- Java is a oop language with advanced & simplified features. it is free to access & can run on any platform.

JAVA :- created at SUN Miccosystem
- James Gosling led the Team
- 1st release - 1995
- 2010 - ORACLE acquired sun miccosystem
- 1st name given was OAK & after JAVA

USES

↳ mobile applications
↳ Desktop applications
↳ Web app
↳ Web services
↳ Games
↳ Data connection

- latest version SE 18

* check if JAVA is installed on your system

command prompt >> java -version
if Java is installed it will show the version

- There are 4 platforms java have
↳ java SE (Standard Edition)
↳ java EE (Enterprise Edition)
↳ java ME (Micro Edition)
↳ java Fx

• Features of JAVA •

- **Simple** : JAVA is very simple to learn, syntax is simple

- **Object oriented programming:**

  : - JAVA is object oriented unlike c++,
  - supports every OOP's concept
    - Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism
  - JAVA programs are developed using objects & classes
  - JAVA is not 100% OOPL because of primitive datatypes as they are not objects

- **Platform Independent:**

  - means JAVA program compiled on one OS/machine can be executed on any OS/machine
  - write once, Run anywhere!
  - 

  - Java source code compiled using compiler
    compiler converts source code in to byte code
    JVM can execute byte code on any platform

- **Secure** : - Java is secure language that ensures program
  - cannot gain access to memory locations without authorisation.
  - it has access modifiers to check memory access

- **Robust** : Java is a Robust language that can handle run-time errors by checking code during compilation & runtime.

- **Multi-Threaded** :
  - Java support multithreaded programming
  - multi threaded refers to creating multiple threads to handle multiple tasks simultaneously.
  - JVM uses multiple threads to execute diff block of code of same program in parallel
    - improves CPU & main memory utilization
    - Saves time & cost

- **High Performance** : Java offers high performance using JIT (Just in time)
  - compiler only compiles that method which is being called

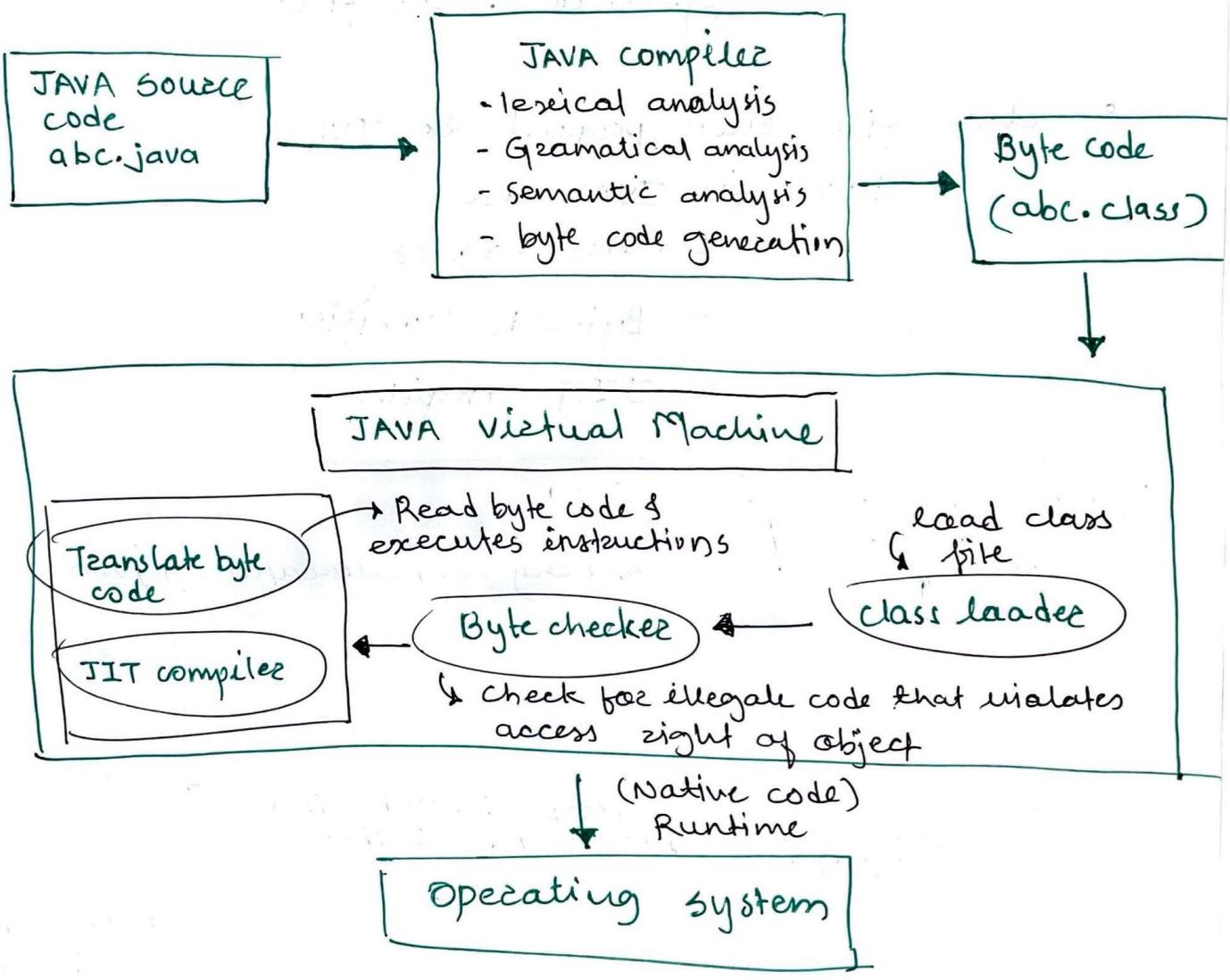
- **why java is called Platform Independent ?**

↳ As we know java is based of WORA principle (write once read anywhere) it means if we have java program which is already running on windows OS. that can run on linux or ios as well because of byte code which is independent of platform

java.class ⟶ byte code ⟶ native code
↳ platform independent

• INTERNAL WORKING OF JAVA PROGRAM •

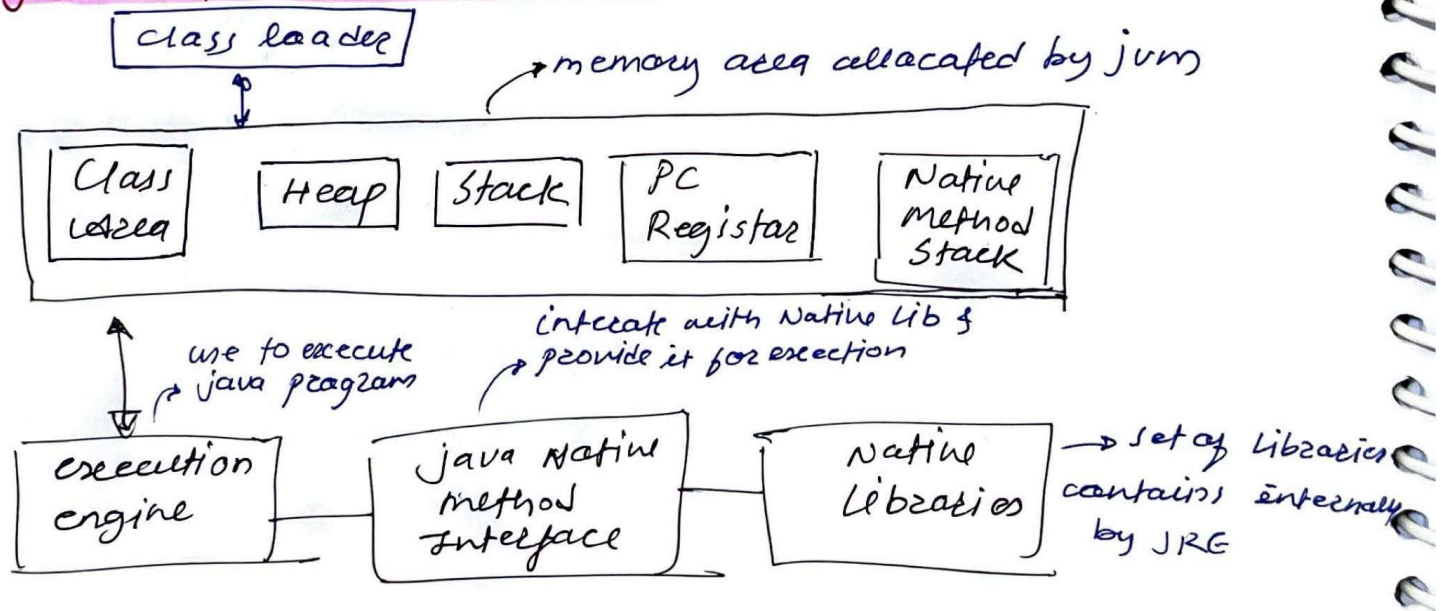
- Execution of JAVA program consist of below steps
  - creation of JAVA program
  - compilation
  - loading in memory by JVM
  - JVM verifies byte code
  - execution.



- .java file passed through compiler which converts source code into machine independent byte code
  - while converting compiler follows below steps
    - Read set of .java source file
    - scan complete source code & show the errors
    - check geomatical / syntax errors
    - Generate .class file.

- class file then passed to JVM
  Stages in JVM are
    - class loader
    - Bytecode verifies
    - JIT compiler

- Java Virtual Machine (JVM)



- <u>class loader</u> → part of JRE which dynamically loads the class into jvm.
- Class area → class body written inside { } / Stores method code

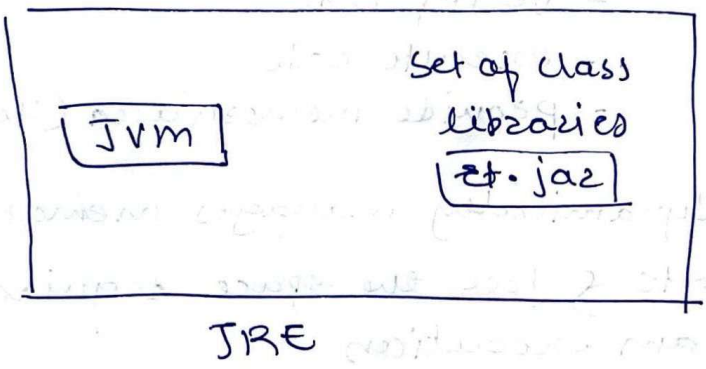
JDK , JRE , JVM

- **JVM (JAVA Virtual Machine) :**
  - essential component of java runtime environment.
  - main Role - execute java program
  - JVM enables portability of java programs allowing java program to be platform independent.
  - JVM working
    - loads byte code (.class file)
    - verify code
    - execute code
    - provide memory area (Heap & stack)
  - JVM dynamically manages memory to allocate & free the space required for program execution
  - JVM performs a Garbage Collection (GC) to free the memory occupied by objects that are no longer refered

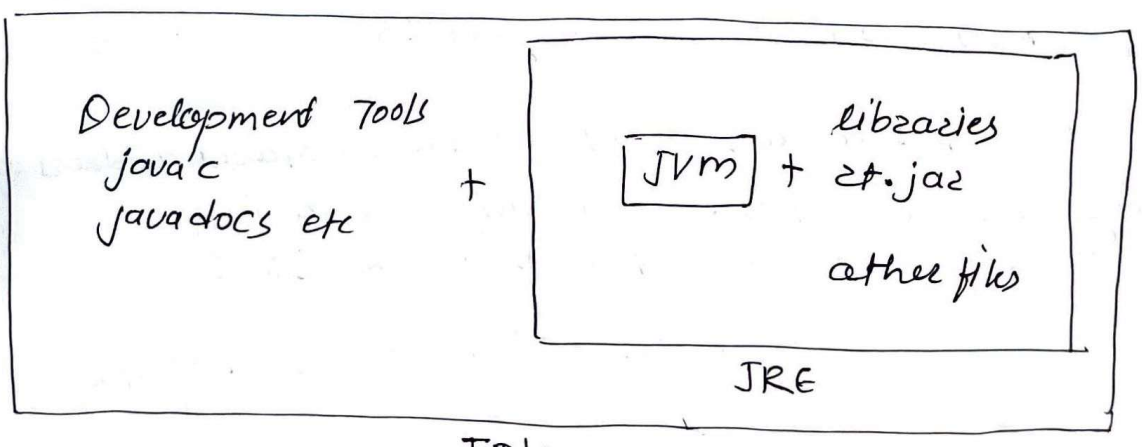
---

- **Heap** → Runtime area which stores object & global variable
- **Stack** → used to store local variable
- **PC Register** → program counter register. stores address of jvm instructions which are currently executing every thread have it own PC Register.
- **Native Method Stack** → Holds instructions of Native code

- **JRE** (Java Runtime Environment):
  - → it is a set of software tools used for developing java apps
  - → it includes JVM, class libraries, rt.jar & other components
  - → rt.jar : it includes all compiles & Bootstrap classes.



- **JDK (Java Development kit)**
  - ↳ it provides environment for both Run & develop java program
  - ↳ contains JRE + development kit
  - ↳ physically exists & need to have installed on our system



• JAVA VARIABLE'S •

- it is a container which holds value while java program is executed
- variable is assigned with datatype.
- three types of variable
  - local
  - instance (Global)
  - static

```
// Declaration
<datatype> <variable name>;
// initialisation
<datatype> <varname> =
        <literals/value>;
```

- variable is name of memory location

- **local variable :**
  - declare inside body of method
  - can be used only inside that method
  - cannot define with static keyword.
  - stored inside Stack memory
  - must initialise when declare (ie. int a = 3;)

- **Instance Variable : (Global var)**
  - for every obj there is seperate copy of Global var
  - declare inside class & outside of method
  - cannot declare as static
  - can be used throughout the class
  - store in Heap memory
  - Not mandatory to initialise

- **Static variable**
  - variable declared with static keyword

```
// public class Demo {
    int a = 30;
    String b = "Atul";        } Instance variables
    Static int m = 100; → static variable
  void method () {
    int n = 90; ------ local variable
  }
  public static void main(String[] args ) {
    int data = 50; ------ local variable
  }
}
```

Static variable

- variable becomes class variable
- it is shared among all instances of class
- must declare inside class & outside of method
- if one instance change its value it is seen by all other instances

CLASS

- class is blueprint or template from which we can create several objects
- it is not a real life entity
- does not occupy memory
- java is oop language. to create object we have to define properties of that object & object can be made by help of class

- **Types of class**:

① **Built in class**

↳ pre the class which is already predefined within package.

↳ eg. java.lang.string java.lang.object
java.lang.util java.lang.class
java.lang.scanner

② **user defined class**

↳ this classes are made by user

* **Syntax**: <<Access specifier>> class <Class Name> {

// body

}

eg. public class Demo{

// code

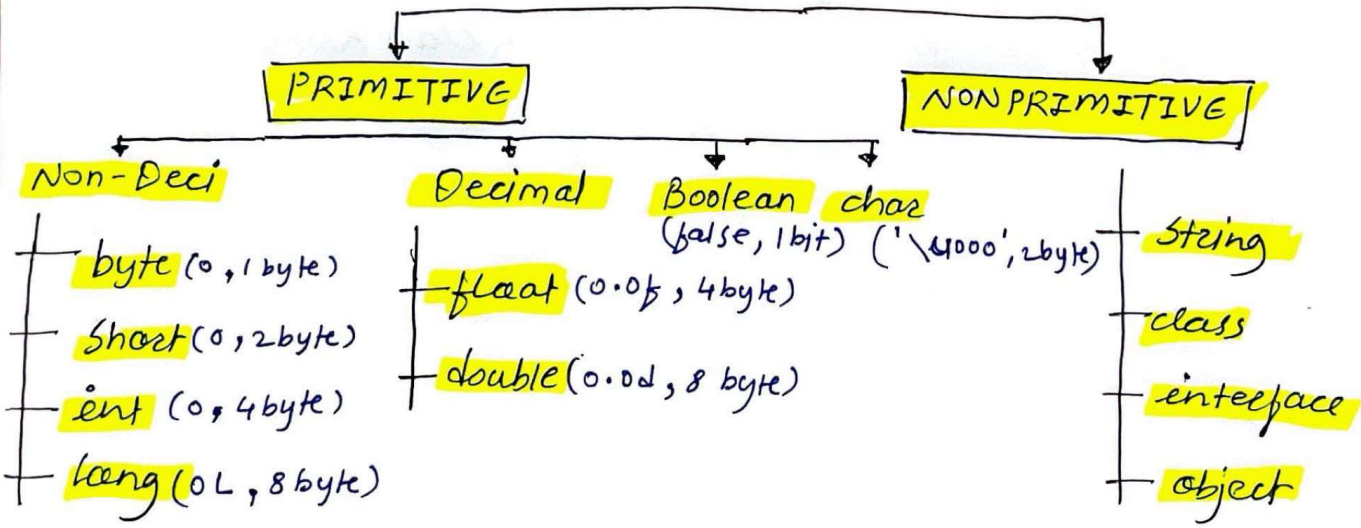
}

Rules :- - class Name always noun & start with upper case letter
- if wants to use two words then 1st letter of both should be upper case eg. (MySchool)
- we cannot use any special symbol
- always there is main method inside class which act as starting point of class/programme

- Elements consist by JAVA class.

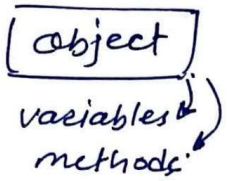
- **Data members / fields:**
  - used to define properties of class
- **Methods:**
  - used to perform specific task which define behaviours of class
- **constructor:**
  - special method with class name & without return type
- **Blocks:**
  - set of instruction within curley brackets
- **Nested class:**
  - class within class

- **Data Types** → Datatype specifies diff type & size & value that can be stored in variable



• OBJECT •

- object is a real world entity which defines its own properties, state & behaviour.
- it is a instance of class
- everything starts from/relates to object only



Syntax

```
<class Name> <object Name> = new <class Name()>;
```

```
Demo obj = new Demo(); // object of Demo class
```

- **Important Methods of object class in java :**

`toString();` → convert obj to string

`hashCode();` → generate unique hashcode for each object.

`equals(obj);` → used to compare two object dynamically

`finalize();` → this method call required to perform Garbage Collection.

`clone();` → returns new object that is exactly same as this objected

`getClass();` → returns class name of object.

• METHODS •

↳ method is a block/of code/statement used to perform specific task.

↳ for eg- we need to perform same task again & again so create a method & call it whenever we want due to method code can be reused to perform same task again & again.

↳ it only runs when called.

→ Name of method should be in camel case, noun etc.
eg. main(), getInt(), myMethod().

Syntax

```
<access specifier> <return type> <methodName> {
        // code (method logic)
    }
```

-
```
public void demo() {
        }        // No return type hence we write
                    Syscout for console
```

-
```
public Int test() {
                // return Int value
    }
```

- main method

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

public : access specifier

Static : method belongs to class & not object hence no need to create object to call method.

void : return type , main : starting point

• Types of methods •

• Static Method :

↳ When we use static keyword before method then that method becomes static

↳ No need of object creation for static method as method belongs to class

↳ We can call static method by three ways
- class name
- method name
- object creation (Not compulsory)

↳ static method can only access other static members.

```
public class Demo {

    static int a = 40; // static variable
    int b = 50; // instance variable

    void display() { // method
        System.out.println(a);
        System.out.println(b); }

    static void staticDisplay() { // static method
        System.out.println(a); }

    public static void main(String[] args) { // main method
        Demo obj = new Demo(); // object creation
        obj.display(); // calling of method
        staticDisplay(); } // calling of static method without object.
}
```

op / 40
50
40

- **Non-static method**:
  - when we dont use static keyword the method becomes non-static
  - it is mandatory to create object to call a method (non static)

• Operators in JAVA •

① Arithmetic operators:

↳ (+), (-), (*), (/), (%)

② Logical operators:

AND	OR
T & T = T	T OR T = T
T & F = F	T OR F = T
F & T = F	F OR T = T
F & F = F	F OR F = F

③ Bitwise operator:

Bitwise AND → [&] → if both 1 then 1 otherwise 0

Bit wise OR → [|] → if both 0 then 0 otherwise 1

Bitwise exclusive OR (XRO) → [^] → return 0 if both are same

Bitwise compliment → [~] → returns inverse ie 0 → 1 & 1 → 0

Shift left → [<<]

Shift right → [>>]

④ Relational operators:

↳ >, <, >=, <=, ==, !=

⑤ Assignment operator → =

↳ used to assign value ie a=1

⑥ **Unary operator**

↳ performs increament or decreament

```
public Class Demo {
    P.S.V.main (String [] args) {
        int a = 10;
Sout (a++);  // 10
Sout (a--);  // 11
Sout (++a);  // 11
Sout (--a);  // 10
```

a++ ↳ even increament
↓
as it is

++a
↓ ↳ as it is
first increament

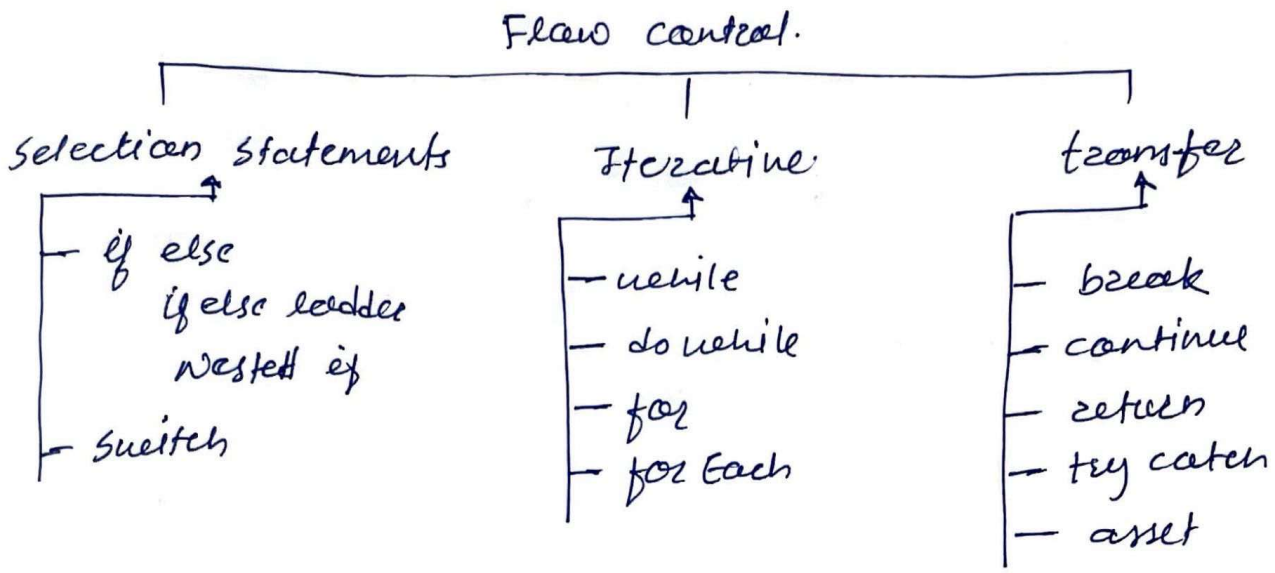
⑦ **Dot operator:**

↳ use to call method, variables etc

eg. obj.a() ... calling a method.

• FLOW CONTROL STATEMENT •

• it is used to control flow of program



• **If else** :

→ boolean form.

```
if (condition)
    { // code to be executed if cond" is true
    } else {
      if condition is false
    }
```

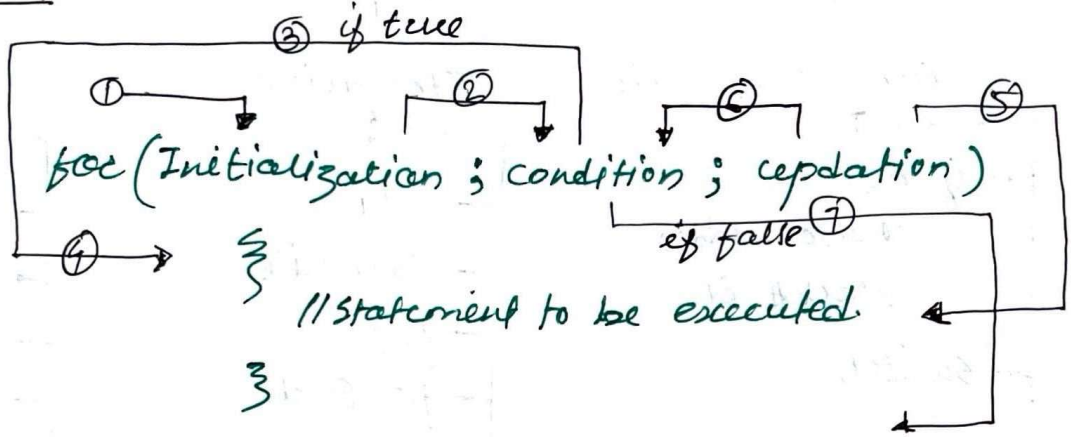
```
int time = 20;
if (time < 18) {
    System.out.println("Good day");
  } else {
    System.out.println("Good evening");
  }
```

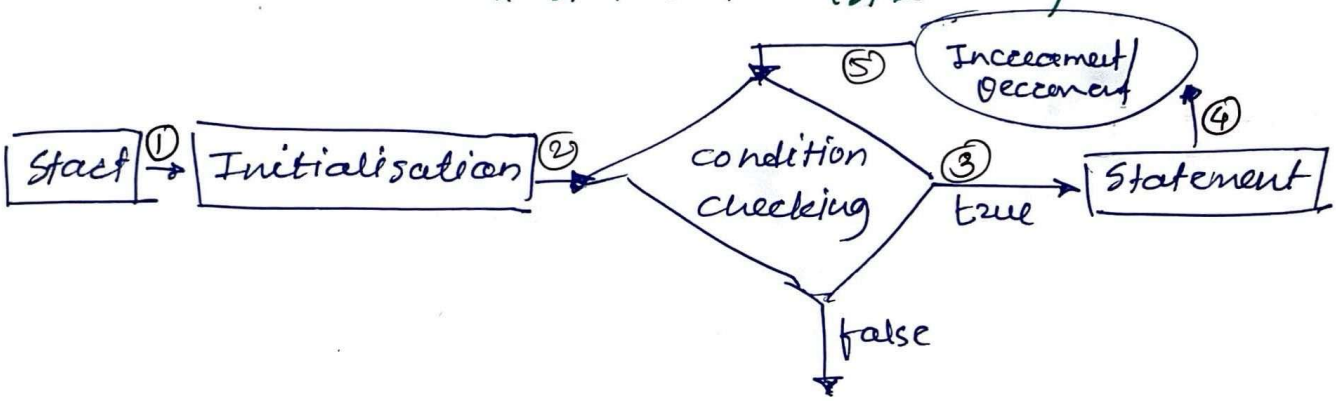
op // → Good evening

cond". time < 18
↳ returns false hence
else block executed.

- **For loop** :

Syntax





eg. print numbers from 0 to 5

```
public class Demo {
    int i;
    P.S.V.m(S.A) {
    for(int i=0; i<=5; i++) {
        System.out.println(i);
    }
    }
}
```

op →
1
2
3
4
5

while loop

↳ when we dont know number of iterations we use while loop

```
while (condition) {
        // code - increment / Decrement.
    }
```

Do while :

```
do {
  // code
  }
while (condition)
```

Switch case

```
switch (var) {
  case 1 ;
    sout( ) ;

  cas 2 ;
    sout( ) ;
```

break

↳ used when out cond^n becomes true. then it stops flow of program

Continue

↳ when cond^n become true it skips that iteration

static

↳ static keyword is used to indicate that particular members (var, methods etc) belongs to class & not the object

↳ static keyword contributes to memory efficiency because it allows variable & methods to be shared among all instances.

eg.

```
int x = 10;
```

obj 1 obj 2 obj 3

int x = 10 int x = 10 int x = 10

(not static variable having seperate copy for each object)

```
static int x = 20;
```

obj 1 obj 2 obj 3

int x = 10

(single copy to all objects)

↳ if we make change in static variable with help of any one object it will effect for all objects

static block :

↳ when we want some statement to be executed before main method then we write it into static block.

• FINAL •

- final keyword is used to restrict the user
- it stops changing the value of variable
- stop method overriding
- stop inheritance for final class & we cannot extend final class.

```
class Bike {
    final int speedlimit = 90; // final var
    void run() {                // method
        speedlimit = 100;       // changing value
    }
    public static void main(String[] args) {
        Bike obj = new Bike(); // obj creation
        obj.run();             // calling method
    }
}
```

op → compile time error.

- cannot change value of final var.

• Packages

↳ collection of all classes is stored in package
↳ java.lang is default package in java.

package com.<company name>.<client name>.
<project name>.<module name>.
<submodule name>

eg. com.infy.gs.pb.gluon.dataloader.

• Access Modifier •

- **public** : scope inside [class, different class & outside package]
- **protected** : inside [same package only] & outside package in case of inheritance
- **private** : inside [same class] only
- **default** : inside [same package] only.

• CONSTRUCTOR •

↳ when object is created, seperate instance variable will be created for each object.

↳ once object is created we need to initialize it compulsory

↳ new keyword is used to create object

* ↳ constructors are specially designed to initialize the object or to perform initialization of object
ie. initialization of instance variable.

→ it is a special method [without any return type] & same name as class

→ when we create object, automatically constructor will be executed to perform initialization of object

→ for every object, constructor will be executed each time. (100 object → 100 time constr. exected)

→ all access modifier can be applied to constructor except final

→ constructor cannot ✗ be abstract, static, final & synchronised.

→ if return type given to constructor then compiler will consider it as a method hence No Return type.

- **Default constructor**

↳ if constructor is NOT created then compiler will create default constructor for every JAVA class

↳ access modifier same as access modifier of class

```
<className> () {
    super();
}
```

- default constructor always comes with super();

- **No-arg constructor:**

↳ constructor without any argument is called as no-arg constructor

```
<className> () {
    // Statement
}
```

- **parameterised constructor**

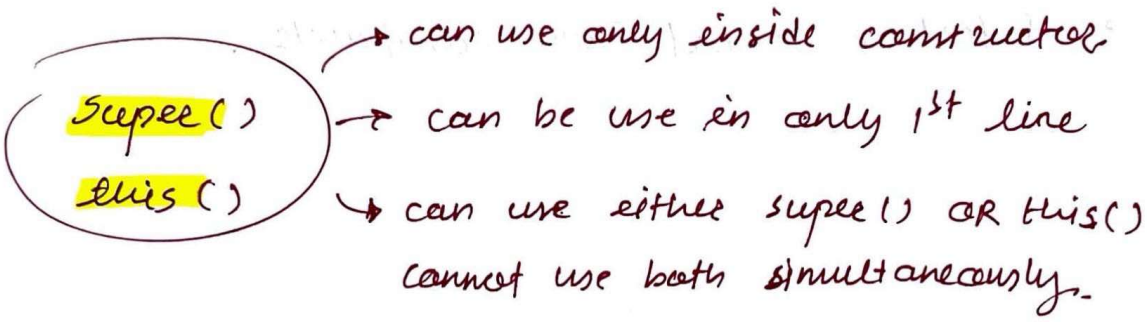
↳ constructor with arguments called as parameterised

```
<className> (arg1, arg2 ...) {

}
```

- all default constructors are no-arg. but no-arg is not default

• super OR this •

- first line in constructor should be always either super(); or this();
  - → if not written, compiler will autogenerate super();
  - → we can use super() or this() only inside constructor if used in method it will give compilation error.




```
class Test {
    String s = "Atul";
}

class Test extends Demo {
    String s = "kangane";
    public static void main(String[] args) {
        Sout(s);          → op : kangane
        Sout(this.s);     → op : kangane
        Sout(super.s);    → op : Atul
    }
```

- super(), this() are related to object & static method is related to class hence inside static method we cannot use super OR this

- constructor can be overloaded
  cannot override
  cannot Inherited

- Interface cannot have constructor.

• STRING, STRING BUFFER & STRING BUILDER •

- String objects → Immutable (Non changeble)
- String buffer → Mutable (changeble)
- String builder →

- ↳ once string object is created we cannot make any change, if we try to change it will create new object

```
Class abc {
public static void main(String[] args){
String s = new String("Atul"); -①
s.concate("Kangane"); -②
s = s.concate("Engineer"); -③
System.out.println(s)
```

↳ op // Atul

```
StringBuffer Sb = new
        StringBuffer("Atul");
Sb.apend("Kangane")
System.out.println(Sb);
```

↳ op // AtulKangane.

↳ String buffer is mutable
↳ can change value.
/ modify the value

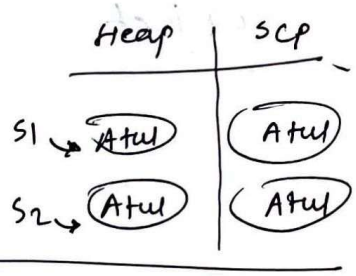
Heap	Scp
s → Atul ...①	Atul ..①
AtulEngineer ...③	
..②	..②
AtulKangane	Kangane
	Engineer ..③

↳ new object is created at step ② with AtulKangane but as there is no reference variable so it it will be eligible for garbage collection (gc)

CASE

```
class Demo {
  public static void main(String[] args);
  String s1 = new String("Atul");
  String s2 = new String("Atul");

  System.out.println(s1 == s2); op // false
  System.out.println(s1.equals(s2)); op // True
```

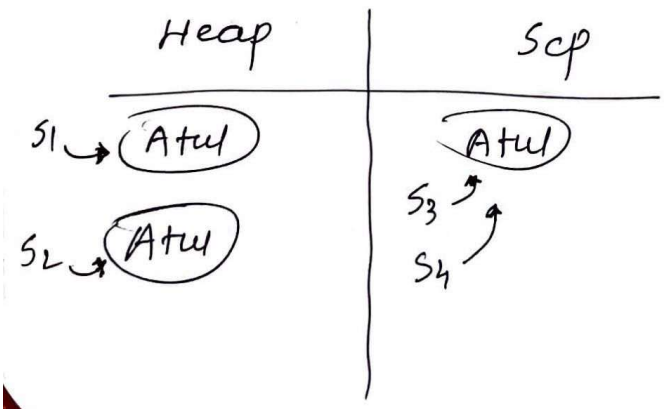
s1 & s2 are refferring to 2 diff objects



- object class have .equals() method which is same as == but in child class String it gets overriden & compare the content of object.
  - .equals() method in String class compare the content of object & in Stringbuffer equals method is NOT overriden & compare the reference

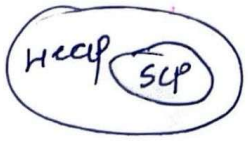
---

```
String s1 = new String("Atul");
String s2 = new String("Atul");
String s3 = "Atul";
String s4 = "Atul";
```

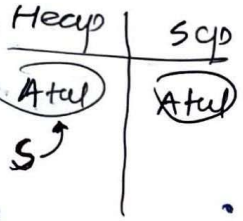


- for every string literal one copy will be created in scp
- in scp area NO two objects will be created with same content

• Heap & String Constant Pool (SCP) •



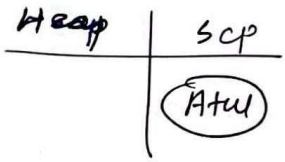
↳ String s = new String ("Atul");



↳ new keyword will create two objects one in Heap area & one in SCP for future referance or reusesability

- SCP was part of method area till 1.6 & from 1.7 it is part of Heap area only
- SCP is only applicable for String & NOT for String buffer
- String is most commonly used object hence memory management (SCP) is only provided for Strig

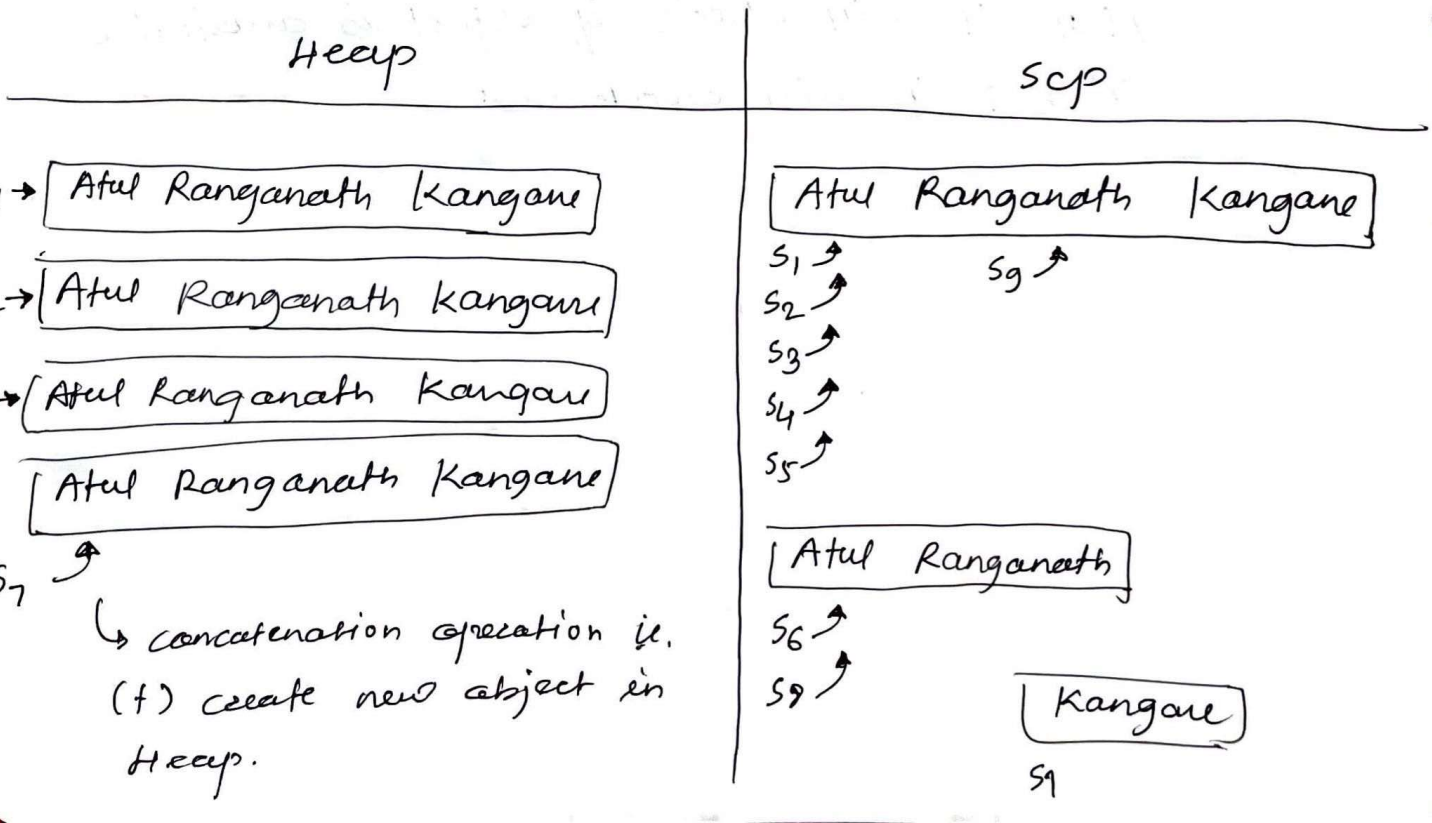
- String s = "Atul" →



↳ in this case only one object will be created in SCP.
first it will check if object is available if NOT it will create new.

case

```
String s1 = new String("Atul Ranganath kangane");
String s2 = new String("Atul Ranganath kangane");
    Sout(s1 == s2); // → False
String s3 = "Atul Ranganath kangane";
    Sout(s1 == s3); → False
String s4 = "Atul Ranganath kangane";
    Sout(s3 == s4); → True
String s5 = "Atul Ranganath" + "kangane";
    Sout(s4 == s5); → True
String s6 = "Atul Ranganath";
String s7 = s6 + "kangane";
    Sout(s4 == s7); → False
final String s8 = "Atul Ranganath";
String s9 = s8 + "kangane";
    Sout(s4 == s9) → True
```

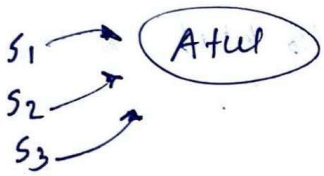


Q. why string object is immutable & stringbuffer is NOT ?

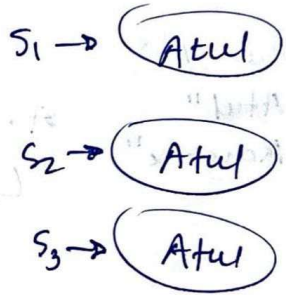
→ String is most commonly used object hence string have special memory management system like SCP
if we change object value it will affect all value refered to that hence string are made immutable

String buffer do not have any concept like reuseability hence there is no need to make stringbuffer immutable.

String



Stringbuffer



- All wrapper class objects are immutable.

• Imp methods of String class.

length()

// Atul.length

↳ 4

isempty()

↳ returns boolean value according to string is empty or not

charAt(int i)

↳ return character at i^{th} index

substring(int i)

Atul.substring(2)

↳ ul

replace()

Atul.replace('A', 'B')

↳ Btul

concate(String str2)

String A = "Atul"

Strig B = "Kargar"

A.concate(B)

↳ AtulKargar

indexof(String s)

Atul.indexof(t)

↳ 1

equals(object obj)

Atul.equals(atul) → false

equalsIgnorecase(object obj)

Atul.equalsIgnorecase("atul")

↳ true

String Builder

↳ every method present in string buffer is synchronized hence at a time only one thread is allowed to operate on string buffer object. to overcome this in Java 1.5 string builder is introduced.

STRING BUFFER	STRING BUILDER
• most methods are synchronized	• no method is synchronized
• at a time only one thread is allowed to operate on object	• multiple threads can operate at a time.
• low performance due to single thread	• high performance
• Introduced in 1.0 version	— in 1.5 version

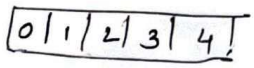
String is always threadsafe.

• Arrays

↳ list of items of same type, Syntax: type[] name = new type[size]

eg. Int[] numbers = new Int[5]; →

↳ this will create array of size 5 & type Int

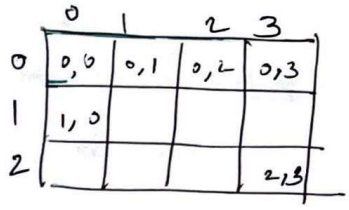


2D Array

Int[][] arrayName = new type[rows][columns];

eg Int[] marks = new Int[3][4];

↳



• ENCAPSULATION •

↳ Encapsulation in java refers to integrating / binding data (variables & methods) in a single unit

↳ In encapsulation class variables are hidden from other classes (private) & can only be accessed by method of same class

↳ It is important to declare data members of class as private it prevents access to data member & method by any external class & improves security.

↳ Encapsulation can be achieved by declaring all variables in class as private & writing public methods in the class to set & get values of variable.
it is more defined with setter & getter method.

```
public class Student {
    private String name ; → private variable

    public String getname() {
        return name ; }
    public String setname() {
    public void Setname (String name) {
        this.name = name ;
    }
}
class Test {
public static void main (String [] args) {
   Student s = new Studen() ; → object of Student class
     s.Setname = "Atul" ; → calling setname method to set value of name
System.out.println(s.getname) → calling getter to get result.
```

Public getter & Setter methods to set & get value

↳ Atul

• INHERITANCE •

↳ It is a mechanism in java by which one class is allowed to inherit features / properties of another class.

↳ creating new class based on existing class

↳ code Reuseability ⇒ child class can use code written in parent class

↳ Method overriding is achievable only through inheritance

• extends keyword is used for inheritance

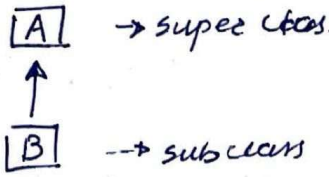
```
class Employee {        ⟶ super / parent class
    int salary = 60000;
}

class Engineer extends Employee {  ⟶ Inherited / sub-class
    int benefits = 10000;              or child class
}

class main {                       ⟶ main class
  public static void main (String[] args) {
    Engineer E1 = new Engineer();
    Sout (E1.salary + " " + E1.benefits);
  }
}
```

op ↳ 60000 10000

• Types of Inheritance

① Single Inheritance

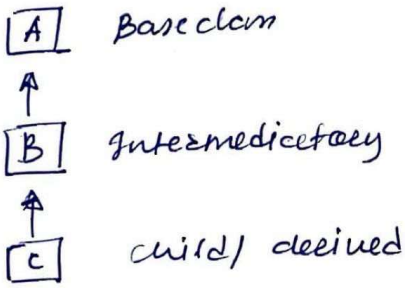
↳ sub-class inherits features of one super class



② multilevel Inheritance :

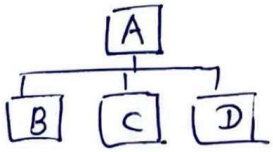
↳ derived class will be inheriting base class & as well as derived class also act as base class for other class

• In java class cannot directly access grandparents members



③ Hierarchical Inheritance

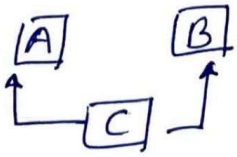
↳ In this one class serves as a super class for multiple sub-classes



④ Multiple Inheritance

↳ a class can have more than one superclass & inherit features from all parent class

↳ java does NOT support multiple inheritance. & can be only achieved through Interface



⑤ **hybrid Inheritance**

↳ mixe of two or more type of above Inheritance.

↳ It can be only achieved through Inheritance

• **upcasting in java** :

↳ reference variable of parent class refers to object of child class it is known as upcasting

↳ it gives us flexibility to access parent class members & the overridden methods in child class

```
class A {        ... parent class
    String a;
void x(){}
    }
class B extends A {  ... child class
@override void x(){
    A a = new B();   .... upcasting
    }
}
```

↳ only access to parent class members

↳ only access to overriden method of child class

• POLYMORPHISM •

↳ polymorphism means many forms, & it occurs when we have multiple classes related to each other by inheritance.

↳ we can perform single action in different ways

↳ Types → ① compile time
② Runtime.

↳ polymorphism in java can be achieved by method overloading & method overriding.

↳ compile time polymorphism is achieved by method overloadig

↳ Runtime polymorphism is a process in which call to overriden method is resolved at runtime.

```
class Bike {                    --- --- parent class
    void run() {
     sout("running") }
    }
class splendor extends Bike {  --- child class
@override
    void run() {
      sout("Bike running")
    }
p.s.v.m(String[] args) {
    Bike b = new Splendor();  ----- upcasting
      b.run();
    }
```

↳ op // Bike running

• Method Overloading :

↳ a class having multiple methods with same name but different parameters is known as method overloading

↳ 2 ways — ① changing number of arguments / parameters
② changing data type.

① changing No. of arguments :

```
class Addee {
static int Add(int a, int b){
    return a+b ;}
static int Add(int a, int b, int c){
    return a+b+c ;}
class Test {
public static void main(String[] args){
    Sysout(Addee.Add(1,2));
    Sysout(Addee.Add(1,2,3));
}
```

} methods with same name but diff parameters

↳ op → 3
6

• changing data type

```
class Addee {
static int add(int a, int b){
    return a+b;}
static double add(int a, int b){
    return a+b;}
}
class Test {
    p.s.v.m(String[] args){
Sout(Addee.add(1,2));
Sout(Addee.add(1.2, 3.8));
```

} method with same name but diff data type

↳ op 3
5

• Method overriding •

↳ If sub-class (child class) has a same method as declared in parent class is known as method overriding

↳ it is used for runtime polymorphism

Rules

↳ method must have same name as in parent class
↳ method must have same parameter as in parent class
↳ there must be an IS-A relation (Inheritance)

```
• class Vehicle {        --- Parent class
    void run() {    ....
      sout("Running");
    }
  }

  class Bike extends Vehicle { -- child class
@Override
      void run() {        ---- overridden method.
    sout("Bike is Running");}
    }
    p.s.v.m(String[]args){
      Bike obj = new Bike();
      obj.run()      --- calling method of object of child class.
    }
```

↳ op // Bike is running

- static method cannot be overriden
-

• ABSTRACTION •

↳ it is a process in java in which we only show essential details / functionality to user. the non essential details are not displayed

→ Abstraction is achieved by interfaces & abstract class. we can achieve 100% abstraction using interface
  ↳ Abstract class → 0 - 100%
  ↳ interface → 100 %

• Abstract class & methods

↳ a class which is declared as abstract is known as abstract class

→ it can have abstract as well as non abstract methods
→ cannot be instantiated
→ can have constructor & static methods as well
→ can have final method

• a method which is declared as abstract & does not have implementation is known as abstract method
  ↳ the implementation is provided by subclass

```
abstract class Bike {          ---- abstract class
    abstract void run();       -- abstract method with no implementation
}

class Honda extends Bike {     -- child class of Bike.
    void run() {
    System.out.println("Bike running");   -- implementation of method
    }
p.s.v.m(String[] args){
    Bike obj = new Honda();    ... object
    obj.run()                  -- calling method.
}
```

→ op // Bike running

* object of abstract class cannot be created but we can create reference var which refers to object of child class.

• Interface

↳ it is a collection of abstract methods
↳ used to achieve abstraction
↳ cannot instantiate an interface
↳ does not contain any constructor
↳ all methods are abstract
↳ all methods declared with empty body & all fields are public, static & final by default

```
interface printable {
    void print();
}
class Abc implements printable {
    public void print() {
    S.out("Hellow"); }
public S.V.m.S[]args() {
    Abc obj = new Abc();
    obj.print();
    }
}
```

↳ op :: Hellow

- Multiple inheritance is not supported by java class but it is possible by an interface.

↳ multiple inheritance is not possible in class because of ambiguity.
supported in interface because there is no implementation is provided by implementation class.

```
interface Printable {
    void print();
}
```
← no implementation (abstract method)

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

```
class Test implements Printable, Showable {
    public void print() {
        System.out("printing");
    }
    P.S.V.m(String[] args) {
        Test obj = new Test();
        obj.print();
    }
}
```
← multiple inheritance

← implementation of abstract method in inherited child class

← obj of child class

↳ op-11 printing

* As we can see there is no implementation of same methods in interfaces & only once it is implemented in child class hence multiple inheritance is supported as it is not creating ambiguity error.

EXCEPTION HANDLING

Exception :

↳ unwanted or unexpected event which occurs during execution of program which disrupts normal flow of program

→ eg. ↳ while interview the zoom call gets disconnected & normal conversation disrupts. is called exception

Exception handling

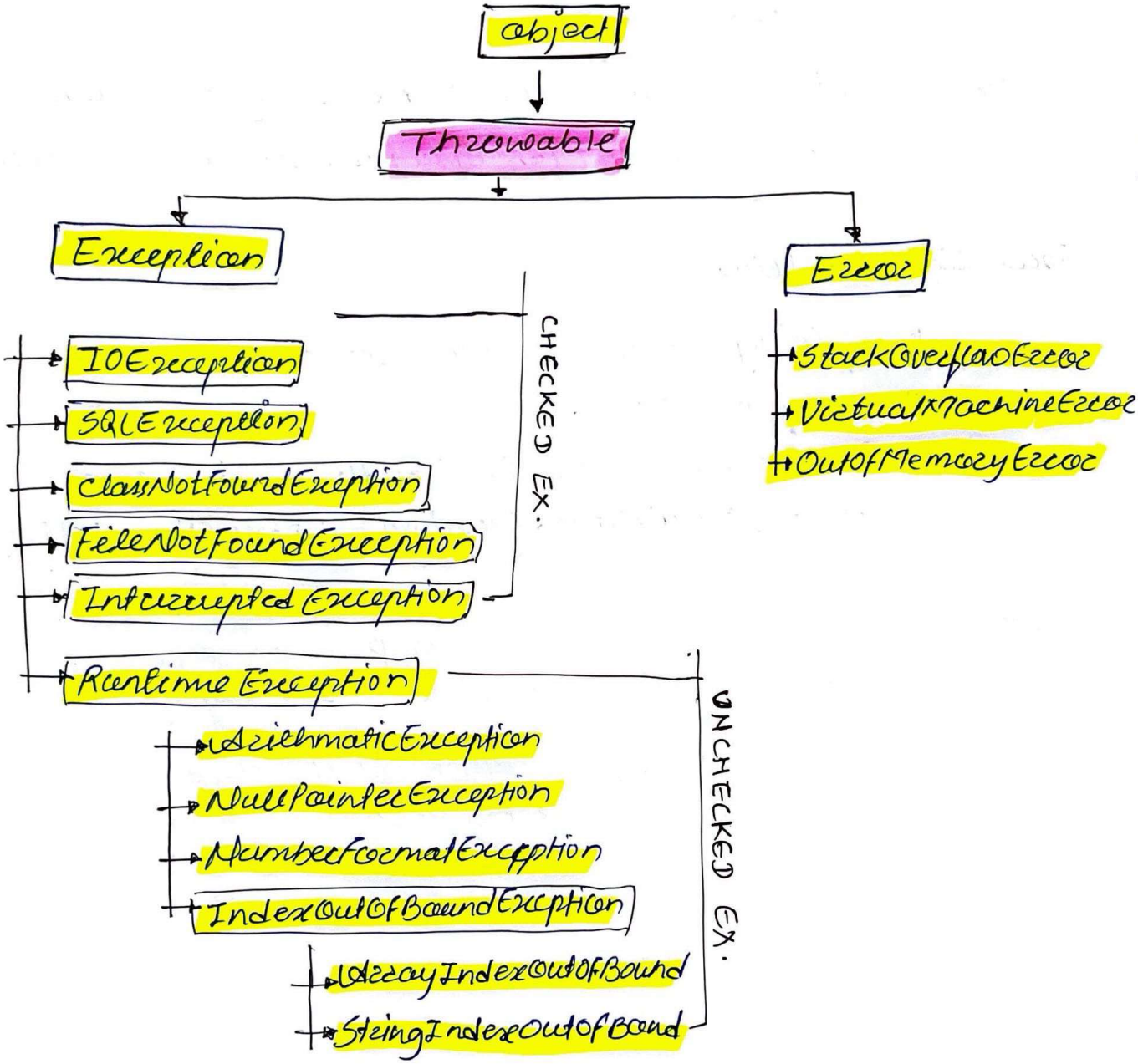
↳ defining alternative way to continue normal flow of program.

↳ it is a mechanism to handle runtime error such as ClassNotFoundException, IOException, SQLException

Error : An error indicates a serious problem that a reasonable application should NOT try to catch

Exception : it indicate a condition that reasonable. application might try to catch

- All exception & error types are subclasses of class Throwable



- Exception can be categorized in two ways
  1. Built-in Exception
     - checked
     - unchecked
  2. User-defined Exception

1. **Built-In Exception**

↳ this are the exceptions available in java libraries & suitable to explain certain error situation.

- **Checked Exception**

↳ checked exception are called compile-time exception because these exception are checked at compile time by compiler.

- **Unchecked Exception:**

↳ the compiler will NOT check this exceptions at compile time &

- only runtime error are unchecked remaining are checked.

- **Partially checked**

↳ when parent is checked & child is NOT checked then its called as partially checked Exception

↳ only throwable & exception are partially checked

2. **User Defined Exception**

↳ sometimes built in exceptions in java is not able to describe certain situation such as cases can be handled by the exception created by user which are called user defined exception.

* Methods to print Exception Information

① printStackTrace()

↳ this method prints exception information in format of Name of Exception : Description of Exception, StackTrace

eg. java.lang.ArithmaticException : / by zero
at GFG.main(File.java:10)

(Name of exception — Description — StackTrace)

② toString()

↳ prints exception info in format of Name of Exc : Descriptn

eg. java.lang.ArithmaticException : / by zero.

③ getMessage()

↳ prints only Description of exception

eg: / by zero.

```
Class Test {
    public static void main(String[] args) {
        int a = 5;
        int b = 0;
    try {
        System.out.println(a/b);
        }
    catch (ArithmaticException e) {
        e.printStackTrace();
        e.toString();
        e.getMessage();
    }
```

op: java.lang.ArithmaticException : / by zero
at GFG.main(file.java:10)

→ result at ② above

→ result at ③ above.

How JVM handles exception?

- **Default exception handling:**

  ↳ whenever inside method, if an exception has occured, the method creates an object known as an Exception object & hand it off to jvm

  ↳ exception object contains name & Description of object & current state of program where exception has occured.

  ↳ creating an exception object & handling it in runtime system is called as throwing an exception there might be list of methods that had been called to get to the method where exception occured this ordered list of method called as CALL STACK

→ runtime system searches call stack to find method that contains a block of code that can handle the occured exception. this block of code is called exception handler.

→ run time system starts searching from the method in which exception has occured & proceeds through call stack in reverse order in which method were called

→ if it finds appropriate handler it passes exception to it if couldn't finds appropriate handler then hand it over to DEFAULT exception handler which prints the exception & terminate program abnormally.

- java exception handling is managed via below five keywords.

try : → it is used to specify a block where we should place exception code i.e. where exception can occur or risky code.

catch : this block is used to handle an exception
↳ must be preceded by try block means we can't use catch block alone

finally : it is used to execute necessary code. it executes whether an exception occurs or NOT

throw : it is used to throw the exception

throws - used to declare an exception

- **Try catch block** :

↳ try block is used to enclosed the code which might throw exception

↳ catch block used to handle exception by declaring type of exception. declared exception must be parent class (eg. Exception) or generated exception type

↳ good approach

```
try {
    // Risky code
    }

catch (Exception e) {
    // Handling code
    }
```

```
class Test {
p.s.v.m (String [] args) {
try {
    Sout ("1");
    Sout ("10/0");
    Sout ("2"); }
catch (Exception e) {
    Sout ("Exp handled");
}
Sout ("4") } // Rest of code
```

Risky code

↳ handling exception

↳ catch block executes only if exception occurred in try

↳ if try & catch both contains exception the abnormal termination happens

↳ if exception is not part of try then abnormal termination

↳ length of try block should be as small as possible.

• try with multiple catch blocks

- Suppose there are multiple exceptions present in try block like, AE, Nullpointer etc & if we have only one catch block then for every exception we will get same result present in catch

↳ if try with multiple catch are there then jvm will start from first catch block to last

↳ catch block sequence from top to bottom i.e from child to parent

↳ we cant use two same exception catch blocks

```
try {
    AE
    NPE   } //multiple exceptions
    IOE
}
catch (AE e) {
}
catch (NPE e) {
}
catch (IOE e) {
}
```

multiple catch blocks.

Nested try block

↳ using try block inside other try block

↳ Sometimes a situation may arise where a part of a block may cause one error & entire block itself may cause another error in such cases exception handlers have to be nested.

```
try {
  // risky code
     try {
       // risky code } inner try block
       }
     catch (Exception e) { —— exception handling for
       // handling } inner try inner exception
       }
catch (Exception e) { —— for outer exception try
    // handling outer try
   }
```

↳ if catch block is NOT present for particular exception of inner try block then catch block of outer (parent) try block is checked for that exception.

finally block

- finally block is a block used to execute important code
- finally block is always executed whether exception is handled or not.
- finally block have highest priority than return statement

```
try {
  // risky code
}
catch (Exception e) {
  // exp handler
}
finally {
  // important code
}
```

throw

↳ it is used to throw exception explicitly

↳ sometimes we required to create own customised exception & handover to jvm then we use throw keyword

```
class Test extends Exception {
    p.s.v.m.(String[] args) {
    throw new Exception();
    }
```

throws

↳ used in signature of method to indicate that this method might throw one of listed type of exception

```
method_name(parameters) throws exceptionList {
    }
```

↳ we can use throws keyword to delegate the responsibility of exception handling to caller (method or JVM)

↳ required only for checked exception

* COLLECTION FRAMEWORK *

- **Need of collections:**

↳ Array is an indexed collection of fixed number of homogeneous data element

↳ main advantage of java is we can represent multiple values with single variable. so reuseability of code will be improved

- limitation on arrays
  - ↳ Arrays are fixed size. once we create array with some size there is no chance to increase or decrease. size
  - ↳ Arrays can hold only homogeneous data elements
  - ↳ we can solve this problem by using object arrays
  - ↳ Array concept is NOT based on any standard data structure hence ready made method support is NOT available. & for every requirement we need to write a code

Arrays	Collection
• fixed in size	• Growable in nature
• holds only homogeneous data structure	• holds Homogeneous & Heterogeneous data structure
• Not based on any Standard Data structure	• based on standard data structure.
• ready made method support is NOT available	• ready made method support is available.

- If we are sure about the size of Array it is recommended to go for Array inted collection due to performace of array is good than collection

Arrays	Collections
• w.r.t. memory arrays are NOT recommended to use	• collection is recommended w.r.t memory
• w.r.t. performance Arrays are recommended	• X
• Array can hold both primitives & object types	• can hold only objects but NOT primitives

---

- **what is collection ?**

↳ If we want to represent a group of individual objects as a single entity then we should go for collections

- **what is collection framework ?**

↳ It defines several classes & Interfaces which can be used a group of objects as single entity
↳ to represent

- usually we can use collections to hold & transfer objects from one place to another place, to provide support for this requirement every collection already implements Serializable & Clonable interfaces.

- **9 key interfaces of collection framework.**

① Collection ② list ③ Set ④ Queue ⑤ SortedSet
⑥ Navigable set ⑦ map ⑧ sorted map ⑨ Navigable map.

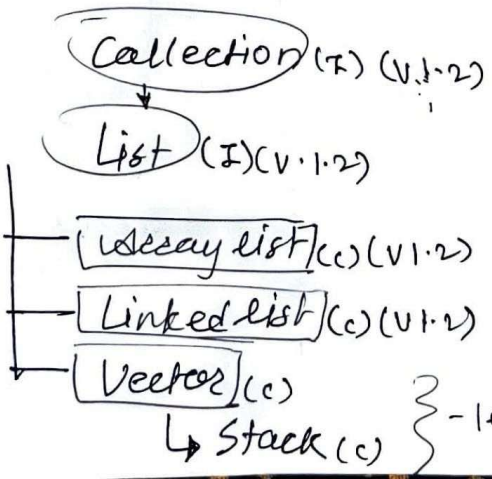
- **collection (I)**

↳ to represent a groups of individual objects as single entity
↳ it defines most common methods which are applicable for any collection object
↳ in general it is considered as root interface of collection framework
↳ there is No concrete class which implements collection interface directly.

* Collection is a interface and Collections is a class

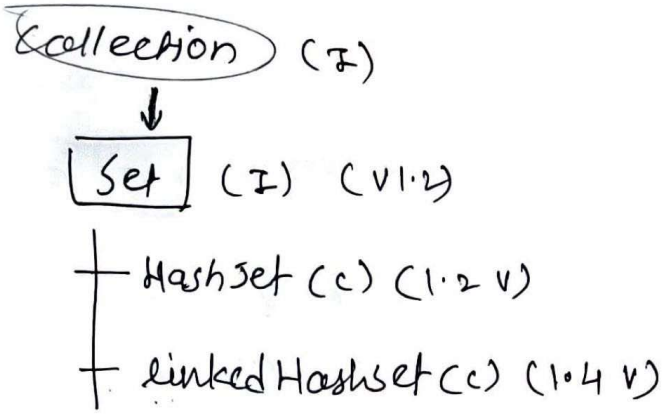
- **List (I):**

↳ child interface of collection
↳ Duplicate allowed
↳ insertion order preserved



③ • Set (I) :

↳ child interface of collection
↳ Duplicate NOT allowed
↳ Insertion object NOT preserved

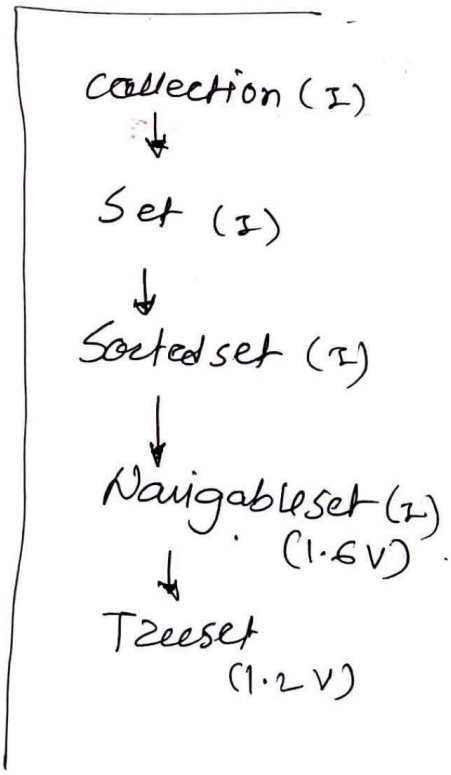


④ • SortedSet (I)

↳ child interface of set
↳ duplicate NOT allowed
↳ inserted according to some sorted order

⑤ Navigable Set (I)

↳ child interface of Sorted set
↳ Defines several methods for Navigation purpose

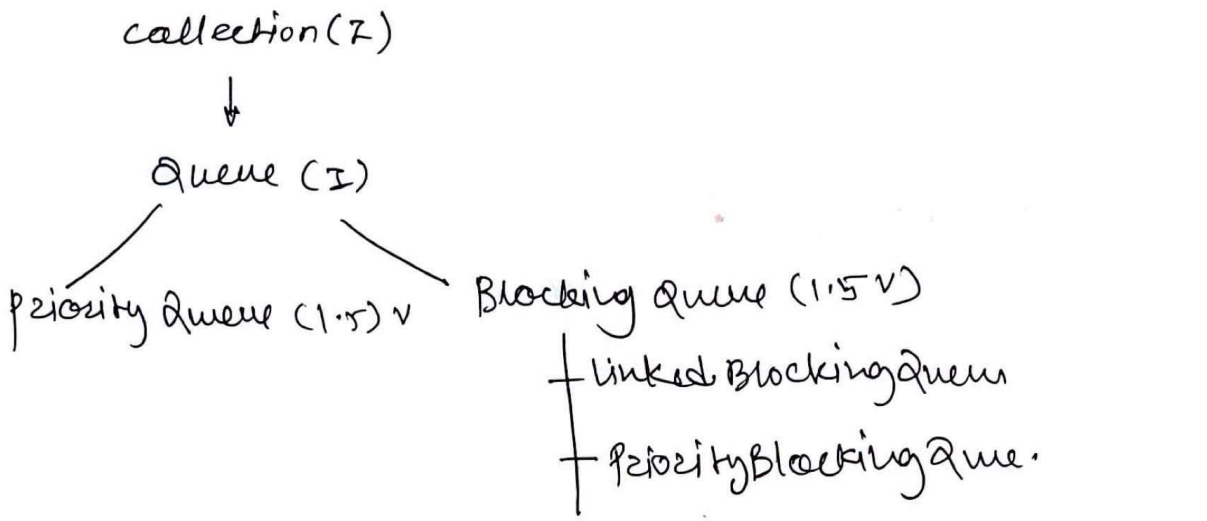


⑥ Queue (1.5 V)

↳ child interface of the collection

↳ represent group of individual object prior to processing

eg. before sending mail we have to store all mail ID's & mail should be deliver first in first out



✱ If we want to represent a group of objects as a key value pair then we should go for [Map interface]

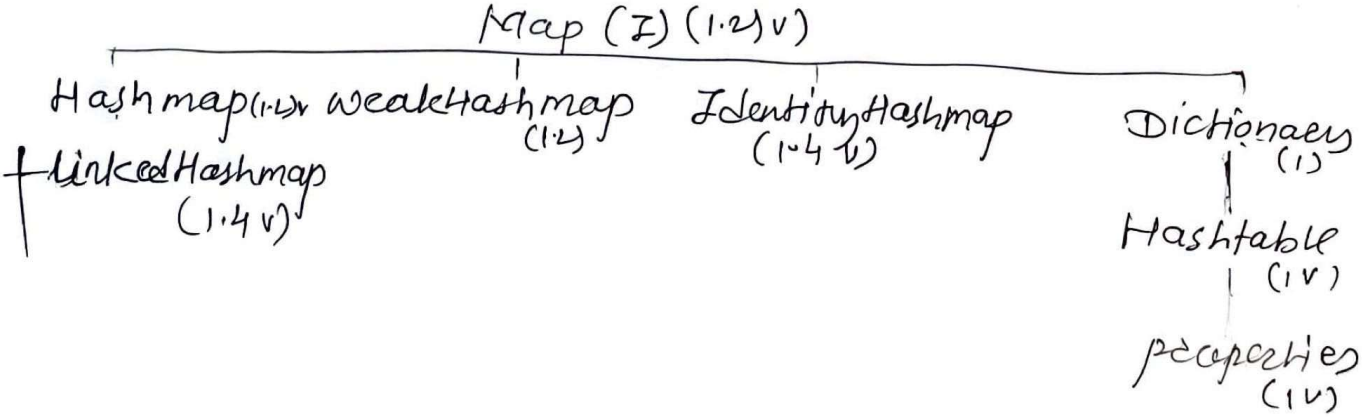
---

⑦ Map (I)

↳ Not child interface of Collection

↳ Key value pair

↳ Both key & value are object & duplicate key are not allowed but dup value is allowed



⑧ **SortedMap : (I)**

↳ child interface of map

↳ key value pair according to some sorting order

Map (I)

↓

SortedMap (I) (1.2)V

⑨ **NavigableMap (I)**

↳ child Interface of Map

↳ defines several utility methods for Navigation purpose

map 1.2V

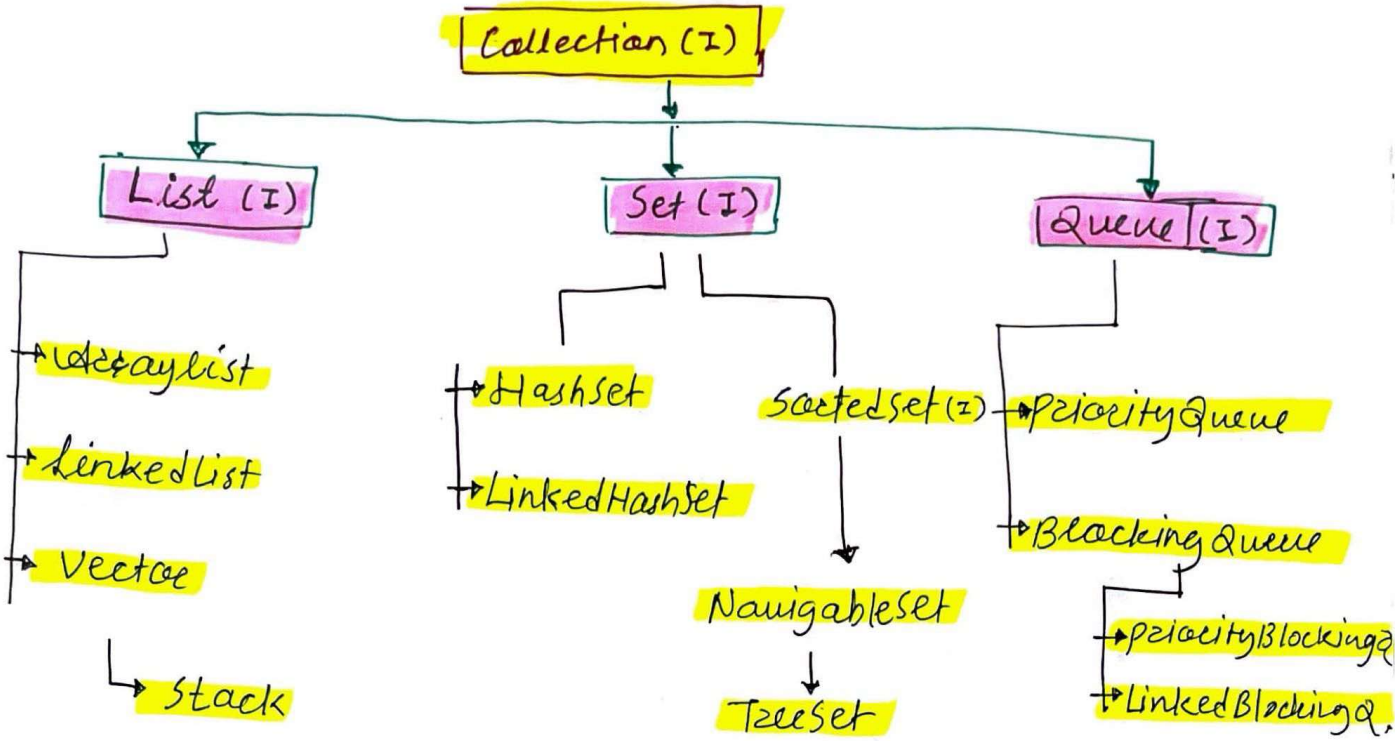
↓

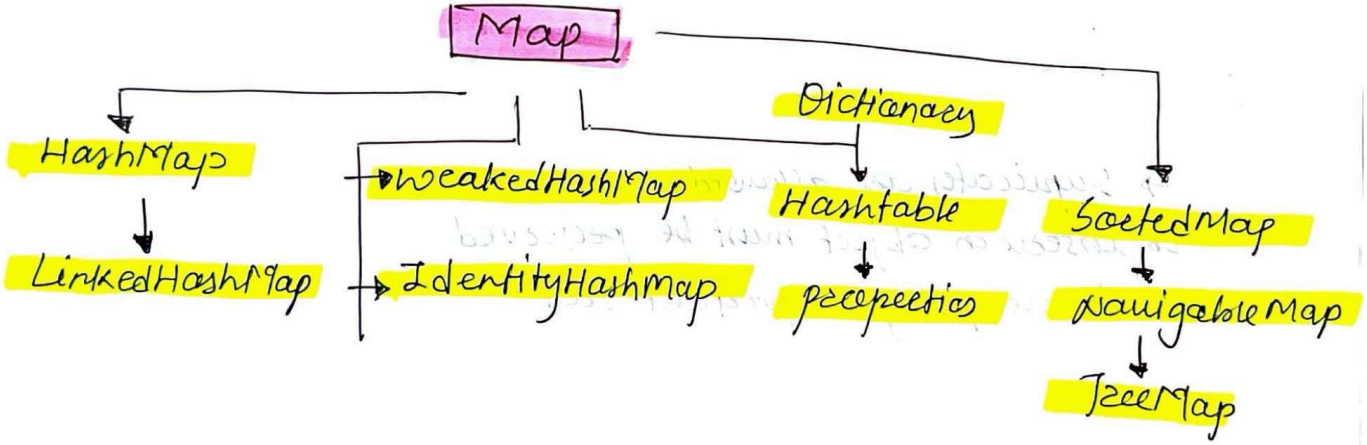
SortedMap 1.2V

↓

NavigableMap 1.6V

↳ TreeMap (C) 1.2V





- Sorting :
  1. comparable
  2. Comparators
- Cursor :
  1. Enumeration
  2. Iterator
  3. ListIterator
- Utility classes
  1. Collection
  2. Arrays

Collection (I)

↳ if we wants to represent group of objects as a single entity then we should go for collection

Imp methods

- add(object o)
- addALL(Collection c)
- remove(object o)
- removeALL(Collection c)
- retainALL(Collection c) ... remove all except c
- clear()
- contains(object o)
- containsAll(~~object o~~ Collection c)
- isEmpty()
- size()
- toArray()
- iterator()

List (I)

↳ Duplicates are allowed
↳ insertion object must be preserved
↳ index play important role

- add(index, object)
- addALL(index, collection)
- get(index)
- remove(index)
- set(index, object)
- indexOf(object)
- lastIndexOf(object)
- ListIterator();

• ArrayList :

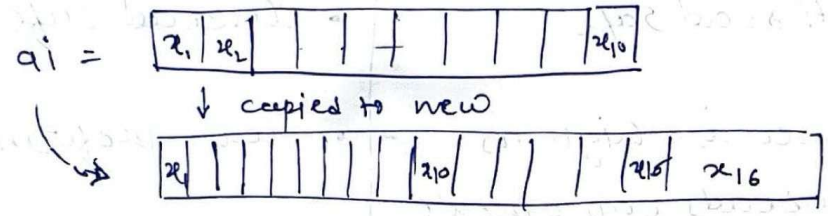
↳ datastructure - Resizable array or Growable array
↳ Duplicates are allowed
↳ Insertion order preserved
↳ Heterogeneous objects are allowed (except treeset & treemap)
↳ Null insertion is possible.

```
ArrayList al = new ArrayList();
```

↳ creates arraylist object with default capacity 10
once it reaches max capacity, new arraylist object will be created with capacity

$$= \left(\text{current capacity} \times \frac{3}{2}\right) + 1$$

$$= (10 \times 3/2) + 1 = 16$$




```
import java.util.*;
class ArrayListDemo {
P.S.V.M(String[] args) {
ArrayList l = new ArrayList();
l.add("A");
l.add(10);
l.add("A");
l.add(null);
Sout(l); // [A, 10, A, null]
l.remove(2);
Sout println(l); // [A, 10, null]
l.add(2, "M");
Sout println(l); // [A, 10, M, null]
}
}
```

- ArrayList & Vector classes implements RandomAccess interface so that any random element we can access with same speed.
- ArrayList is best choice if our frequent operation is retrival operation because it implement RandomAccess interface
- if frequent operation is insertion/deletion, ArrayList is worst choice.

ArrayList	Vector
• Every method is non-synchronised	• every method present is synchronised
• its not thread safe	• thread safe
• performance is high as multiple threads can operate at once	• low performance
• 1.2 version	• 1.0 version hence legacy class

- How to get synchronised version of ArrayList?

→ `ArrayList l1 = new ArrayList();` // Non synchronized.

by default AL is non-synchronized but we can get synchronised version by using collection class synchronisedList() method

→ `List l = Collections.synchronizedList(l1);` // synchronized.

- Similarly we can get synchronized version of Set & Map

```
synchronizedSet(set s1);
synchronizedMap(Map m1);
```

• LinkedList)

↳ Data structure → Double Linked List
↳ Insertion order preserved
↳ Duplicates are allowed
↳ Heterogeneous objects are allowed
↳ Null insertion possible.
↳ it implements serializable & Clonable interfaces but NOT RandomAccess interface.
↳ it is Best choice if frequent operation is insertion & deletion in middle. & worst choice if frequent operation is retrieval.
↳ we can use linked list to implement stacks & queues to provide support for this req. we have below methods in L.L.

- addFirst();
- getFirst();
- removeFirst();
- addLast();
- getLast();
- removeLast();

```
LinkedList L1 = new LinkedList();
    ↳ // create empty LinkedList object
LinkedList L1 = new LinkedList(Collection c);
    ↳ create equivalent L.L object for given collection
```

ArrayList	LinkedList
• best choice → retrieval operation	• best choice → Insertion/deletion
• underlying Data structure ↳ resizable/growable array	• Double Linked List
• implements RandomAccess interface	• Does NOT implement RandomAccess

Vector (legacy class introduction)

↳ underlying DS → resizable array / growable array
↳ Duplicate array allowed
↳ insertion order preserved
↳ Null insertion possible
↳ Heterogeneous objects are allowed
↳ implements serializable, cloanable & RandomAccess interfaces
↳ most of method present in vector are synchronized hence it is thread safe
↳ Best choice if frequent operation is retrieval

- addElement(Object o);
- removeElement(Object o);
- removeElementAt(index);
- removeAllElement();
- elementAt(index);
- size(); • capacity();
- firstElement();
- lastElement();

↳ initial capacity 10, increment = 2x current cap.

```
Vector V = new Vector();
```
↳ create empty vector object

```
Vector V = new Vector(initial capacity);
```
↳ create with same initial cap.

```
Vector V = new Vector(initial cap, incremental cap);
```
↳ create with initial cap & increment i.e. (1000, 5)

```
Vector V = new Vector(Collection c);
```

1005 → 1010 - 1015 etc.

- **Stack**

↳ it is a child class of vector

↳ specially designed for Last in First out order (LIFO)

```
Stack s = new Stack ();
```

methods

- push(object o);
  ↳ insecting object in stack
- pop();
  ↳ removes & returns top of stack
- peak();
  ↳ Returns top of stack without removal of object
- Search(object o);
  ↳ if specified object is available it returns its offset from top otherwise returns -1

eg. offset of D = 1

D	1
C	2
B	3
A	4

```
import java.util.*;
class StackDemo {
p.s.v.m(String[] args){
Stack s = new Stack();
s.push("A");
s.push("B");
s.push("C");
s.o.println(s); // [A, B, C]
s.o.println(s.search("A")); [3]
s.o.println(s.search("Z")); [-1]
  }
}
```

C	1
B	2
A	3

• Cursors of Java

↳ If we want to retrieve objects one by one from collection then we should go for cursors.

↳ Three types:

① - Enumeration
② - Iterator
③ - ListIterator

① Enumeration

↳ 1.0 version
↳ to get objects one by one
↳ we can create enumeration object by using elements() method of vector class
↳ applicable only for legacy classes & NOT universal

```
public Enumeration elements();

Enumeration e = v.elements();   // v: vector object
```

→ we can get only read access & cannot perform remove ope.
↳ methods
- hasMoreElements();
- nextElement();

to overcome this we should go for iterator

```
class EnumerationDemo {
  p.s.v.m(String[] args) {
  Vector v = new Vector();
  for(int i = 0; i <= 10; i++)
      { v.addElement(i);
      }
  s.o.println(v); // [0,1,2,3,4,5,6,7,8,9,10]
  Enumeration e = v.elements();
while(e.hasMoreElements()) {
Integer I = (Integer)e.nextElement();
   if (I%2 == 0)
      s.o.println(I); // [0,2,4,6,8,10]
  }
s.o.println(v); [0,1,2,3...10]
}
```

② Iterator

↳ we can apply this for any collection object hence universal

↳ we can perform both read & remove opr.

↳ we can create iterator object by using iterator()

```
Iterator itr = c.iterator();
```
(c → collection object)

↳ **methods**

- hasNext();
- next();
- remove();

↳ we can move only toward forward direction hence this is single direction cursor

↳ we cant perform replacement operation & to overcome this we should go for ListIterator.

③ ListIterator

↳ using this cursor we can move forward as well as backward (bidirectional)

↳ we can perform replacement & addition of new obj. & read & remove operations

```
ListIterator itr = L.listIterator();
```
(L → list object)

↳ it is child interface of iterator

↳ methods

- hasNext();
- next();
- nextIndex();
- hasPrevious();
- previous();
- previousIndex();
- remove();
- set();
- add();

↳ it is applicable only for List implemented class object & it is NOT universal cursor.

• Set

↳ it is child interface of collection

↳ it represents a group of an individual objects as a single entity where duplicates are not allowed & insertion order is NOT preserved

↳ it doesn't contains any new method so we have only collection interface method.

• HashSet

↳ underlying data structure is hashtable

↳ Duplicates are NOT allowed. if we try to insert null we will NOT get any error. add() method returns false if we try to add dupe.

↳ Insertion order is NOT preserved & all objects will be inserted based on hash-code of objects

↳ Heterogeneous objects are allowed

↳ 'null' insertion possible

↳ implements serializable & clonable interfaces but NOT RandomAccess

↳ if frequent operation is search then HashSet is best choice.

```
HashSet h = new HashSet();          // creates empty hashset with initial cap 16 & fill ration 0.75
HashSet h = new HashSet(int initialCap);          // create with specified initial capacity
HashSet h = new HashSet(intinitialCap, loadFactor);          // Specified initial cap & fill ratio (loadfactor)
HashSet h = new HashSet(Collection c);
```

fill ratio: after filling 0.75 % of Hashset then new Hashset will be created in memory

LinkedHashSet

↳ it is child class of HashSet

↳ 1.4 version

↳ exactly same as HashSet except following diff.

→ underlying data structure is Hash table + Linked list

→ insertion order preserved

↳ it is best option to develop cache based application where duplicates are NOT allowed & insertion order must be preserved.

SortedSet

↳ child interface of set

↳ duplicates NOT allowed & some sorting order is present

↳ SortedSet specific Methods: {10, 20, 30, 40, 50}

- object first(); → returns 1st element {10
- Last(); → last element of SortedSet {50}
- headSet(object obj); → returns elements < obj [10, 20, 30]
- tailSet(object obj); → returns elements >= obj [30, 40, 50]
- subSet(obj1, obj2); → element >= obj1 & < obj2 ↳ [20, 30, 40]
- comparator();

Treeset

↳ datastructure → Balanced tree

↳ Duplicate object NOT allowed

↳ Insertion order NOT preserved but object are inserted according to some sorting order

↳ Heterogeneous objects are NOT allowed if we try to insert heterogeneous we will get runtime exception as ClassCastException

↳ Null insertion allowed but only once.

→ for empty Treeset as first element null insertion is possible but after that if we try to insert null we will get NullPointerException

→ for Non empty treeset if we try to insert null we will get Nullpointer Exception.

→ If we are depend on default natural sorting order then objects should be homogeneous & comparable otherwise we will get runtime exception saying ClassCastException

→ an object said to be comparable if and only if corresponding class implements comparable interface

→ String & other wrapper classes already implements comparable but StringBuffer doesn't

• Comparable Interface •

↳ present in java.lang package
↳ contains only one method compareTo().

obj1.compareTo(obj2)
→ returns -ve if obj1 has to come before obj2
→ returns +ve if obj1 has to come after obj2
→ returns 0 if obj1 & obj2 are equal

↳ default natural sorting order.

• Comparator Interface

↳ it is for customised sorting order
↳ present in java.util package
↳ define two methods compare & equals

public int compare(obj1, obj2);
→ returns -ve if obj1 has to come before obj2
→ returns +ve if obj1 has to come after obj2
→ returns 0 if obj1 & obj2 are equal

public boolean equals();

COMPARABLE	COMPARATOR
• meant for default sorting order	• meant for customized sorting order
• present in java.lang package	• present in java.util package
• defines one method: compareTo()	• defines two methods compare() & equals()
• all wrapper classes & String class implements this	• only implemented classes are Collator & RuleBasedCollator

- **Queue**
  - extends collection interface
  - First in First out

- **priority Queue :**
  - child class of queue
  - processing objects on basis of priority

```
priorityQueue queue = new PriorityQueue();
```

- **Dequeue**
  - child class of queue
  - unlike queue we can add & remove elements from both ends
  - Null Not allowed
  - Not threadsafe
  - Deque can be used as stack or queue because Stack follows Last in first out (LIFO) operation & queue follows (FIFO) & Dequeue follows both

* Map

↳ contains values on basis of key & value pair
↳ key's are unique but value can be duplicate
↳ Map is useful if we have to search, update or delete elements on basis of key

* HashMap & LinkedHashMap allows both key & value as ~~duplicate~~ null
* Treemap doesn't allows any ~~da~~ null key or value

↳ Each key & value pair considered as entry hence map considered as collection of entry object

* methods of Map

- Object put(key, value);
  ↳ to add entry to map
  ↳ if key is already present then old value will be replaced with new
- putAll();
- get(key); → returns value associated with key
- remove(key); → remove entry of that key
- contains(key); , ~~contains(value)~~
- containsValue(value);
- isEmpty();
- clear();

HashMap

- underlying datastructure is HashMap
- insertion order is NOT preserved
- Duplicate key NOT allowed
- Heterogeneous object allowed
- null key allowed only once
- best choice if frequent operation is search

LinkedHashMap

- child class of HashMap
- insertion order preserved
- underlying DS → LinkedList + HashTable

• WeakHashMap

↳ Almost same as Hashmap except in case of WeakHashmap if object is specified as key doesn't contain any refercances - it is eligible for garbage collection even though it is associated with weakHashmap.

↳ both null value & key is supported

↳ NOT Synchronized

• SortedMap :

↳ it is child interface of map

↳ Sorting based on key

↳ underlying DS → Red-Black Tree.

eg. 101 - A
102 - B
103 - C
104 - D
105 - E

firstKey(); → 101
lastKey(); - 105
headMap(103); → (101 - A, 102 - B).
tailMap(103); (103 - C, 104 - D, 105 - E);
subMap(102, 104); (102 - B, 103 - C);

• TreeMap

↳ DS → Red-Black Tree

↳ duplicate key NOT allowed, value allowed

↳ Null key NOT allow, value allow

↳ Sorting will be key basis

↳ Heterogeneous key ✗, Heterogeneous value ✓