out.println("You can mari each other wish u good luck");
	}
	} Note : Run has     java  -ea Program (-ea means enable assertion)
```

Output

```
Enter the age of GIRL and BOY
15
23
Exception in thread "main" java.lang.AssertionError: girl is too young
	at com.vikas.jspiders.AssertionProgram.main(AssertionProgram.java:13)
```

Enumeration

The Java programming language contains the enum keyword, which represents a special data type that enables for a variable to belong to a set of predefined constants. The variable must be equal to one of the values that have been predefined for it.

The values defined inside an enum are constants. Also, these values are implicitly **static** and **final** and cannot be changed, after their declaration. If an enum is a member of a class, then it is implicitly defined as **static**. Finally, you should use an enum, when you need to define and represent a fixed set of constants.

```
class Test
{
	enum E
	{
		a, b, c, d;
	}
	public static void main(String[] args)
	{
		E e1 = E.b;
		System.out.println(e1);
		E e2 = E.d;
		System.out.println(e2);
		System.out.println(e1.ordinal());//ordinal() is used to get the position of that enum
		System.out.println(e2.ordinal());
		System.out.println(E.valueOf("a").ordinal());
	}
}
```

Output

```
b
d
1
3
0
```

Enums in Java are considered to be reference types, like class or Interface and thus, a programmer can define constructor, methods and variables, inside an enum

An enum is an abstract data type with values that each take on exactly one of a finite set of identifiers that you specify. Apex provides built-in enums, such as LoggingLevel, and you can define your own enum.

All Apex enums, whether user-defined enums or built-in enums, have the following common method that takes no arguments.

values

This method returns the values of the Enum as a list of the same Enum type. Each Enum value has the following methods that take no arguments.

name

Returns the name of the Enum item as a String.

ordinal

Returns the position of the item, as an Integer, in the list of Enum values starting with zero.

Example2:

```
class Test
{
	enum Month
	{
		JAN(31), FEB(28), MAR(31);

		int days;

		Month(int days)
		{
			this.days = days;
		}

		int getDays()
		{
			return days;
		}
	}

	public static void main(String[] args)
	{
		Month m1 = Month.FEB;
		System.out.println(m1);
		System.out.println(m1.getDays());
		System.out.println(m1.days);
		System.out.println("--------");
		Month m2 = Month.JAN;
		System.out.println(m2);
		System.out.println(m2.getDays());
		System.out.println(m2.days);
		System.out.println("--------");
	}
}
```

Output

```
FEB
28
28
--------
JAN
31
31
```

Example 3:

```
class Test
{
	enum A
	{
		CON1, CON2,
		CON3
		{
			void test()
			{
				System.out.println("CSCB-test");
			}
		},CON4, CON5;
```

```
                void test()
                {
                        System.out.println("test");
                }
        }

        public static void main(String[] args)
        {
                A a1 = A.CON5;
                A a2 = A.CON1;
                A a3 = A.CON3;
                A a4 = A.CON4;
                System.out.println(a1);
                System.out.println(a2);
                System.out.println(a3);
                System.out.println(a4);
                a1.test();
                a2.test();
                a3.test();//this line will call the method inside it in the séance CON3
                a4.test();
        }
}
```

Output

```
CON5
CON1
CON3
CON4
test
test
CSCB-test
test
```

Array

Array is a collection of similar type of elements that, all elements are stored in same memory location. We can store the fixed number of elements in the array(i mean the mentioned size of elements).

There are two types of array

One dimensional array
Multi dimensional array

To check the size of array

Syntax : **arrayName.length**

Declaring Array

One Dimensional Array : dataType variable[ ] = new dataType[size];

Example :

int[ ] a = new int[5];
int [ ]a = new int[5];
int a[ ] = new int[5];
int a[ ] = {1,2,3,4,5,6}; //Declaration & initialization in a single line

Two dimensional array : dataType variable[ ][ ] = new dataType[row-size][column-size];

Example :

```
int[ ][ ] a = new int[3][3];
int  [ ]a[ ] = new int[3][3];
int a[ ][ ] = new int[3][3];
int [ ][ ]a = new int[3][3];
int a[ ][ ] = {{1,2,3},{4,5,6}};
```

Jagged Array

Array in which number of columns different in each row called jagged array.

```
int a[ ][ ] = new int[4][ ];

a[0] = new int[5];
a[1] = new int[2];
a[2] = new int[1];
a[3] = new int[4];
```

a[0]th row contains 5-columns
a[1]th row contains 2-columns
a[2]th row contains 1-columns
a[3]th row contains 4-columns

Note : -

To check the size of array or to get the size of array

arrayName.length

To check the size of individual row of array

arrayName[row-number].length

```
public class Program
{
	public static void main(String[] args) throws IOException {
		Scanner s = new Scanner(System.in);
		int a[][] = new int[3][];
		a[0]=new int[4];
		a[1]=new int[1];
		a[2]=new int[2];
		for(int i=0;i<a.length;i++)
		{
			System.out.println("Enter "+a[i].length+" elements");
			for(int j=0;j<a[i].length;j++)
			{
				a[i][j]=s.nextInt();
			}
		}
		System.out.println("Entered elements are");
		for(int i=0;i<a.length;i++)
		{
			for(int j=0;j<a[i].length;j++)
			{
				System.out.print(a[i][j]+" ");
			}
			System.out.println();
		}
	}
}
```

Arrays is a class in java it is present in **java.util** package. The Arrays class contain lot of inbuilt methods which are going to help us to do the operations, like sort , search , copy one array to other and etc... and these methods are static methods. You can accesses them like **Arrays.methodname** and pass the required fields has the arguments. The methods in this class throw NullPointerException if the specified array reference is null.

- **public static void sort(int[] a)**
  Sorts the specified array into ascending numerical order.

- **public static int binarySearch(int[] a , int key)**
  Searches the specified array of ints for the specified value using the binary search algorithm. The array must be sorted (as by the sort(int[]) method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.
- **public static boolean equals(int[] a,int[] a2)**
  Returns true if the two specified arrays of ints are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are null.
- **public static void fill(int[] a,int val)**
  Assigns the specified int value to each element of the specified array of ints.
- **public static int[] copyOf(int[] original,int newLength)**
  Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain 0. Such indices will exist if and only if the specified length is greater than that of the original array.
- **public static int[] copyOfRange(int[] original,int from,int to)**
  Copies the specified range of the specified array into a new array. The initial index of the range (from) must lie between zero and original.length, inclusive. The value at original[from] is placed into the initial element of the copy (unless from == original.length or from == to). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (to), which must be greater than or equal to from, may be greater than original.length, in which case 0 is placed in all elements of the copy whose index is greater than or equal to original.length - from. The length of the returned array will be to - from.

- **public static String toString(int[] a)**
  Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters ", " (a comma followed by a space). Elements are converted to strings as by String.valueOf(int). Returns "null" if a is null.

There are few more methods of Arrays class refer them from
http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Wrapper classes

Wrapper classes are used to convert any primitive type into an object. The primitive data types are not objects, they do not belong to any class, they are defined in the language itself. While storing in data structures which support only objects, it is required to convert the primitive type to object first, so we go for wrapper class. Wrapper classes are used to convert any data type into an object.

Premitive data type	Wrpaer class
char	**Character**
byte	**Byte**
short	**Short**
int	**Integer**
long	**Long**
float	**Float**
double	**Double**
boolean	**Boolean**

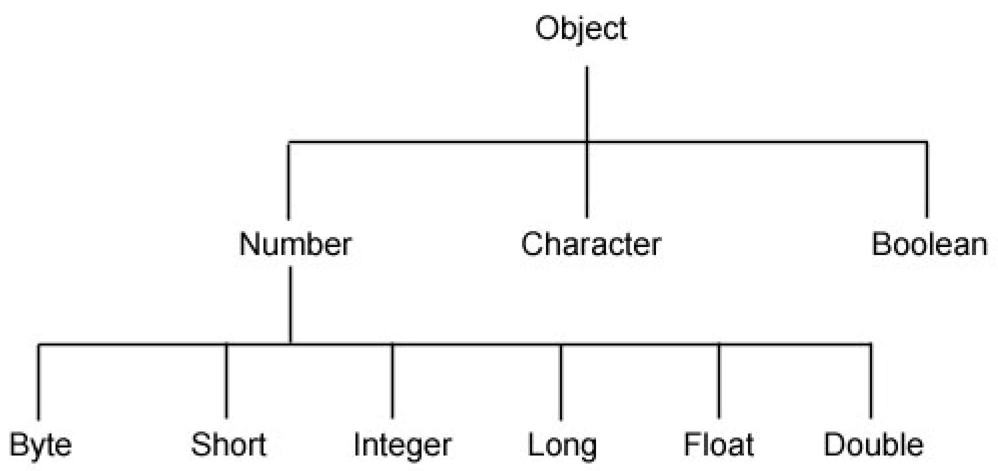


Figure : Wrapper classes Hierarchy

As you can observe in the above hierarchy, the super class of all numeric wrapper classes is Number and the super class for Character and Boolean is Object. All the wrapper classes are defined as final and thus designers prevented them from inheritance

Number class is an abstract class. It has some following methods
byte byteValue() this method converts calling object into byte value
short shortVlaue()
int intValue()
long longValue()
folat floatValue()
double doubleValue()

Boxing and unboxing in java

Boxing, otherwise known as wrapping, is the process of placing a primitive type within an object so that the primitive can be used as a reference object

Auto-boxing is the **automatic** conversion that the **Java** compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on

Un-boxing is a process by which the value of object is automatically extracted from a type wrapper

Integer i = 100; //autoboxing
int j = i; //unboxing

Parsing in java

Parsing is to read the value of one object to convert it to another type. For example you may have a string with a value of "10". Internally that string contains the Unicode characters '1' and '0' not the actual number 10. The method Integer.parseInt takes that string value and returns a real number.

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

		String s1 = "80";
		String s2 = "90";

		int i = Integer.parseInt(s1);
		int j = Integer.parseInt(s2);
		System.out.println(i+" + "+j+" = "+i+j);
	 }
}
```

Note : similarly you can use paserseDouble, parseFloat and so on.....
but not parseChar

Character class

This provides many special method few are listed in the below program.

```
public class CharWrap {
public static void main(String[] args) {
        char c = 'a';
String s ="vikas 123 @ \n gamil";
        char cc[] = s.toCharArray();
        for(int i =0;i<cc.length;i++){
        Character ci = new Character(cc[i]);
        if(ci.isDigit(cc[i]))
        {
                System.out.println(cc[i]+" is digit");
        }
        else if(ci.isLetter(cc[i]))
        {
                System.out.println(cc[i]+" is letter");
        }
        else if(ci.isSpaceChar(cc[i]))
        {
                System.out.println("this is a space by pressing space bar");
        }
        else if(ci.isWhitespace(cc[i]))
        {
                System.out.println("this is a space by pressing enter, tab or backspace buttons");
        }
        else if(ci.isLetterOrDigit(cc[i]))
        {
                System.out.println("this is a number or a alpha character");
        }
        else
        {
                System.out.println(cc[i]+" its a special character");
        }
        }
}
```

Output

```
v is letter
i is letter
k is letter
a is letter
s is letter
this is a space by pressing space bar
1 is digit
2 is digit
3 is digit
this is a space by pressing space bar
@ its a special character
this is a space by pressing space bar
this is a space by pressing enter, tab or backspace buttons
this is a space by pressing space bar
g is letter
a is letter
m is letter
i is letter
l is letter
```

Date class

Java provides the **Date** class that is available in **java.util** package, this class gives the current date and time.

```
import java.util.Date;
public class DateTime {
	public static void main(String[] args) {
		Date d = new Date();
		System.out.println(d);
	 }
}
```
Output
```
Tue Jun 24 23:08:29 IST 2014
```

You can use
getDay()
getMonth()
getYear()
getMinutes()
getHour()
getSeconds()
getTime() //it will give in milliseconds

Eg: Date d = new Date();

d.getDay(); similarly u can use any date class **get** methods or **set** methods using the Date class reference.

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class C {

	public static void main(String[] args) {
		Date d = new Date();
		System.out.println(d);
	System.out.println("******************");
		SimpleDateFormat ss = new SimpleDateFormat("E dd-MM-YYYY hh:mm
a");
		System.out.println(ss.format(d));
	}
}
```

Output

```
Tue Sep 30 17:30:34 IST 2014
******************
Tue 30-09-2014 05:30 PM
```

Simple DateFormat format codes:

To specify the time format use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following:

Character	Description	Example
G	Era designator	AD

y	Year in four digits	2001
M	Month in year	July or 07
d	Day in month	10
h	Hour in A.M./P.M. (1~12)	12
H	Hour in day (0~23)	22
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	234
E	Day in week	Tuesday
D	Day in year	360
F	Day of week in month	2 (second Wed. in July)
w	Week in year	40
W	Week in month	1
a	A.M./P.M. marker	PM
k	Hour in day (1~24)	24
K	Hour in A.M./P.M. (0~11)	10
z	Time zone	Eastern Standard Time

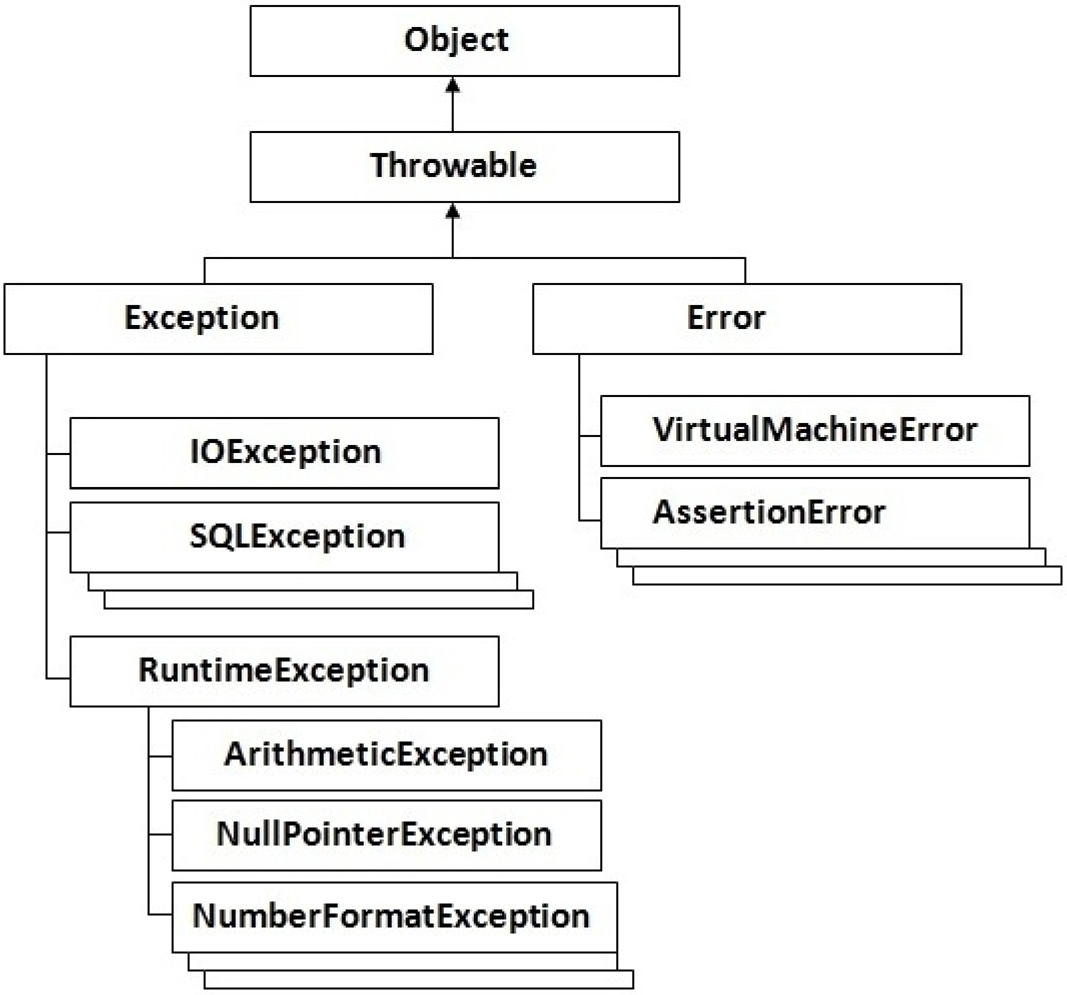
Calender Class

This class is used to get the system date and time. Even this class is also present in the java.utli package.

```
import java.util.Calendar;
public class DateTime {
	public static void main(String[] args) {

	Calendar cal = Calendar.getInstance();
	int day = cal.DATE;
	int mon = cal.MONTH;
	int yer = cal.YEAR;
	System.out.println("Date is "+day+" - "+mon+" - "+yer);

	int hour = cal.HOUR;
	int min = cal.MINUTE;
	int sec = cal.SECOND;
	System.out.println("Time is "+hour+" - "+min+" - "+sec);
	}
}
```

Exception Handling

An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. Exception handling is one of the technique which is used to handle the run time error. Exception means the abnormal condition.



The **Throwable** class is the superclass of all errors and exceptions in the **Java** language. Only objects that are instances of this class (or one of its subclasses) are thrown by the **Java** Virtual Machine or can be thrown by the **Java** throw statement.

Errors: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

In **exception** there are two types

Checked exception
Unchecked Exception

Checked Exceptions:

The exception that extends Exception is called has checked exceptions. The checked exceptions are the exceptions that need to be handled at the compile time itself. It is compulsory or mandatory to handle the checked exception or else its a compile time error. (Please see the example for checked exception in file handling and JDBC programs)

Eg: ClassNotFound
SqlException
FileNotFoundException
IOException

InterruptedException
CloneNotSupportedException

Unchecked Exception

The exception class that extends the RuntimeException are called has unchecked exceptions. It is not mandatory to handle the unchecked exceptions.

Eg: ArithmeticException
ArrayIndexOutOfBoundsException
NullPointerException
NumberFormatException

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

		String s = "vikas";
		int i = Integer.parseInt(s);//java.lang.NumberFormatException

		String name=null;
		int len = name.length();//java.lang.NullPointerException

		int a[] = new int[5];
		a[6]=95;//java.lang.ArrayIndexOutOfBoundsException
	}
}
```

Hear the **int** i = Integer.*parseInt*(s); is going to generate the exception, but we are not handling it, so the JVM will handle it and it will terminate the program. And rest part of the program will not get executed.

So we need to handle the exception where ever it might get generated, so we can handle the exception using the try catch block.

```
	try {
	   }
	catch (Exception e)
	{
	}
```

If we handle the exception than the program will continue to execute or else it will terminate from the point where the exception occurs. So its a good practice to handle all the exceptions before they get generated.

```
public class UnchkExp {
	public static void main(String[] args) {
		String s = "vikas";
		try {
			int i = Integer.parseInt(s);
		}
		catch (NumberFormatException e)
		{
			System.out.println("Sorry "+s+" cannot be converted to int");
		}

		String name=null;
		try {
			int len = name.length();
		}
```

```
                catch (NullPointerException e)
                {
                System.out.println("Sorry you are trying to get something from null");
                }

                int a[] = new int[5];
                try {
                        a[6] = 95;
                }
                catch (ArrayIndexOutOfBoundsException e)
                {
                System.out.println("Sorry you are crossing the size limit of array");
                }
        }
}
```

In exception handling we come across five keywords they are.

try
catch
finally
throw
throws

Note:
See the class notes for more and clear examples.

Finally : finally is the block which contain the code that need to executed even though the exception occurs(after catch block execution) and also even after execution of try block.

```
try {

} catch (Exception e) {

}
finally
{

}
```

We cant write only try block without a catch block, but we can write a try block with the finally block

Throw v/s Throws in java

1. **Throws clause** in used to declare an exception and **throw** keyword is used to throw an exception explicitly.
2. If we see syntax wise than **throw** is followed by an instance variable and **throws** is followed by exception class names.
3. The keyword **throw** is used inside method body to invoke an exception and **throws clause** is used in method declaration (signature).

Creating custom exception

Creating Unchecked exception

To create our own unchecked exception, we need to extend RuntimeException class. To get the proper message has you write, override getMessage() or toString() method.

```
public class MockException extends RuntimeException
{
	public String getMessage()
	{
		return "Sorry you have less mock rating not eligible for interview";
	}
	public String toString()
	{
		return "Sorry you have less mock rating not eligible for interview";
	}
}


public class Verify
{
	void check(double rate)
	{
		if(rate<4)
		{
			MockException m = new MockException();
			System.out.println(m);
		}
		else
		{
			System.out.println("Wish u good luck for interview");
		}
	}
}

import java.util.Scanner;
public class StudentDetails {
public static void main(String[] args) {
	Scanner scan = new Scanner(System.in);
	System.out.println("Enter the jspider mock rate");
	double rate = scan.nextDouble();
	Verify v = new Verify();
		v.check(rate);
	}
}
```

Output:

```
Enter the jspider mock rate
4
Wish u good luck for interview
```

Note:-If exception occurs than output will be

```
Enter the mock rate
2
Sorry you have less mock rating not eligible for interview
```

Creating Checked exception

To create our own unchecked exception, we need to extend Exception class. To get the proper message has you write, override getMessage() or toString() method. This is the Checked exception, so when ever you through a checked exception you need to mention a **throws** declaration in the method, and you also should handle that excetion from the place where you are calling that method, or else again you can use the throws declarations to through it back to JVM but has a developers we should not through any exception back to JVM.

```
public class Pub extends Exception
{
	public String toString()
	{
		return "Sorry you are too young to enter a Pub";
	}
	public String getMessage()
	{
		return "Sorry you are still a kid to enter the pub";
	}
}
public class Verify
{
	void check(double age) throws Pub
	{
		if(age<18)
		{
			throw new Pub();
		}
		else
		{
			System.out.println("Welcome to pub enjoy have masthi");
		}
	}
}

import java.util.Scanner;
public class People {
public static void main(String[] args) {
	Scanner scan = new Scanner(System.in);
	System.out.println("Enter your age");
	double age = scan.nextDouble();
	Verify v = new Verify();

	try
	{
		v.check(age);
	}
	catch (Pub e)
	{
		System.out.println(e);
		//or
		System.out.println(e.getMessage());
	}
}
}
```

Output

```
Enter your age
22
Welcome to pub enjoy have masthi
```

Note:-If exception occurs than output will be

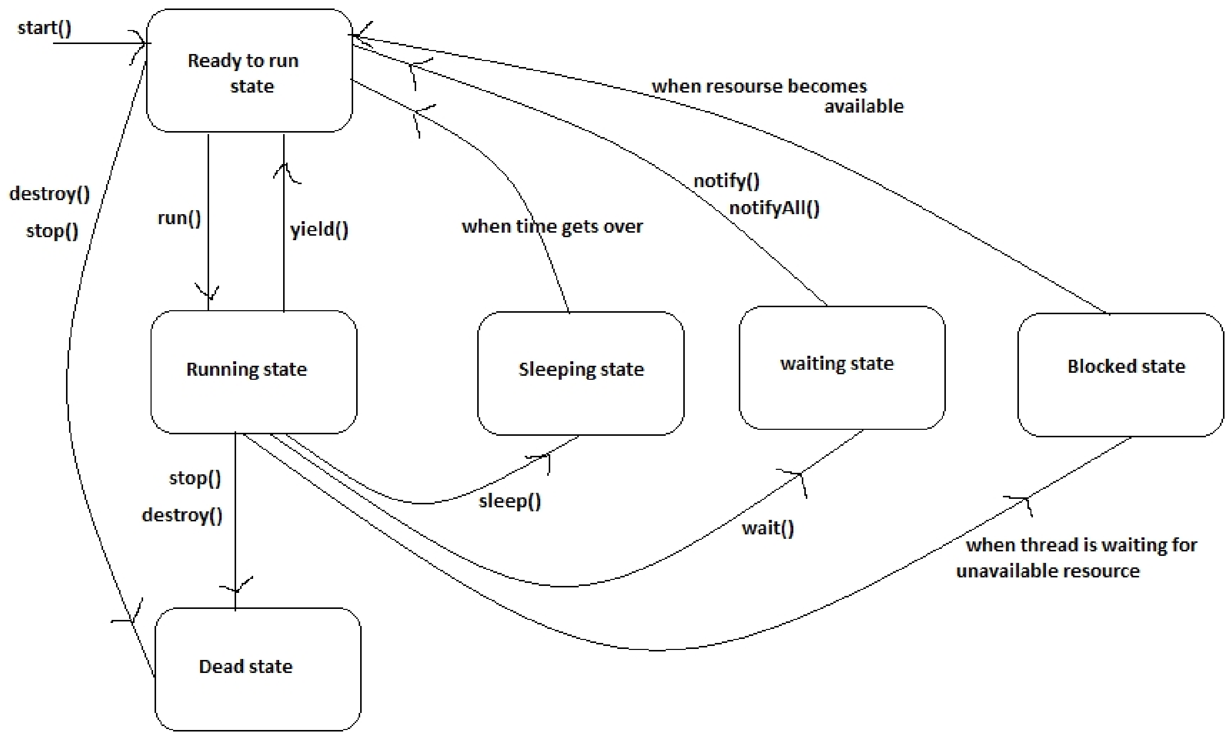
```
Enter your age
15
Sorry you are too young to enter a Pub
Sorry you are still a kid to enter the pub (if you call e.getMessage( ))
```

Threads

Diffrence between Multithreading and multitasking

Multithreading	Multitasking
The program is organised or divided into multiple paths of execution and are executed concurrently by the CPU	Multiple tasks are executed concerrently by the CPU
Multithreading is a feature provided by programming language	Multitasking is a feature provided by the OS
The process has to switch between the diffrent lines of execution	The process has to switch between diffrent process(or programs)
It is loght weight and hence is a highly efficent	It is heavy weight and hence is inefficent
The smallest unit in multithreading is a thread	The smallest unit in multitasking is a task
All the threads would use same memory	Every program would have its own memory

Thread Life Cycle



Thread is a program which has separate path of execution. A program would contain many threads. Multi threading ensures that the cpu time is not wasted.

The JVM will create main thread and this main thread is the one which is going to run main method. Any thread that is created by the user is called has child thread.

For each thread there will be the priority, we can set it using setPriority(int i) method, where i is the value from 1-10, 1 means low priority and 10 is the maximum priority. The main thread priority is 5.

We can get the priority of the thread by calling getPriority( ) method. If we don’t set any priority the default priority will be 5.

```
public class CreatingThreads {
public static void main(String[] args) {
        Thread t1 = new Thread();
        Thread t2 = new Thread();
        t1.setPriority(2);
        t2.setPriority(8);

        System.out.println("Priority of thread "+t2.getPriority());
        System.out.println("thread name Before setting");
        System.out.println(t1.getName());
        System.out.println(t2.getName());

         t1.setName("vikas");
         t2.setName("jspiders");
         System.out.println("thread name after setting");
         System.out.println(t1.getName());
         System.out.println(t2.getName()); }
}
```

The above program show how to set and get the name and priority of the thread

Output

```
Priority of thread 8
thread name Before setting
Thread-0
Thread-1
thread name after setting
vikas
jspiders
```

Creating the thread class can be done by extending Thread class or implementing Runnable interface (refer class notes for the examples)

1) Implementing Runnable is the preferred way to do it. Here, you’re not really specializing or modifying the thread’s behavior. You’re just giving the thread something to run. That means composition is the better way to go.

2) Java only supports single inheritance, so you can only extend one class.

3) Instantiating an interface gives a cleane separation between your code and the implementation of threads.

4) Implementing Runnable makes your class more flexible. If you extend thread then the action you’re doing is always going to be in a thread. However, if you extend Runnable it doesn’t have to be. You can run it in a thread, or pass it to some kind of executor service, or just pass it around as a task within a single threaded application.

5) By extending Thread, each of your threads has a unique object associated with it, whereas implementing Runnable, many threads can share the same runnable instance.

Daemon thread

Daemon thread is a service provider thread. It provides the service to the user created thread. When all the user thread dies, the JVM will also terminate the daemon thread.

- **It provides service to user thread**
- **It is alive only till the user thread is alive**

- **It has low priority**
- **setDaemon(true) method is used to make thread has daemon, isDaemon() method is used to check weather the thread is Daemon or not.**

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

Note

- In the above example t1 is the demon thread
- The thread need to be declared has demon before it starts. By using the method setDemon(true).
- Demon thread fails or stop executing has soon has the parent thread dies or terminates.
- t2 is not demon thread , because by default for all threads setDemon(false) is made by the compiler, so we are explicitly making the t1 has demon by setDemon(true).

What is the difference between sleep( ) , yield( ) and wait( )

sleep( ) will ask the thread to sleep for a while(until the given interval of time), or to stop executing for a particular interval of time. wait( ) is used to ask the thread to go for waiting state until some one notify it. yield( ) is the method which is used to allow the other thread to execute

What is the difference between notify( ) , notifyAll( )

notify( ) method is used to notify only one thread which is present in the waiting pool, it notify the thread which went for the waiting state for the first time. notifyAll( ) is the method which is used to notify all the thread which are in waiting pool.

What is the use of join( ) method

join( ) is the method which is used to wait for the other thread until it joins the thread which is waiting for it. Please refer class notes for example

Synchronized block

In a method if you want to make only few lines has a thread safe, than put those lines inside the synchronized block. If you put all the lines of method inside the synchronized block it works similar to 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 refrence)
          {
                    //code
          }
```

The below example shows you how to use the Synchronized block, hear I have written sleep method just to feel the thread effect, while executing. All the lines present inside the method are not thread safe, only the line4,line5, line6 are thread safe because they are present inside the Synchronized block, but the line1 , line2 , line3 are not thread safe, such that they can be accessed by the multiple threads at a time

```
public class SyncBlockProg{
        void m()
        {
                try{

                        System.out.println("line1");
                        Thread.sleep(600);
                        System.out.println("line2");
                        Thread.sleep(600);
                        System.out.println("line3");
                        Thread.sleep(600);

                        synchronized (this)
                        {
                                System.out.println("line4");
                                Thread.sleep(600);
                                System.out.println("line5");
                                Thread.sleep(600);
                                System.out.println("line6");
                        }
                }
                catch(InterruptedException e)
                {
                        System.out.println("Sleep was disterbed");
                }
        }
}

public class SyncBlockMain
{
        public static void main(String[] args) {
                final C c = new C();
                Thread t1 = new Thread()
                {
                        public void run() {
                                c.m();
                        }
                };
                Thread t2 = new Thread()
                {
                        public void run() {
                                c.m();
                        }
                };
                t1.start();
                t2.start();
        }
}
```

Output
line1
line1
line2
line2
line3
line3
line4
line5
line6
line4
line5
line6

Deadlock

It is the situation that occurs in the multi threading programs. Hear the each thread depend on one another, each thread waits for other thread to release the lock. OR

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

```
public class A
{
        public static B b1 = new B();
        public static B b2 = new B();
        public static void main(String[] args)
        {
                Thread t1 = new Thread()
                {
                        public void run()
                        {
                                synchronized (b1)
                                {
                                        System.out.println("t1 locked b1");
                                        System.out.println("t1 Waiting for b2");

                                        synchronized (b2)
                                        {
                                                System.out.println("t1 locked by b2 & b1");
                                        }
                                }

                        }
                };
                Thread t2 = new Thread()
```

```
		{
			public void run()
			{
				synchronized (b2)
				{
					System.out.println("t2 locked b2");
					System.out.println("t2 Waiting for b1");

					synchronized (b1)
					{
						System.out.println("t2 locked b1 & b2");
					}
				}
			}
		};
		t1.start();
		t2.start();
	}
}
```

Output

```
t2 locked b2
t1 locked b1
t2 Waiting for b1
t1 Waiting for b2
```

In the above example we can see that both the threads are started and they are running, the Thread-1 (t1) is holding the b1 object, and it is waiting for the b2 object, but the Thread-2 is holding b2 object and it is waiting for the b1. Thread-2 keep waiting for b1 but the b1 object is locked by Thread-1. Similarly Thread-1 is waiting for b2 and it is locked by Thread-2, both the threads are waiting for the object release, which are never going to be released hence both the thread go into the deadlock state.

Has a developers we should not write any program which rise into dead lock condition, to over come this condition we need to take care that all the threads accesses the objects in the same order, has shown below.

```
public class A
{
	public static B b1 = new B();
	public static B b2 = new B();
	public static void main(String[] args)
	{
		Thread t1 = new Thread()
		{
			public void run()
			{
				synchronized (b1)
				{
					System.out.println("t1 locked b1");
					System.out.println("t1 Waiting for b2");

					synchronized (b2)
					{
						System.out.println("t1 locked by b2 & b1");
					}
				}
				}
			};
			Thread t2 = new Thread()
			{
				public void run()
				{
					synchronized (b1)
					{
						System.out.println("t2 locked b2");
						System.out.println("t2 Waiting for b1");

						synchronized (b2)
						{
							System.out.println("t2 locked b1 & b2");
						}
					}
				}
			};
			t1.start();
			t2.start();
		}
}
```

Output

```
t1 locked b1
t1 Waiting for b2
t1 locked by b2 & b1
t2 locked b2
t2 Waiting for b1
t2 locked b1 & b2
```

In the above program Thread-1 has locked b1, so until Thread-1 release the b1 the Thread-2 cant lock b1, so that block is not going to execute until b1 is released by Thread-1, has soon has Thread-1 releases b1, the Thread-2 will take the charge.

Collection Frame Work

- Frame work is a collection of many utilities which are present in a class or an interface.
- Collection frame work defines several class and interfaces which can be used to represent a group of objects as a single entiry
- Collection frame work gives many readymade utilities, has a developers it's our responsibility to use the readymade things as per the program requirement
- All the class and interface of collection frame work is present in **java.util** package

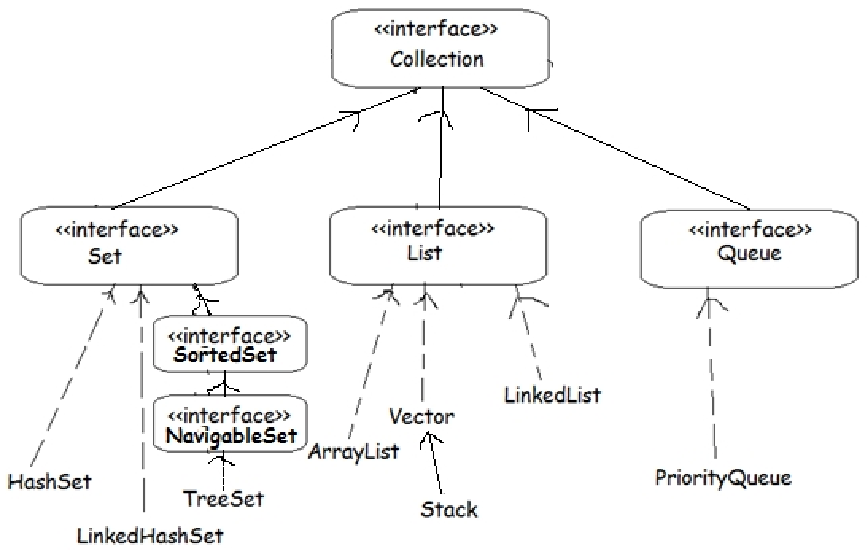
Difference between array and collection

Array	Collection
Stores primitive has well has non primitive.	Store only non primitive (Objects)
It's a fixed size	Its dynamically growing
We don't have any methods to deal with any data hence may not give object oriented approach	We have many utility methods and also many methods that deals with data hence gives object oriented approach
Memory point of view array is not recommended	It is recommended, because memory increases has per the load

Collection

A group of individual objects as a single entity is called has collection.

- If we want to represent a group of individual objects into one entity than we should go for Collection
- The Collection is a interface present in the java.util package
- The Collection interface contains lot of methods which can be applicable for any Collection type Objects



- Collection and Map interfaces are the part of collection framework but they are two separate vertical.
- Map is not a child of collection interface
- Collection and Map both can be used with generics or can be used independently without generics
- We cant use any primitives(int,float,double….etc) with generics
- Generic should be any object type not any other type (Integer, Double, Student……etc)

List

- It's a child interface of Collection, such that List interface extends Collection interface
- If you want to represent the group of objects in the order of insertion and if you want to allow the duplicate values than you can go for List.
- Index based
- Ordered (insertion order)
- Accepts duplicate
- Multiple null values are accepted
- List has many child class
    - ArrayList **Class**
    - LinkedList **Class**
    - Vector **Class**
        - Stack **sub Class of Vector**

ArrayList

The object of ArrayList can be created in three ways has shown below

```
ArrayList al1 = new ArrayList();
ArrayList al2 = new ArrayList(int   inetialCapacity);
ArrayList al3 = new ArrayList(Collection   ref);
```

Collection means any Collection type object

LinkedList

The object of LinkedList can be created in two ways has shown below

```
LinkedList li1 = new LinkedList();
LinkedList li2 = new LinkedList(Collection);
```

Collection means any Collection type object

Vector

The object of Vector can be created in four ways

```
Vector v1 = new Vector();
 Vector v2 = new Vector(Collection);
Vector v3 = new Vector(int inetialCapacity);
Vector v4 = new Vector(int inetialCapacity, int growthRate);
```

Growth rate means the rate at which the vector grows after filling the allotted memory.

Stack

The object of Stack can be created only in one way

```
Stack s1 = new Stack();
```

Special methods of stack class

Object push(Object o) push the object into stack
Object pop( ) remove the top most Object from stack
Object peek( ) gives you the top most Object of stack
boolean empty( ) gives true if stack is empty else false
int search(Object o) return position or the index of the object if found or else it return -1

Set

- It's a child interface of Collection, such that List interface extends Collection interface
- Non-index based
- No duplicates are allowed
- Un-ordered
- Accepts only one NULL
- Set has some child class and interfaces
    - HashSet **Class**
        - LinkedHashSet **sub Class of HashSet**
    - SortedSet **child interface of Set**
        - NavigableSet **child interface of SortedSet**

- TreeSet **Class**

SortedSet

It’s a child interface of Set

If we want to represent a group of individual objects according to some sorting order than we should go for SortedSet

NavigableSet

It is the child interface of SortedSet to provide several methods for navigation purpose

It was introduced in 1.6

HashSet

The object of HashSet class can be created in the four ways by calling any one of the below constructor

```
HashSet hs1 = new HashSet();
HashSet hs2 = new HashSet(Collection);
HashSet hs3 = new HashSet(int inetialCapacity);
HashSet hs4 = new HashSet(int inetialCapacity , float incrementRatio);
```

incrementRatio it can be give in the form of floating values , like 0.25 which is nothing but the HashSet need to grow by 25% of its size whenever it is filled full.

LinkedHashSet

The object of LinkedHashSet class can be created in the four ways by calling any one of the below constructor

```
LinkedHashSet hs1 = new LinkedHashSet ();
LinkedHashSet hs2 = new LinkedHashSet (Collection);
LinkedHashSet hs3 = new LinkedHashSet (int inetialCapacity);
LinkedHashSet hs4 = new LinkedHashSet (int inetialCapacity , float incrementRatio);
```

incrementRatio it can be give in the form of floating values , like 0.25 which is nothing but the HashSet need to grow by 25% of its size whenever it is filled full.

TreeSet

The object of TreeSet can be created in four ways has shown below

```
TreeSet ts1 = new TreeSet();
TreeSet ts2 = new TreeSet(Collection);
TreeSet ts3 = new TreeSet(Comparator);
TreeSet ts4 = new TreeSet(SortedSet);
```

Comparator ,means any Object of the class which implements the Comparator interface

SortedSet , means any Object of the Class which implements the SortedSet interface

Queue

Is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations.

Priority queue

- This is the data structure to hold a group of individual objects prior to processing according to some priority
- The priority can be either default natural sorting order or customized sorting order
- If we want to sort user created objects than they should be Comparable otherwise we will get a ClassCastException
- Duplicate objects are not allowed
- Insertion order is not preserved
- Null cannot be inserted

```
import java.util.PriorityQueue;
public class QueExample {
```

```
public static void main(String[] args) {
        PriorityQueue p = new PriorityQueue();

        p.add("vikas");
        p.add("anamika");
        p.add("sindrela");
        p.add("santhu");
        System.out.println("Elements of queue are ");
        System.out.println(p);

        System.out.println("Use of peek()");
        System.out.println(p.peek());

        System.out.println("Queue after using peek()");
        System.out.println(p);

        System.out.println("Use of pool()");
        System.out.println(p.poll());

        System.out.println("Queue after using pool()");
        System.out.println(p);
}
}
```

OUTPUT

```
Elements of queue are
[anamika, santhu, sindrela, vikas]
Use of peek()
anamika
Queue after using peek()
[anamika, santhu, sindrela, vikas]
Use of pool()
anamika
Queue after using pool()
[santhu, vikas, sindrela]
```

boolean offer(Object obj) to add the objects into the queue
Object element() to return head element of the Queue. If the Queue is empty than we will get RuntimeException saying NoSuchElementException
Object peek() it gives the top most element present in the queue, but it will not remove the topmost element. If the Queue is empty than this method will retun null
Object pool() it give u top most element present in the queue, but it will also remove that element.
Object remove() to remove and return the head element of the queue. If the Queue is empty than we will get RuntimeException saying NoSuchElementException
All elements in the queue will be getting arranged in their values order

Legacy class

There was no collection framework in the early version of java. There were several classes and interfaces that provided the methods for storing the objects. These classes where re constructed in 1.2, and these classes are called has legacy class.

They are

1. Dictonary
2. HashTable
3. Properties
4. Stack
5. Vector

There is one Legacy interface that is **Enumeration**.

Cursors

If we want to get the Objects one-by-one from the collection we should go for cursors

There are three types of cursors

Enumeration

Iterator

ListIterator

Enumeration

It is applicable only for the legacy class

We can create Enumeration object by using elements( ) method

It has two more methods

boolean hasMoreElements( )

Object nextElement( )

Difference between LIST and SET

List	Set
List allow duplicate elements	No duplicates allowed
List maintains insertion order	No order
All multiple null values	Allow only one null value
List is index based	Not index based

Difference between ArrayList and vector

ArrayList	Vector
ArrayList is not synchronized	Vector is synchronized
ArrayList is fast	Vector is slow
ArrayList increases by half of its size when its size is increased.	Vector doubles the size of its array when its size is increased.
ArrayList came after JDK 1.2 has a part of collecyion frame work	Vector came along with JDK 1.0, later it became a part of collection frame work
iterator and listIterator returned by ArrayList are fail-fast	Enumeration returned by Vector is not fail-fast.

Difference between ArrayList and LinkedList

ArralyList	LinkedList
Search operation is pretty fast	search operation is slow when compared to arraylist
Adding and removing the elements is slow	Adding and removing the elements is fast
Iterating the elements is fast	Iterating the elements is slow

Memory consumption is less	Memory consumption is more

Difference between HashMap and HashSet

Hash Map	Hash Set
HashMap is a implementation of Map interface	HashSet is an implementation of Set Interface
HashMap Stores data in form of key value pair	HashSet Store only objects
Put method is used to add element in map	Add method is used to add element is Set
In hash map hashcode value is calculated using key object	Here member object is used for calculating hashcode value which can be same for two objects so equal () method is used to check for equality if it returns false that means two objects are different.
HashMap is faster than hashset because unique key is used to access object	HashSet is slower than Hashmap

Difference between hashmap and hashtable

HashMap	HashTable
is non-synchronized	is synchronized
HashMap allows one null key and any number of null values	Hashtable does not allow null keys or values.
HashMap is not synchronized it perform better than Hashtable.	Less performance

Difference between Iterator and ListIterator

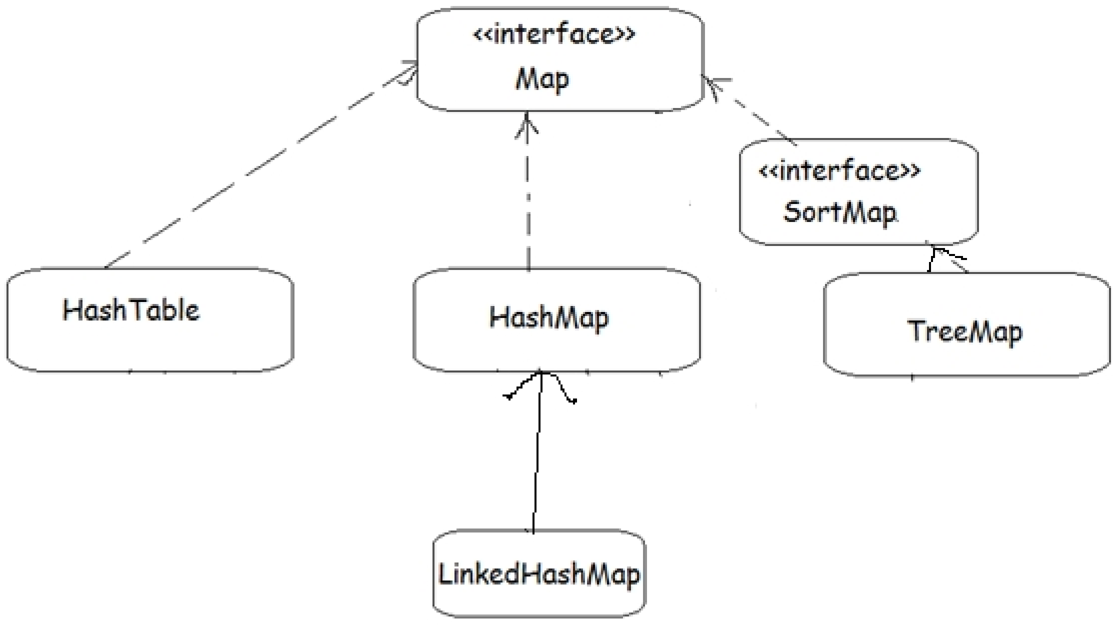
Iterator	ListIterator
Iterator to traverse Set and List and also Map type of Objects.	List Iterator can be used to traverse for List type Objects, but not for Set type of Objects.
Methods in Iterator : 1. hasNext() 2. next() 3. remove()	Methods in ListIterator 1. hasNext() 2. next() 3. previous() 4.hasPrevious() 4. remove() 5. nextIndex() 6. previousIndex()
iterator you can move only forward	List Iterator you can move back word also while reading the elements.
Not possible with iterator	list iterator you can add new element at any point of time

not possible with iterator	list iterator you can modify an element while traversing

Comparable	**Comparator**
We can use Comparable to define default natural sorting order	We can use Comparator to define customized sorting order
This interface present in java.lang package	This interface present in java.util package
Defines only one method that is compareTo( )	Defines 2 methods compare( ) equals( )
Class implements Comparable interface	Implements Comparator interface

The Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value.



- When ever we need to represent a group of Objects has pair of key and value we should go with map
- Both the key and the value should be of Object type
- Duplicate key is not allowed but the values can be duplicated
- Each key-value pair is called Entry
- There is no relation ship between Collection and Map
- **Methods**
  Object put(Object key , Object value)
  Object putAll(Map map)
  Object get(Object key)
  Object remove(Object key)
  boolean isEmpty( )
  boolean containsKey(Object key)
  boolean containsValue(Object value)
  int size( )

  Set keySet( )
  Set entrySet( )
  Collection values( )

Entry (interface)

- Key-value pair is called has Entry
- Without existing map object there is no chance of Entry hence, interface entry is defined inside Map Interface

HashMap	**HashTable**
Is not synchronized	It is synchronized
Any number of threads can operate simultaneously on the HashMap methods	Only one thread can act on the methods of HashTabel, hence it is thread safe
It accepts null for both key and value	No null is allowed for both key and values
The performance is high	Comparatively performance is low
Introduced from 1.2 version	It from 1.1 version

HashMap	**LinkedHashMap**
HashMap is parent class of LinkedHashmap	It is a child class of HashMap
Does not maintain the insertion order	Maintains the insertion order
From 1.2 version	From 1.4 version

TreeMap

- The insertion order in not maintained, all elements are inserted in some sorting order
- If we want to sort our objects than need to make it has comparable type
- No duplicate key but values can be duplicated.
- We can also sort based on the value only with Comparable

Note – refer class notes for examples

Streams and Files

Streams are used to transfer the data from one place to another, for example to recive the data from keyboard and to write into a file we need one stream. Similarly to read the data from file we need one more stream. So without streams we cant read or write the data into the file. All the related class are present in **java.io** package

To read the data from the keyboard we need to attach the keyboard to the input stream.

DataInputStream data = new DataInputStream(System.in);

System.in : represents the keyboard, it is a InputStream object
System.out : represents the monitor, it is a PrintStream object
System.err : represents the monitor, it is a PrintStream object

Input means reading the data (from file or input device)
Output means writing the data (to file)

Stream is of two types

- byte stream
- text stream

Byte stream represents the data in the form of bytes. If the class name ends with streams than it is a byte stream.

- FileInputStream
- FileOutputStream
- BufferedInputStream
- BufferedOutputStream

Text stream represents the data in the form of characters of 2 bytes each. If the class name ends with Writer or Reader than it's a text stream.

- FileReader
- FileWriter
- BufferedReader
- BufferedWriter

Examples please refer class notes

Serialization and deserialization of Objects

Serialization is a process of storing the object into the file. To write the object into the file we need to make use of ObjectOutputStream.

Deserialization is a process of reading the object from the file.

Transient: is a keyword in java which is used to declare only the variables, any variables declared with this keyword will not be serialized

NOTE : the static and transient variables cannot be serialized.

Nested Class

The class declared inside one more class is called has nested class or inner class. We use this nested class to group all the related class into one unit so it would be more readable. The inner class can accesses all the outer class members including the private members.

The inner class is has-a relationship, but it’s not a is-a relationship.

There are two types of nested class

1. Non-static nested class or inner class
    - Normal inner class
    - Local inner class
    - Anonymous inner class
2. static nested class

Normal inner class

A class that is declared inside the class is called has member inner class. First we need to create the Object of outer class and using this Object or the reference we need to create the object of inner class and then accesses the members of inner class.

The inner class cannot have the static members inside it.

The inner class can accesses all the members of outer class directly.

Outer class can accesses the inner class members only by creating the instance of the inner class

The outer class can be declared

public , default , final , abstract , strictfp

The inner class can be declared with

public , default , final , abstract , strictfp **, private , protected , static**

Local Inner class

The class that is created inside a method is known as local inner class. We cant call this class directly from outside, we need to create the object of this class only inside the method that where the class is present.

We can’t invoke local inner class from outside the method

We can accesses only the final members of the method from the inner class.

We should create the object of inner class only after the definition of the class (inside the method)

We can declare the method inner class inside Instance method or the static method.

If we declare the inner class inside the instance method than we can accesses both static and non-static members of outer class directly.

If we declare the inner class inside the static method than we can accesses only static members of outer class directly.

The method inner class can be declared only with the **final , abstract , strictfp**

Anonymous inner class

A class that has no name is known as anonymous inner class. Some times we can declare the class without the name and such type of name less inner class are called anonymous inner class.

There are three types of anonymous inner class

1) Anonymous inner class that extends Class
2) Anonymous inner class that implements interface
3) Anonymous inner class that defined inside method argument

By using the parent reference to the child object we can access only the parent class methods but we cant accesses the child specified method. Similarly we can declare the new methods inside the anonymous class but we can’t accesses them from the parent reference.

Static inner class

If we declare a inner class with the static keyword than that inner class is called has static inner class.

The static inner class can contain the static members inside it

We need not to create the object of outer class to accesses the static-members of inner class. We can accesses directly by using the class name.