/># Core Java Notes By Ayushi Jain

Link to Headings

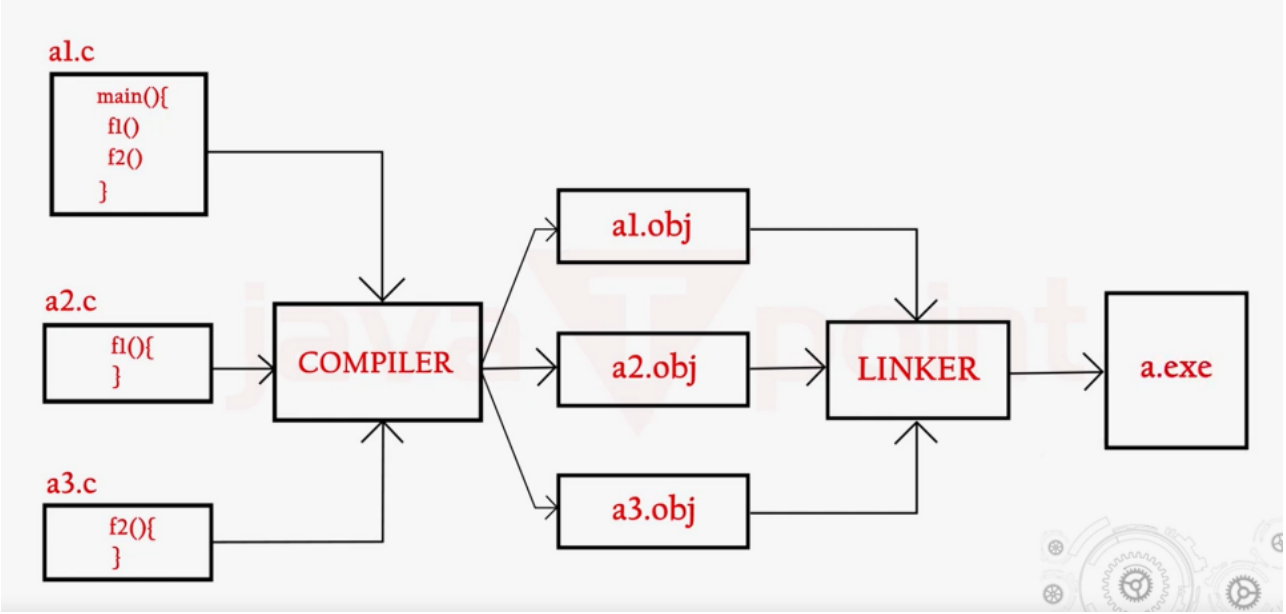
1. Basic

Java

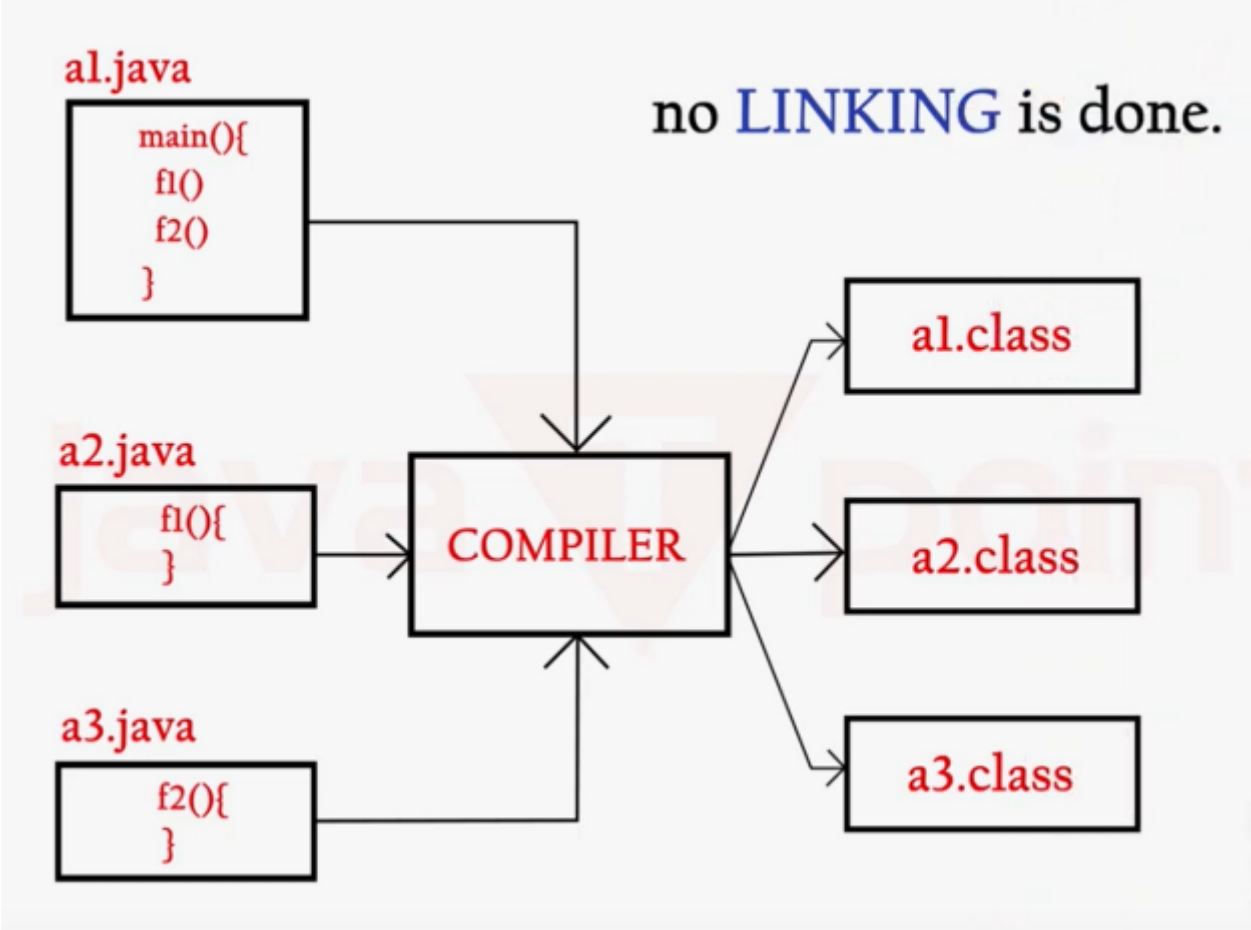
- Java is a programming language.
- Java is a high level, robust, object-oriented and secure programming language.
- The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic"
- Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year `1995`. `James Gosling` is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Java Basics

- We can have only one public class in a single java file
- Name of the file should be same as the same of public class
- In absence of public class, any class name can be given to the file name
- WORA - Write Once Run Anywhere
- Java code (.java) -> Compiler-Javac -> Byte Code(.class)



- NO Linker here



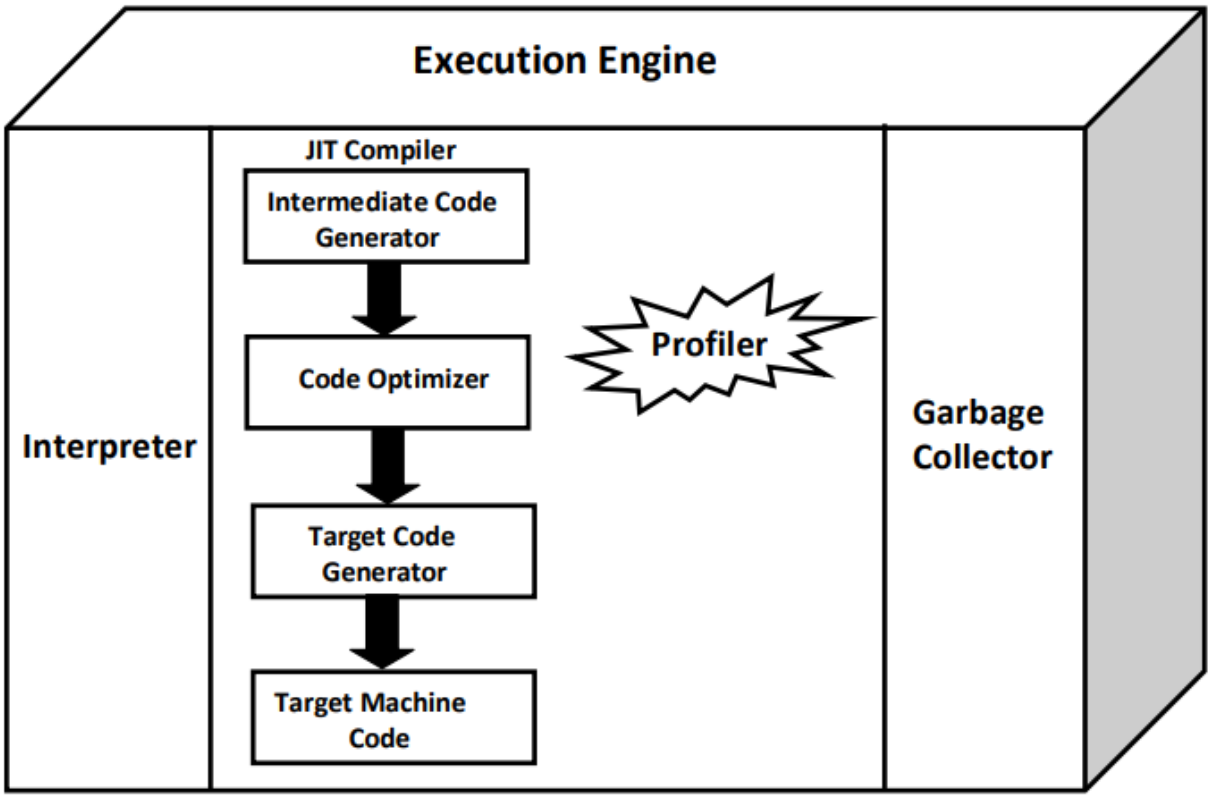
Interpreter

- It is Responsible to Read Byte Code and Interpret (Convert) into Machine Code (Native Code) and Execute that Machine Code Line by Line.
- `The Problem with Interpreter` is it Interprets Every Time Even the Same Method Multiple Times. Which Reduces Performance of the System.

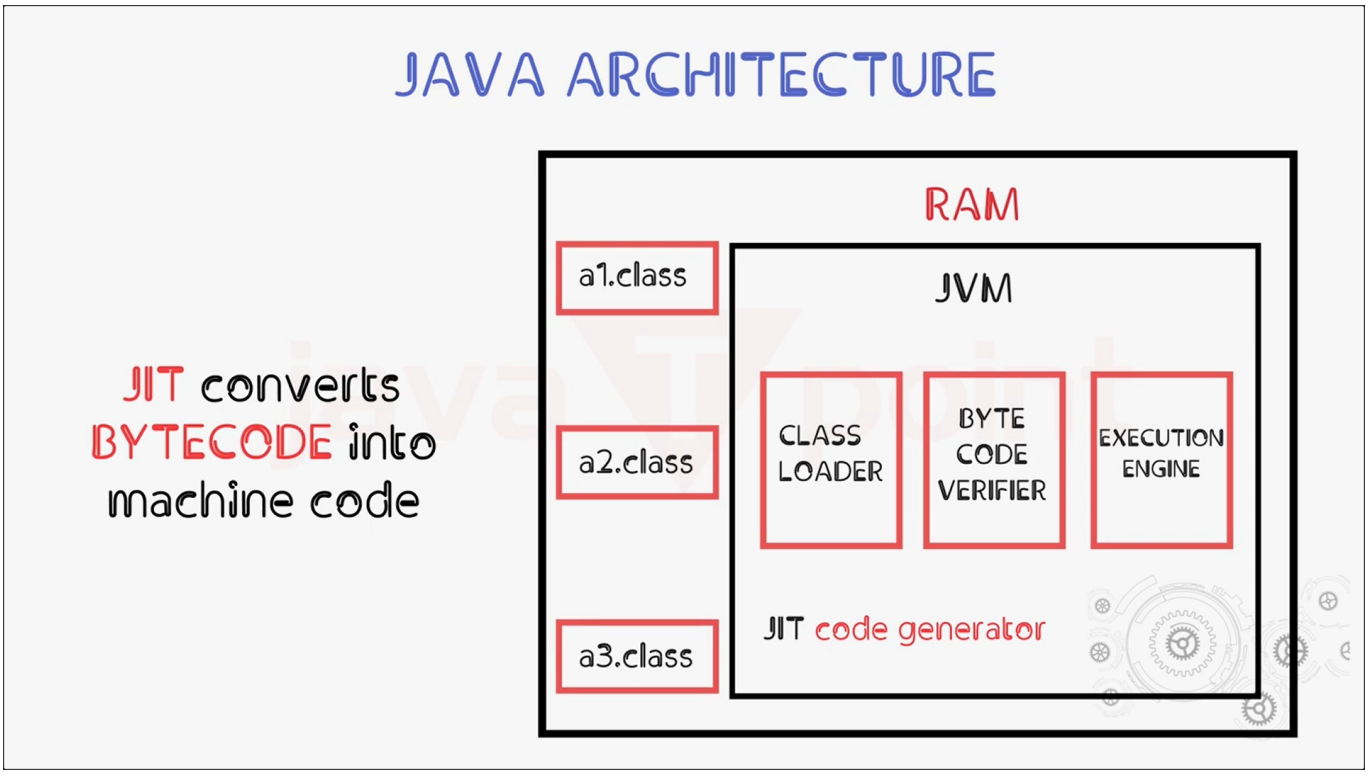
JIT Compiler

- Just-In-Time compiler
- Converts Byte code into machine code
- The Main Purpose of JIT Compiler is to Improve Performance.
- Internally JIT Compiler Maintains a `Separate Count for Every Method` whenever JVM Come Across any Method Call.
- First that Method will be interpreted normally by the Interpreter and JIT Compiler Increments the corresponding Count Variable.
- `HOTSPOT` : JIT Compiler Identifies that Method Repeatedly used Method
- `Profiler` : Part of JIT Compiler is Responsible to IdentifY HOT SPOTS

- The Threshold Count Value varied from JVM to JVM.



- JIT is like Interpreter though is a compiler, It does not converts all the file in one go like compiler, but converts each line and while it is processed converts another line and stores like interpreter.
- Not in one go - unlike compiler but yet sends before fully converting
- Line by line - like interpreter but also keeps sending line as it converts and stores next
- `JVM Interprets Total Program Line by Line at least Once`.
- `JIT Compilation` is Applicable Only for `Repeatedly invoked Methods`. But Not for Every Method.

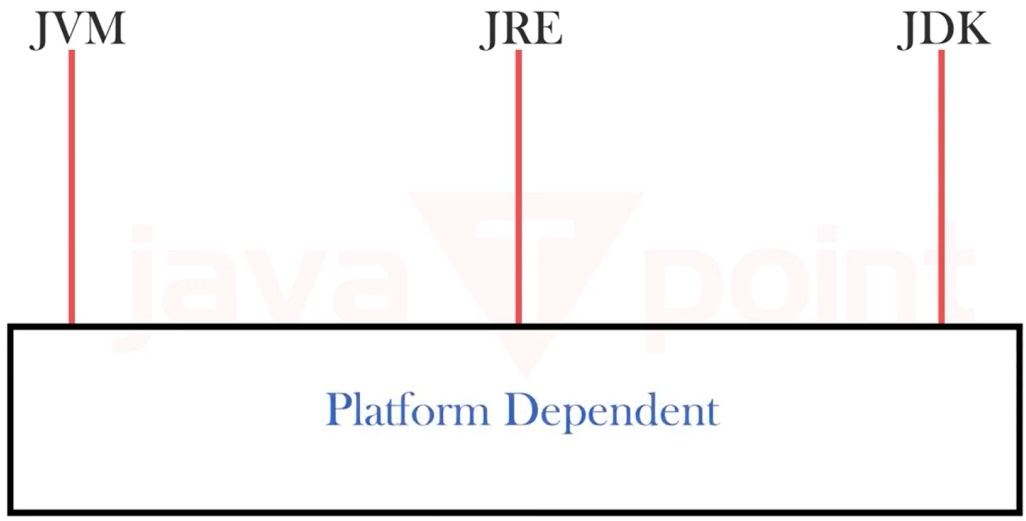


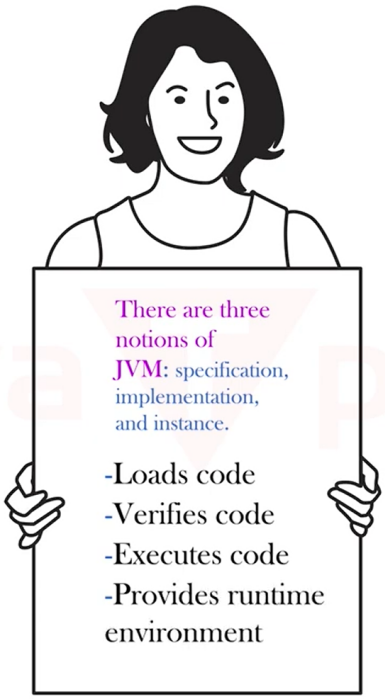
Run Commands

```
javac Hello.java
java Hello
or
java Hello.java
```

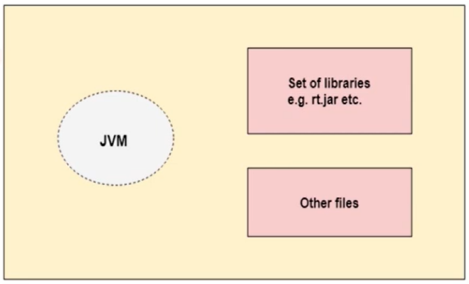
JDK, JRE, JVM

- JDK - Java Development Kit - `JRE + Tool` for Java lang - javac, jar | Physically exists
- JRE - Java Runtime Environment - `JVM + Inbuilt-extra libraries-classes` | Physically exists
- JVM - Java Virtual Machine - Virtual layer | `Code executes here` | Virtually exists
- All are platform dependent.
- Java is platform Independent.


JVM
JRE
JDK
Platform Dependent


There are three notions of JVM: specification, implementation, and instance.
-Loads code
-Verifies code
-Executes code
-Provides runtime environment

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



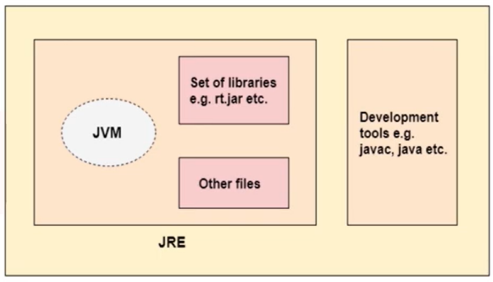
JRE

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

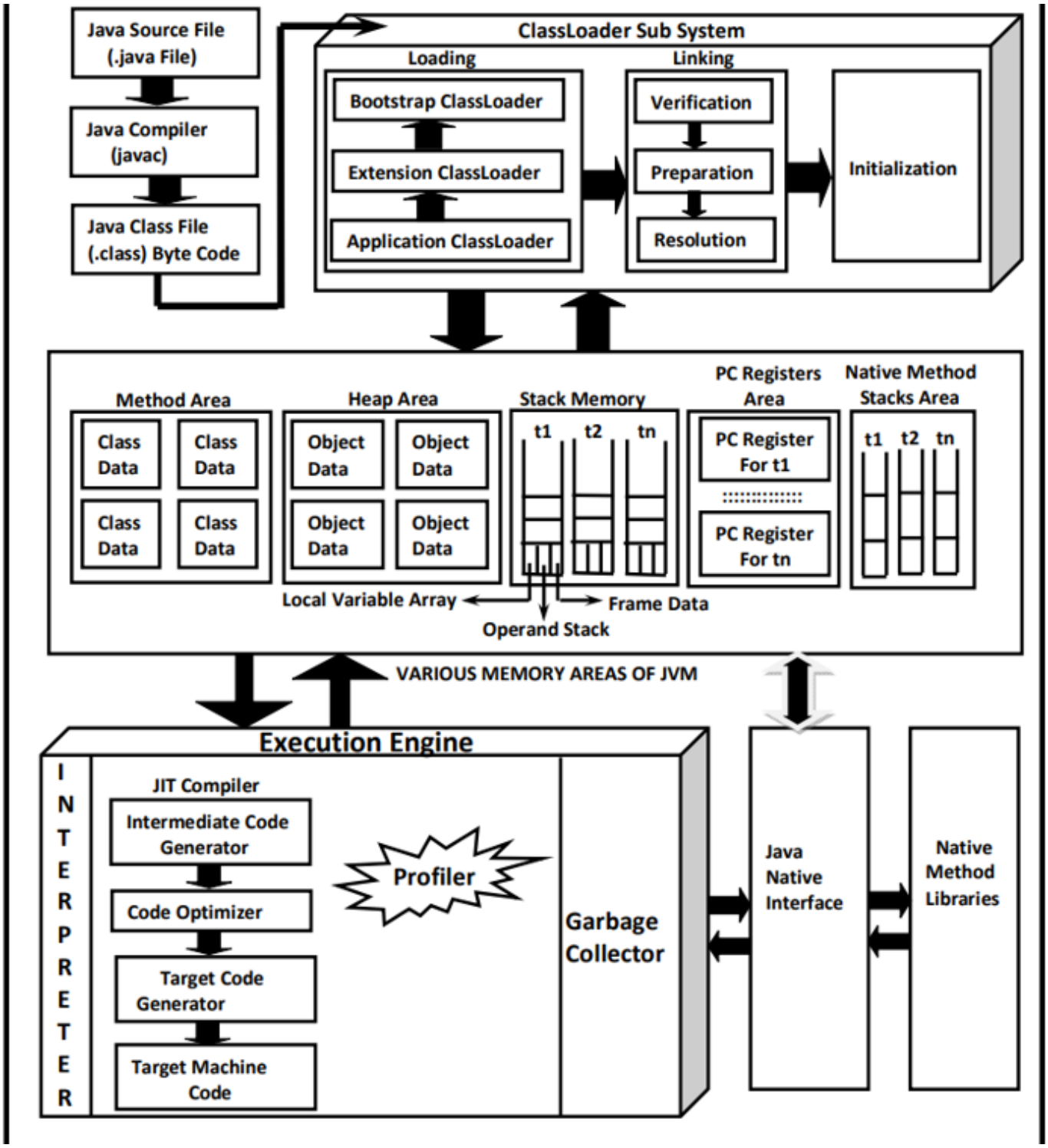
JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

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

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



JDK



Data Types

1. Integer
    - int : 4 bytes
    - long : 8 bytes
    - short : 2 bytes
    - byte : 1 bytes : 2^7 to 2^7-1 : -128 to 127
2. Float
    - float : 4 bytes : 5.6f
    - double : 8 bytes : 5.6
3. Character : 2 bytes | UNICODE

4. Boolean : Machine dependent | true/false | NO 0/1

Extra

```
// System.out.println(10/0); // Exception in thread "main"
java.lang.ArithmeticException: / by zero
System.out.println(10/0.0); // Infinity
System.out.println(-10/0.0); // -Infinity

// System.out.println(0/0); // Exception in thread "main"
java.lang.ArithmeticException: / by zero
System.out.println(0/0.0); // NaN : Undefined
System.out.println(0.0/0.0); // NaN : Undefined
System.out.println(-0.0/0.0); // NaN : Undefined

System.out.println(Float.NaN == Float.NaN ); // false
System.out.println(Float.NaN != Float.NaN ); // true

int i=1;
i+=++i + i++ + ++i + i++; // i=i + ++i + i++ + ++i + i++
System.out.println(i); //13
```

Literals

- int num = 0x7E : Hexadecimal
- int num1 = 0b101 : Binary
- int num2 = 10*00_00_000 :* For 0 seperation
- int double num3= 12e10/1.2e11
- long l=5854l;
- float f=5.8f;

```
public class hello {
    public static void main(String[] args) {
        int num1=9;
        byte by=127;
        short sh=558;
        long l=58541;

        float f=5.8f;
        double d=5.8;

        char c='k';

        boolean b=true;
    }
}
```

Type Conversion and Casting

- byt b = 127;
- int a = 12;

Conversion

- Lower to upper
- a=b : Conversion

Casting

- Upper to lower
- b=a NO => b = (byte)a : Casting
- `Explicitly`
- In case of out of bound : `number%Range`
- byt b = 127;
- int a = 257; : byte max can be 128
- b = (byte)a : b will be 1 (257%256) if >256 or
- byte a = 130;
- b = byte(a) : a-256 : 130-256 : -126

```
public class Basic {
    public static void main(String args[]){
        int a = 257;
        byte b = 127;
        System.out.println("a - "+a+" b - "+b); // a - 257 b - 127
        // a=b;
        System.out.println("a - "+a+" b - "+b); // a - 127 b - 127
        // b=a; // Error
        b=(byte)a;
        System.out.println("a - "+a+" b - "+b); // a - 257 b - 1

        int a2=2567;
        byte b2=(byte)a2;
        System.out.println("b2 - "+b2); // b2 - 7

        int a2=130;
        byte b2=(byte)a2;
        System.out.println("b2 - "+b2); // b2 - -126

        float f=5.6f;
        int t=(int)f;
        System.out.println("t - "+t); // t - 5

        byte a3=10;
        byte b4=20;
        int t1=a3*b4;
```

```
        System.out.println("t - "+t1); // t1 - 200
    }
}
```

- If we perform `downcasting by typecasting`, `ClassCastException` is thrown at runtime in case of classes : Use instanceof to avoid exception.

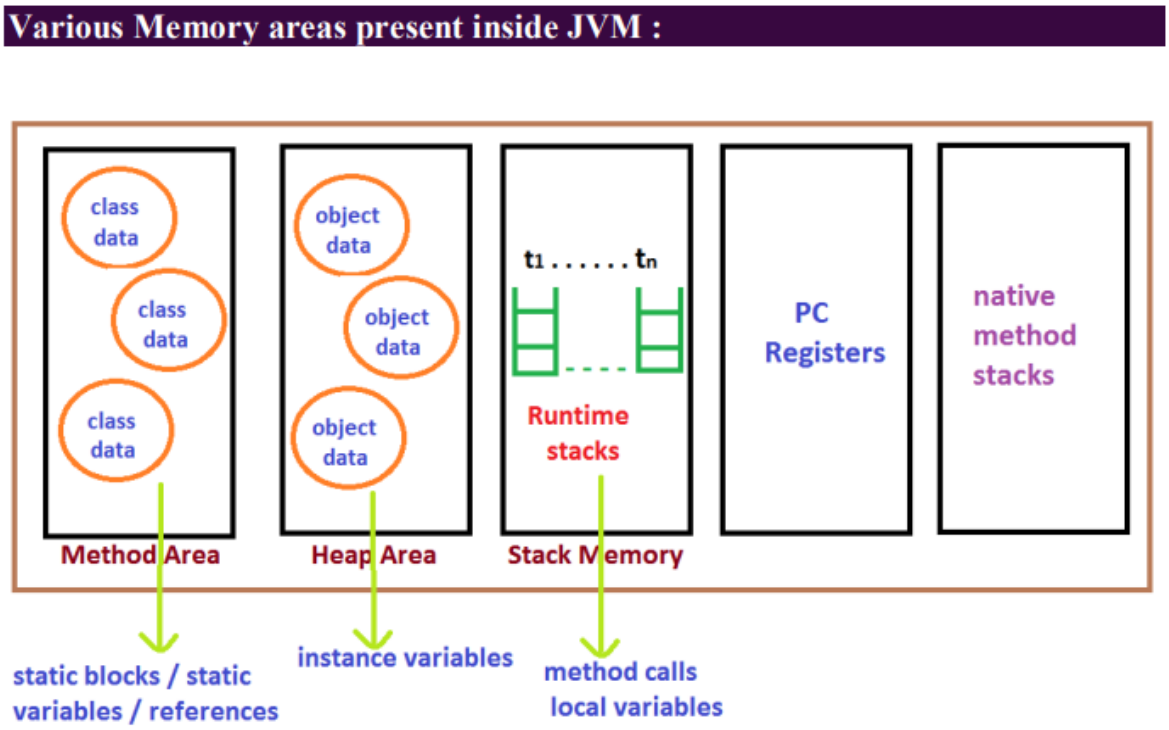
```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime

if(a instanceof Dog){
    Dog d=(Dog)new Animal();//downcasting
}
```

Memory Areas

- `Major Memory Areas` : `Method Area, Heap Area and Stack Area` with Respect to Programmers Point of View.
- Static Variables will be stored in Method Area
- Instance Variables will be stored in Heap Area
- Local Variables will be stored in Stack Area.

- Strings in String Constant Pool in Heap Area



1. **Class level binary data includung static variables will be stored in method area.**
2. **Objects and corresponding instance variables will be stored in Heap area.**
3. **For every method the JVM will create a Runtime stack all method calls performed by that Thread and corresponding local variables will be stored in that stack.**
   Every entry in stack is called Stack Frame or Action Record.
4. **The instruction which has to execute next will be stored in the corresponding PC Registers.**
5. **Native method invocations will be stored in native method stacks.**

- `Method Area and Heap Area` are for `JVM`.
- `Stack Area, PC Registers Area and Native Method Stack Area` are for `Thread` :
- One Separate Heap for Every JVM
- One Separate Method Area for Every JVM
- One Separate Stack for Every Thread
- One Separate PC Register for Every Thread
- One Separate Native Method Stack for Every Thread

---

2. Wrapper Classes

- Are used to manipulate primitive values as objects.
- Are `final`.
- `Objects of wrapper classes are immutable`.
- Java is not said to be 100% Object Oriented due to its non-object primitive types.

- As a solution to this problem, Java allows us to `include the primitives in the family of objects` by using what are called as Wrapper classes.
- There is a wrapper class for every primitive data types in Java.

```
int -> Integer
byte -> Byte
```

Methods of wrapper Classes

- valueOf() - Integer.valueOf() - Static method - Object reference of that class - `Character class does not define this method`.
- parseXXX() - Integer.parseInt() - Static Method - returns XXX type value
- XXXValue() - Integer.intValue() - Instance Method - returns XXX primitive type
- int compareTo(WrapperType obj2)
- .MIN_VALUE
- .MAX_VALUE

Auto Boxing & Unboxing

Boxing

- Representing primitive in the form of object is termed as boxing

Unboxing

- Converting it back to primitive from reference is termed as unboxing.

Auto Boxing

- Auto Boxing is the process where the primitive data member automatically gets converted into its respective wrapper object.

Auto Unboxing

- Auto unboxing is the process of automatically extracting the primitive value wrapped inside the object.

```
Integer intObj1 = new Integer(10); //boxing
Integer intObj2 = 10; //auto boxing

int num1 = intObj1.intValue(); //unboxing
int num2 = intObj2; //auto unboxing
```

Packages and Import

- Its nothing more than the way we `organize files into different directories` according to their functionality, usability, category they should belong to!
- Different package have different functionality
- java.io - IO related, java.net - Network related
- Help us `avoid class name collision`
- Provides ease of maintenance, organization and increase collaboration among developers
- Create package in java file - `package folderName;`
- Compile - `javac -d . fileName.java`
- Run - `java packageName.fileName`
- `import packageName.ClassName`

Classes

When we create obj :-

1. class is loaded : static{} calls
2. object is instantiated

- when obj is created then only class would be compiled :- .java to .class
- no obj no class loader, static{} won't work - use `Class.forName(Student)` : class loaded now

Access Modifiers

1. private - only same class
2. default - only same pkg
3. protected - not outside pkg, but in subclass which can be outside pkg

Private < Default < Protected < Public

Entity	Declaration context	Accessibility Modifier	Enclosing Instance	Direct Access to enclosing context	Declarations in entity body
Top level class (or interface)	Package	public or default	No	N/A	All that are valid in a class or interface body
static member class (or interface)	as static member of encloing class or interface	All	No	Static members in enclosing context	All that are valid in a class or interface body
Non-static member class	As non-static member of encloing class or interface	All	Yes	All members in enclosing context	Only non static declaration + final static fields
Local Class	In block with non static context	None	Yes	All members in enclosing context + final local variables	Only non static declaration + final static fields
	In block with static context	None	No	Static members in enclosing context + final local variables	Only non static declaration + final static fields
Anonymous class	As expression in non static context	None	Yes	All members in enclosing context + final local variables	Only non static declaration + final static fields
	As expression in static context	None	No	Static members in enclosing context + final local variables	Only non static declaration + final static fields

Access modifiers	Class	Package	Subclass within the package	Subclass outside the package	World
private	✓	✗	✗	✗	✗
default/No Modifier	✓	✓	✓	✗	✗
protected	✓	✓	✓	✓	✗
public	✓	✓	✓	✓	✓

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

The only applicable modifiers for Top Level `classes` are:

1. Public
2. Default
3. Final
4. Abstract
5. Strictfp

The only applicable modifiers for `inner classes` are:

1. Public

2. Default
3. Final
4. Abstract
5. Strictfp
6. private
7. protected
8. static

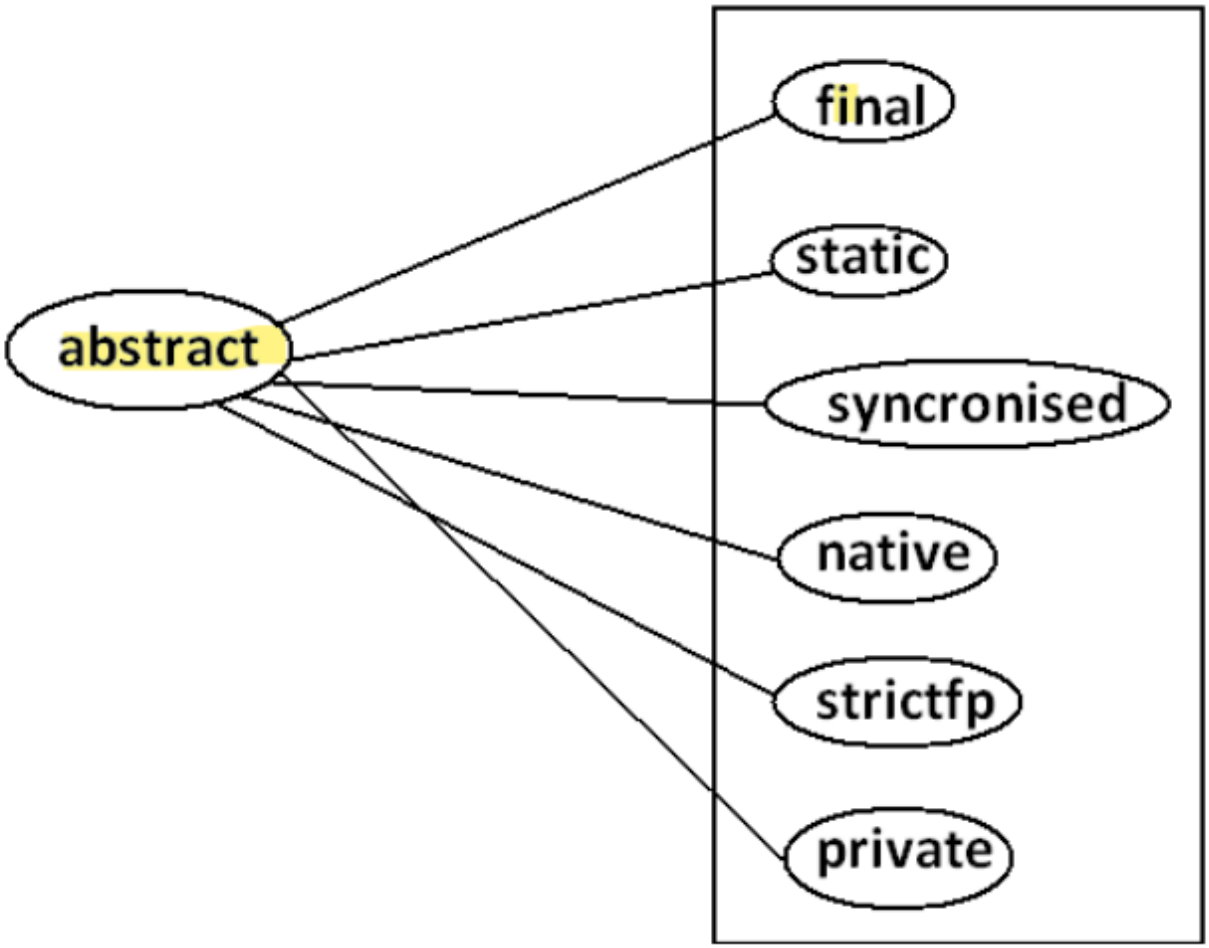
Things you can’t mark as static

- Static block gets executed before main method gets executed.
- Constructors
- Classes
- Interfaces
- Local variables
- Inner classes
- Inner class methods and instance variable

Illegal Combinations

The following are the various illegal combinations for methods.

Diagram:



All the 6 combinations are illegal.

Transient

- Transient is the modifier applicable only `for variables` but not for methods and classes.
- At the time of serialization if we don't want to serialize the value of a particular variable to meet the security constraints then we should declare that variable with transient modifier.
- `transient means "not to serialize"`.

Violatile

- If the value of variable keeps on changing such type of variables we have to declare with volatile modifier.
- If a variable declared as volatile then for every thread a separate local copy will be created by the jvm, all intermediate modifications performed by the thread will takes place in the local copy instead of master copy.

- Once the value got finalized before terminating the thread that final value will be updated in master copy.

Final

- variable - can't change value
- class - can't inherit : Against Inheritance
- method - can't override : Against Overriding concept - Polymorphism
- `Final class cannot contain abstract methods whereas abstract class can contain final method.`

Applicable Modifiers

Summary of modifier:

Modifiers	Classes		Methods	Variables	Blocks	Interfaces		enum		Constructors
	Outer	Inner				Outer	inner	Outer	Inner	
public	√	√	√	√	×	√	√	√	√	√
private	×	√	√	√	×	×	√	×	√	√
protected	×	√	√	√	×	×	√	×	√	√
<default>	√	√	√	√	×	√	√	√	√	√
final	√	√	√	√	×	×	×	×	×	×
static	×	√	√	√	√	×	√	×	√	×
synchronized	×	×	√	×	√	×	×	×	×	×
abstract	√	√	√	×	×	√	√	×	×	×
native	×	×	√	×	×	×	×	×	×	×
strictfp	√	√	√	×	×	√	√	√	√	×
transient	×	×	×	√	×	×	×	×	×	×
volatile	×	×	×	√	×	×	×	×	×	×

- The Only Applicable Modifiers for `Local Variable` is final.
- The Modifiers which are Applicable for `Variables` are transient and volatile.
- The Only Modifier which is Applicable for `Methods` is native.
- The Only Applicable Modifiers for `Constructors` are public, private, protected, and default.
- The Only Modifier which is applicable for Classes but `Not for Interfaces` is final.
- The Modifiers which are Applicable for Classes but `Not for enum` are final and abstract.
- The Modifiers which are Applicable `for Inner Classes` but `Not for Outer Classes` are public, protected, and static.

Factory method

- The method that returns the instance of a class is known as factory method.

```
class GetPlanFactory{

   //use getPlan method to get object of type Plan
       public Plan getPlan(String planType){
            if(planType == null){
             return null;
            }
          if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
                return new DomesticPlan();
              }
           else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
                return new CommercialPlan();
            }
          else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
                return new InstitutionalPlan();
          }
      return null;
   }
}//end of GetPlanFactory class.


// Usage
GetPlanFactory planFactory = new GetPlanFactory();
//  getPlan is factory method
Plan p = planFactory.getPlan("DOMESTICPLAN");
```

OOPS

1. Data Hiding

- Our internal data should not go out directly that is outside person can't access our internal data directly.
- By using private modifier we can implement data hiding.

2. Abstraction

- Hide internal implementation and just highlight the set of services, is called abstraction.
- By using abstract classes and interfaces we can implement abstraction

3. Encapsulation

- Binding of data and corresponding methods into a single unit is called Encapsulation .
- If any java class follows data hiding and abstraction such type of class is said to be encapsulated class.
- Encapsulation = Datahiding + Abstraction : Every data member should be declared as private and for every member we have to maintain getter & Setter methods.
- Getters and setters

4. Inheritance

- Parent-child relationship
- By using "extends" keywords we can implement IS-A relationship.
- IS-A : extends
- HAS-A : reference and obj of another class
- `Association(bank, emp)` -> aggre-> compo
- `Composition: Strong - University-Departmnet` | one object contains other objects as a part of its state | containing objects do not have an independent existence
- Without existing container object if there is no chance of existing contained objects then the relationship between container object and contained object is called composition which is a strong association.
- `Aggregation : Weak - University-Professor` | one object contains other objects as a part of its state | HAS - A
- Without existing container object if there is a chance of existing contained objects such type of relationship is called aggregation. In aggregation objects have weak association.

super(), this()	super, this
These are constructors calls.	These are keywords
We can use these to invoke super class & current constructors directly	We can use refers parent class and current class instance members.
We should use only inside constructors as first line, if we are using outside of constructor we will get compile time error.	We can use anywhere (i.e., instance area) except static area , other wise we will get compile time error .

5. Polymorphism

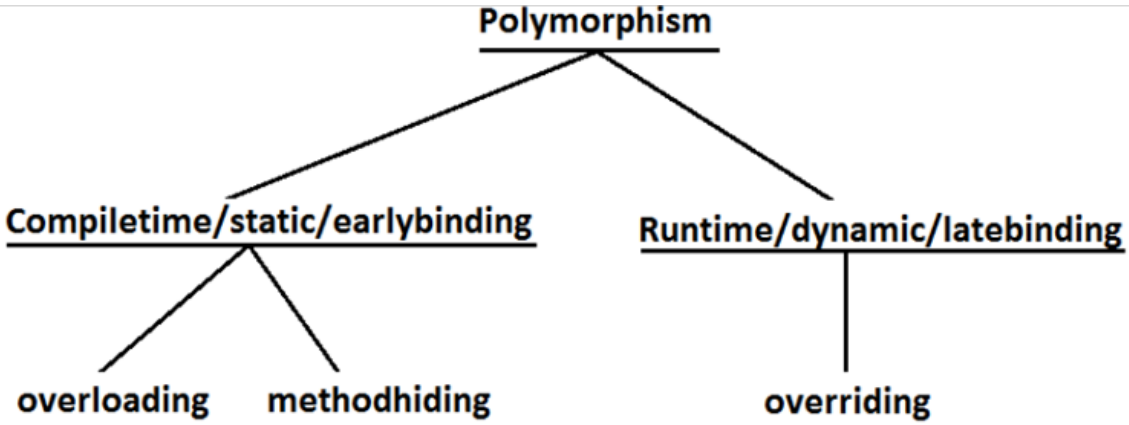
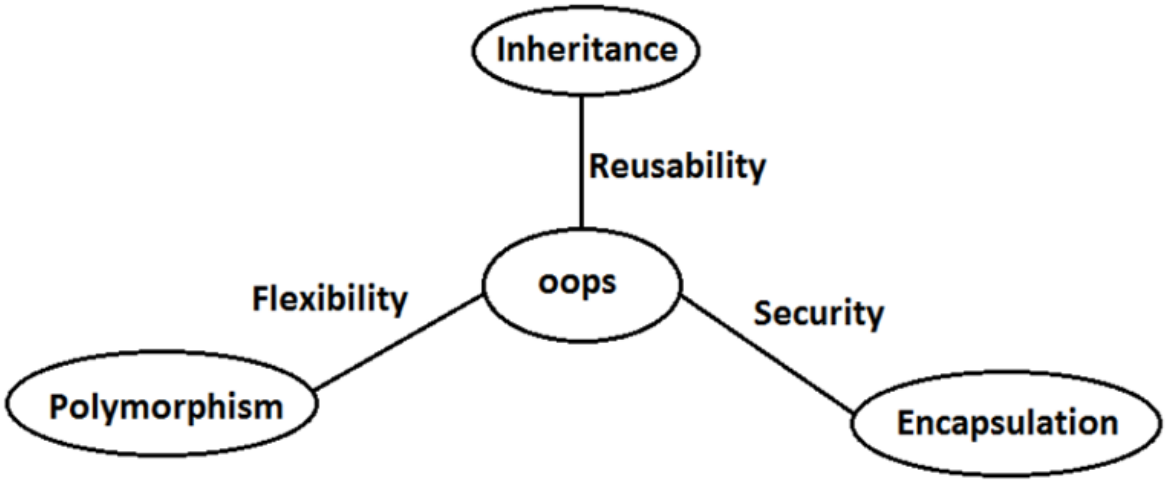


Diagram: 3 Pillars of OOPS



Method Overloading

- `reference type`
- `only same name all 2 diff`
- Having the same name and different argument types is called method overloading
- Comile time polymorphism
- Ex
- abs(int), abs(float)...
- List reference, arrayList, vector obj
- Automatic promotion in overloading
- `In overloading method resolution is always based on reference type and runtime object won't play any role in overloading.`
- While resolving overloaded methods exact match will always get high priority,
- While resolving overloaded methods child class will get the more priority than parent class
- var-arg method will get less priority

Method Overriding - dynamic method dispatch

- `object`
- `all 3 same including arg order`method names and arguments must be same. That is method signature must be same. *The process of overriding method resolution is also known as dynamic method dispatch.
- `In overriding runtime object will play the role and reference type is dummy.`
- We can override a non-abstract method as abstract
- In overriding we should compulsory check everything like method names, arguments, return types, `throws keyword, modifiers` etc.
- `Co-variant return type` (exact type or types child) concept is applicable only for object types but not for primitives. It is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type.
- Java doesn't allow the return type-based overloading, but JVM always allows return type-based overloading. JVM uses the full signature of a method for lookup/resolution.

```
class E{
    E get(){return this;}
}

class F extends E{
    @Override
    F get(){return this;}
    void message(){System.out.println("welcome to covariant return type");}

    public static void main(String args[]){
        new F().get().message();
    }
}
```

- The return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.
- You `cannot override the static and final methods` of a superclass

Overloading V/s Overriding

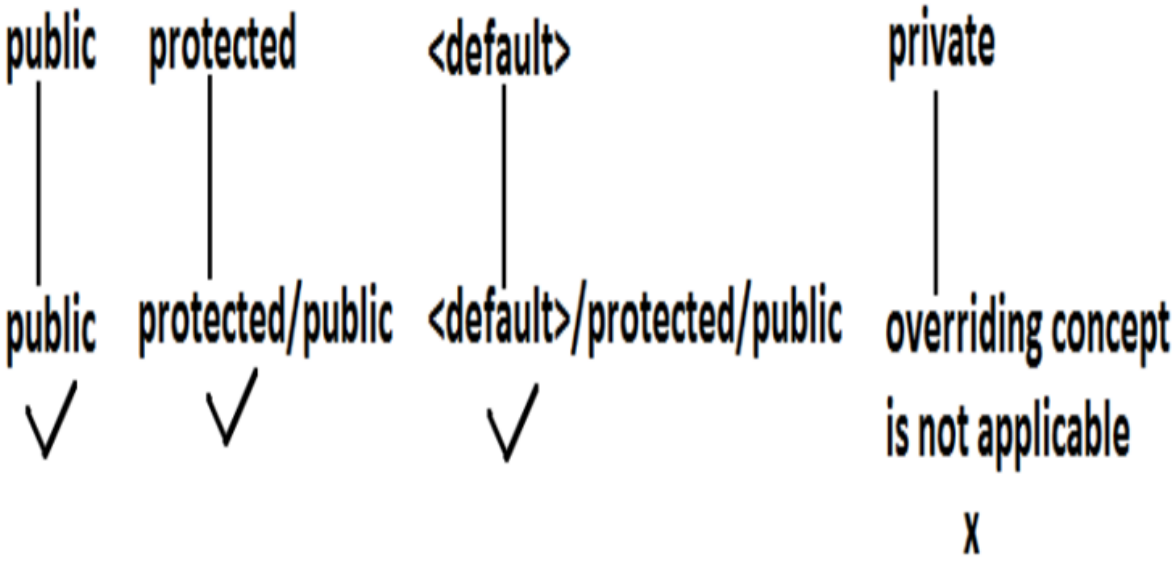
Property	Overloading	Overriding
1) Method names	Must be same.	Must be same.
2) Argument type	Must be different(at least order)	Must be same including order.
3) Method signature	Must be different.	Must be same.
4) Return types	No restrictions.	Must be same until 1.4v but from 1.5v onwards we can take co-variant return types also.
5) private, static, final methods	Can be overloaded.	Can not be overridden.
6) Access modifiers	No restrictions.	Weakering/reducing is not allowed.
7) Throws clause	No restrictions.	If child class method throws any checked exception compulsory parent class method should throw the same checked exceptions or its parent but no restrictions for un-checked exceptions.
8) Method resolution	Is always takes care by compiler based on referenced type.	Is always takes care by JVM based on runtime object.
9) Also known as	Compile time polymorphism (or) static(or)early binding.	Runtime polymorphism (or) dynamic (or) late binding.

Method Overriding

`object`

1. Both Parent and Child class methods should be `non static`.
2. Method resolution is always takes care by JVM based on `runtime` object.
3. Overriding is also considered as runtime polymorphism (or) dynamic polymorphism (or) late binding.

- static method can be overloaded, but cannot be overridden



private < default < protected < public

Method hiding

`reference type`

1. Both Parent and Child class methods should be `static`.
2. Method resolution is always takes care by `compiler` based on reference type.
3. Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early biding.

Overriding V/s Hiding

Overriding	Method hiding
1. Both Parent and Child class methods should be non static.	1. Both Parent and Child class methods should be static.
2. Method resolution is always takes care by JVM based on runtime object.	2. Method resolution is always takes care by compiler based on reference type.
3. Overriding is also considered as runtime polymorphism (or) dynamic polymorphism (or) late binding.	3. Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early biding.

Coupling

The degree of dependency between the components is called coupling.

```
class A
{
 static int i=B.j;
```

```
}
class B extends A
{
 static int j=C.methodOne();
}
class C extends B
{
 public static int methodOne()
 {
 return D.k;
 }
}
class D extends C
{
 static int k=10;
 public static void main(String[] args)
 {
 D d=new D();
 }
}
```

- The above components are said to be tightly coupled to each other because the dependency between the components is more.
- Tightly coupling is not a good programming practice

Cohesion

- For every component we have to maintain a `clear well defined functionality` such type of component is said to be follow high cohesion.
- High cohesion is always good programming practice

Instance Initializer Block

- Rules for instance initializer block :

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Static Block -> Parent Constructor/ super() -> Instance Block -> Child constructor

- Static will be invoked only once for a class

```
class H{
    H(){
        System.out.println("2. Parent class constructor invoked");
    }
```

```
}

class B3 extends H{
    B3(){
        super();
        System.out.println("4. Child class constructor invoked");
    }

    B3(int a){
        super();
        System.out.println("4. Child class constructor invoked "+a);
    }
    static {
        System.out.println("1. Static block invoked");
    }
    {System.out.println("3. Instance initializer block invoked");}

    public static void main(String args[]){
        B3 b1=new B3();
        System.out.println();
        B3 b2=new B3(10);
    }
}

// 1. Static block invoked
// 2. Parent class constructor invoked
// 3. Instance initializer block invoked
// 4. Child class constructor invoked

// 2. Parent class constructor invoked
// 3. Instance initializer block invoked
// 4. Child class constructor invoked 10
```

Object Class

- Methods

1. `equals(Object ref)` returns true if both objects are equal
2. `finalize( )` method called when an object's memory is destroyed
3. `getClass( )` returns class to which the object belongs
4. `hashCode( )` returns the hashcode of the class object
5. `toString()` return the string equivalent of the object name
6. `notify( )` method to give message to a synchronized methods
7. `notifyAll( )` method to give message to all synchronized methods
8. `wait()` suspends a thread for a while
9. `wait(...)` suspends a thread for specified time in seconds

String Class

- String Literals is a reference to a String object.
- `immutable`
- String str = “Hello!!!!”;

StringBuilder is exactly same as StringBuffer(includung constructors and methods) except the following differences :

StringBuffer	StringBuilder
Every method present in StringBuffer is synchronized.	No method present in StringBuilder is synchronized.
At a time only one thread is allow to operate on the StringBuffer object hence StringBuffer object is Thread safe.	At a time Multiple Threads are allowed to operate simultaneously on the StringBuilder object hence StringBuilder is not Thread safe.
It increases waiting time of the Thread and hence relatively performance is low.	Threads are not required to wait and hence relatively performance is high.
Introduced in 1.0 version.	Introduced in 1.5 versions.

String vs StringBuffer vs StringBuilder :

1. **If the content is fixed and won't change frequently then we should go for String.**
2. **If the content will change frequently but Thread safety is required then we should go for StringBuffer.**
3. **If the content will change frequently and Thread safety is not required then we should go for StringBuilder.**

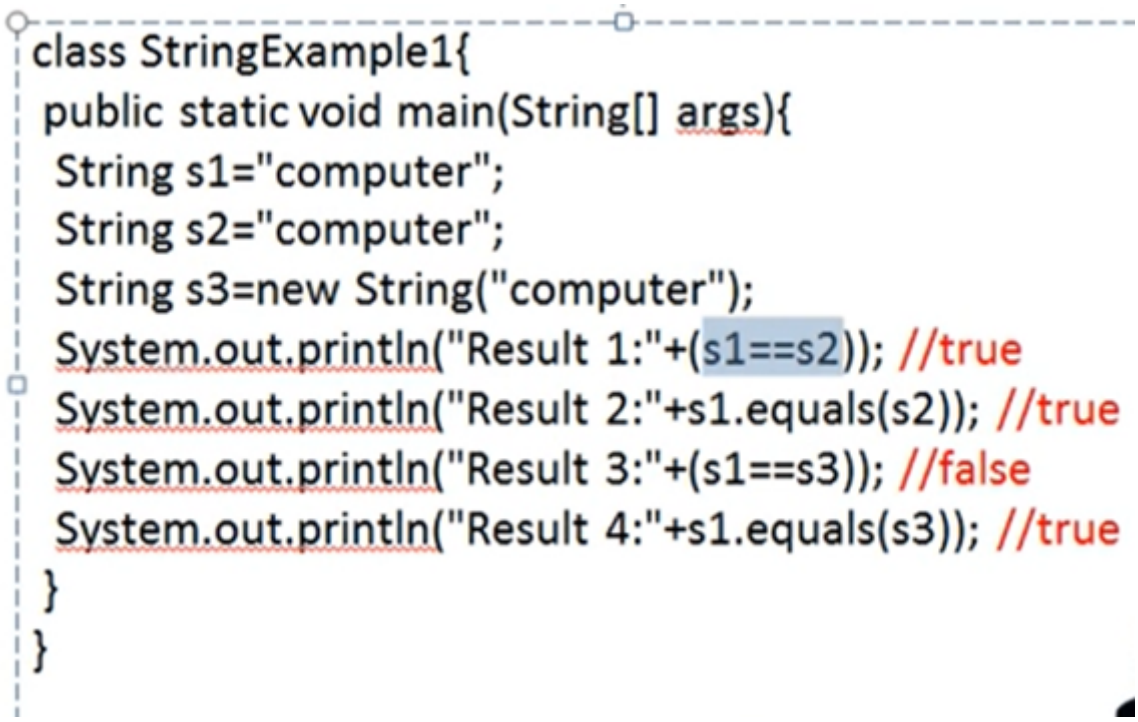
equals()

- Implements String object value equality as two String objects having the same sequence of characters. `boolean equals(Object obj)` `boolean equalsIgnoreCase(String str2)`
- In general we can use == (double equal operator) for reference comparison whereas .equals() method for content comparison.

==

- Compares string location

- by hashcode



```
class StringExample1{
 public static void main(String[] args){
  String s1="computer";
  String s2="computer";
  String s3=new String("computer");
  System.out.println("Result 1:"+(s1==s2)); //true
  System.out.println("Result 2:"+s1.equals(s2)); //true
  System.out.println("Result 3:"+(s1==s3)); //false
  System.out.println("Result 4:"+s1.equals(s3)); //true
 }
}
```

compareTo()

- Returns a value based on
- 0 : if current string is equal to str2
- Less than 0 : if current string is lexicographically less than str2
- Greater than 0 : if current string is lexicographically greater than to str2

StringBuffer Class

- `StringBuffer strBuf1 = new StringBuffer("Java")`
- implements `mutable character strings`.
- capacity of the string buffer can also change dynamically.
- The StringBuffer class provides various facilities for manipulating string buffers:
- constructing string buffers
- changing, deleting, and reading characters in string buffers
- constructing strings from string buffers
- appending, inserting, and deleting in string buffers
- controlling string buffer capacity
- Three constructors:

1. StringBuffer(String s) : Capacity: length of argument string + 16 more characters.
2. StringBuffer(int length) : Capacity: value of argument length
3. StringBuffer() : Capacity: `16 characters`

Enums

- Enum is a data type that contains fixed set of constants, that specifies the possible values to be used.

- The constants declared in enum are by default `static & final`.
- Declared outside class : default & public are valid access modifiers.
- Declared inside class : private, default, protected & public are valid access modifiers. Can also be declared as static.

Enum iteration using values() method

- values() method can be used to iterate over the enum & fetch the available values.
- This method returns array of type enum that holds all the values.
- Level levels[] = Level.values();

```
public enum Seasons{
SUMMER, WINTER, AUTUMN, FALL
}

Seasons season; //variable of type Seasons
Season season = Seasons.WINTER; //assigning value to enum variable

if(x==Seasons.WINTER){...}

switch(season){
    case WINTER :
    ...
}

Seasons season[] = Seasons.values();
for (Seasons x : season) {
    System.out.println(x);
}
```

- implicitly extend java.lang.Enum class.
- can not extend any class as implicitly extends java.lang.Enum class.
- can improve type check
- can be used in if statements & switch statements.
- can have fields, methods and constructors.
- may implement one or more interfaces.
- `can not be instantiated`, as constructor is private by default.

enumSet

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
    // Traversing elements
```

```
    Iterator<days> iter = set.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());
  }
}

// TUESDAY
// WEDNESDAY
```

enumMap

```
import java.util.*;
public class EnumMapExample {
   // create an enum
   public enum Days {
   Monday, Tuesday, Wednesday, Thursday
   };
   public static void main(String[] args) {
   //create and populate enum map
   EnumMap<Days, String> map = new EnumMap<Days, String>(Days.class);
   map.put(Days.Monday, "1");
   map.put(Days.Tuesday, "2");
   map.put(Days.Wednesday, "3");
   map.put(Days.Thursday, "4");
   // print the map
   for(Map.Entry m:map.entrySet()){
       System.out.println(m.getKey()+" "+m.getValue());
      }
   }
}

// Monday 1
// Tuesday 2
// Wednesday 3
// Thursday 4
```

5. Abstract

Abstract Classes

- An abstract class `cannot be instantiated`, i.e Object can't be made
- `We can create Reference variable of abstract class`
- Made by using abstract keyword in Java
- Abstract class in Cpp making - virtual keyword
- Example - `Person abstract class, Student and Faculty - SubClass`

- No need of Person object here, what we just need is common methods and properties of Student and Faculty at a single place to avoid repeatability.
- Used to declare common characteristics of sub classes
- Abstract class does not compulsorily have Abstract method
- `It can have constructor` - No object so why constructor? For instance variables. And this constructor can be called by concrete classes as super()

Abstract Methods

- Methods that contain `no implementation`
- If a class has any Abstract method, entire class will be made abstract class.
- If parent is abstract class, and subclass is not implementing abstract method, it has to be made abstract too and then its subclass has to implement.
- Declaration must end with semicolon rather than block.
- Example - Account has method calculateInterest() - Abstract class, Saving Acccount is now forced to override calculateInterest()

```
abstract class A{
    int x;
    abstract void func1();
    abstract void func2(int i);
    void func3(){
        ...
    }
}
abstract class B extends A{
    ....
}
class C extends A{
    abstract void func1(){
        ...
    }
    abstract void func2(int i){
        ...
    }
}
class D{
    public static void main(String[] args){
        A a = new A(); // NO
        B b = new B(); // NO
        C c = new C(); // YES
    }
}
```

6. Interface

- Keyword is interface
- Any `service requirement specification (srs)` is called an interface
- Example1: Sun people responsible to define JDBC API and database vendor will provide implementation for that.
- It `cannot be instantiated`
- `Method` declarations are implicitly - `public abstract`. `public static abstract` in java 8
- `Variable` are implicitly - `public static final`
- public: To make it available for every implementation class.
- static: Without existing object also we have to access this variable.
- final: Implementation class can access this value but cannot modify.
- static implemented method
- default implemented method
- `It does not have constructor` - No object so why constructor? Since static variables, so no role of object?
- Like pure virtual function in C++
- If parent is abstract class, and subclass is not implementing abstract method, it has to be made abstract too and then its subclass has to implement.
- We just know prototype.
- Example - Admission, Student, Employee - make function definitions as per need
- Multiple Inheritence possible - Implementing class would
- interface I3 extends I1, I2
- class B extends A implements I3, I4
- As every interface method is always public and abstract we can't use the following modifiers for interface methods. `Abstract NO Private, protected, final, static, synchronized, native, strictfp.`
- class-class : extends
- class-interface : implements
- interface-interface : extends

3 types of interfaces :

1. `Normal Interfaces`
2. `Functional Interfaces / SAM` - 1 abstract method and many default/static methods - Feature of lambda expression
3. `Marker Interface` - Empty interface - It delivers the run-time type information about an object. It is the reason that the JVM and compiler have additional information about an object. Ex - Serializable and Cloneable

- If an interface `doesn't contain any methods` and by implementing that interface if our objects will get `some ability` such type of interfaces are called Marker interface (or) Tag interface (or) Ability interface.
- Example:

1. Serializable
2. Cloneable
3. RandomAccess
4. SingleThreadModel - These are marked for some ability

```
interface A{
    int x;
    void func1();
    void func2(int i);
}
abstract class B implements A{
    ....
}
class C implements A{
    public void func1(){
        ...
    }
    public void func2(int i){
        ...
    }
    public void func3(int i){
        ...
    }
}
class D{
    public static void main(String[] args){
        A a = new A(); // NO - Can't create object of interface
        B b = new B(); // NO - Can't create object of interface
        C c = new C(); // YES
    }
}
```

Difference Between Abstract Class and Interface

1. Abstract can have any `access modifier` for members | Interface can have only public members.
2. Abstract class may or may not contain `abstract method` | Interface cannot have defined methods. (Changed now)
3. Abstract class can have `static or non-static` members | Interfaces can have only static member variables.
4. Abstract class can have `final or non-final` members | Interfaces can have only final member variables.
5. Abstract class can have `constructor` | Interface can't have constructor
6. Abstract class can have `0 to 100% abstraction` | Interface have `100% abstraction`
7. Abstract class doesn't support `multiple inheritance` | Interface support `multiple inheritance`

Differences between interface with default methods and abstract class

Eventhough we can add concrete methods in the form of default methods to the interface , it wont be equal to abstract class.

Interface with Default Methods	Abstract Class
Inside interface every variable is Always public static final and there is No chance of instance variables	Inside abstract class there may be a Chance of instance variables which Are required to the child class.
Interface never talks about state of Object.	Abstract class can talk about state of Object.
Inside interface we can't declare Constructors.	Inside abstract class we can declare Constructors.
Inside interface we can't declare Instance and static blocks.	Inside abstract class we can declare Instance and static blocks.
Functional interface with default Methods Can refer lambda expression.	Abstract class can't refer lambda Expressions.
Inside interface we can't override Object class methods.	Inside abstract class we can override Object class methods.

Interface with default method != abstract class

1. If we don't know anything about `implementation` just we have requirement specification then we should go for `interface`.
2. If we are talking about `implementation but not completely` (partial implementation) then we should go for `abstract class`.
3. If we are talking about `implementation completely and ready to provide service` then we should go for `concrete class`.

Any Way we can't Create Object for Abstract Class and Interface. But Abstract Class can contain Constructor but Interface doesn't Why?

- The Main Purpose of Constructor is to Perform Initialization of an Object i.e. to Provide Values for Instance Variables.
- Abstract Class can contain Instance Variables which are required for Child Class Object to Perform Initialization of these Instance Variables Constructor is required Inside Abstract Class.
- But Every Variable Present Inside Interface is Always public, static and final whether we are declaring OR Not and Every Interface Variable we should Perform Initialization at the Time of Declaration and Hence Inside Interface there is No Chance of existing Instance Variable.
- Due to this Initialization of Instance Variables Concept Not Applicable for Interfaces.
- Hence Constructor Concept Not required for Interface.

Adapter class:

- Adapter class is a simple java class that `implements an interface only with empty implementation for every method.`
- If we implement an interface directly for each and every method compulsory we should provide implementation whether it is required or not. This approach increases length of the code and reduces readability.
- We can resolve this problem by using adapter class.
- `Instead of implementing an interface if we can extend adapter class we have to` provide implementation only for required methods but not for all methods` of that interface.
- This approach decreases length of the code and improves readability.

Object Reference

- Object reference of interface can refer to any of its subclass type
- `Object Reference of Interface - Functions in parent class only can be called`

```
interface I1{
    void func1();
}
interface I2{
    void func2(int i);
}
class A implements I1, I2{
    public void func1(){
        ...
    }
    public void func2(int i){
        ...
    }
    public void func3(int i){
        ...
    }
}
class Example{
    public static void main(String[] args){
        // 1. Object Reference of class
        A obj1 = new A();
        obj1.f1(); // YES
        obj1.f2(); // YES
        obj1.f3(); // YES

        // 2. Object Reference of Interface - Functions in parent class only can
be called
        I1 obj2 = new A();
        obj2.f1(); // YES
        obj2.f2(); // NO - error
        obj2.f3(); // NO - error

        I1 obj3 = new A();
        obj3.f1(); // NO - error
        obj3.f2(); // YES
```

```
        obj3.f3(); // NO - error
    }
}
```

---

7. Input from User

- System.in responsible for *Keyboard -> Buffer*
- Scanner's method nextInt() *Buffer -> Variable*

Scanner Class

- Used to get data from Buffer
- We can read input from *System.in* object
- Scanner is *final class*, this cannot be extended
- Scanner class is a part of java.util package
- Picks from buffer till delimiter(space, tab) is found
- `Scanner sc = new Scanner(System.in)`
- `x = sc.nextInt()` - buffer data to int
- `x = sc.nextFloat()` etc.
- `x = sc.next()` - one word string `no space`
- `x = sc.next()` - one line words

```
import java.util.Scanner; // Import the Scanner class
public class Input {
    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in); // Create a Scanner object
        System.out.print("Enter a number - ");

        int num = sc.nextInt(); // Read user input
        System.out.println("You Entered - "+num);
        sc.close();
    }
}
```

`System.out.println()`

```
- System is a class
- PrintStream is a class
- out is an object in PrintStream class
- println is a method in PrintStream class
```

3 Input Ways - User Input using BufferReader and Scanner

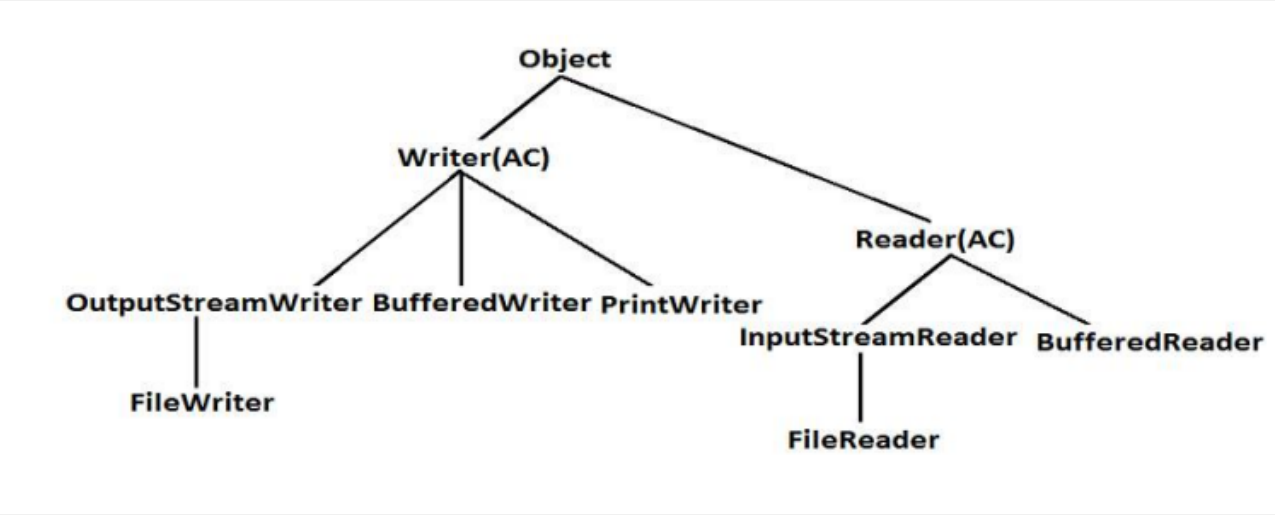
1. System.in.read()
    - reads ASCII | So do num-48 | But fails for big numbers
2. BufferReader
    - import java.io.BufferedReader;
    - import java.io.IOException;
    - import java.io.InputStreamReader;
    - InputStreamReader in = new InputStreamReader(System.in);
    - BufferedReader bf = new BufferedReader(in);
    - int num = Integer.parseInt(bf.readLine());
    - System.out.println(num);
3. Scanner
    - import java.util.Scanner; import java.io.IOException;
    - Scanner sc = new Scanner(System.in);
    - int num = sc.nextInt();
    - System.out.println(num);

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

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

    System.out.println("Enter a number");
//      1
    int num=System.in.read();
    System.out.println(num);
    System.out.println(num-48);
//      2
    InputStreamReader in=new InputStreamReader(System.in);
    BufferedReader bf=new BufferedReader(in);
    int num=Integer.parseInt(bf.readLine());
    System.out.println(num);
    BufferedReader bf=new BufferedReader(null);
    System.out.println(num-48);
//      3
    Scanner sc=new Scanner(System.in);
    int num=sc.nextInt();
    System.out.println(num);
    }
}
```

File IO Package



>

java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format()*, printf()* print(), println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()
			*Discussed later

1. File

- `File f=new File("abc.txt");`
- Checks whether abc.txt file is already available (or) not if it is already available then "f" simply refers that file.
- If it is not already available then it `won't create any physical file` just creates a java File object represents name of the file.
- File class constructors:

1. `File f=new File(String name);` Creates a java File object that represents name of the file or directory in current working directory.
2. `File f=new File(String subdirname,String name);` Creates a File object that represents name of the file or directory present in specified sub directory.
3. `File f=new File(File subdir,String name);`

- Create File :

1. `f.createNewFile();`
2. `f.mkdir();`

Methods of file class:

1. `boolean exists();`

- Returns true if the physical file or directory available.

2. `boolean createNewFile();`

- This method 1st checks whether the physical file is already available or not if it is already available then this method simply returns false without creating any physical file.
- If this file is not already available then it will create a new file and returns true

3. `boolean mkdir();`

- This method 1st checks whether the directory is already available or not if it is already available then this method simply returns false without creating any directory.
- If this directory is not already available then it will create a new directory and returns true

4. `boolean isFile();`

- Returns true if the File object represents a physical file.

5. `booleanisDirectory();`

- Returns true if the File object represents a directory.

6. `String[] list();`

- It returns the names of all files and subdirectories present in the specified directory.

7. `long length();`

- Returns the no of characters present in the file.

8. `boolean delete();`

- To delete a file or directory.

```
public static void main(String[] args)throws IOException {
    File f1=new File("SaiCharan123");
    f1.mkdir();
    File f2=new File("SaiCharan123","abc.txt");
    f2.createNewFile();
}
```

2. FileWriter

- By using FileWriter object we can write character data to the file.
- If the specified physical file is not already available then these constructors will create that file.
- Constructors:
- `FileWriter fw=new FileWriter(String name,boolean append);`
- `FileWriter fw=new FileWriter(File f,boolean append);`
- The main problem with FileWriter is we have to insert line separator manually , which is difficult to the programmer. ('\n')

Methods

1. `write(int ch);`

- To write a single character to the file.

2. `write(char[] ch);`

- To write an array of characters to the file.

3. `write(String s);`

- To write a String to the file.

4. `flush();`

- To give the guarantee the total data include last character also written to the file.

5. `close();`

- To close the stream.

```
public static void main(String[] args)throws IOException {
    FileWriterfw=new FileWriter("cricket.txt",true);
    fw.write(99);//adding a single character
    fw.write("haran\nsoftware solutions");
    fw.write("\n");
    char[] ch={'a','b','c'};
    fw.write(ch);
    fw.write("\n");
    fw.flush();
    fw.close();
}
```

3. FileReader

- By using FileReader object we can read character data from the file.
- Constructors:
- `FileReaderfr=new FileReader(String name);`
- FileReaderfr=new FileReader (File f); `

Methods:

1. `int read();`

- It attempts to read next character from the file and return its Unicode value. If the next character is not available then we will get -1.

2. `int i=fr.read();`
3. System.out.println(`(char)i`);

- As this method returns unicodevalue , while printing we have to perform type casting.

4. `int read(char[] ch);`

- It attempts to read enough characters from the file into char[] array and returns the no of characters copied from the file into char[] array.

5. `File f=new File("abc.txt");`
6. Char[] ch=new Char[(int)f.length()];
7. `void close();`

```
public static void main(String[] args)throws IOException {
    File f=new File("cricket.txt");
    FileReaderfr=new FileReader(f);
    char[] ch=new char[(int)f.length()]; //small
    amount of data
    fr.read(ch);
    for(char ch1:ch) {
        System.out.print(ch1);
    }
 }
```

- Usage of `FileWriter and FileReader is not recommended` because :

1. While writing data by FileWriter compulsory we should insert line separator(\n) manually which is a bigger headache to the programmer.
2. While reading data by FileReader we have to read character by character instead of line by line which is not convenient to the programmer.
3. To overcome these limitations we should go for BufferedWriter and BufferedReader concepts.

4. BufferedWriter

- By using BufferedWriter object we can write character data to the file.
- BufferedWriter never communicates directly with the file it should communicates via some writer object.
- Constructors:
- `BufferedWriter bw=new BufferedWriter(writer w);`
- `BufferedWriter bw=new BufferedWriter(writer w,intbuffersize);`
- When ever we are closing BufferedWriter automatically underlying writer will be closed and we are not required to close explicitly.

Methods:

1. `write(int ch);`
2. `write(char[] ch);`
3. `write(String s);`
4. `flush();`
5. `close();`
6. `newline();` : Inserting a new line character to the file

```
public static void main(String[] args)throws IOException {
    FileWriterfw=new FileWriter("cricket.txt");
    BufferedWriterbw=new BufferedWriter(fw);
    bw.write(100);
    bw.newLine();
    char[] ch={'a','b','c','d'};
    bw.write(ch);
    bw.newLine();
    bw.write("SaiCharan");
    bw.newLine();
    bw.write("software solutions");
    bw.flush();
    bw.close();
}
```

5. BufferedReader:

- This is the most enhanced(better) Reader to read character data from the file.
- Constructors:
- `BufferedReaderbr=new BufferedReader(Reader r);`
- `BufferedReaderbr=new BufferedReader(Reader r,intbuffersize);`
- BufferedReadercan not communicate directly with the File it should communicate via some Reader object.
- The main advantage of BufferedReader over FileReader is we can read data line by line instead of character by character.
- Whenever we are closing BufferedReader automatically underlying FileReader will be closed it is not required to close explicitly.

Methods:

1. `int read();`
2. `int read(char[] ch);`
3. `String readLine();` :It attempts to read next line and return it , from the File. if the next line is not available then this method returns null.
4. `void close();`

```
public static void main(String[] args)throws IOException {
 FileReaderfr=new FileReader("cricket.txt");
```

```
    BufferedReaderbr=new BufferedReader(fr);
    String line=br.readLine();
    while(line!=null) {
        System.out.println(line);
        line=br.readLine();
    }
    br.close();
}
```

6. PrintWriter:

- This is the most enhanced Writer to write text data to the file.
- By using FileWriter and BufferedWriter we can write only character data to the File but by using PrintWriter we can write any type of data to the File.
- Constructors:
- `PrintWriter pw=new PrintWriter(String name);`
- `PrintWriter pw=new PrintWriter(File f);`
- `PrintWriter pw=new PrintWriter(Writer w);`
- PrintWriter can communicate either directly to the File or via some Writer object also.

Methods:

1. write(intch);
2. write (char[] ch);
3. write(String s);
4. flush();
5. close();
6. print(char ch);
7. print (int i);
8. print (double d);
9. print (boolean b);
10. print (String s);
11. println(char ch);
12. println (int i);
13. println(double d);
14. println(boolean b);
15. println(String s);

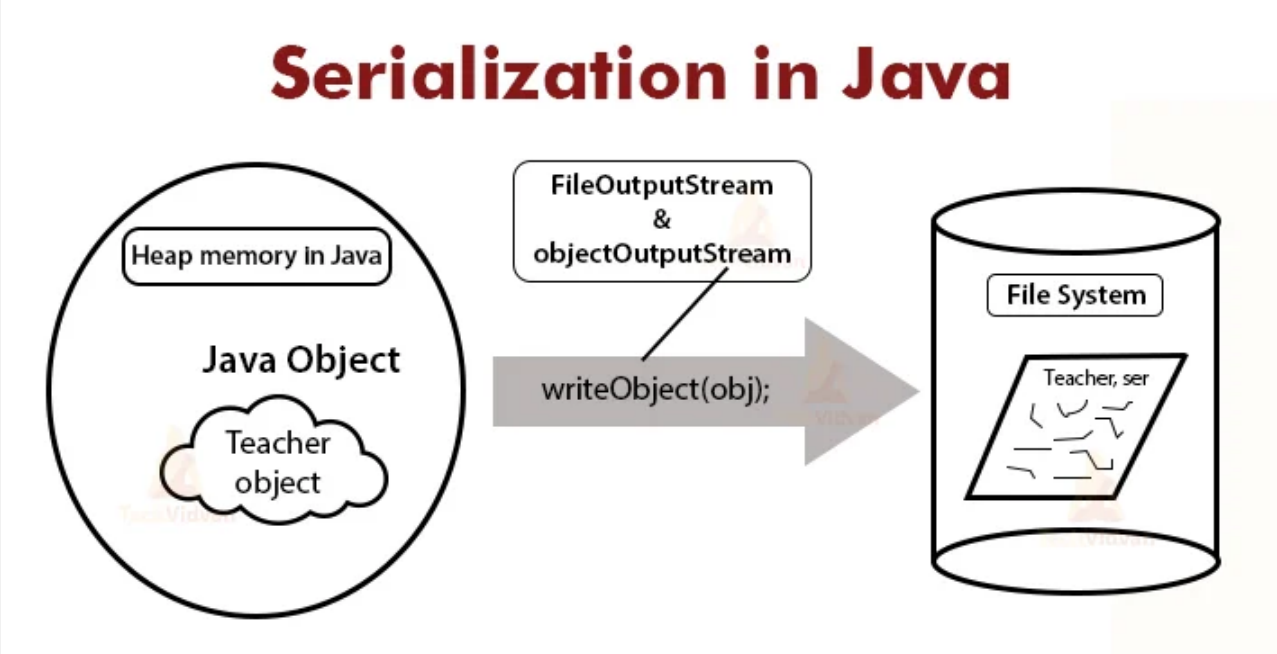
```
public static void main(String[] args)throws IOException {
    FileWriterfw=new FileWriter("cricket.txt");
    PrintWriter out=new PrintWriter(fw);
    out.write(100);
    out.println(100);
    out.println(true);
    out.println('c');
    out.println("SaiCharan");
    out.flush();
```

```
        out.close();
    }
```

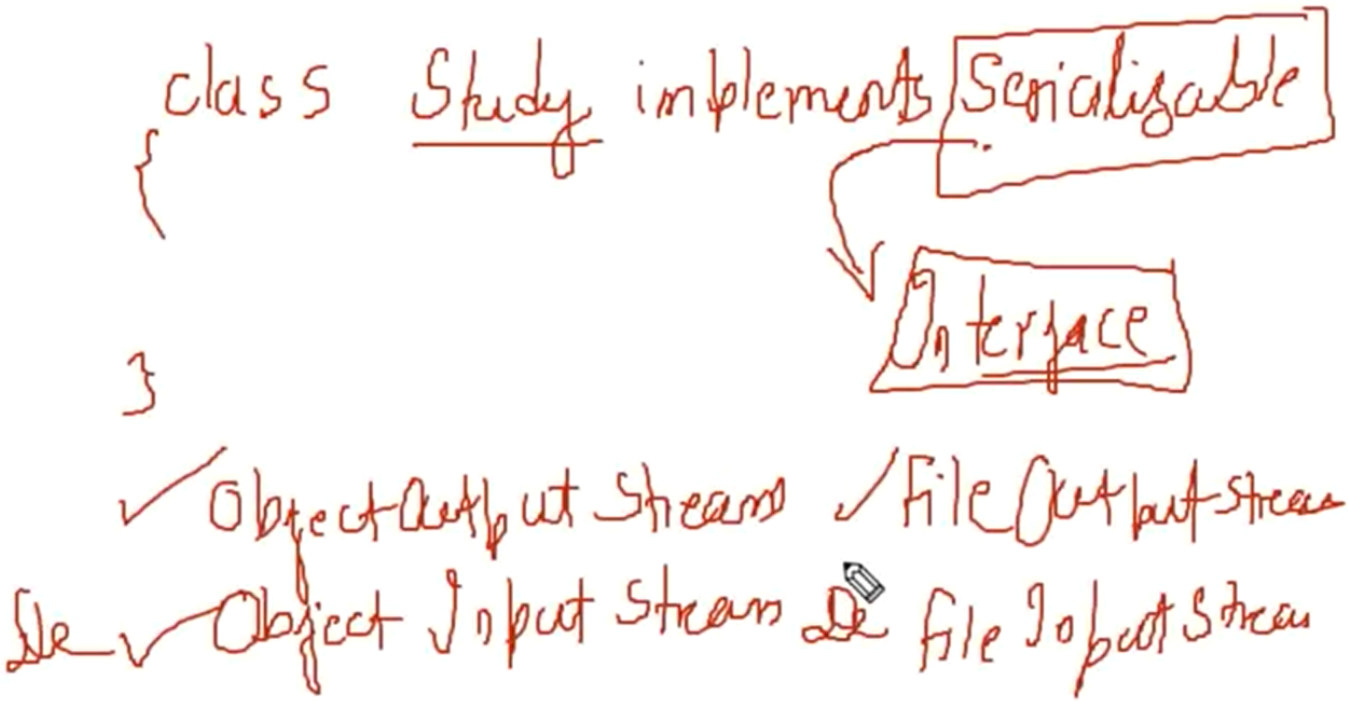
write(100) v/s print(100)

- In the case of write(100) the corresponding character "d" will be added to the File but in the case of print(100) "100" value will be added directly to the File.
- The most enhanced Reader to read character data from the File is BufferedReader.
- The most enhanced Writer to write character data to the File is PrintWriter.
- We can use OutputStream to write binary data to the File and we can use InputStream to read binary data from the File.

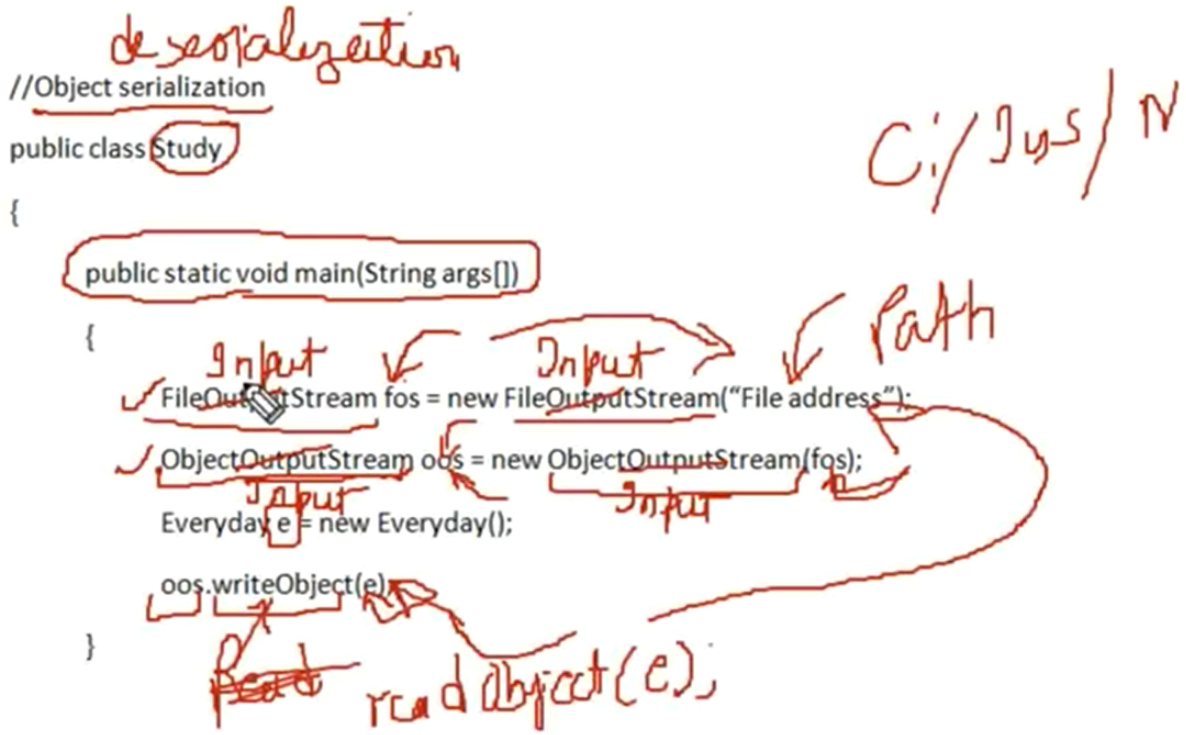
Serialization



- `Serialization` : Java obj -> stream
- It convert an Object to stream that we can send over the network or save it as file or store in DB for later usage.
- Serialization is the process of converting the state of an object into a form that can be persisted or transported.
- `Deserialization` : Object stream -> Java Object
- It is the process of converting Object stream to actual Java Object to be used in our program.




```
//Object serialization
public class Study
{
    public static void main(String args[])
    {
        FileOutputStream fos = new FileOutputStream("File address");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        Everyday e = new Everyday();
        oos.writeObject(e);
    }
```



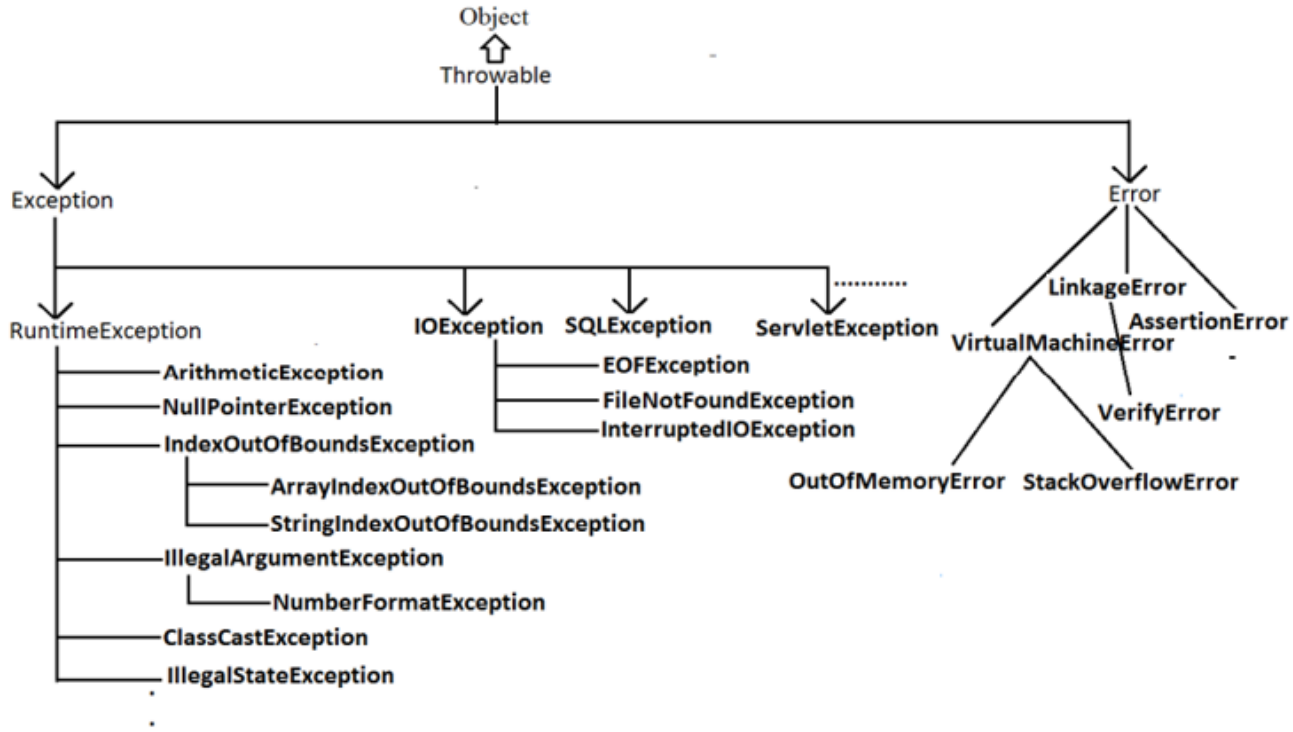
Exception Handling

Types of error:-

1. Complile - time error
2. Runtime error -> Exception handling
3. Logical error

Exception Hierarchy

Diagram:



1. Object
2. Throwable
3. Error
    1. Thread Death
    2. Virtual Machine error (Out of memory)
    3. IO Error
4. Exception
    1. Runtime Exception (Unchecked Exception)
    2. SQL Exception (Checked Exception)
    3. IO Exception (Checked Exception)

Error is divided into :-

1. Thread Death
2. Virtual Machine error (Out of memory)
3. IO Error

Exception is divided into:-

- Unchecked Exception - It is your choose to handle or not
- `Checked Exception - Mandatory to handle`

1. Runtime Exception (Unchecked Exception)

- It is your choose to handle or not
    - Arithmetic

- ArrayIndexOutOfBound
- Null POinter

2. SQL Exception (Checked Exception)

- It is necessary to handle

3. IO Exception (Checked Exception)

- It is necessary to handle

Compile Time Error

```
public class Demo {
    public static void main(String[] args) {
    System.out.Println();
    }
}
```

Logical Error

```
public class Demo {
    public static void main(String[] args) {
    System.out.println(2+2); /// 4 Needed 5
    }
}
```

Exception handling keywords summary:

1. **try: To maintain risky code.**
2. **catch: To maintain handling code.**
3. **finally: To maintain cleanup code.**
4. **throw: To handover our created exception object to the JVM manually.**
5. **throws: To delegate responsibility of exception handling to the caller method.**

try with multiple catch block

- `Parent Exception should be last`

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

        int i=2;
  //    int i=0;
        int j=0;
```

```
        int nums[]=new int[5];
        String str=null;

        try
        {
            j=18/i; // ArithmeticException
            System.out.println(str.length());
            System.out.println(nums[1]);
            System.out.println(nums[5]); // ArrayIndexOutOfBoundsException
        }
//      catch(Exception e)
//      {
//          System.out.println("Something went wrong."+e);
//      }
        catch(ArithmeticException e)
        {
            System.out.println("Cannot divide by zero");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Stay in your limit.");
        }
        catch(Exception e)
        {
            System.out.println("Something went wrong."+e);
        }
        System.out.println(j);
        System.out.println("Bye");
    }
}
```

throw keyword

- `throw new ArithmeticException("message");`
- throw keyword is used `throw an exception explicitly` in the code, `inside the function or the block of code`.

```
public class Demo {
    public static void main(String[] args) {
//      int i=2;
        int i=0;
        int j=0;

        try{
            j=18/i;
            if(j==0)
                throw new ArithmeticException("I don't want to do print zero");
        }
```

```
        catch(ArithmeticException e){
            j=18/i;
            System.out.println("that is default output"+e);
//          System.out.println("Cannot divide by zero");
        }

        catch(Exception e){
            System.out.println("Something went wrong."+e);
        }
        System.out.println(j);
        System.out.println("Bye");
    }
}
```

Custom Exception

```
// Syntax
class AyushiException extends Exception{
    public AyushiException(String string){
        super(string);
    }
}

throw new AyushiException("I don't want to do print zero");
```

```
class AyushiException extends Exception{
    public AyushiException(String string){
        super(string);
    }
}
public class Demo {
    public static void main(String[] args) {
//      int i=2;
//      int i=0;
        int i=20;
        int j=0;
        try{
            j=18/i;
            if(j==0)
//              throw new Exception("I don't want to do print zero");
                throw new AyushiException("I don't want to do print zero");
        }
        catch(AyushiException e){
            System.out.println("Custom Exception...."+e);
        }
        catch(ArithmeticException e){
            j=18/i;
            System.out.println("that is default output"+e);
```

```
//          System.out.println("Cannot divide by zero");
        }
        catch(Exception e){
            System.out.println("Something went wrong."+e);
        }
        System.out.println(j);
        System.out.println("Bye");
    }
}
```

throws - Ducking Exception using throws

- Someone will have to handle the exception
- `If you just do throws Exception, parent or parent of parent will ahve to try catch ie handle it`
- throws - public void show() throws ClassNotFoundException or do try-catch
- throws keyword is `used in the method signature` to declare an `exception which might be thrown by the function` while the execution of the code.

```
class AyushiException extends Exception{
    public AyushiException(String string){
        super(string);
    }
}

class A{
    public void show() throws ClassNotFoundException{
//      try{
//          Class.forName("Calc");
//      }
//      catch(ClassNotFoundException e){
//          System.out.println("Not able to find theh class");
//      }
        Class.forName("Calc"); // throws for this
    }
}

public class Demo {
    static {
        System.out.println("Class Loader");
    }
    public static void main(String[] args) {

//      try{
//          Class.forName("Class");
//      }
//      catch(ClassNotFoundException e){
//          System.out.println("Not able to find theh class");
//      }
```

```
        A obj=new A();
        try {
            obj.show();
            throw new AyushiException("I don't want to do print zero");
        }
        catch(ClassNotFoundException e){
            e.printStackTrace();
        }
    }
}
```

try with resources

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

public class Demo {
    public static void main(String[] args) throws NumberFormatException {

        int i=0;
        int j=0;
        try{
            j=18/i;
        }
        catch(Exception e){
            System.out.println("Someting went wrong.");
            System.out.println("Bye");
        }
        finally{
            System.out.println("Bye");
        }

        int num=0;
        //BufferedReader br=null;
        try(BufferedReader br=new BufferedReader(new
InputStreamReader(System.in))){
//          InputStreamReader in =new InputStreamReader(System.in);
//          BufferedReader br=new BufferedReader(in);
            num=Integer.parseInt(br.readLine());
            System.out.println(num);
        }
        finally{
            //br.close();
}}}
```

Various possible compile time errors in exception handling:

1. **Exception XXX has already been caught.**
2. **Unreported exception XXX must be caught or declared to be thrown.**
3. **Exception XXX is never thrown in body of corresponding try statement.**
4. **Try without catch or finally.**
5. **Catch without try.**
6. **Finally without try.**
7. **Incompatible types.**

```
found:Test
    requried:java.lang.Throwable;
```

8. **Unreachable statement.**

Exception/Error	Raised by
1. AIOOBE 2. NPE(NullPointerException) 3. StackOverFlowError 4. NoClassDefFoundError 5. CCE(ClassCastException)	Raised automatically by JVM(JVM Exceptions)
6. ExceptionInInitializerError	
1. IAE(IllegalArgumentException) 2. NFE(NumberFormatException) 3. ISE(IllegalStateException) 4. AE(AssertionError)	Raised explicitly either by programmer or by API developer (Programatic Exceptions).

Final TryCatchFinallyThrowThrows

```
class AyushiException extends RuntimeException{
    public AyushiException(String string){
        super(string);
    }
}
public class TryCatchFinallyThrowThrows throws ClassNotFoundException{
    public static void main(String[] args) {
        // 0 - no exp
        // int i=2;
        // int j=20;
        // 1 - /0 exp
        // int i=0;
        // int j=10;
        // 2 j-0 throw exp - not working
        int i=2;
         int j=0;
        try{
```

```
            // Class.forName("Calc");
            // j=18/i;
            if(j==0)
                throw new AyushiException("I don't want to do print zero");
        }

        catch(ArithmeticException e){
            // j=18/i;
            // System.out.println("that is default output"+e);
            System.out.println("Cannot divide by zero...."+e);
        }
        catch(AyushiException e){
            System.out.println("Custom Exception...."+e);
        }
        catch(Exception e){
            System.out.println("Something went wrong."+e);
        }
        finally{
        System.out.println(j);
        System.out.println("Bye");
        }
    }
}
```

MultiThreading

- `Multitasking`
- Executing several tasks simultaneously.
- `Process based multitasking`
- Executing several tasks simultaneously where each task is a separate independent process such type of multitasking.
- Best suitable at "os level"
- `Thread based multitasking`
- Executing several tasks simultaneously where each task is a separate independent part of the same program. Each independent part is called a "`Thread`".
- Best suitable for "programatic level".
- `Thread`
- A thread is a lightweight sub-process, the smallest unit of processing.
- We use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory and context-switching between the threads takes less time than process.

The ways to define instantiate and start a new Thread

1. By extending Thread class.

```
class MyThread extends Thread {
    public void run(){
        for(int i=0; i<10; i++>){
            SOUT("Child Thread");
        }
    }
}

class Demo{
    public static void main(String[] args){
        // 1
        MyThread t = new MyThread(); // Instantiation of thread
        // 2
        t.start(); // Starting of thread
        ...
    }
}
```

2. By implementing Runnable interface.

- Runnable has only run()
- Best Approach
- Provides Inheritence feature

```
class MyRunnable extends Runnable {
    public void run(){
        for(int i=0; i<10; i++>){
            SOUT("Child Thread");
        }
    }
}

class Demo{
    public static void main(String[] args){
        // 1
        MyRunnable r = new MyRunnable();
        // 2
        Thread t = new MyThread(r);
        // 3
        t.start(); // Starting of thread
        ...
    }
}
```

- If multiple Threads are waiting to execute then which Thread will execute 1st is decided by "`Thread Scheduler`" which is part of JVM, which algorithm or behavior followed by Thread Scheduler we can't expect exactly it is `JVM vendor dependent.`
- Thread class constructors:

1. Thread t=new Thread();
2. Thread t=new Thread(Runnable r);
3. Thread t=new Thread(String name);
4. Thread t=new Thread(Runnable r,String name);
5. Thread t=new Thread(ThreadGroupg,String name);
6. Thread t=new Thread(ThreadGroupg,Runnable r);
7. Thread t=new Thread(ThreadGroupg,Runnabler,String name);
8. Thread t=new Thread(ThreadGroupg,Runnabler,Stringname,longstackSize);

t.start() V/S t.run()

- `t.start()` : a `new Thread will be created` which is responsible for the execution of run() method.

1. Register Thread with Thread Scheduler
2. All other mandatory low level activities.
3. Invoke or calling run() method.

- Never recommended to override start() method. If we override start() method then our start() method will be executed just like a normal method call and no new Thread will be started.
- `t.run()` : no new Thread will be created and run() method will be executed just like a `normal method` by the main Thread.
- We can overload run() method but Thread class start() method always invokes `no argument run()` method the other overload run() methods we have to call explicitly.

Play with run() and start()

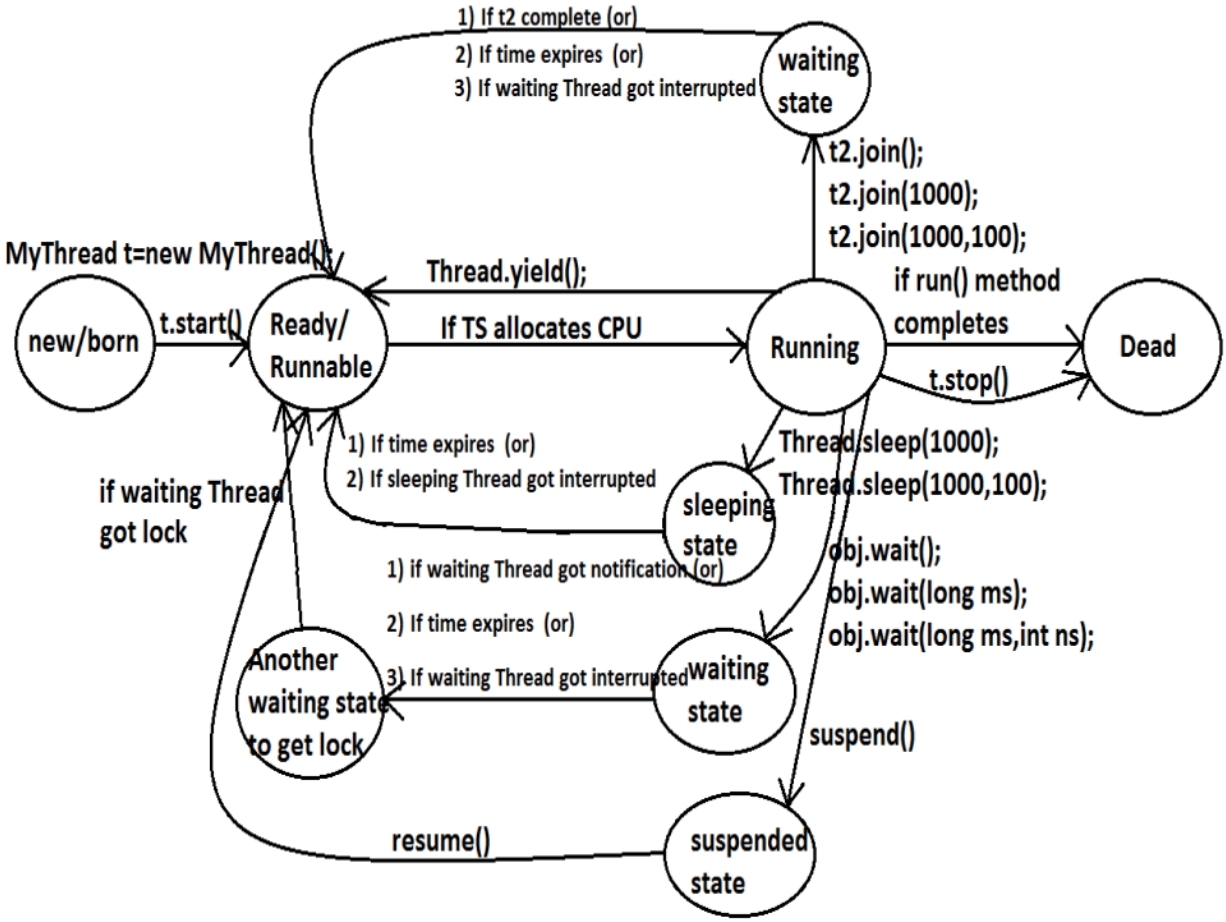
```
MyRunnable r=new MyRunnable();
Thread t1=new Thread();
Thread t2=new Thread(r);
```

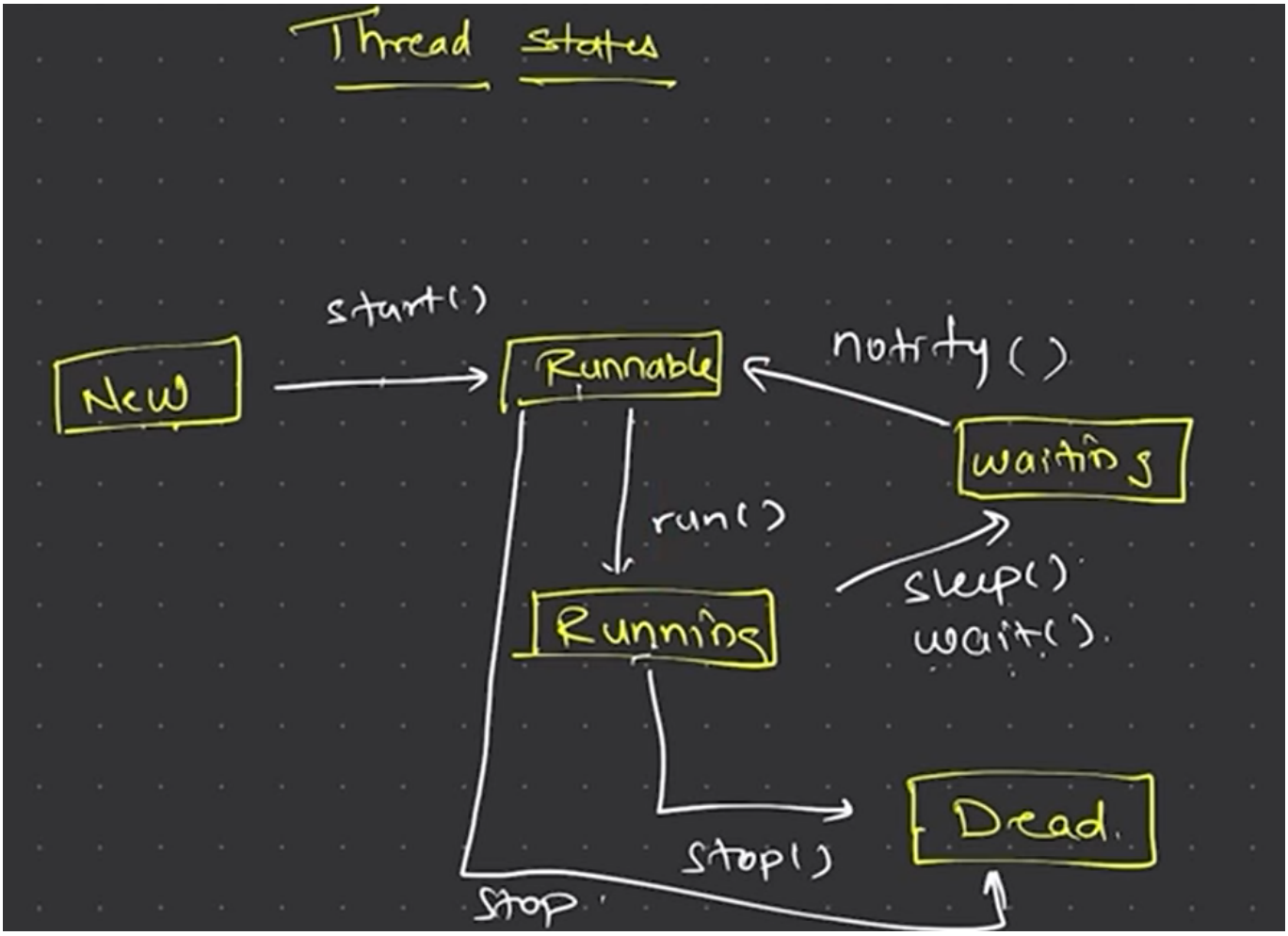
1. t1.start() : New thread created + Executes Thread-run()
2. t1.run() : No thread creation + Executes Thread-run() like normal method call
3. t2.start() : New thread created + Executes MyRunnable-run()
4. t2.run() : No thread creation + Executes MyRunnable-run() like normal method call
5. r.start() : Compile Time Error : `No start() in MyRunnable class`
6. r.run() : No thread creation + Executes MyRunnable-run() like normal method call

Life Cycle of Thread

- `Thread.State getState( )`

1. New State : Craete Thread Object
2. Ready/ Runnable : Call start()
3. Running : Thread Scheduler allocated CPU
4. Waiting/Sleeping State : join() / sleep()
5. Dead : run() completes


1) If t2 complete (or)
2) If time expires (or)
3) If waiting Thread got interrupted
waiting state
t2.join();
t2.join(1000);
t2.join(1000,100);
MyThread t=new MyThread();
Thread.yield();
if run() method completes
new/born
t.start()
Ready/ Runnable
If TS allocates CPU
Running
Dead
t.stop()
1) If time expires (or)
2) If sleeping Thread got interrupted
Thread.sleep(1000);
Thread.sleep(1000,100);
if waiting Thread got lock
sleeping state
obj.wait();
obj.wait(long ms);
obj.wait(long ms,int ns);
1) if waiting Thread got notification (or)
2) If time expires (or)
3) If waiting Thread got interrupted
Another waiting state to get lock
waiting state
suspend()
resume()
suspended state


Thread States
start()
New
Runnable
notify()
waiting
run()
Running
sleep()
wait()
Dead
stop()
Stop

Thread Priorities

- Default priority(5) generated by JVM or explicitly provided by the programmer.
- Priority Range : 1 to 10 (1 lowest)
- Default priority will be inheriting from parent to child.
- Standard thread priorities constants :

1. Thread. MIN_PRIORITY : 1
2. Thread. MAX_PRIORITY : 10
3. Thread. NORM_PRIORITY : 5

Get and Set Thread Priorities

1. public final int getPriority()
2. public final void setPriority(int newPriority) : the allowed values are 1 to 10

Get and Set Thread Name

1. public final String getName()
2. public final void setName(String name)

```
System.out.println(Thread.currentThread().getName()) // main
System.out.println(t.getName()); //Thread-0

Thread.currentThread().setName("Ayushi Thread");
```

Illegal Exceptions

1. `IllegalThreadStateException`

- After starting a Thread we are `not allowed to restart the same Thread` once again otherwise we will get runtime exception.

2. `IllegalArgumentException`

- The allowed values are 1 to 10 otherwise we will get runtime exception.

3. `IllegalMonitorStateException`

- Calling wait(), notify() and notifyAll() methods apart from synchronized area

4. `IllegalThreadStateException`

- If after starting the Thread, we are trying to change the daemon nature.s

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalCallerException	A method cannot be legally executed by the calling code.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
LayerInstantiationException	A module layer cannot be created.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Methods to Prevent Thread from Execution

1. yield()
2. join()
3. sleep()

yield()

- To pause current executing Thread for giving the `chance of remaining waiting Threads` of same priority.
- If all waiting threads are low priority OR No waiting thread : Same thread execution continues
- If many same priority waiting threads : Can't expect order of execution
- Current thread will be re run when again given chance by Thread Scheduler
- Thread comes from Running to Runnable state.

1. public static native void yield();

join()

- waits until the thread on which it is called terminates
- If a Thread `wants to wait` until completing some other Thread.
- Thread comes from Running to Waiting state.
- If both parent and child thread calls join() : Deadlock
- throws InterruptedException

1. public final void join() throws InterruptedException
2. public final void join(long ms) throws InterruptedException
3. public final void join(long ms,int ns) throws InterruptedException

sleep()

- If a Thread don't want to perform any operation for a `particular amount of time`.
- throws InterruptedException

1. public static native void sleep(long ms) throws InterruptedException
2. public static void sleep(long ms,int ns) throws InterruptedException

Compression of yield, join and sleep() method?

property	Yield()	Join()	Sleep()
1) Purpose?	To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.	If a Thread wants to wait until completing some other Thread then we should go for join.	If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
2) Is it static?	yes	no	yes
3) Is it final?	no	yes	no
4) Is it overloaded?	No	yes	yes
5) Is it throws InterruptedException?	no	yes	yes
6) Is it native method?	yes	no	sleep(long ms) -->native sleep(long ms,int ns) -->non-native

Interrupting a Thread

- If a Thread can `interrupt a sleeping or waiting Thread` by using interrupt()(break off) method of Thread class.
- Only applies when thread is in sleep or waiting state : so we may not see effect immediately
- Whenever we are calling interrupt() method we may not see the effect immediately, if the target Thread is in sleeping or waiting state it will be interrupted immediately.

- If the target Thread is not in sleeping or waiting state then interrupt call will wait until target Thread will enter into sleeping or waiting state. Once target Thread entered into sleeping or waiting state it will effect immediately.
- In its lifetime if the target Thread never entered into sleeping or waiting state then there is no impact of interrupt call simply interrupt call will be wasted.

Race Condition

- Executing multiple Threads simultaneously and causing data inconsistency problems is nothing but Race condition
- Can resolve race condition by using synchronized keyword.

Synchronization

- Only for methods and blocks
- If a method or block declared as the synchronized then `at a time only one Thread is allow to execute` that method or block on the given object.
- Advantage : Resolve Data Inconsistency Problems.
- Disadvantage : Increases waiting time of the Thread and effects performance of the system.
- Internally synchronization concept is implemented by using lock concept. Every object has a unique lock.
- If a Thread wants to execute any synchronized method on the given object : `Get the lock` of that object. Once a Thread got the lock of that object then it's allow to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.
- Remaining Threads are allowed to execute any non-synchronized method simultaneously
- `lock concept is implemented based on object but not based on method`
- If multiple threads are operating on multiple objects then there is no impact of Syncronization.
- Class level lock - static synchronized method
- Synchronized Block

```
// 1 : Synchronized method
synchronized void func(){
    ...
}
// 2 : Synchronized block
{
    synchronized(lock) { // lock = object reference expression
    ...
    }
}
//3 : Static synchronized method
static synchronized void func(int n){
    ...
}
// 4 : Static synchronized block
class A{
    static void func(int n) {
```

```
        synchronized (lock) { // lock = A.class
            ...
        }
    }
}
```

- lock concept is dependent only for objects and classes but not for primitives

Inter Thread Communication

1. wait() - Running -> Waiting after releasing lock
2. notify() - give the notification for 1 waiting Threads
3. notifyAll() - give the notification for all waiting Threads

Except these (wait(),notify(),notifyAll()) methods there is no other place(method) where the lock release will be happen.

Method	Is Thread Releases Lock?
yield()	No
join()	No
sleep()	No
wait()	Yes
notify()	Yes
notifyAll()	Yes

- wait(), notify() and notifyAll() methods are available in Object class but not in Thread class
- *Once a Thread calls wait() on any Object it immediately releases the lock of that particular Object and entered into waiting state*
- *Once a Thread calls notify() on any Object it releases the lock of that Object but may not immediately*
- To call wait(), notify() and notifyAll() methods compulsory :
    1. Current Thread should be owner of that object
    2. Current Thread should has lock of that object
    3. Current Thread should be in synchronized area.
- Can call these only from synchronized area else `IllegalMonitorStateException`
- `Lock release happen in only these 3` that too on that particular object not all.

```
Stack s1 = new Stack()
Stack s2 = new Stack()

synchronized(s1){
    s1.wait() // Valid
```

```
    s2.wait() // Invalid
}
```

1. public final void wait() throws InterruptedException
2. public final native void wait(long ms) throws InterruptedException
3. public final void wait(long ms,int ns) throws InterruptedException
4. public final native void notify()
5. public final void notifyAll()

```
class ThreadA {
    public static void main(String[] args)throws InterruptedException {
        ThreadB b=new ThreadB();
        b.start();
        synchronized(b){
            System.out.println("main Thread calling wait() method");//step-1
            b.wait();
            System.out.println("main Thread got notification call");//step-4
            System.out.println(b.total);
        }
    }
}
class ThreadB extends Thread {
    int total=0;
    public void run() {
        synchronized(this) {
            System.out.println("child thread starts calcuation");//step-2
            for(int i=0;i<=100;i++) {
                total=total+i;
            }
            System.out.println("child thread giving notification call"); //step-3
            this.notify();
        }
    }
}

Output:
main Thread calling wait() method
child thread starts calculation
child thread giving notification call
main Thread got notification call
5050
```

Producer - Consumer

```
import java.util.LinkedList;
import java.util.*;
```

```
public class InterThreadCommunicationPractice {
    public static void main(String[] args) {
        String key = "LOCK";
        Queue<Integer> queue = new LinkedList<>(); // Plate
        int plateSize = 5;

        Thread producer = new Thread(new Runnable() {

            public void run() {
                int count = 0;
                while(true){
                    synchronized (key){
                        try{
                            while(queue.size()==plateSize)
                                key.wait(); // producer thread should stop if 5
are produced
                            queue.offer(count++);
                            key.notifyAll();
                            Thread.sleep(1000);
                            System.out.println("Produced : "+queue.size());
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }

                    }
                }
            }
        });

        Thread consumer = new Thread(new Runnable() {
            public void run() {
                while(true){
                    synchronized (key){
                        try{
                            while(queue.size()==0)
                                key.wait(); // If no momo consumer thread should
wait
                            queue.poll();
                            key.notifyAll();
                            System.out.println("Consumed : "+queue.size());
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }

                    }
                }
            }
        });
        producer.start();
        consumer.start();
    }
}
```

- With Blocking Queue :

```
BlockingQueue<Integer> q = new ArrayBlockingQueue<>(10);
In consumer : q.take()
In producer : q.put(i++)
```

Deadlock:

- A long waiting of a Thread which never ends
- If 2 Threads are waiting for each other forever(without end) such type of situation(infinite waiting) is called dead lock.
- There are no resolution techniques for dead lock but several prevention(avoidance) techniques are possible.
- `Synchronized keyword is the cause for deadlock` hence whenever we are using synchronized keyword we have to take special care.

```
public class DeadlockPractice {
    public static void main(String[] args) {
        final String lock1="A";
        final String lock2="B";
        Thread t1 = new Thread(new Runnable(){
            @Override
            public void run() {
                synchronized (lock1){
                    System.out.println("Inside Thread1");
                    synchronized (lock2){
                        System.out.println("Lock1 executed properly");
                    }
                }
            }
        });
        Thread t2 = new Thread(new Runnable(){
            @Override
            public void run() {
                synchronized (lock2){
                    System.out.println("Inside Thread2");
                    synchronized (lock1){
                        System.out.println("Lock2 executed properly");
                    }
                }
            }
        });
        t1.start();
        t2.start();
        System.out.println("End");
    }
}
```

```
End
Inside Thread1
Inside Thread2
Hang.....
```

Starvation

- A long waiting of a Thread which ends at certain point.
- A low priority Thread has to wait until completing all high priority Threads.

Daemon Threads:

- The Threads which are `executing in the background` are called daemon Threads Provides support for non-daemon Threads like main Thread. Ex - Garbage Collector
- We can change daemon nature `before starting` Thread only. That is after starting the Thread if we are trying to change the daemon nature we will get R.E saying IllegalThreadStateException. (main thread always non-daemon as it always starts in beginning)
- Nature gets inherited from parent
- Last non-daemon ends -> All daemons end

1. public final boolean isDaemon() - If thread is daemon or not
2. public final void setDaemon(boolean b) - Change thread nature

Lazy Thread

- Last non-daemon ends -> All daemons end
- If parent is non-daemon and child is daemon - If parent ends : child also ends

```
class MyThread1 extends Thread{
    public void run() {
        for(int i=0;i<10;i++) {
            System.out.println("Lazy thread");
            try{
                Thread.sleep(2000);
            }
            catch(InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

public class DaemonThreadPractice {
    public static void main(String[] args)
    {
        MyThread1 t=new MyThread1();
//        t.setDaemon(true); //-->1
```

```
        t.start();
        System.out.println("end of main Thread");
    }
}

// Uncomment
//  end of main Thread
//  lazy thread

// Comment
//  end of main Thread
//  lazy thread
//  lazy thread
//  lazy thread
```

Green Thread - deprecated

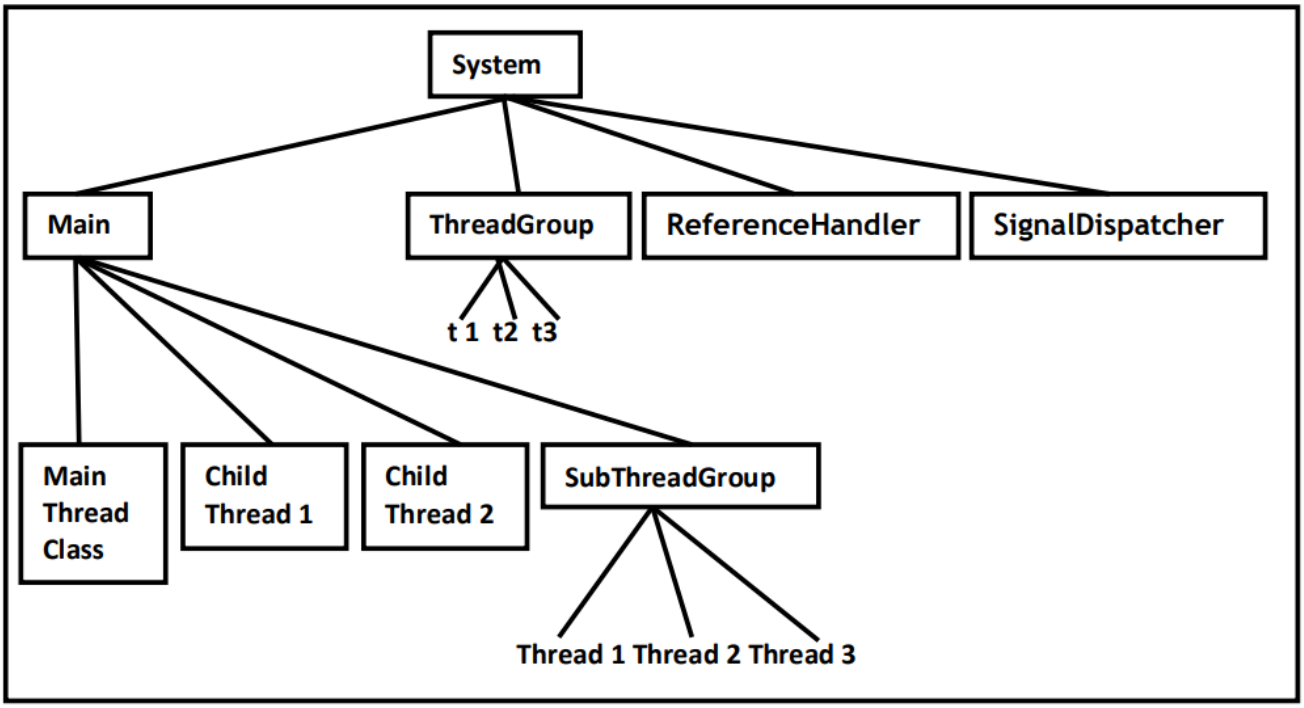
- Threads which are managed completely by JVM without taking support for underlying OS
- Only JVM, Without OS

Native OS

- The Threads which are managed with the help of underlying OS.

ThreadGroup

- Set of Threads
- Based on functionality we can group threads as a single unit.
- `System Group` Acts as Root for all ThreadGroup's in Java



- Create ThreadGroup : `ThreadGroup g = new ThreadGroup(String gName);`
- Attach a thread to ThreadGroup : `Thread t=new Thread(ThreadGroup g, String name);`

```
ThreadGroup g=new ThreadGroup("Printing Threads");

MyThread t1=new MyThread(g,"Header Printing");
MyThread t2=new MyThread(g,"Footer Printing");
MyThread t3=new MyThread(g,"Body Printing");


classThreadGroupDemo {
public static void main(String[] args) {
    System.out.println(Thread.currentThread().getThreadGroup().getName());

System.out.println(Thread.currentThread().getThreadGroup().getParent().getName());
    ThreadGroup pg = new ThreadGroup("Parent Group");
    System.out.println(pg.getParent().getName());
    ThreadGroup cg = new ThreadGroup(pg, "Child Group");
    System.out.println(cg.getParent().getName());
}
}

// main
// system
// main
// Parent Group
```

Methods of ThreadGroup Class

1. `String getName();`: Returns Name of the ThreadGroup.
2. `int getMaxPriority();` : Returns the Maximum Priority of ThreadGroup.
3. `void setMaxPriority();`

- To Set Maximum Priority of ThreadGroup.
- The Default Maximum Priority is 10.
- Threads in the ThreadGroup that Already have Higher Priority. Not effected in older but Newly Added Threads this MaxPriority is Applicable.

4. `ThreadGroup getParent()` : Returns Parent Group of Current ThreadGroup.
5. `void list()` : It Prints Information about ThreadGroup to the Console.
6. `int activeCount()` : Returns Number of Active Threads Present in the ThreadGroup.
7. `int activeGroupCount()`: It Returns Number of Active ThreadGroups Present in the Current ThreadGroup.
8. `int enumerate(Thread[] t)` : To Copy All Active Threads of this Group into provided Thread Array. In this Case SubThreadGroup Threads also will be Considered.
9. `int enumerate(ThreadGroup[] g)` : To Copy All `Active` SubThreadGroups into ThreadGroupArray.
10. `boolean isDaemon()`
11. `void setDaemon(boolean b)`
12. `void interrupt()` : To Interrupt All Threads Present in the ThreadGroup.

13. `void destroy()` : To Destroy ThreadGroup and its SubThreadGroups.

```
class MyThread2 extends Thread {
        MyThread2(ThreadGroup g, String name) {
        super(g, name);
        }
        public void run() {
            System.out.println("Child Thread");
            try {
            Thread.sleep(2000);
            }
            catch (InterruptedException e) {}
            }
        }

public class ThreadGroupPractice {
    public static void main(String[] args) throws InterruptedException {
        ThreadGroup pg = new ThreadGroup("Parent Group");
        ThreadGroup cg = new ThreadGroup(pg, "Child Group");
        MyThread2 t1 = new MyThread2(pg, "Child Thread 1");
        MyThread2 t2 = new MyThread2(pg, "Child Thread 2");
        t1.start();
        t2.start();
        System.out.println(pg.activeCount());
        System.out.println(pg.activeGroupCount());
        pg.list();
        Thread.sleep(5000);
        System.out.println(pg.activeCount());
        pg.list();
    }
}

//Child Thread
//Child Thread
//1
//java.lang.ThreadGroup[name=Parent Group,maxpri=10]
//Thread[Child Thread 1,5,Parent Group]
//Thread[Child Thread 2,5,Parent Group]
//java.lang.ThreadGroup[name=Child Group,maxpri=10]
//0
//java.lang.ThreadGroup[name=Parent Group,maxpri=10]
//java.lang.ThreadGroup[name=Child Group,maxpri=10]


ThreadGroup system = Thread.currentThread().getThreadGroup().getParent();
Thread[] t = new Thread[system.activeCount()];
system.enumerate(t);
for (Thread t1: t) {
    System.out.println(t1.getName()+"-------"+t1.isDaemon());
}

// Reference Handler-------true
// Finalizer-------true
```

```
// Signal Dispatcher-------true
// Attach Listener-------true
// Notification Thread-------true
// main-------false
// Monitor Ctrl-Break-------true
// Common-Cleaner-------true
```

Thread Pools

- Pool of Already Created Threads Ready to do Our Job.
- Java Thread pool represents a group of worker threads that are waiting for the job and reused many times.
- To overcome problem of : Creating a New Thread for Every Job May Create Performance and Memory Problems
- `Thread Pool Framework/ Executor Framework` to Implement Thread Pools.
- Create thread pool : `ExecutorService es = Executors.newFixedThreadPool(3)`
- Submit Runnable job : `es.submit(job)`
- Shutdown ExecutorService : `es.shutdown()`

ThreadLocal

- Each thread has its own local resources
- `ThreadLocal<String> tl = new ThreadLocal<>();`
- `tl.set()` : Set value
- `tl.get()` : Get value
- `ThreadLocal<String> tl = new ThreadLocal<>(){ String initialValue(){ return "hi"}}`
- If not used in following example number will be shared by all
- Since println will have common out object
- With threadlocal each thread will create different number

```
import java.beans.IntrospectionException;


public class ThreadLocalPractice implements Runnable {
//    1
//    public static Integer number = 10;
        static ThreadLocal<Integer> number = new ThreadLocal<>();
    public static void main(String[] args) {
        Thread t1 = new Thread(new ThreadLocalPractice());
        Thread t2 = new Thread(new ThreadLocalPractice());
        Thread t3 = new Thread(new ThreadLocalPractice());
        t1.start();
        t2.start();
        t3.start();
        try{
            t1.join();
```

```
            t2.join();
            t3.join();
        }catch(InterruptedException e){ e.printStackTrace();}

    }
    public void run(){
//        2
//        number = (int)(Math.random()*100);
        number.set((int)(Math.random()*100));
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
//        3
//        System.out.println(number);
        System.out.println(number.get());
    }
}

// Without threadlocal (commented) : 18 18 18
// With : 12 56 89
```

Java Shutdown Hook

- A special construct that facilitates the developers to add some code that has to be run when the Java Virtual Machine (JVM) is shutting down is known as the Java shutdown hook.
- The JVM shuts down when:

1. user presses ctrl+c on the command prompt
2. System.exit(int) method is invoked
3. user logoff
4. user shutdown etc.

- The addShutdownHook() method of the Runtime class is used to register the thread with the Virtual Machine. `Runtime.getRuntime().addShutdownHook(new MyThread());`
- The object of the Runtime class can be obtained by calling the static factory method getRuntime(). `Runtime r=Runtime.getRuntime();`
- The removeShutdownHook() method of the Runtime class is invoked to remove the registration of the already registered shutdown hooks. `Runtime.getRuntime().removeShutdownHook(new MyThread());`

```
class MyThread extends Thread{
    public void run(){
        System.out.println("shut down hook task completed..");
    }
}

public class TestShutdown1{
```

```
    public static void main(String[] args)throws Exception {
        Runtime r=Runtime.getRuntime();
        r.addShutdownHook(new MyThread());

        System.out.println("Now main sleeping... press ctrl+c to exit");
        try{Thread.sleep(3000);}catch (Exception e) {}
    }
}
Output:

// Now main sleeping... press ctrl+c to exit
// shut down hook task completed.
```

Executor Framework

- A framework having a bunch of components that are used for managing threads efficiently.
- Executors manage thread execution.
- Top interface - `Executor` : which is used to initiate a thread.
- `ExecutorService` extends Exec : Provides methods that manage execution.

3 implementations of ExecutorService:

- java.util.concurrent also defines the Executors utility class, which includes a number of static methods that simplify the creation of various executors.

1. ThreadPoolExecutor
    1. Executors.newSingleThreadExecutor()
    2. Executors.newFixedThreadPool(5)
    3. Executors.newWorkStealingPool()
2. ScheduledThreadPoolExecutor
    1. Executors.schedule(task, 3, TimeUnit.SECONDS) : After delay task will be executed
    2. Executors.scheduleAtFixedRate(task, waitTime, Period, TimeUnit.SECONDS) : After waitTime it starts and after every Period it executes again : 0 ->1->2 (after every 1 sec)
    3. Executors.scheduleWithFixedDelay(task, waitTime, Period, TimeUnit.SECONDS) : After waitTime it starts and after every Period it executes again : 0->3->6(task 2sec + 1sec of Period) | Best when we don't know task duration
3. ForkJoinPool.

Fork/Join Framework

- Pool of threads using parallelism : Worker threds under a thread
- It simplifies the creation and use of multiple threads
- it automatically makes use of multiple processors
- `ForkJoinTask<V>` is an abstract class that defines a task that can be managed by a ForkJoinPool
- ForkJoinTasks are executed by threads managed by a thread pool of type ForkJoinPool

```java
import java.util.ArrayList;
import java.util.List;

public class ForkJoinPoolPractice {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for(int i=0; i<10; i++)
            list.add(i);
        long initialTime = System.currentTimeMillis();
        list.stream().forEach(a->{ // 1086
//        list.parallelStream().forEach(a->{ // 214
            try{
                Thread.sleep(100);
                System.out.println(Thread.currentThread());
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        });
        long finalTime = System.currentTimeMillis();
        System.out.println(finalTime-initialTime);

    }
}

//with stream
//Thread[main,5,main]
//Thread[main,5,main]
//Thread[main,5,main]
//Thread[main,5,main]
//Thread[main,5,main]
//Thread[main,5,main]
//Thread[main,5,main]
//Thread[main,5,main]
//Thread[main,5,main]
//Thread[main,5,main]
//1086

// with parallelStream
//Thread[ForkJoinPool.commonPool-worker-5,5,main]
//Thread[ForkJoinPool.commonPool-worker-1,5,main]
//Thread[ForkJoinPool.commonPool-worker-4,5,main]
//Thread[ForkJoinPool.commonPool-worker-3,5,main]
//Thread[ForkJoinPool.commonPool-worker-2,5,main]
//Thread[ForkJoinPool.commonPool-worker-6,5,main]
//Thread[ForkJoinPool.commonPool-worker-7,5,main]
//Thread[main,5,main]
//Thread[ForkJoinPool.commonPool-worker-5,5,main]
//Thread[ForkJoinPool.commonPool-worker-1,5,main]
//214
```

Executor Service

1. `execute()` : es.execute(Runnable); : `NO return type`
2. `shutDown()` : es.shutdown(); - Doesn't wait for whole task to execute of es and continues main task
3. `shutDownNow` : Interrupts all running threads and shut down executor down
4. `awaitTermination()` : es.awaitTermination(10, TimeUnit.SECONDS) - wait till 10sec and then main task will be continued
5. `submit(new Runnable())` : returns type - null
6. `submit(new Callable())` : returns type - in the form of Future
7. `invokeAny()` : Executes all callable but returns any one (mainly which takes less time) | Accepts collection of callables
8. `invokeAll()` : Executes all callable and returns list of future | Accepts collection of callables
9. `Callable` : Like Runnable but has call() instead of run() and returns something called Future

```
class Service2 implements Callable<String>{
    public String call() throws InterruptedException, ExecutionException{
        return(i+" "+Thread.currentThread().getName()); // Returns Future
    }
}
```

10. `Future` : Returned by callable is stored as future object. It has future.get(), future.isDone()

- `Future is tightly coupled with Executor Service` : So every non-terminated future will throw exceptions if you shut down the executor.
- Any call to get() will block and wait - Until the underlying callable has been terminated. In the worst case a callable will run forever, thus making app unresponsive

```
Future<String> future = es.submit(task)
```

1. `FutureTask` : Provide Callable Object in FutureTask itself and this will be passed in submit

```
ExecutorService es = Executors.newSingleThreadExecutor();

FutureTask<String> futureTaskObj = new FutureTask<> (new Service());
es.submit(futureTaskObj);
sout(futureTask.get());
```

1. execute() and shutDown() and awaitTermination()

- execute() returns nothing
- shutdown() : Doesn't wait for es to complete and start running main
- awaitTermination() : Wait till complete execution of es and then start main

```
import java.util.concurrent.*;
import java.util.Date;

class Service implements Runnable{
    int i;
    public Service(int i){this.i = i;}
    public void run(){
        System.out.println(i+" "+Thread.currentThread().getName());
        try{Thread.sleep(1000);}catch(InterruptedException e)
{e.printStackTrace();}
    }
}
public class ExecutorServicePractice {
   public static void main(String[] args) throws InterruptedException {
       ExecutorService es = Executors.newFixedThreadPool(5);
       System.out.println(new Date());
       for(int i=1; i<=10; i++){
           es.execute(new Service(i));
       }
       es.shutdown();
//     es.awaitTermination(10, TimeUnit.SECONDS);
       System.out.println(new Date());
   }
}

With shutDown()
//Wed Apr 26 20:35:30 IST 2023
//Wed Apr 26 20:35:30 IST 2023
//4 pool-1-thread-4
//2 pool-1-thread-2
//3 pool-1-thread-3
//5 pool-1-thread-5
//1 pool-1-thread-1
//6 pool-1-thread-2
//7 pool-1-thread-5
//8 pool-1-thread-4
//9 pool-1-thread-3
//10 pool-1-thread-1

with awaitTermination()
//Wed Apr 26 20:34:17 IST 2023
//1 pool-1-thread-1
//3 pool-1-thread-3
//5 pool-1-thread-5
//2 pool-1-thread-2
//4 pool-1-thread-4
//7 pool-1-thread-5
//6 pool-1-thread-3
//8 pool-1-thread-1
//9 pool-1-thread-2
//10 pool-1-thread-4
//Wed Apr 26 20:34:27 IST 2023
```

2. submit() with Runnable

- Returns null as stored in future

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;
import java.util.Date;

class Service implements Runnable{
    int i;
    public Service(int i){this.i = i;}
    public void run(){
        System.out.println(i+" "+Thread.currentThread().getName());
        try{Thread.sleep(1000);}catch(InterruptedException e)
{e.printStackTrace();}
    }
}

public class ExecutorServicePractice {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService es = Executors.newFixedThreadPool(5);
        System.out.println(new Date());
        List<Future<String>> futureList = new ArrayList<>();
        for(int i=1; i<=10; i++){
            futureList.add((Future<String>) es.submit(new Service(i)));
        }
        es.awaitTermination(10, TimeUnit.SECONDS);
        for(Future<String> fut : futureList)
            System.out.println(fut.get()); // null
        System.out.println(new Date());
    }
}

// Wed Apr 26 20:44:10 IST 2023
// 3 pool-1-thread-3
// 1 pool-1-thread-1
// 2 pool-1-thread-2
// 5 pool-1-thread-5
// 4 pool-1-thread-4
// 7 pool-1-thread-1
// 6 pool-1-thread-2
// 8 pool-1-thread-3
// 9 pool-1-thread-5
// 10 pool-1-thread-4
// null
// null
// null
// null
// null
// null
```

```
// null
// null
// null
// null
// Wed Apr 26 20:44:20 IST 2023
```

3. submit() with Callable

- Returns as Future

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;
import java.util.Date;

class Service2 implements Callable<String>{
    int i;
    public Service2(int i){this.i = i;}
    public String call() throws InterruptedException, ExecutionException{
        return(i+" "+Thread.currentThread().getName());
    }
}

public class ExecutorServicePractice {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService es = Executors.newFixedThreadPool(5);
        System.out.println(new Date());
        List<Future<String>> futureList = new ArrayList<>();
        for(int i=1; i<=10; i++){
            futureList.add((Future<String>) es.submit(new Service2(i)));
        }
        es.awaitTermination(10, TimeUnit.SECONDS);
        for(Future<String> fut : futureList)
            System.out.println(fut.get());
        System.out.println(new Date());
    }
}

// Wed Apr 26 20:58:15 IST 2023
// 1 pool-1-thread-1
// 2 pool-1-thread-2
// 3 pool-1-thread-3
// 4 pool-1-thread-4
// 5 pool-1-thread-5
// 6 pool-1-thread-4
// 7 pool-1-thread-5
// 8 pool-1-thread-4
// 9 pool-1-thread-4
// 10 pool-1-thread-4
// Wed Apr 26 20:58:25 IST 2023
```

4. invokeAny()

- Executes all callable but invokes any one
- Returns only 1 future

```
class Service3 implements Callable<String>{
    int i;
    public Service3(int i){this.i = i;}
    public String call() throws InterruptedException, ExecutionException{
        System.out.println(i+" "+Thread.currentThread().getName());
        return("Returning "+i+" "+Thread.currentThread().getName());
    }
}

public class ExecutorServicePractice {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService es = Executors.newFixedThreadPool(5);
        System.out.println(new Date());
        List<Callable<String>> callableList = new ArrayList<>();
        callableList.add(new Service3(1));
        callableList.add(new Service3(2));
        callableList.add(new Service3(4));
        callableList.add(new Service3(3));
        String str = es.invokeAny(callableList);
        es.awaitTermination(10, TimeUnit.SECONDS);
        System.out.println(str);
        System.out.println(new Date());
    }
}

// Wed Apr 26 21:14:13 IST 2023
// 1 pool-1-thread-1
// 3 pool-1-thread-4
// 4 pool-1-thread-3
// 2 pool-1-thread-2
// Returning 3 pool-1-thread-4
// Wed Apr 26 21:14:23 IST 2023
```

5. invokeAll()

- Executes all callable but invokes all
- Returns list of all future objects

```
class Service3 implements Callable<String>{
    int i;
    public Service3(int i){this.i = i;}
    public String call() throws InterruptedException, ExecutionException{
```

```
        System.out.println(i+" "+Thread.currentThread().getName());
        return("Returning "+i+" "+Thread.currentThread().getName());
    }
}

public class ExecutorServicePractice {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService es = Executors.newFixedThreadPool(5);
        System.out.println(new Date());
        List<Callable<String>> callableList = new ArrayList<>();
        callableList.add(new Service3(1));
        callableList.add(new Service3(2));
        callableList.add(new Service3(4));
        callableList.add(new Service3(3));
        List<Future<String>> futureList =  es.invokeAll(callableList);
        es.awaitTermination(10, TimeUnit.SECONDS);
        for(Future<String> fut : futureList)
            System.out.println(fut.get());
        System.out.println(new Date());
    }
}

// Wed Apr 26 21:24:57 IST 2023
// 3 pool-1-thread-4
// 2 pool-1-thread-2
// 4 pool-1-thread-3
// 1 pool-1-thread-1
// Returning 1 pool-1-thread-1
// Returning 2 pool-1-thread-2
// Returning 4 pool-1-thread-3
// Returning 3 pool-1-thread-4
// Wed Apr 26 21:25:07 IST 2023
```

6. Future V/S FutureTask

- In Future : Get future from Callable Object | Implements Runnable

```
ExecutorService es = Executors.newSingleThreadExecutor();

Future<String> futureObj = es.submit(new Service());
sout(futureObj.get());
```

- In FutureTask : Provide Callable Object in FutureTask itself and this will be passed in submit | Implements RunnableFuture Interface -> Runnable and Future Interface

```
ExecutorService es = Executors.newSingleThreadExecutor();

FutureTask<String> futureTaskObj = new FutureTask<> (new Service());
```

```
es.submit(futureTaskObj);
sout(futureTask.get());
```

Limitations of Future

1. Cannot be completed explicitly
2. Actions cannot be performed until the result is available(since it blocks and wait)
3. Attaching callback function not possible
4. Mutiple futures can't be Chained together
5. Mutiple futures can't be Combined together
6. No Exception Handling

Completable Future

- Works on all these limitations : It has methods for Creating + Chaining + Combining + Exception Hnadling
- Implements Future + CompletionStage interface
- Restriction : Static data only can be used in CF

runAsync()

```
import java.util.concurrent.CompletableFuture;

public class CompletableFuturePractice {
    private static int data=5;
    public static void main(String[] args) throws InterruptedException {
        System.out.println(" Before "+data);
        CompletableFuture.runAsync(()->{
            try{
                Thread.sleep(5000);
            } catch(InterruptedException e){e.printStackTrace();}
            data=10;
        });
        System.out.println("Main Thraed ");
        System.out.println("Data in progress in main "+data);
        Thread.sleep(2500);
        System.out.println("Data ready in main "+data);

    }
}

// Before 5
// Main Thraed
// Data in progress in main 5
// Data ready in main 5
```

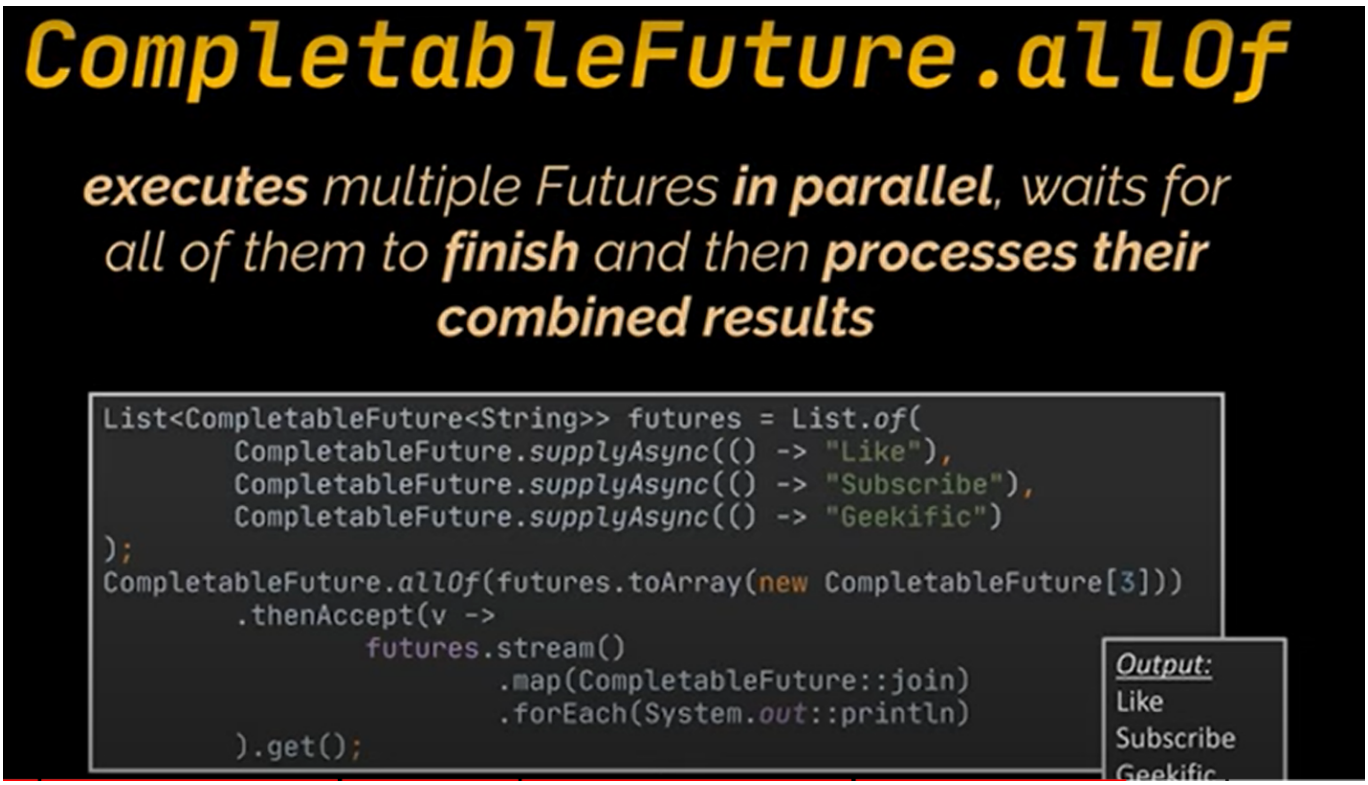
supplyAsync

- Returns CompletableFuture Object

```
public class CompletableFuturePractice {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        CompletableFuture<String> data = CompletableFuture.supplyAsync(()->{
            try{
                Thread.sleep(5000);
            } catch(InterruptedException e){e.printStackTrace();}
            return "data from CF";
        });
        System.out.println("Main Thraed ");
        System.out.println("Data in progress in main "+data.get());
        Thread.sleep(2500);
        System.out.println("Data ready in main "+data.get());

    }
}

// Main Thraed
// Data in progress in main data from CF
// Data ready in main data from CF
```



Locks

- Problems with Traditional synchronized Key Word :

1. If a Thread Releases the Lock then `which waiting Thread will get that Lock` we are Not having any Control on this.
2. We can't Specify `Maximum waiting Time for a Thread` to get Lock so that it will Wait until getting Lock, which May Effect Performance of the System and Causes Dead Lock.

- A Lock Object is Similar to Implicit Lock acquired by a Thread to Execute synchronized Method OR synchronized Block

Methods of Lock

1. `void lock()` : It Locks the Lock Object | If Lock Object is Already Locked by Other Thread then it will wait until it is Unlocked.
2. `boolean tryLock()` : To Acquire the Lock if it is Available | If the Lock is Available then Thread Acquires the Lock and Returns true else false and continues execution | Never blocks thread

```
if (l.tryLock()) {
Perform Safe Operations
}
else {
 Perform Alternative Operations
}
```

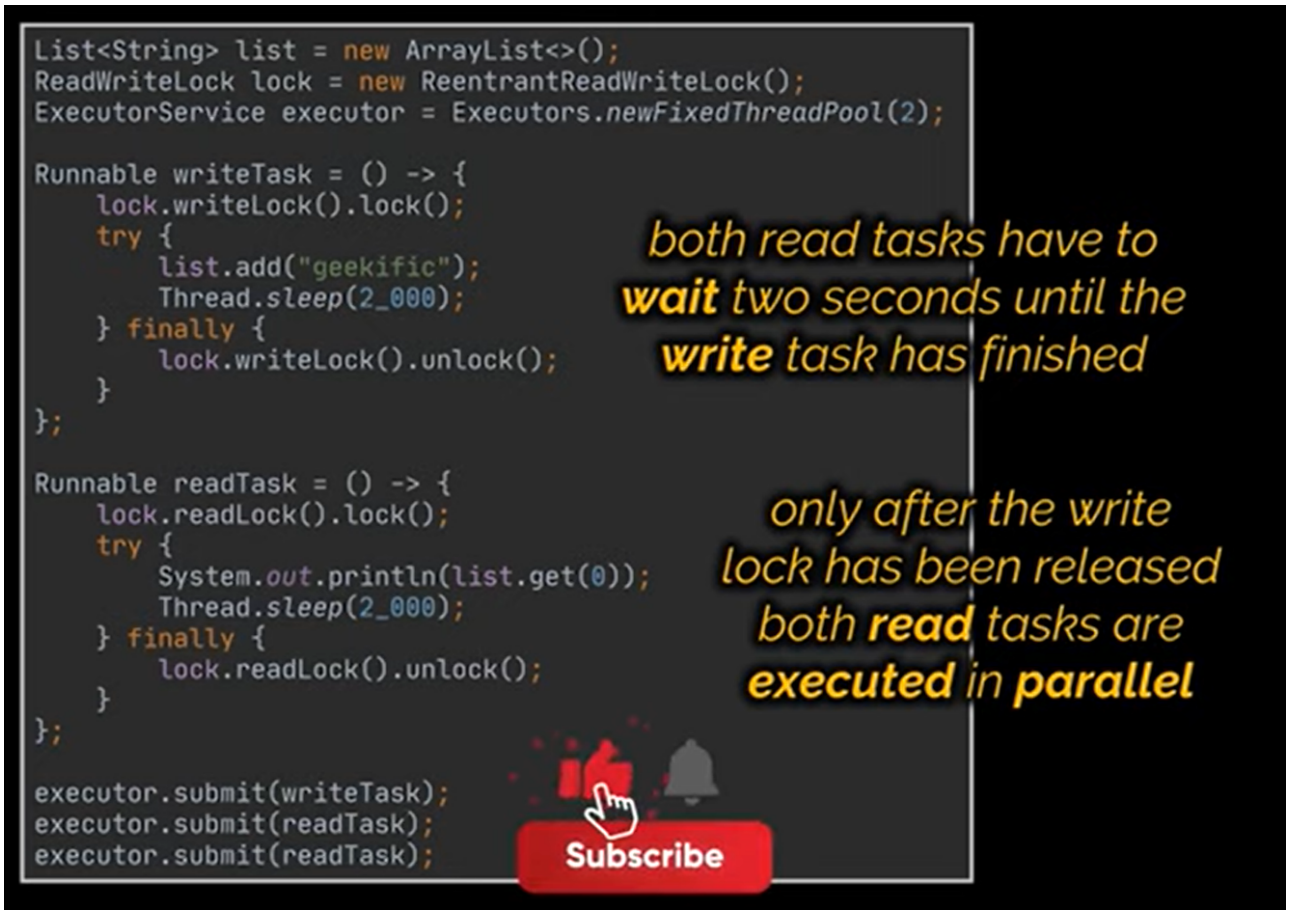
3. `boolen tryLock(long time, TimeUnit unit)` : To Acquire the Lock if it is Available | If the Lock is Unavailable then Thread can Wait until specified Amount of Time | Still if the Lock is Unavailable then Thread can Continue its Execution.
4. `void lockInterruptedly()` : Acquires the Lock if it is Available and Returns Immediately | If it is Unavailable then the Thread will wait while waiting if it is Interrupted then it won't get the Lock.
5. `void unlock()` : To Release the Lock.

```
public class SharedObject {
    //...
    ReentrantLock lock = new ReentrantLock();
    int counter = 0;

    public void perform() {
        lock.lock();
        try {
            // Critical section here
            count++;
        } finally {
            lock.unlock();
        }
    }
    //...
}
```

1. `Reentrant Lock` : Default | Implicit Monitor | Mutual Exclusion

2. `Read Write Lock` : Read concurrently as long as noone's writing
3. `Stamped Lock` : Returns stamp represented by long value



Synchronizers

- Synchronizers offer high-level ways of synchronizing the interactions between multiple threads. The synchronizer classes defined by java.util.concurrent are

1. `Semaphore` Implements the classic semaphore.
2. `CountDownLatch` Waits until a specified number of events have occurred.
3. `CyclicBarrier` Enables a group of threads to wait at a predefined execution point.
4. `Exchanger` Exchanges data between two threads.
5. `Phaser` Synchronizes threads that advance through multiple phases of an operation.

1. Semaphore

- A semaphore controls access to a shared resource through the use of a counter
- If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource.
- Count increases when released, Decreases when acquired.

1. `Semaphore(int num)` : num specifies the initial permit count. Thus, num specifies the number of threads that can access a shared resource at any one time
2. `Semaphore(int num, boolean how)` : By setting how to true, you can ensure that waiting threads are granted a permit in the order in which they requested access
3. `void acquire( ) throws InterruptedException` : To acquire a permit

4. `void acquire(int num) throws InterruptedException`
5. `void release( )` : To release a permit
6. `void release(int num)`

- To use a semaphore to control access to a resource, each thread that wants to use that resource must first call acquire( ) before accessing the resource. When the thread is done with the resource, it must call release( )

```
import java.util.concurrent.*;
class SemDemo {
    public static void main(String[] args) {
        Semaphore sem = new Semaphore(1);
        new Thread(new IncThread(sem, "A")).start();
        new Thread(new DecThread(sem, "B")).start();
    }
}
// A shared resource.
class Shared {
    static int count = 0;
}
// A thread of execution that increments count.
class IncThread implements Runnable {
    String name;
    Semaphore sem;
    IncThread(Semaphore s, String n) {
        sem = s;
        name = n;
    }
    public void run() {
        System.out.println("Starting " + name);
        try {
            // First, get a permit.
            System.out.println(name + " is waiting for a permit.");
            sem.acquire();
            System.out.println(name + " gets a permit.");
            // Now, access shared resource.
            for(int i=0; i < 5; i++) {
                Shared.count++;
                System.out.println(name + ": " + Shared.count);
                // Now, allow a context switch -- if possible.
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
        // Release the permit.
        System.out.println(name + " releases the permit.");
        sem.release();
    }
}
// A thread of execution that decrements count.
class DecThread implements Runnable {
```

```
    String name;
    Semaphore sem;
    DecThread(Semaphore s, String n) {
        sem = s;
        name = n;
    }
    public void run() {
        System.out.println("Starting " + name);
        try {
            // First, get a permit.
            System.out.println(name + " is waiting for a permit.");
            sem.acquire();
            System.out.println(name + " gets a permit.");
            // Now, access shared resource.
            for(int i=0; i < 5; i++) {
                Shared.count--;
                System.out.println(name + ": " + Shared.count);
                // Now, allow a context switch -- if possible.
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
        // Release the permit.
        System.out.println(name + " releases the permit.");
        sem.release();
    }
}

Starting B
Starting A
B is waiting for a permit.
B gets a permit.
A is waiting for a permit.
B: -1
B: -2
B: -3
B: -4
B: -5
B releases the permit.
A gets a permit.
A: -4
A: -3
A: -2
A: -1
A: 0
A releases the permit.
```

2. CountDownLatch

- Wait until one or more events have occurred

- It is initially created with a count of the number of events that must occur before the latch is released. Each time an event happens, the count is decremented. When the count reaches zero, the latch opens.

1. `CountDownLatch(int num)` : num = e number of events that must occur in order for the latch to open
2. `void await( ) throws InterruptedException` : To wait till CountDownLatch reaches zero. It returns false if the time limit is reached and true if the countdown reaches zero.
3. `boolean await(long wait, TimeUnit tu) throws InterruptedException`
4. `void countDown( )` : decrements the count associated with the invoking object

```
import java.util.concurrent.CountDownLatch;
class CDLDemo {
    public static void main(String[] args) {
        // 1
        CountDownLatch cdl = new CountDownLatch(5);
        System.out.println("Starting");
        new Thread(new MyThread(cdl)).start();
        try {
            // 2
            cdl.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
        System.out.println("Done");
    }
}
class MyThread implements Runnable {
    CountDownLatch latch;
    MyThread(CountDownLatch c) {
        // 3
        latch = c;
    }
    public void run() {
        for(int i = 0; i<5; i++) {
            System.out.println(i+" "+latch);
            // 4
            latch.countDown(); // decrement count
        }
    }
}

Starting
0 java.util.concurrent.CountDownLatch@50ab8444[Count = 5]
1 java.util.concurrent.CountDownLatch@50ab8444[Count = 4]
2 java.util.concurrent.CountDownLatch@50ab8444[Count = 3]
3 java.util.concurrent.CountDownLatch@50ab8444[Count = 2]
4 java.util.concurrent.CountDownLatch@50ab8444[Count = 1]
Done
```

3. CyclicBarrier

- When a set of two or more threads must wait at a predetermined execution point until all threads in the set have reached that point.
- Suspends until the specified number of threads has reached the barrier point

1. `CyclicBarrier(int numThreads)` : numThreads specifies the number of threads that must reach the barrier before execution continues
2. `CyclicBarrier(int numThreads, Runnable action)` : action specifies a thread that will be executed when the barrier is reached
3. `int await( ) throws InterruptedException, BrokenBarrierException` : This will pause execution of the thread until all of the other threads also call await( ). Returns a value that indicates the order that the threads arrive at the barrier point.
4. `int await(long wait, TimeUnit tu) throws InterruptedException, BrokenBarrierException, TimeoutException`

- The first thread returns a value equal to the number of threads waited upon minus one. The last thread returns zero.

```
import java.util.concurrent.*;
class BarDemo {
    public static void main(String[] args) {
        // 1
        CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );
        System.out.println("Starting");
        new Thread(new MyThread4(cb, "A")).start();
        new Thread(new MyThread4(cb, "B")).start();
        new Thread(new MyThread4(cb, "C")).start();
    }
}
// A thread of execution that uses a CyclicBarrier.
class MyThread4 implements Runnable {
    // 2
    CyclicBarrier cbar;
    String name;
    MyThread4(CyclicBarrier c, String n) {
        cbar = c;
        name = n;
    }
    public void run() {
        System.out.println(name);
        try {
            // 3
            cbar.await();
        } catch (BrokenBarrierException | InterruptedException exc) {
            System.out.println(exc);
        }
    }
}
// An object of this class is called when the
// CyclicBarrier ends.
class BarAction implements Runnable {
    public void run() {
```

```
            System.out.println("Barrier Reached!");
        }
    }

    Starting
    A
    C
    B
    Barrier Reached!
```

4. Exchanger

- Exchange of data between two threads
- It simply waits until two separate threads call its exchange( )
- Exchanger is a generic class : Exchanger
- `Exchanger<String> exgr = new Exchanger<String>();`

1. `V exchange(V objRef) throws InterruptedException` : objRef= data to be exchanged
2. `V exchange(V objRef, long wait, TimeUnit tu) throws InterruptedException, TimeoutException`

- For example, one th]read might prepare a buffer for receiving information over a network connection. Another thread might fill that buffer with the information from the connection. The two threads work together so that each time a new buffer is needed, an exchange is made.

```
// package ExecutorFrameworkPractice;

import java.util.concurrent.Exchanger;
class ExgrDemo {
    public static void main(String[] args) {
        // 1
        Exchanger<String> exgr = new Exchanger<String>();
        new Thread(new UseString(exgr)).start();
        new Thread(new MakeString(exgr)).start();
    }
}
class MakeString implements Runnable {
    // 2
    Exchanger<String> ex;
    String str;
    MakeString(Exchanger<String> c) {
        // 3
        ex = c;
        str = new String();
    }
    public void run() {
        char ch = 'A';
        for(int i = 0; i < 3; i++) {
            // Fill Buffer
            for(int j = 0; j < 5; j++)
```

```
                str += ch++;
            try {
                // Exchange a full buffer for an empty one.
                System.out.println("Making "+str);
                // 4
                str = ex.exchange(str);
            } catch(InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}
// A Thread that uses a string.
class UseString implements Runnable {
    // 2
    Exchanger<String> ex;
    String str;
    UseString(Exchanger<String> c) {
        // 3
        ex = c;
    }
    public void run() {
        for(int i=0; i < 3; i++) {
            try {
                // Exchange an empty buffer for a full one.
                // 4
                str = ex.exchange(new String());
                System.out.println("Got: " + str);
            } catch(InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}

Making ABCDE
Making FGHIJ
Got: ABCDE
Got: FGHIJ
Making KLMNO
Got: KLMNO
```

5. Phaser

- Enable the synchronization of threads that represent one or more phases of activity.

1. `Phaser( )` : Creates phaser with registration count = Zero
2. `Phaser(int numParties)` : Creates phaser with registration count = numParties
3. `register( )` : Register parties to phaser. Returns phase number of the phase to which it is registered
4. `arrive( )` : To signal that a party has completed a phase. When the number of arrivals equals the number of registered parties, the phase is completed and the Phaser moves on to the next phase (if there is one). If the phaser has been terminated, then it returns a negative value.

5. `arriveAndAwaitAdvance( )` : To indicate the completion of a phase and then wait until all other registrants have also completed that phase. . It returns the next phase number or a negative value if the phaser has been terminated.
6. `arriveAndDeregister( )` : A thread can arrive and then deregister itself. It returns the current phase number or a negative value if the phaser has been terminated.
7. `getPhase( )` : To obtain the current phase number

- *When a Phaser is created, the first phase will be 0, the second phase 1, the third phase 2, and so on. A negative value is returned if the invoking Phaser has been terminated.*
- For each registered party, have the phaser wait until all registered parties complete a phase.
- Phaser works a bit like a CyclicBarrier, except that it supports multiple phases.
- For example, you might have a set of threads that implement three phases of an order-processing application.In the first phase, separate threads are used to validate customer information, check inventory, and confirm pricing. When that phase is complete, the second phase has two threads that compute shipping costs and all applicable tax. After that, a final phase confirms payment and determines estimated shipping time.

```
// package ExecutorFrameworkPractice;

import java.util.concurrent.*;
class PhaserDemo {
    public static void main(String[] args) {
        // 1
        Phaser phsr = new Phaser(1);
        int curPhase;
        System.out.println("Starting");
        new Thread(new MyThread5(phsr, "A")).start();
        new Thread(new MyThread5(phsr, "B")).start();
        new Thread(new MyThread5(phsr, "C")).start();

        // Wait for all threads to complete phase one.
        curPhase = phsr.getPhase();
        // 2
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        // Wait for all threads to complete phase two.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        // Wait for all threads to complete phase three.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        // Deregister the main thread.
        phsr.arriveAndDeregister();
        if(phsr.isTerminated())
            System.out.println("The Phaser is terminated");
    }
```

```
}
// A thread of execution that uses a Phaser.
class MyThread5 implements Runnable {
    Phaser phsr;
    String name;
    MyThread5(Phaser p, String n) {
        phsr = p;
        name = n;
        // 3
        phsr.register();
    }
    public void run() {

        System.out.println("Thread " + name + " Beginning Phase One");
        // 4
        phsr.arriveAndAwaitAdvance(); // Signal arrival.
        // Pause a bit to prevent jumbled output. This is for illustration
        // only. It is not required for the proper operation of the phaser.
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Thread " + name + " Beginning Phase Two");
        phsr.arriveAndAwaitAdvance(); // Signal arrival.
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Thread " + name + " Beginning Phase Three");
        phsr.arriveAndDeregister(); // Signal arrival and deregister.
    }
}

Starting
Thread C Beginning Phase One
Thread A Beginning Phase One
Thread B Beginning Phase One
Phase 0 Complete
Thread C Beginning Phase Two
Thread B Beginning Phase Two
Thread A Beginning Phase Two
Phase 1 Complete
Thread B Beginning Phase Three
Thread C Beginning Phase Three
Thread A Beginning Phase Three
Phase 2 Complete
The Phaser is terminated
```

17. Collection

1. Collection API / Collection Framework- Concept
2. Collection - Interface - List, Set
3. Collections - Class

Collection Framework

- Collection Framework is a `Java API` which `provides architecture` to store and manipulate group of objects.
- Well designed `set of collection interfaces and classes for storing and maipulationg group of data as a single unit`, as collection
- It contains :-

1. `Interfaces` - abstract data types that represent collection
2. `Classes` - Implementation - These are the concrete implementations of collection interfaces
3. `Algorithms` - Methods for computation on objects

Collection

- Interface - Root Interface
- `Group of objects as single unit`

```
Interface I1{
    void f1();
    int f2(int a);
}

Class A implements I1{
    public void f1(){
        ...
    }
    public int f2(int a){
        ...
    }
}
```

Array V/S Collection : Why Collection?

Differences Between Arrays And Collections:

Arrays	Collections
Arrays are Fixed in Size.	Collections are Growable in Nature.
With Respect to Memory Arrays are Not Recommended to Use.	With Respect to Memory Collections are Recommended to Use.
With Respect to Performance Arrays are Recommended to Use.	With Respect to Performance Collections are Not Recommended to Use.
Arrays can Hold Only Homogeneous Data Elements.	Collections can Hold Both *Homogeneous* and *Heterogeneous* Elements.
Arrays can Hold Both Primitives and Objects.	Collections can Hold Only Objects but Not Primitives.
Arrays Concept is Not implemented based on Some Standard Data Structure. Hence Readymade Method Support is Not Available.	For every Collection class underlying Data Structure is Available Hence Readymade Method Support is Available for Every Requirement.

Array

- Similar type of data collection
- Already size is known mostly
- Operations - have to define methods for each operation
- Due to `fast execution` it `consumes more memory`
- `Better when Size is known` - better choice in terms of performance(but takes more time)

- Limitations of Array :

1. To Use Array concept compulsorily we should `Know the Size in Advance` which May Not be Possible Always.
2. Arrays can hold Only `Homogeneous Data Type` Elements.
3. `NO Readymade Methods Support` as array concept is Not implemented based on Some Standard Data Structure

Collection

- Group of Individual Objects as a Single Entity
- Size expandable - `Takes more time`
- Homogenous + Heterogenous Data `Except TreeSet and TreeMap`
- Operations - Can use classes - abstract data type - structure + methods
- Flexibility in memory (but increases with also increases time)

- Advantages Of Collection :

1. Collections are `Growable` in Nature. That is based on Our Requirement we can Increase OR Decrease the Size.
2. Collections can `Hold Both Homogeneous and Heterogeneous` Elements.
3. Every Collection Class is implemented based on Some Standard Data Structure. Hence for Every Requirement `Readymade Method Support is Available`. Being a Programmer we have to Use those Methods and we are Not Responsible to Provide Implementation.

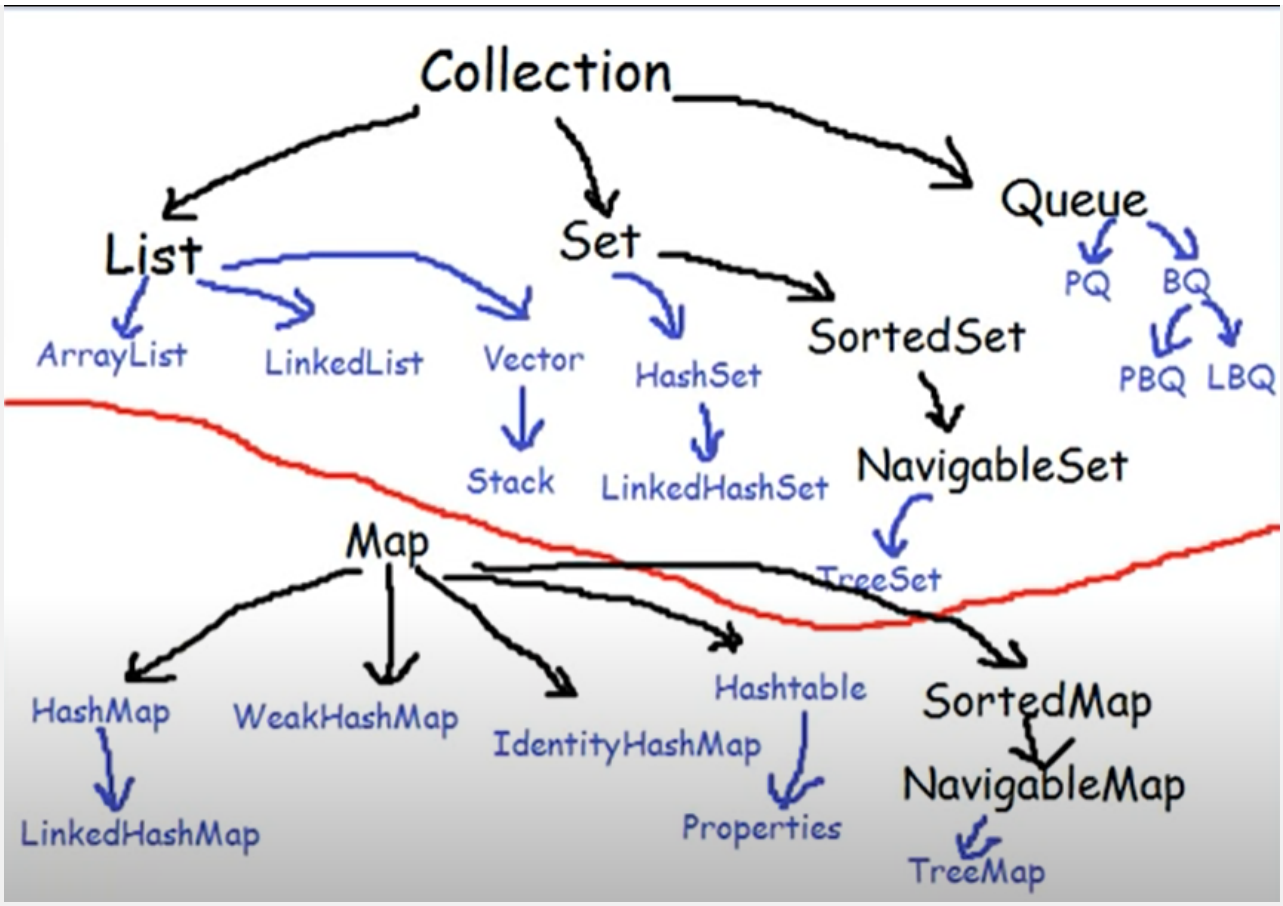
- To Hold and Transfer Data (Objects) form One Location to Another Location - Every Collection Class Implements Serializable and Cloneable Interfaces.

Methods of Collection

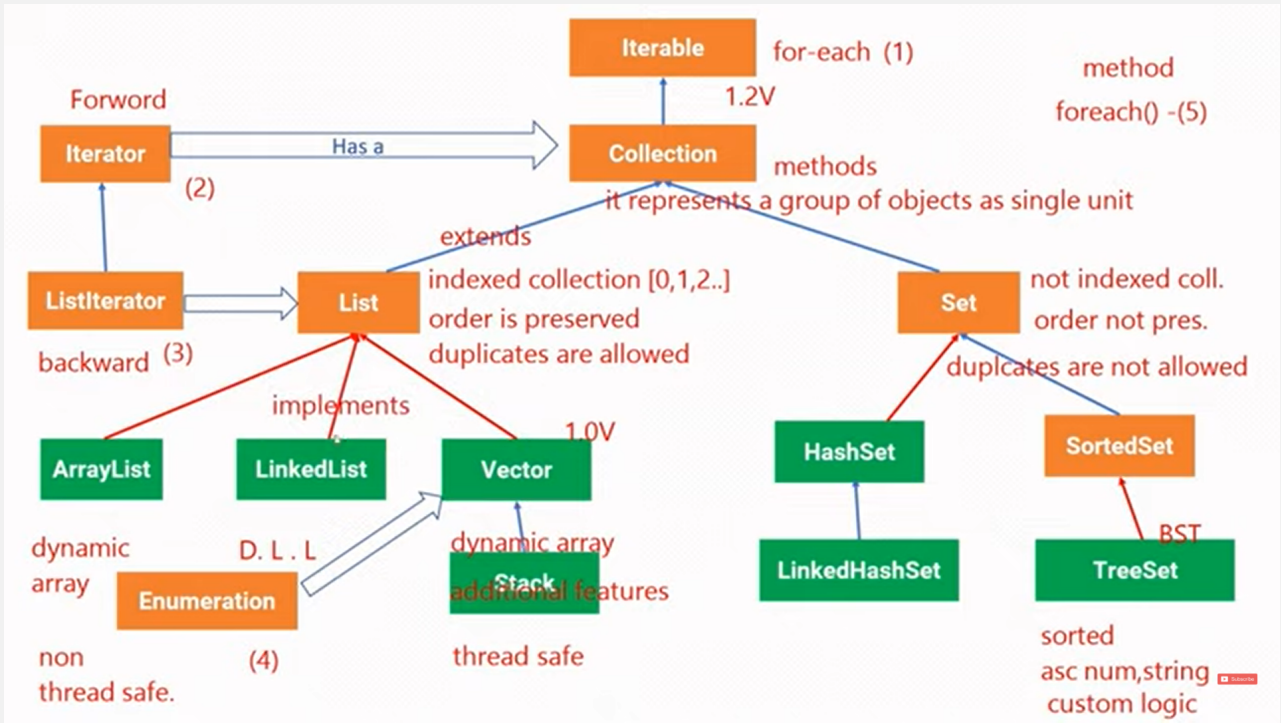
- boolean add(Object o); - l.add("Red")
- boolean addAll(Collection c); - l.addAll(lst)
- boolean remove(Object o); - l.remove("Red")
- boolean removeAll(Collection c); - l.removeAll(lst)
- boolean retainAll(Collection c)); - l.retainAll(lst) : To Remove All Objects Except those Present in c.
- int size(); - l.size()
- boolean contains(Object o); - l.contains("Red")
- boolean containsAll(Collection c) - l.containsAll(lst)
- void clear(); - l.clear()
- boolean isEmpty(); - l.isEmpty()
- Object[] toArray()
- Iterator iterator()

```
Iterator itr = lst.iterator();
while(itr.hasNext()){
    Sout(itr.next());
}
```

Collection


Collection
List
Set
Queue
ArrayList
LinkedList
Vector
HashSet
SortedSet
PQ
BQ
PBQ
LBQ
Stack
LinkedHashSet
NavigableSet
Map
TreeSet
HashMap
WeakHashMap
IdentityHashMap
Hashtable
SortedMap
NavigableMap
LinkedHashMap
Properties
TreeMap

>


Iterable
for-each (1)
1.2V
method
foreach() -(5)
Forword
Iterator
Has a
Collection
methods
(2)
it represents a group of objects as single unit
extends
ListIterator
List
indexed collection [0,1,2..]
order is preserved
duplicates are allowed
Set
not indexed coll.
order not pres.
backward (3)
duplcates are not allowed
implements
1.0V
ArrayList
LinkedList
Vector
HashSet
SortedSet
dynamic
array
D. L . L
dynamic array
BST
Enumeration
Stack
additional features
LinkedHashSet
TreeSet
non
thread safe.
(4)
thread safe
sorted
asc num,string
custom logic

>

- The criteria to choose the collection

Implementation	Iteration order
ArrayList	Insertion Order
LinkedList	Insertion Order
HashSet	Random order
LinkedHashSet	Insertion Order
TreeSet	Ascending order based on Comparable / Comparator
HashMap	Random Order
LinkedHashMap	Insertion Order as per Key
TreeMap	Ascending order of keys based on Comparable / Comparator

- Interface
- Root Interface of Collection Framework
- There's no concrete class which implements Collection Interface directly.
- `Collection` -> `List` - `Set` - `Queue`
- `List` -> ArrayList - LinkedList - Vector -> Stack
- `Set` -> HashSet -> LinkedHashSet - `SortedSet` -> `NavigableSet` -> TreeSet
- `Queue` -> PriorityQueue - BlockingQueue -> PriorityBlockingQueue - LinkedBlockingQueue
- `Map` -> HashMap -> LinkedHashMap - WeakHashMap - IdentityHashMap - Hashtable-> Properties - `SortedMap`-> `NavigableMap`-> TreeMap
- Only TreeMap and TreeSet can't contain heterogenous

List

- Interface - child of Collection Interface
- `Duplicates allowed`
- `Insertion order Preserved`
- Index will Play Very Important Role in List
- ArrayList and Vector Classes Implements RandomAccess Interface. So that we can Access any Random Element with the Same Speed. `RandomAccess Interface` Present in java.util Package and it doesn't contain any Methods. Hence it is a Marker Interface
- ArrayList, LinkedList, Vector -> Stack are the Implementation classes

Methods of List

- void add(int index, E element) : shifts and insert
- E set(int index, E element) : replace and insert
- E get(int index)
- int indexOf(Object o) : first Occurence : if -1: element not present
- int lastIndexOf(Object o) : last Occurence : if -1: element not present
- E remove(int index)

- list subList(int fromIndex, int toIndex) : returns portion of list (from,to]

ArrayList

- Defined using dynamic arrays - Resizable Array or Growable Array
- `Duplicates allowed`
- `Insertion order Preserved`
- `Null Insertion possible`
- `Heterogenous objects allowed`
- Contiguous memory allocation - have to shift before inserting reduces performance
- It is resizable
- ArrayList is Best Suitable if Our Frequent Operation is Retrieval Operation due to RandomAccess Interface.
- ArrayList is Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle. Because it required Several Shift Operations Internally.
- default capacity "10"
- If reaches max capacity : New Capacity=(current capacity*3/2)+1
- Non – Synchronized
- NOT Thread safe
- `ArrayList lst = new ArrayList()`
- `ArrayList lst = new ArrayList(int initialCapacity)`
- `ArrayList lst = new ArrayList(Collection c)`
- Get Synchronized Version of ArrayList Object by using the following Method of Collections Class

```
ArrayList lst = new ArrayList();
List l = Collections.synchronizedList(lst);
// lst - Non - Synchronized Version
// l - Synchronized Version
```

LinkedList

- Uses Doubly Linked List
- `Duplicates allowed`
- `Insertion order Preserved`
- `Null Insertion possible`
- `Heterogenous objects allowed`
- Logically contiguous, but actually not - no need to shift, just create another node - cost of inserting less
- Implements Serializable and Cloneable Interfaces but Not RandomAccess Interface.
- Best Choice if Our Frequent Operation is Insertion OR Deletion in the Middle.
- Worst Choice if Our Frequent Operation is Retrieval - Accessing nth element is costly
- We can Use LinkedList to Implement Stacks and Queues - 6 methods
- `LinkedList llst = new LinkedList()`
- `LinkedList llst = new LinkedList(Collection c)`

Methods of LinkedList

- void addFirst(Object o)
- void addLast(Object o)
- Objects.getFirst()
- Objects.getLast()
- Objects.removeFirst()
- Objects.removeLast()

Vector

Differences between *ArrayList* and *Vector*:

ArrayList	Vector
Every Method Present Inside ArrayList is Non – Synchronized.	Every Method Present in Vector is Synchronized.
At a Time Multiple Threads are allow to Operate on ArrayList Simultaneously and Hence ArrayList Object is Not Thread Safe.	At a Time Only One Thread is allow to Operate on Vector Object and Hence Vector Object is Always Thread Safe.
Relatively Performance is High because Threads are Not required to Wait.	Relatively Performance is Low because Threads are required to Wait.
Introduced in 1.2 Version and it is Non – Legacy.	Introduced in 1.0 Version and it is Legacy.

- Defined using dynamic arrays - Resizable Array or Growable Array
- `Duplicates allowed`
- `Insertion order Preserved`
- `Null Insertion possible`
- `Heterogenous objects allowed`
- Low performance than arrayList
- Introduced in 1.0, reengineered in 1.2 as Legacy class
- Has `Enumeration` loop option
- Implements Serializable, Cloneable and RandomAccess interfaces.
- `Every Method is Synchronized` so `Thread safe`
- Vector is the Best Choice if Our Frequent Operation is Retrieval.
- Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle
- Default Initial Capacity "10"
- New Capacity = Current Capacity * 2
- `Vector v = new Vector()`
- `Vector v = new Vector(int initialCapacity)`
- `Vector v = new Vector(int initialCapacity, int incrementalCapacity);`
- `Vector v = new Vector(Collection c)`

Methods of Vector

1. To Add Elements:

- add(Object o) : Collection
- add(int index, Object o) : List
- addElement(Object o) : Vector

2. To Remove Elements:

- remove(Object o) : Collection
- remove(int index) : List
- removeElement(Object o) : Vector
- removeElementAt(int index) : Vector
- clear() : Collection
- removeAllElements() : Vector

3. To Retrive Elements:

- Object get(int index) : List
- Object elementAt(int index) : Vector
- Object firstElement() : Vector
- Object lastElement() : Vector

4. Some Other Methods:

- int size()
- int capacity()
- Enumeration element()

Stack

- LIFO principle
- Subclass of Vector class
- `Stack lst = new Stack()`

Methods of Stack

- boolean empty()
- E peek() - Return Top of the Stack without Removal
- E pop() - Remove and Return Top of the Stack
- E push(E element)
- int search(Object o) - `Returns Offset if the Element is Available Otherwise Returns -1`

Set

- Interface - child of Collection Interface
- `Duplicates NOT allowed`
- `Insertion order NOT Preserved`
- HashSet, LinkedHashSet are the Implementation classes
- Has only methods of Collection, `NO new method`

Comparison Table of Set implemented Classes:

Property	HashSet	LinkedHashSet	TreeSet
Underlying Data Structure	Hashtable	Hashtable and LinkedList	Balanced Tree
Insertion Order	Not Preserved	Preserved	Not Preserved
Sorting Order	Not Applicable	Not Applicable	Applicable
Heterogeneous Objects	Allowed	Allowed	Not Allowed
Duplicate Objects	Not Allowed	Not Allowed	Not Allowed
null Acceptance	Allowed (Only Once)	Allowed (Only Once)	For Empty TreeSet as the 1st Element null Insertion is Possible. In all Other Cases we will get NullPointerException.

HashSet

- Class - child of Set Interface
- Uses Hash Table data structure
- `Insertion order NOT Preserved` - But, `Inserted based on Hash Code`
- `Heterogenous objects allowed`
- `Null Insertion possible`
- `Provides efficient searching`
- *If you do add and try to insert duplicate object - Returns false and not error*
- HashSet implements Serializable and Cloneable Interfaces but Not RandomAccess.
- If Our `Frequent Operation is Search Operation, then HashSet is the Best Choice`
- Default capacity - "16"
- After 75% is filled, it grows dynamically - Fill Ratio/ Load factor - 0.75
- Load Factor/Fill Ratio 0.75 Means After Filling 75% Automatically a New HashSet Object will be Created.
- `HashSet hset = new HashSet()`
- `HashSet hset = new HashSet(int capacity)`
- `HashSet hset = new HashSet(int capacity, float loadFactor)`
- `HashSet hset = new HashSet(Collection c)`
- hset.add("Z"); System.out.println(hset.add("Z")); // false and not error

LinkedHashSet

- SubClass - subclass of HashSet Class
- Uses `Hash Table + Doubly Linked List` data structure
- `_Insertion order Preserved_`

- `Heterogenous objects allowed`
- `Null Insertion possible`
- `Provides efficient searching`
- *If you do add and try to insert duplicate object - Returns false and not error*
- Default capacity - "16"
- After 75% is filled, it grows dynamically - Fill Ratio/ Load factor - 0.75
- `LinkedHashSet hset = new LinkedHashSet()`
- `LinkedHashSet hset = new LinkedHashSet(int capacity)`
- `LinkedHashSet hset = new LinkedHashSet(int capacity, float loadFactor)`
- `LinkedHashSet hset = new LinkedHashSet(Collection c)`
- We can Use LinkedHashSet and LinkedHashMap to *Develop Cache Based Applications* where Duplicates are Not Allowed and Insertion Order Must be Preserved.

SortedSet

- Interface - child of Set Interface
- `Duplicates NOT allowed`
- `Insertion order NOT Preserved`
- `Some Logical Sorted order` : Sorting can be Either Default Natural Sorting OR Customized Sorting Order (For String Objects Default Natural Sorting is Alphabetical Order. For Numbers Default Natural Sorting is Ascending Order.)
- 6 new methods

Methods of Sorted Set

- Object first() : first as per sorted order
- Object last() : last as per sorted order
- SortedSet headSet(E toElement) : Portion less than toElement
- SortedSet tailSet(E fromElement) : Portion greater than or equal to fromElement
- SortedSet subSet(E fromElement, E toElement) : Portion [fromElement, toElement)
- Comparator comparator() - returns comparator (used to order elements in this set), or null (if this set uses natural ordering of its elements)

NavigableSet

- Interface - child of SortedSet Interface
- `Duplicates NOT allowed`
- `Insertion order NOT Preserved`
- `Some Logical Sorted order`
- Defines methods for navigation purposes
- TreeSet is a Implementation class
- 6 new methods

Methods of NavigableSet

- ceiling(E e) : greatest ele >= e or null(if not present)
- floor(E e) : lowest <= e or null(if not present)
- higher(E e) : greater ele strictly > e or null(if not present)
- lower(E e) : lower ele strictly < e or null(if not present)
- pollFirst() : retrieves and removes first(lowest) ele or null(if not present)
- pollLast() : retrieves and removes last(highest) ele or null(if not present)
- `descendingSet()` : It Returns NavigableSet in Reverse Order.

TreeSet

- Interface - child of NavigableSet Interface
- TreeSet is a Implementation class
- Defines methods for navigation purposes
- `Duplicates NOT allowed`
- `Insertion order NOT Preserved`
- `Some Logical Sorted order` - Natural Order - Integer asc, String dictionary, StringBuffer not
- `_Heterogenous elements NOT allowed_` : have to be Homogenous else `classCastException`
- Implements Serializable and Cloneable Interfaces but Not RandomAccess Interface.
- `Null Insertion possible in Empty TreeSet only once, not in Non-empty/Null Empty TreeSet till version 6` otherwise mostly not possible >6 version ??
- `TreeSet tset = new TreeSet()` : empty + sorted in natural order
- `TreeSet tset = new TreeSet(Comparator c)` : empty + sorted in specified comparator order : Comparator c object, either in same class or diff
- `TreeSet tset = new TreeSet(Collection c)` : Collection ele + sorted in natural order
- `TreeSet tset = new TreeSet(SortedSet s)` : new TreeSet with same logic as s - same Collection ele + same sorted order
- By default before adding any element, compareTo() of `Comparable` Interface is called and then arranged - Wrapper Classes, Integer, String or by a `Comparator`.
- StringBuffer Class doesn't Implement Comparable Interface. Hence we get ClassCastException
- t.add(new StringBuffer("B")); System.out.println(t) // ClassCastException
- `Exception` : If we defining Our Own Sorting by Comparator then Objects Need Not be Homogeneous and Comparable. That is `we can Add Heterogeneous Non Comparable Objects to the TreeSet`.

Methods of TreeSet

- TreeSet tset = new TreeSet(Comparator c)

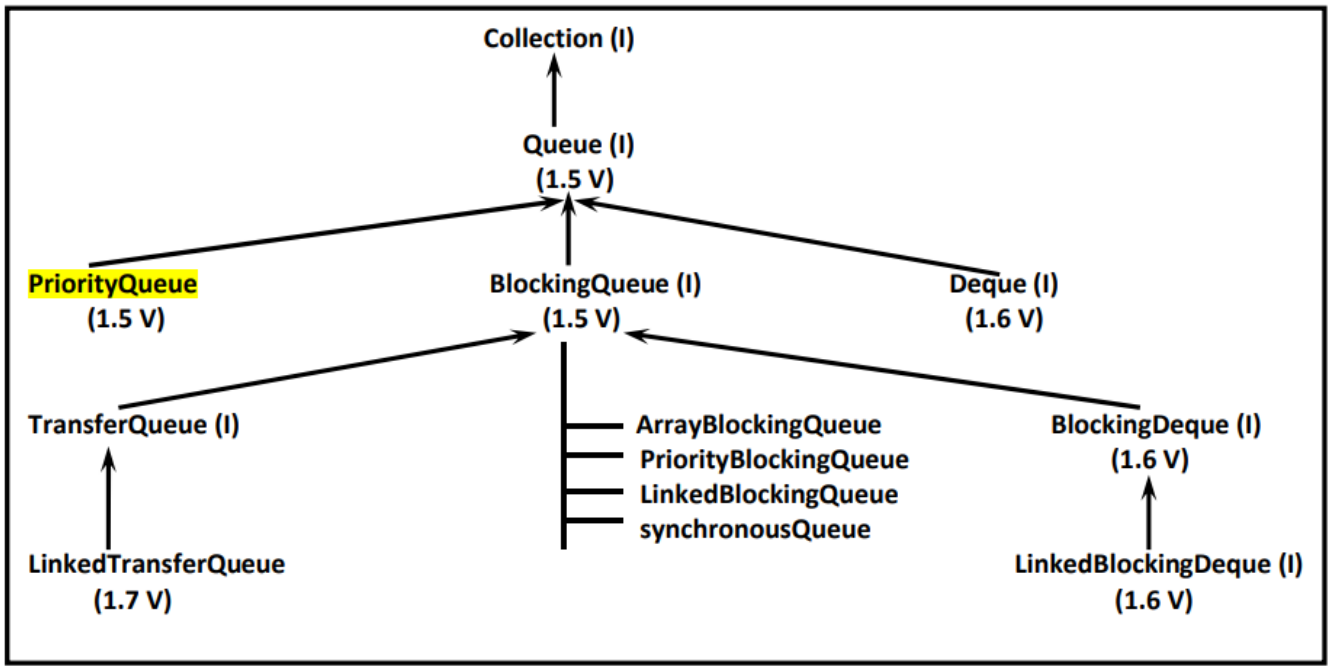
Queue

- Interface - child of Collection Interface
- `Duplicates allowed`
- `Null Insertion NOT allowed`
- `Heterogenous elements allowed`

- Orders in FIFO principle - But Based on Our Requirement we can Implement Our Own Priorities Also (PriorityQueue)
- LinkedList based Implementation of Queue always follows FIFO Order
- If we want to `Represent a Group of Individual Objects Prior to Processing then we should go for Queue`.
- Eg: Before sending a Mail we have to Store All MailID's in Some Data Structure and in which Order we added MailID's in the Same Order Only Mails should be delivered (FIFO). For this Requirement Queue is Best Suitable.
- From 1.5 Version onwards LinkedList also implements Queue Interface
- PriorityQueue, BlockingQueue -> PriorityBlockingQueue, LinkedBlockingQueue are the Implementation classes

Methods of Queue

- `Returns special value(null or false) ON Fail`
- boolean offer(E e)
- E poll()
- E peek() : view element
- `Throws exception ON Fail`
- boolean add(E e)
- E remove()
- E element() : view element



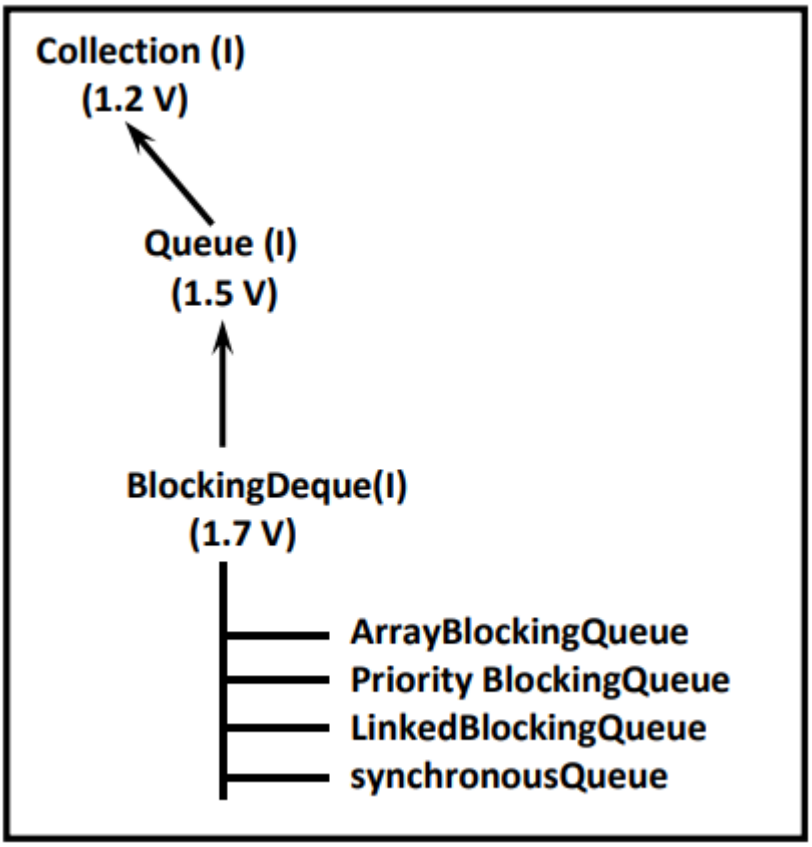
PriorityQueue

- PriorityQueue - Implementation class of Queue. Actually, Queue Interface -> AbstractQueue(implementing class of Queue) -> ProrityQueue
- `Null Insertion NOT allowed`

- `NOT Thread Safe`
- `Use PriorityBlockingQueue for Thread Safe`
- Natural Order of sorting by `Comparable` or sorted by a `Comparator` provided at Queue Construction Time.
- If we want to Represent a Group of Individual Objects Prior to processing according to Priority then we should go for PriorityQueue.
- Default capacity "11"
- `PriorityQueue pq = new PriorityQueue()`
- `PriorityQueue pq = new PriorityQueue(Collection c)`
- `PriorityQueue pq = new PriorityQueue(int capacity)`
- `PriorityQueue pq = new PriorityQueue(int capacity, Comparator c)`
- `PriorityQueue pq = new PriorityQueue(SortedSet s)`
- `PriorityQueue pq = new PriorityQueue(PriorityQueue p)`
- Offer order - Can be any order
- `Poll Order - Natural Sorting Order`

BlockingQueue

- Child interface of Queue
- It is a `Thread Safe` Collection.
- It is a specially designed Collection Not Only to Store Elements but also `Supports Flow Control by Blocking Mechanism`.
- If Queue is Empty `take()` (Retrieval Operation) will be Blocked until Queue will be Updated with Items.
- `put()` will be blocked if Queue is Full until Space Availability.
- This Property Makes BlockingQueue `Best Choice for Producer Consumer Problem`. When One Thread producing Items to the Queue and the Other Thread consuming Items from the Queue

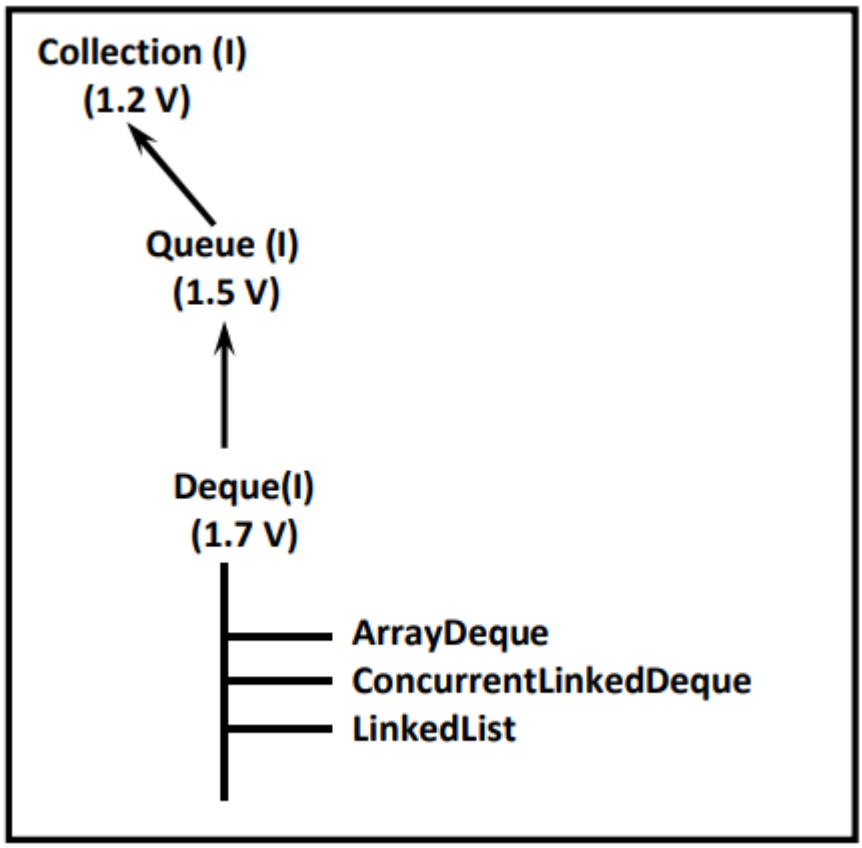


TransferQueue

- In BlockingQueue we can Only Put Elements into the Queue and if Queue is Full then Our put() will be blocked until Space is Available.
- But in TransferQueue we can also `Block until Other Thread receiving Our Element`. Hence this is the Behavior of `transfer()`.
- In BlockingQueue we are Not required to wait until Other Threads Receive Our Element but in TransferQueue we have to `wait until Some Other Thread Receive Our Element`.
- TrasferQueue is the Best Choice for `Message Passing Application` where Guarantee for the Delivery.

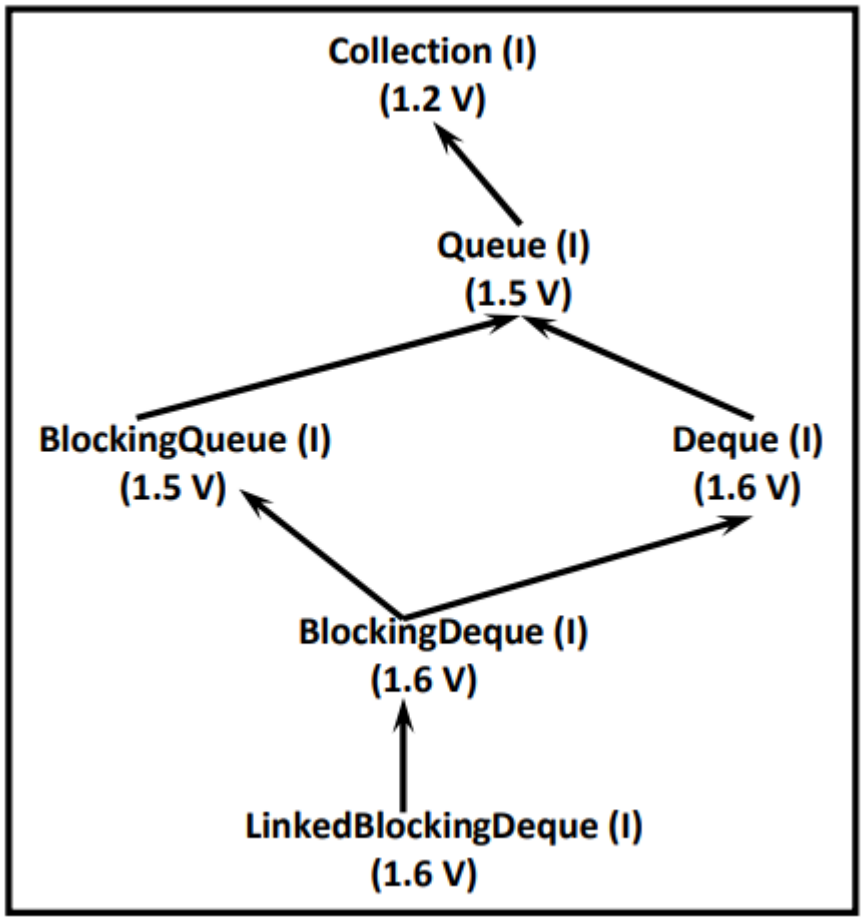
Deque

- Deque Means Double Ended Queue



BlockingDeque

- Child Interface of BlockingQueue and Deque.
- It is a Simple Deque with Blocking Operations but wait for the Deque to become Non Empty for Retrieval Operation and wait for Space to Store Element.



Map

- Interface - `NOT a child of Collection Interface`
- Represents group of objects as `key-value pairs`
- Key value both are objects - `Key unique`
- Duplicate Keys are Not allowed. But Values can be Duplicated.
- Each Key - Value Pair is Called an Entry
- HashMap -> LinkedHashMap, WeakHashMap, IdentityHashMap, Hashtable-> Properties, SortedMap-> NavigableMap-> TreeMap are the Implementation classes
- All map implementation classes `should provide two "Standard" constructors` : ??

1. A void `no arguments` constructor which creates an empty map
2. A constructor with a `Single argument` of type Map, which creates a new map with the same key-value mappings as its argument.

Methods of Map

- Object put(Object key, Object value) : inserts an entry in this map : To Add One Key - Value Pair. If the specified Key is Already Available then Old Value will be Replaced with New Value and Returns Old Value.
- Object putAll(Map map) : inserts specified map in this map
- Object remove(Object key) : deletes an entry for specified key
- Object get(Object key) : returns the value for specified key

- boolean containsKey(Object key) :
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- boolean isEmpty()
- int size()
- void clear()
- Collection Views of Map -
- Set keySet() : Set view containing `all the keys`
- Set entrySet() : Set view containing `all the keys and values`
- Collection values()

Map.Entry

- Inner Interface of Map Interface
- A map entry(key-value pair)
- Without existing Map Object there is No Chance of existing Entry Object. Hence Interface Entry is Define Inside Map Interface.
- 2 methods - getKey() and getValue() and setValue(Object new) ??

Methods of Entry

- Object getKey() : used to obtain key
- Object getValue() : used to obtain value
- Object setValue(Object new)

HashMap

- Hash Table used as data structure
- `Contains only Unique keys`
- `Heterogeneous Objects allowed` for Both Keys and Values.
- `Insertion Order NOT preserved` and `it is based on hash code of the keys`.
- `NULL value allowed ONCE for keys`
- It may have *one null key and multiple null values* - Next attempt to enter null as key, it simply discards without any error
- Initial capacity "16"
- Default Fill Ratio/loadFactor = "0.75"
- `HashMap hmap = new HashMap()`
- `HashMap hmap = new HashMap(int capacity)`
- `HashMap hmap = new HashMap(int capacity, float loadFactor)`
- `HashMap hmap = new HashMap(Map<? extends K,? extends V> m)`
- Non-Synchronized by default.
- Get Synchronized Version of HashMap by using the following Method of Collections Class

```
HashMap hmap = new HashMap();
HashMap h = Collections.synchronizedMap(hmap);
// hmap - Non - Synchronized Version
// h - Synchronized Version
```

Methods of HashMap

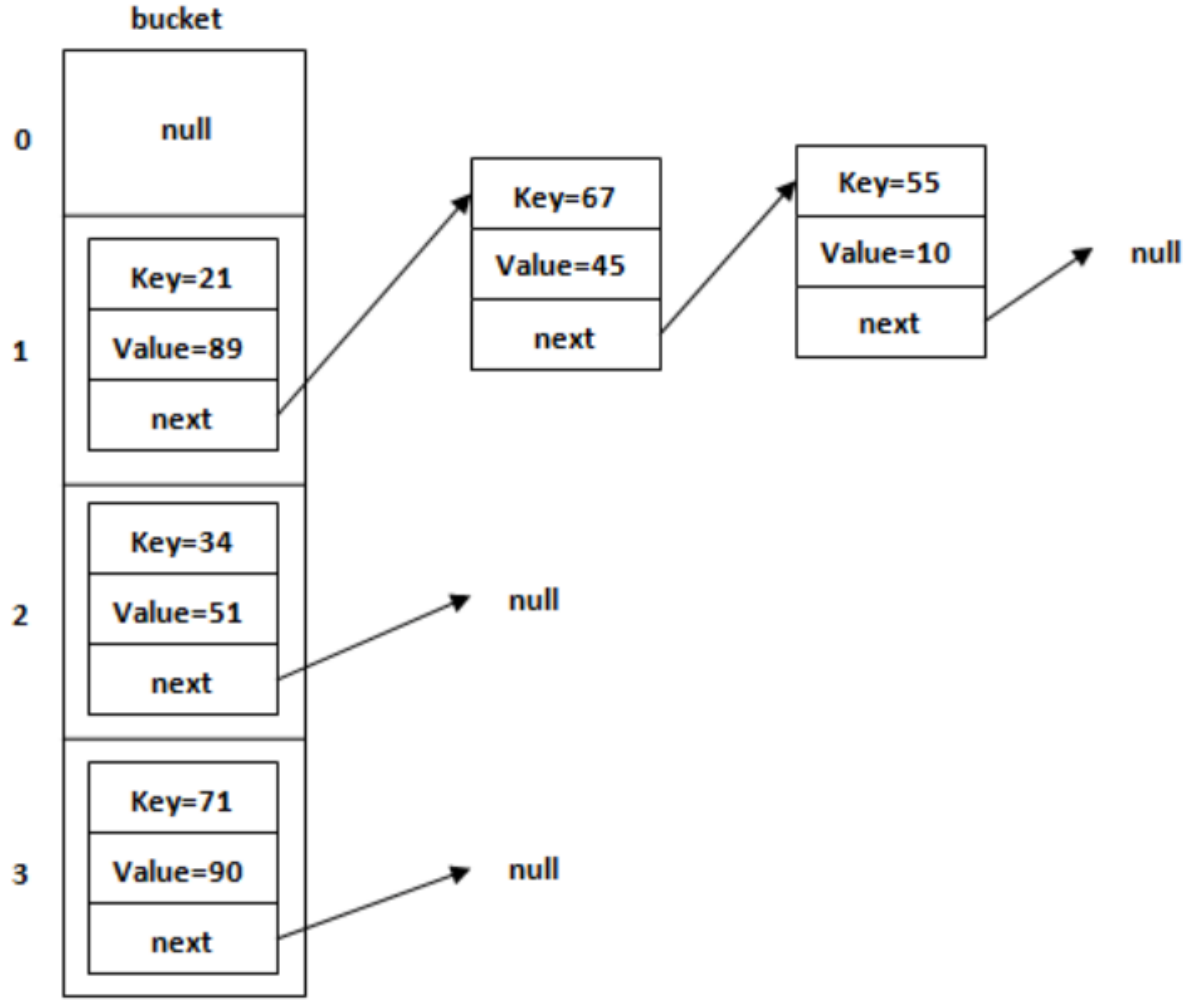
- int size() - # of key-value pairs
- boolean isEmpty()
- V get(Object key) : returns the value for specified key or null if no mapping for the key. The integer value helps in indexing and faster searches.

Working of HashMap

- Hashing - It is the process of converting an object into an integer value.
- HashMap contains an array of the nodes, and the node is represented as a class.
- Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.
- There are four fields in HashMap / Node:

1. int hash
2. K key
3. V value

4. Node<K,V> next



Index = hashcode(Key) & (size-1)

LinkedHashMap

- Subclass of HashMap
- `Insertion order preserved`
- Slower insertion and deletion
- Hybrid DS - DLL + Hash Table
- We can Use LinkedHashSet and LinkedHashMap to *Develop Cache Based Applications* where Duplicates are Not Allowed and Insertion Order Must be Preserved.

HashMap	LinkedHashMap
The Underlying Data Structure is Hashtable.	The Underlying Data Structure is Combination of Hashtable and LinkedList.
Insertion is Not Preserved.	Insertion Order is Preserved.
Introduced in 1.2 Version.	Introduced in 1.4 Version.

IdentityHashMap

- It is exactly same as HashMap, except the following difference.
- In HashMap JVM will Use .equals() to Identify Duplicate Keys, which is Meant for Content Comparision.
- In IdentityHashMap JVM will uses == Operator to Identify Duplicate Keys, which is Meant for Reference Comparison.

```
// HashMap - equals() - Content Comparison.
hmap.put(new Integer(10),"Pawan");
hmap.put(new Integer(10),"Kalyan");
System.out.println(m); //{10=Kalyan} // Because I1 and I2 are Duplicate as
I1.equals(I2) Returns true.

// IdentityHashMap - == - Reference Comparison
ihmap.put(new Integer(10),"Pawan");
ihmap.put(new Integer(10),"Kalyan");
System.out.println(m); //{10=Pawan, 10=Kalyan} // Because I1 and I2 are Not
Duplicate as I1 == I2 Returns false.
```

WeakHashMap

- It is exactly same as HashMap Except the following difference :
- In Case of HashMap, HashMap dominates Garbage Collector. That is if Object doesn't have any Reference Still it is Not Eligible for Garbage Collector if it is associated with HashMap.
- In Case of WeakHashMap, Garbage Collector Dominates WeakHashMap.That is if an Object doesn't contain any References then it is Always Eligible for GC Even though it is associated with WeakHashMap.

```
import java.util.HashMap;
class WeakHashMapDemo {
        public static void main(String[] args) throws InterruptedException {
        HashMap m = new HashMap();
        Temp t = new Temp();
        m.put(t, "Durga");
        System.out.println(m);
        t = null;
        System.gc();
        Thread.sleep(5000);
        System.out.println(m);
        }
    }
    class Temp {
        public String toString() {
            return "temp";
        }
        public void finalize() {
            System.out.println("finalize() Called");
```

```
        }
}

// With HashMap
{temp=Durga}
{temp=Durga}
// With WeakHashMap
{temp=Durga}
finalize() Called
```

Hashtable

- `_Thread safe_` version of HashMap
- `Duplicate Keys NOT Allowed`
- `Insertion Order NOT Preserved`
- Unsorted and Unordered
- `NO Sorting`
- `Order NOT preserved
- `NULL value not allowed` for Both Key and Values. Otherwise we will get Runtime Exception Saying NullPointerException.unlike HashMap
- Default Initial Capacity "11"
- Default Fill Ratio "0.75"
- `Hashtable h = new Hashtable();` * `Hashtable h = new Hashtable(int initialcapacity);`
- `Hashtable h = new Hashtable(int initialcapacity, float fillRatio);`
- `Hashtable h = new Hashtable(Map m);`
- Table data print : Top to Bottom, Right to Left

Differences between and HashMap and Hashtable:

HashMap	Hashtable
No Method Present in HashMap is Synchronized.	Every Method Present in Hashtable is Synchronized.
At a Time Multiple Threads are allowed to Operate on HashMap Object simultaneously and Hence it is Not Thread Safe.	At a Time Only One Thread is allowed to Operate on the Hashtable Object and Hence it is Thread Safe.
Relatively Performance is High.	Relatively Performance is Low.
null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values. Otherwise we will get NPE.
Introduced in 1.2 Version and it is Non – Legacy.	Introduced in 1.0 Version and it is Legacy.

Properties

- SubClass - Child of Hashtable

- Properties can be used to Represent a Group of Key – Value Pairs where Both Key and Value should be String Type.
- In Our Program if `anything which Changes Frequently` (Like Database User Name, Password, Database URLs Etc) Never Recommended to Hard Code in Java Program. Because for Every Change in Source File we have to Recompile, Rebuild and Redeploying Application and Sometimes Server Restart Also Required, which Creates Business Impact to the Client.
- To Overcome this Problem we have to Configure Such Type of Properties in Properties File.
- The Main Advantage in this Approach is if a there is a Change in Properties File, to Reflect that Change Just Redeployment is Enough, which won't Create any Business Impact.
- We can Use Properties Object to Hold Properties which are coming from Properties File.
- `Properties p = new Properties();`

Methods of Properties

- public String getProperty(String pname); To Get the Value associated with specified Property.
- public String setProperty(String pname, String pvalue); To Set a New Property.
- public Enumeration propertyNames(); It Returns All Property Names.
- public void load(InputStream is); To Load Properties from Properties File into Java Properties Object.
- public void store(OutputStream os, String comment); To Store Properties from Java Properties Object into Properties File.

SortedMap

- Interface - child of Map Interface
- `Some Sorting order`
- `Sorting logic applies on keys`

Methods of SortedMap

- Object firstKey();
- Object lastKey();
- SortedMap headMap(Object key)
- SortedMap tailMap(Object key)
- SortedMap subMap(Object key1, Object key2)
- Comparator comparator()

NavigableMap

- Interface - child of SortedMap Interface
- Defines methods for navigation purposes
- TreeMap is a Implementation class

Methods of NavigableMap

- ceiling(E e) : greatest ele >= e or null(if not present)
- floor(E e) : lowest <= e or null(if not present)
- higher(E e) : greater ele strictly > e or null(if not present)
- lower(E e) : lower ele strictly < e or null(if not present)
- pollFirst() : retrieves and removes first(lowest) ele or null(if not present)
- pollLast() : retrieves and removes last(highest) ele or null(if not present)
- `descendingSet()` : It Returns NavigableSet in Reverse Order.

TreeMap

- Class - child of NavigableMap Interface
- Red-Black tree based.
- `NOT Thread Safe`
- `Insertion order NOT Preserved`
- `Some Logical Sorted order` - Natural Order - Integer asc, String dictionary, (StringBuffer is not) or use comparator
- `_Heterogenous keys NOT allowed-Heterogenous Values allowed_` : have to be Homogenous else `classCastException`
- `Null Insertion possible in Empty TreeSet only once, not in Non-empty/Null Empty TreeSet till version 6` otherwise mostly not possible >6 version ??
- `TreeMap tmap = new TreeMap()` : empty + sorted in natural order
- `TreeMap tmap = new TreeMap(Comparator c)` : empty + sorted in specified comparator order : Comparator c object, either in same class or diff
- `TreeMap tmap = new TreeMap(Collection c)` : Collection ele + sorted in natural order
- `TreeMap tmap = new TreeMap(SortedMap m)` : Inter Conversion between Map Objects.
- `TreeMap tmap = new TreeMap(Map m)` : Inter Conversion between Map Objects.
- By default before adding any element, compareTo() of `Comparable` Interface is called and then arranged - Wrapper Classes, Integer, String or by a `Comparator on keys`.

Comparable - Sorting

- Interface
- `default sorting`

Comparator - Sorting

- Interface
- `Customized sorting`

Utility Classes

- Collections
- Arrays

Legacy Classes

- Vector

- Stack

1. Enumeration (I)
2. Dictionary (Abstract Class)
3. Vector (Concrete Class)
4. Stack (Concrete Class)
5. Hashtable (Concrete Class)
6. Properties (Concrete Class)

Methods

Methods of Collection

- boolean add(Object o); - l.add("Red")
- boolean addAll(Collection c); - l.addAll(lst)
- boolean remove(Object o); - l.remove("Red")
- boolean removeAll(Collection c); - l.removeAll(lst)
- boolean retainAll(Collection c)); - l.retainAll(lst) : To Remove All Objects Except those Present in c.
- int size(); - l.size()
- boolean contains(Object o); - l.contains("Red")
- boolean containsAll(Collection c) - l.containsAll(lst)
- void clear(); - l.clear()
- boolean isEmpty(); - l.isEmpty()
- Object[] toArray()
- Iterator iterator()

```
Iterator itr = lst.iterator();
while(itr.hasNext()){
    Sout(itr.next());
}
```

Methods of List

- void add(int index, E element) : shifts and insert
- E set(int index, E element) : replace and insert
- E get(int index)
- int indexOf(Object o) : first Occurence : if -1: element not present
- int lastIndexOf(Object o) : last Occurence : if -1: element not present
- E remove(int index)
- list subList(int fromIndex, int toIndex) : returns portion of list (from,to]

Methods of LinkedList

- void addFirst(Object o)
- void addLast(Object o)
- Objects.getFirst()
- Objects.getLast()
- Objects.removeFirst()
- Objects.removeLast()

Methods of Vector

1. To Add Elements:

- add(Object o) : Collection
- add(int index, Object o) : List
- addElement(Object o) : Vector

2. To Remove Elements:

- remove(Object o) : Collection
- remove(int index) : List
- removeElement(Object o) : Vector
- removeElementAt(int index) : Vector
- clear() : Collection
- removeAllElements() : Vector

3. To Retrive Elements:

- Object get(int index) : List
- Object elementAt(int index) : Vector
- Object firstElement() : Vector
- Object lastElement() : Vector

4. Some Other Methods:

- int size()
- int capacity()
- Enumeration element()

Methods of Stack

- boolean empty()
- E peek() - Return Top of the Stack without Removal
- E pop() - Remove and Return Top of the Stack
- E push(E element)
- int search(Object o) - `Returns Offset if the Element is Available Otherwise Returns -1`

Methods of Sorted Set

- Object first() : first as per sorted order
- Object last() : last as per sorted order
- SortedSet headSet(E toElement) : Portion less than toElement
- SortedSet tailSet(E fromElement) : Portion greater than or equal to fromElement
- SortedSet subSet(E fromElement, E toElement) : Portion [fromElement, toElement)
- Comparator comparator() - returns comparator (used to order elements in this set), or null (if this set uses natural ordering of its elements)

Methods of NavigableSet

- ceiling(E e) : greatest ele >= e or null(if not present)
- floor(E e) : lowest <= e or null(if not present)
- higher(E e) : greater ele strictly > e or null(if not present)
- lower(E e) : lower ele strictly < e or null(if not present)
- pollFirst() : retrieves and removes first(lowest) ele or null(if not present)
- pollLast() : retrieves and removes last(highest) ele or null(if not present)
- `descendingSet()` : It Returns NavigableSet in Reverse Order.

Methods of TreeSet

- TreeSet tset = new TreeSet(Comparator c)

Methods of Queue

- `Returns special value(null or false) ON Fail`
- boolean offer(E e)
- E poll()
- E peek() : view element
- `Throws exception ON Fail`
- boolean add(E e)
- E remove()
- E element() : view element

Methods of Entry

- Object getKey() : used to obtain key
- Object getValue() : used to obtain value
- Object setValue(Object new)

Methods of Map

- Object put(Object key, Object value) : inserts an entry in this map : To Add One Key - Value Pair. If the specified Key is Already Available then Old Value will be Replaced with New Value and Returns Old Value.
- Object putAll(Map map) : inserts specified map in this map
- Object remove(Object key) : deletes an entry for specified key
- Object get(Object key) : returns the value for specified key
- boolean containsKey(Object key) :
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- boolean isEmpty()
- int size()
- void clear()
- Collection Views of Map -
- Set keySet() : Set view containing `all the keys`
- Set entrySet() : Set view containing `all the keys and values`
- Collection values()

Methods of HashMap

- int size() - # of key-value pairs
- boolean isEmpty()
- V get(Object key) : returns the value for specified key or null if no mapping for the key

Methods of SortedMap

- Object firstKey();
- Object lastKey();
- SortedMap headMap(Object key)
- SortedMap tailMap(Object key)
- SortedMap subMap(Object key1, Object key2)
- Comparator comparator()

Methods of NavigableMap

- ceiling(E e) : greatest ele >= e or null(if not present)
- floor(E e) : lowest <= e or null(if not present)
- higher(E e) : greater ele strictly > e or null(if not present)
- lower(E e) : lower ele strictly < e or null(if not present)
- pollFirst() : retrieves and removes first(lowest) ele or null(if not present)
- pollLast() : retrieves and removes last(highest) ele or null(if not present)
- `descendingSet()` : It Returns NavigableSet in Reverse Order.

Methods of Properties

- public String getProperty(String pname); To Get the Value associated with specified Property.
- public String setProperty(String pname, String pvalue); To Set a New Property.
- public Enumeration propertyNames(); It Returns All Property Names.
- public void load(InputStream is); To Load Properties from Properties File into Java Properties Object.
- public void store(OutputStream os, String comment); To Store Properties from Java Properties Object into Properties File.

Cursors

- Interface actually, here its a way to retrieve data from collection object one by one
- 3 cursors in java
- Facility Order :- `ListIterator > Iterator > Enumeration`

Comparison Table of 3 Cursors:

Property	Enumeration	Iterator	ListIterator
Applicable For	Only Legacy Classes	Any Collection Objects	Only List Objects
Movement	Single Direction (Only Forward)	Single Direction (Only Forward)	Bi-Direction
How To Get	By using elements()	By using iterator()	By using listIterator() of List (I)
Accessability	Only Read	Read and Remove	Read , Remove, Replace And Addition of New Objects
Methods	hasMoreElements() nextElement()	hasNext() next() remove()	9 Methods
Is it legacy?	Yes (1.0 Version)	No (1.2 Version)	No (1.2 Version)

1. Enumeration

- Introduced in 1.0 for legacy classes - Vector, stack
- Enumeration Concept is Applicable Only for Legacy Classes and it is Not a Universal Cursor
- Limitations -

1. `Forward Cursor` (Single direction)
2. `Only read operation available`
3. `Unable to add or replace any element`

```
public interface Enumeration<E>{
    boolean hasMoreElements();
    E nextElement();
}

public Enumeration elements()
```

- `v` - vector object (Vector v = new Vector())
- `Enumeration` - Interface
- `elements()` - Method of collection classes
- `v.elements()` - Object of enumeration implementing class
- `Enumeration e` - Reference of enumeration interface
- `e.hasMoreElements()` - Calling method - gives boolean res if ele is present or not
- `e.nextElement()` - Calling method - give ele
- Have to typecast based on object type

```
Enumeration e = v.elements();
while(e.hasMoreElements()){
    Sout(e.nextElement())
    String str = (String)e.nextElement();
}
```

2. Iterator

- Available for all collection implemented classes.
- We can Apply Iterator Concept for any Collection Object. Hence it is Universal Cursor.
- `void remove(); : Not availabe in Enumeration`
- Limitations -

1. `Forward Cursor` (Single direction)
2. `Only read and remove operation available`
3. `Unable to add or replace any element`

```
public interface Iterator<E>{
    boolean hasNext();
    Object next();
    void remove(); // Not availabe in Enumeration
}

public Iterator iterator()
```

- `Iterator` - Interface
- `iterator()` - Method of classes
- Have to typecast based on object type

```
Iterator itr = lst.iterator();
while(itr.hasNext()){
    Sout(itr.next())
    String str = (String)itr.next();
    if(str=="Abc"){ itr.remove() }
}
```

3. ListIterator

- Interface - Child Interface of iterator
- `Forward+Backward : BiDirectional Cursor`
- The `Most Powerful Cursor` is ListIterator
- Limitation -

1. It is `Only applicable for List Objects`

```
public interface ListIterator<E>{
    // Forward direction
    boolean hasNext();
    Object next();
    int nextIndex(); // Not availabe in Enumeration and Iterator
    // Forward direction
    boolean hasPrevious(); // Not availabe in Enumeration and Iterator
    Object previous(); // Not availabe in Enumeration and Iterator
    int previousIndex(); // Not availabe in Enumeration and Iterator
    //
    void remove(); // Not availabe in Enumeration
    void set(E e) // Replace & Insert - Not availabe in Enumeration and Iterator
    void add(E e) // Shift & Insert - Not availabe in Enumeration and Iterator
}

public ListIterator listIterator()
```

- `ListIterator` - Interface
- `listIterator()` - Method of classes
- Have to typecast based on object type

```
ListIterator litr = lst.listIterator();
while(litr.hasNext()){
    Sout(litr.next())
    String str = (String)itr.next();
    if(str=="Abc"){ litr.remove() }
    if(str=="A"){ litr.set("A1") } // Replace A with A1
    if(str=="B"){ litr.add("C") } // Add C after B
}
```

Sorting

Comparison of Comparable and Comparator:

Comparable	Comparator
Present in java.lang Package	Present in java.util Package
It is Meant for Default Natural Sorting Order.	It is Meant for Customized Sorting Order.
Defines Only One Method compareTo().	Defines 2 Methods *compare()* and *equals()*.
All Wrapper Classes and String Class implements Comparable Interface.	The Only implemented Classes of Comparator are *Collator* and *RuleBaseCollator*.

- If we are Depending on `Natural Sorting Order` then the Objects should be `Homogeneous and Comparable` otherwise we will get `ClassCastException`.
- If we are defining Our Own `Sorting by Comparator` then the Objects Need Not be `Homogeneous and Comparable`.

When we go for Comparable and When we go for Comparator: Comparable Vs Comparator:

- For Predefined Comparable Classes (Like String) Default Natural Sorting Order is Already Available. If we are Not satisfied with that we can Define Our Own Sorting by Comparator Object.
- For Predefine Non - Comparable Classes (Like StringBuffer) Default Natural Sorting Order is Not Already Available. If we want to Define Our Own Sorting we can Use Comparator Object.
- For Our Own Classes (Like Employee) the Person who is writing Employee Class he is Responsible to Define Default Natural Sorting Order implementing Comparable Interface.
- The Person who is using Our Own Class if he is Not satisfied with Default Natural Sorting Order he can Define his Own Sorting by using Comparator Object.
- If Person is satisfied with Default Natural Sorting Order then he can Use Directly Our Class.

Comparable

- Interface
- Comparable Interface Present in java.lang Package
- Has `1 method` - compareTo(Object o)
- `Natural Sorting Order`
- Wrapper Classes have implemented Comparable Interface and have compareTo(). So `only Wrapper Classes have natural sorted order` facility.(StringBuffer ahs not implemented so natural order isn't there and gives classCastException)
- `Default natural sorting works for` :

1. `Homogenous` Objects
2. `Comparable` Objects

- System.out.println("Z".compareTo(null)); //RE: java.lang.NullPointerException

```
interface Comparable{
    int compareTo(Object obj)
}
```

- Whenever we are Depending on Default Natural Sorting Order and if we are trying to Insert Elements then `Internally JVM will Call compareTo()` to Identify Sorting Order.

1. Returns –ve if and Only if obj1 has to Come Before obj2
2. Returns +ve if and Only if obj1 has to Come After obj2
3. Returns 0 if and Only if obj1 and obj2 are Equal

- Comparable : Default Natural Sorting Order | gives power to class to compare its obj
- Comparator : Customized Sorting Order | logic to custom sort

Comparator

- Interface in java.util.package
- Used for `Custom sorting order`
- Whenever we are implementing Comparator Interface, `Compulsory we should Provide Implementation for compare()`.
- Has `2 method` -

1. compare(Object obj1, Object obj2) - Compulsory
2. equals() - Optional

- `NOT compulsory to override equals()` Since equals() is already inherited from Object class

```
interface Comparator{
    int compare(Object obj1, Object obj2); // -ve if Obj1<Obj2, +ve if Obj1>Obj2,
Zero if Obj1==Obj2
    equals(); // Optional to implement
}

TreeSet t = new TreeSet(new MyComparator());
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        }}
```

- If we are Not Passing Comparator Object as an Argument then Internally JVM will Call compareTo(), which is Meant for Default Natural Sorting Order (Ascending Order)

- `If we are Passing Comparator Object then JVM will Call compare() Instead of compareTo()`, which is Meant for Customized Sorting.

- Various Possible Implementations of compare():

```
public int compare(Object obj1, Object obj2) {
    Integer I1 = (Integer)obj1;
    Integer I2 = (Integer)obj2;
    return I1.compareTo(I2); //[0, 5, 10, 15, 20] Ascending Order
    return -I1.compareTo(I2); //[20, 15, 10, 5.0] Descending Order
```

```
        return I2.compareTo(I1); //[20, 15, 10, 5.0]
        return -I2.compareTo(I1); //[0, 5, 10, 15, 20]
        return +1; //[10, 0, 15, 5, 20, 20] Insertion Order
        return -1; //[20, 20, 5, 15, 0, 10] Reverse of Insertion Order
        return 0; //[10] Only 1st Inserted Element Present And All Remaining Elements
    Treated as Duplicates
    }
```

Collections

- Class in java.util package
- `Defines Several Utility Methods for Collection Objects`.
- It consists exclusively of static methods that operate on collections or return collections.
- `Searching Points` :-
- Internally the Above Search Methods will Use Binary Search Algorithm.
- `Before performing Search` Operation `Compulsorily List should be Sorted`. Otherwise we will get Unpredictable Results.
- Successful Search Returns Index.
- Unsuccessful Search Returns Insertion Point. (Insertion Point is the Location where we can Insert the Target Element in the SortedList.)
- If the List is Sorted according to Comparator then at the Time of Search Operation, Also we should Pass the Same Comparator Object. Otherwise we will get Unpredictable Results.
- Note: For the List of n Elements | A B Z : 0 1 2 ??
    1. Successful Result Range: 0 To n-1
    2. Unsuccessful Result Range: -(n+1) To -1
    3. Total Result Range: -(n+1) To n-1
- Eg: For the List of 3 Elements | A B Z : -1 -2 -3
    1. Range of Successful Search: 0 To 2
    2. Range of Unsuccessful Search: -4 To -1
    3. Total Result Range: -4 To 2

Methods of Collections

- public static void sort(List lst) - List should Not contain null Otherwise we will get NullPointerException.
- public static void sort(List lst, Comparator c)
- public static int binarySearch(List lst, T key) : returns index
- public static int binarySearch(List lst, T key, Comparator c) : returns index
- public static void reverse(List lst)
- public static void shuffleswap(List lst)
- public static void sort(List lst, int index1, int index2)
- public static void copy(List dest, List src)
- public static T min(Collection c)
- public static T min(Collection c, Comparator comp)
- public static T max(Collection c)

- public static T max(Collection c, Comparator comp)
- reverse() Vs reverseOrder():
- We can Use reverse() to Reverse Order of Elements of List.
- We can Use reverseOrder() to get Reversed Comparator.
- Comparator c1 = Collecctions.reverseOrder(Comparator c);
- c1 - descending, c - ascending

```
Collections.sort(lst);
Collections.sort(lst,cmp);
```

Arrays

- Class in java.util package
- `Defines Several Utility Methods for Array`
- It consists exclusively of static methods for manipulating array - for different primittive types

Methods of Arrays

- Polymorphic versions of sort() :-

- public static void sort(primitive[] p); To Sort According to Natural Sorting Order.
- public static void sort(primitive[] a, int fromIndex, int toIndex)
- public static void sort(Object[] o); To Sort According to Natural Sorting Order.
- public static void sort(Object[] o, Comparator c); To Sort According to Customized Sorting Order
- public static void sort(int[] a)
- public static void sort(int[] a, int f\romIndex, int toIndex)
- public static void sort(long[] a)
- public static void sort(long[] a, int fromIndex, int toIndex)

- Search :-

- public static int binarySearch(primitive[] p, primitive target); - If the Primitive Array Sorted According to Natural Sorting Order then we have to Use this Method.
- public static int binarySearch(Object[] a, Object target); - If the Primitive Array Sorted According to Natural Sorting Order then we have to Use this Method.
- public static int binarySearch(Object[] a, Object target, Comparator c); - If the Object Array Sorted According to Comparator then we have to Use this Method.
- public static int binarySearch(long[] a, long key) : returns index
- public static int binarySearch(long[] a, int fromIndex, int toIndex, long key) : returns index
- public static boolean equals(int[] a1, int[] a2) : if corresponding index ele are equal
- public static void fill(int[] a, int val)
- public static void fill(int[] a, int fromIndex, int toIndex2, int val)
- public static void copy(List dest, List src)
- public static T min(Collection c)
- public static T min(Collection c, Comparator comp)

- public static T max(Collection c)
- public static T max(Collection c, Comparator comp)
- `asList()` - Conversion of Array to List

```
Collections.sort(lst);
Collections.sort(lst,cmp);
```

Conversion of Array to List

- Arrays Class contains `asList()`
- this Method `won't Create an Independent List Object`, Just we are Viewing existing Array in List Form
- By using Array Reference if we Perform any Change Automatically that Change will be reflected to List Reference.
- Similarly by using List Reference if we Perform any Change Automatically that Change will be reflected to Array.
- By using List Reference if we are trying to Perform any Operation which `Varies the Size` then we will get Runtime Exception Saying `UnsuportedOperationException`.
- By using List Reference if we are trying to `Replace with Heterogeneous Objects` then we will get Runtime Exception Saying `ArrayStoreException`.
- `List lst = Arrays.asList(arr);`

Concurrent Collections

- Already existing Thread Safe Collections :-
- Vector
- Hashtable
- synchronizedList()
- synchronizedSet()
- synchronizedMap()
- Problems with traditional collections :

1. Performance wise Not Upto the Mark(Because for Every Operation Even for Read Operation Also Total Collection will be loaded by Only One Thread at a Time and it Increases waiting Time of Threads)
2. While One Thread iterating Collection, the Other Threads are Not allowed to Modify Collection Object simultaneously if we are trying to Modify then we will get `ConcurrentModificationException`.

```
while (itr.hasNext()) {
String s = (String)itr.next();
System.out.println(s);
al.add("D"); // RE: java.util.ConcurrentModificationException
}
```

- Hence these Traditional Collection Objects are Not Suitable for Scalable Multi Threaded Applications.
- Advantages of Thread Safe :

1. Concurrent Collections are `Always Thread Safe`.
2. When compared with Traditional Thread Safe Collections `Performance is More` because of different Locking Mechanism.
3. `While One Thread interacting` Collection the `Other Threads are allowed to Modify Collection` in Safe Manner.

- Hence Concurrent Collections `Never throw ConcurrentModificationException`.
- The Important Concurrent Classes are
- ConcurrentHashMap : Map -> ConcurrentMap -> ConcurrentHashMap
- CopyOnWriteArrayList
- CopyOnWriteArraySet

ConcurrentMap

- Interface
- 3 methods :-

1. `Object putIfAbsent(Object Key, Object Value)` : To Add Entry to the Map if the specified Key is Not Already Available.

```
ConcurrentHashMap m = new ConcurrentHashMap();
m.put(101, "Durga");
m.put(101, "Ravi");
System.out.println(m); //{101=Ravi}
m.putIfAbsent(101, "Siva");
System.out.println(m); //{101=Ravi}
```

- `put()` : If the Key is Already Available, Old Value will be replaced with New Value and Returns Old Value.
- `putIfAbsent()` : If the Key is Already Present then Entry won't be added and Returns Old associated Value. If the Key is Not Available then Only Entry will be added.

2. `boolean remove(Object key, Object value)` : Removes the Entry if the Key associated with specified Value Only.

```
ConcurrentHashMap m = new ConcurrentHashMap();
m.put(101, "Durga");
m.remove(101, "Ravi"); //Value Not Matched with Key So Not Removed
System.out.println(m); //{101=Durga}
m.remove(101, "Durga");
System.out.println(m); //{}
```

3. `boolean replace(Object key, Object oldValue, Object newValue)` : If the Key Value Matched then Replace with newValue

```
ConcurrentHashMap m = new ConcurrentHashMap();
m.put(101, "Durga");
m.replace(101, "Ravi", "Siva");
System.out.println(m); //{101=Durga} // Value Not Matched with Key So Not Replaced
m.replace(101, "Durga", "Ravi");
System.out.println(m); //{101=Ravi}
```

ConcurrentHashMap

- DS Used is Hashtable
- ConcurrentHashMap allows `Concurrent Read and Thread Safe Update Operations`.
- To Perform Read Operation Thread won't require any Lock. But to Perform Update Operation Thread requires Lock but it is the Lock of Only a Particular Part of Map (`Bucket Level Lock`).
- Instead of Whole Map Concurrent Update achieved by *Internally dividing Map into Smaller Portion* which is defined by *Concurrency Level*.
- Default Concurrency Level "16"
- Default Fill Ratio "0.75"
- That is ConcurrentHashMap Allows simultaneous Read Operation and *simultaneously 16 Write (Update) Operations*.
- `NULL insertion NOT Allowed for Both Keys and Values`.
- While One Thread iterating the Other Thread can Perform Update Operation and ConcurrentHashMap Never throw ConcurrentModificationException.
- `ConcurrentHashMap m = new ConcurrentHashMap();`
- `ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity);`
- `ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio);`
- `ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio, int concurrencyLevel);`
- `ConcurrentHashMap m = new ConcurrentHashMap(Map m);`
- If we Replace ConcurrentHashMap with HashMap then we will get ConcurrentModificationException.
- In the Case of ConcurrentHashMap iterator creates a Read Only Copy of Map Object and iterates over that Copy if any Changes to the Map after getting iterator it won't be affected/reflected.

```
ConcurrentHashMap m = new ConcurrentHashMap();
m.put(101, "A");
m.put(102, "B");
Iterator itr = m.keySet().iterator();
m.put(103, "C");
while (itr.hasNext()) {
Integer I1 = (Integer) itr.next();
System.out.println(I1+"............."+m.get(I1));
Thread.sleep(3000);
}
```

```
System.out.println(m);

// 102.............B ??
// 101.............A
// {103=C, 102=B, 101=A}
```

Difference between HashMap and ConcurrentHashMap

HashMap	ConcurrentHashMap
It is Not Thread Safe.	It is Thread Safe.
Relatively Performance is High because Threads are Not required to wait to Operate on HashMap.	Relatively Performance is Low because Some Times Threads are required to wait to Operate on ConcurrentHashMap.
While One Thread iterating HashMap the Other Threads are Not allowed to Modify Map Objects Otherwise we will get Runtime Exception Saying ConcurrentModificationException.	While One Thread iterating ConcurrentHashMap the Other Threads are allowed to Modify Map Objects in Safe Manner and it won't throw ConcurrentModificationException.
Iterator of HashMap is Fail-Fast and it throws ConcurrentModificationException.	Iterator of ConcurrentHashMap is Fail-Safe and it won't throws ConcurrentModificationException.
null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values. Otherwise we will get NullPointerException.
Introduced in 1.2 Version.	Introduced in 1.5 Version.

Difference between ConcurrentHashMap, synchronizedMap() and Hashtable

ConcurrentHashMap	synchronizedMap()	Hashtable
We will get Thread Safety without locking Total Map Object Just with Bucket Level Lock.	We will get Thread Safety by locking Whole Map Object.	We will get Thread Safety by locking Whole Map Object.
At a Time Multiple Threads are allowed to Operate on Map Object in Safe Manner.	At a Time Only One Thread is allowed to Perform any Operation on Map Object.	At a Time Only One Thread is allowed to Operate on Map Object.
Read Operation can be performed without Lock but write Operation can be performed with Bucket Level Lock.	Every Read and Write Operations require Total Map Object Lock.	Every Read and Write Operations require Total Map Object Lock.
While One Thread iterating Map Object, the Other Threads are allowed to Modify Map and we won't get ConcurrentModificationException.	While One Thread iterating Map Object, the Other Threads are Not allowed to Modify Map. Otherwise we will get ConcurrentModificationException	While One Thread iterating Map Object, the Other Threads are Not allowed to Modify Map. Otherwise we will get ConcurrentModificationException
Iterator of ConcurrentHashMap is Fail-Safe and won't raise ConcurrentModificationException.	Iterator of synchronizedMap is Fail-Fast and it will raise ConcurrentModificationException.	Iterator of synchronizedMap is Fail-Fast and it will raise ConcurrentModificationException.
null is Not allowed for Both Keys and Values.	null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values.
Introduced in 1.5 Version.	Introduced in 1.2 Version.	Introduced in 1.0 Version.

CopyOnWriteArrayList

- Class of List Interface
- `Thread Safe Version of ArrayList`
- `Update Operation will be performed on cloned Copy` there is No Effect for the Threads which performs Read Operation
- It is `Costly to Use` because for every Update Operation a cloned Copy will be Created.
- CopyOnWriteArrayList is the Best Choice if Several Read Operations and Less Number of Write Operations are required to Perform.
- `Insertion Order Preserved`
- `Duplicate Objects Allowed`
- `Heterogeneous Objects Allowed`
- `null Insertion Possible`
- It implements Serializable, Clonable and RandomAccess Interfaces.
- While One Thread iterating CopyOnWriteArrayList, the Other Threads are allowed to Modify and we `won't get ConcurrentModificationException`. That is `iterator is Fail Safe`.
- Iterator of ArrayList can Perform Remove Operation but `Iterator of CopyOnWriteArrayList can't Perform Remove Operation`. Otherwise we will get RuntimeException Saying `UnsupportedOperationException`.
- CopyOnWriteArrayList l = new CopyOnWriteArrayList();
- CopyOnWriteArrayList l = new CopyOnWriteArrayList(Collection c);
- CopyOnWriteArrayList l = new CopyOnWriteArrayList(Object[] a);

- Methods -

- `boolean addIfAbsent(Object o)` : The Element will be Added if and Only if List doesn't contain this Element
- `int addAllAbsent(Collection c)`: The Elements of Collection will be Added to the List if Elements are Absent and Returns Number of Elements Added.
- If we Replace CopyOnWriteArrayList with ArrayList then we will get ConcurrentModificationException.
- Every Update Operation will be performed on Separate Copy Hence After getting iterator if we are trying to Perform any Modification to the List it won't be reflected to the iterator.

```
CopyOnWriteArrayList l = new CopyOnWriteArrayList();
l.add("A");
l.add("B");
l.add("C");
Iterator itr = l.iterator();
l.add("D");
while (itr.hasNext()) {
String s = (String)itr.next();
System.out.println(s);
}

// A
```

```
// B
// C
```

- In the Above Program if we Replace CopyOnWriteArrayList with ArrayList then we will get RuntimeException: java.util.ConcurrentModificationException.

Differences between ArrayList and CopyOnWriteArrayList

ArrayList	CopyOnWriteArrayList
It is Not Thread Safe.	It is Not Thread Safe because Every Update Operation will be performed on Separate cloned Coy.
While One Thread iterating List Object, the Other Threads are Not allowed to Modify List Otherwise we will get ConcurrentModificationException.	While One Thread iterating List Object, the Other Threads are allowed to Modify List in Safe Manner and we won't get ConcurrentModificationException.
Iterator is Fail-Fsat.	Iterator is Fail-Safe.
Iterator of ArrayList can Perform Remove Operation.	Iterator of CopyOnWriteArrayList can't Perform Remove Operation Otherwise we will get RuntimeException: UnsupportedOperationException.
Introduced in 1.2 Version.	Introduced in 1.5 Version.

Differences between CopyOnWriteArrayList, synchronizedList() and vector()

CopyOnWriteArrayList	synchronizedList()	vector()
We will get Thread Safety because Every Update Operation will be performed on Separate cloned Copy.	We will get Thread Safety because at a Time List can be accessed by Only One Thread at a Time.	We will get Thread Safety because at a Time Only One Thread is allowed to Access Vector Object.
At a Time Multiple Threads are allowed to Access/ Operate on CopyOnWriteArrayList.	At a Time Only One Thread is allowed to Perform any Operation on List Object.	At a Time Only One Thread is allowed to Operate on Vector Object.
While One Thread iterating List Object, the Other Threads are allowed to Modify Map and we won't get ConcurrentModificationException.	While One Thread iterating , the Other Threads are Not allowed to Modify List. Otherwise we will get ConcurrentModificationException	While One Thread iterating, the Other Threads are Not allowed to Modify Vector. Otherwise we will get ConcurrentModificationException
Iterator is Fail-Safe and won't raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.
Iterator can't Perform Remove Operation Otherwise we will get UnsupportedOperationException.	Iterator can Perform Remove Operation.	Iterator can Perform Remove Operation.
Introduced in 1.5 Version.	Introduced in 1.2 Version.	Introduced in 1.0 Version.

CopyOnWriteArraySet

- Child Class of Set Interface

- `Thread Safe Version of ArrayList`
- Internally implemented by CopyOnWriteArrayList.
- `Insertion Order Preserved`.
- `Duplicate Objects NOT allowed`.
- Multiple Threads can Able to Perform Read Operation simultaneously but for `Every Update Operation a Separate cloned Copy` will be Created.
- As for Every `Update Operation` a Separate cloned Copy will be Created which is `Costly`. Hence if `Multiple Update Operation are required then it is Not recommended` to Use CopyOnWriteArraySet.
- While One Thread iterating Set the Other Threads are allowed to Modify Set and we `won't get ConcurrentModificationException`.
- Iterator of CopyOnWriteArraySet can Perform Only Read Operation and `won't Perform Remove Operation`. Otherwise we will get RuntimeException: `UnsupportedOperatonException`.

- Methods :-

- `CopyOnWriteArraySet s = new CopyOnWriteArraySet ();` : Creates an Empty CopyOnWriteArraySet Object.
- `CopyOnWriteArraySet s = new CopyOnWriteArraySet(Collection c);` : Creates CopyOnWriteArraySet Object which is Equivalent to given Collection Object.

- Methods :

- Methods Present in Collection and Set Interfaces are the Only Methods Applicable for CopyOnWriteArraySet and there are No Special Methods.

Differences between CopyOnWriteArraySet() and synchronizedSet()

CopyOnWriteArraySet()	synchronizedSet()
It is Thread Safe because Every Update Operation will be performed on Separate Cloned Copy.	It is Thread Safe because at a Time Only One Thread can Perform Operation.
While One Thread iterating Set, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException.	While One Thread iterating, the Other Threads are Not allowed to Modify Seta Otherwise we will get ConcurrentModificationException.
Iterator is Fail Safe.	Iterator is Fail Fast.
Iterator can Perform Only Read Operation and can't Perform Remove Operation Otherwise we will get RuntimeException Saying UnsupportedOperationException.	Iterator can Perform Both Read and Remove Operations.
Introduced in 1.5 Version.	Introduced in 1.7 Version.

Fail Fast Iterator

Fail Fast Iterator

- While One Thread iterating Collection if Other Thread trying to Perform any Structural Modification to the underlying Collection then immediately Iterator Fails by raising

`ConcurrentModificationExcepion`. Such Type of Iterators are Called Fail Fast Iterators.

- : Internally Fail Fast Iterator will Use Some `Flag named with MOD` to Check underlying Collection is Modified OR Not while iterating.

```
ArrayList l = new ArrayList();
l.add("A");
l.add("B");
Iterator itr = l.iterator();
while(itr.hasNext()) {
    String s = (String)itr.next();
    System.out.println(s); //A
    l.add("C"); // java.util.ConcurrentModificationException
}
```

- `ArrayList, Vector, HashMap, HashSet`

Fail Safe Iterator

- While One Thread iterating if the Other Threads are allowed to Perform any Structural Changes to the underlying Collection, Such Type of Iterators are Called Fail Safe Iterators.
- Fail Safe Iterators `won't raise ConcurrentModificationException` because Every Update Operation will be performed on `Separate cloned Copy`.
- Memory Problem

```
CopyOnWriteArraySet l = new CopyOnWriteArraySet();
l.add("A");
l.add("B");
Iterator itr = l.iterator();
while(itr.hasNext()) {
    String s = (String)itr.next();
    System.out.println(s); //A
    l.add("C");
}
```

- `ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet`

Differences between Fail Fast and Fail Safe Iterators:

Property	Fail Fast	Fail Safe
Does it through ConcurrentModificationException?	Yes	No
Is the Cloned Copy will be Created?	No	Yes
Memory Problems	No	Yes
Examples	ArrayList, Vector, HashMap, HashSet	ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet

Generics

- The main objective of Generics is to provide `Type-Safety` and to `resolve Type-Casting` problems.
- Generics concept is applicable `only at compile time`, at runtime there is no such type of concept.

- Problems :-

1. Type Safety

- We `can't provide guarantee for the type of elements` present inside collections that is collections are not safe to use with respect to type.
- By mistake if we are trying to add any other type we won't get any compile time error but the program may fail at runtime - `ClassCastException`

1. Type Casting

- In the case of collection at the time of retrieval compulsory we should perform type casting otherwise we will get compile time error.

```
String name = lst.get(0); // Compile Time Error - Incompatible  type
String name = (String)lst.get(0); // Type casting at retrieval time is mandatory
```

- Collections type casting is bigger headache

- Solution is Generics :-

1. To provide type safety to the collections.
2. To resolve type casting problems. `ArrayList<String> lst = new ArrayList<String>();`

- Collections concept applicable only for objects , Hence for the parameter type we can use any class or interface name but `not primitive value(type)`.Otherwise we will get compile time error.

No Polymorphism for parameter type

- Polymorphism concept is applicable only for the base type but `not for parameter type` [usage of parent reference to hold child object is called polymorphism].

```
ArrayList<String> l1 = new ArrayList<String>(); // Correct
List<String> l2 = new ArrayList<String>(); // Correct
Collection<String> l3 = new ArrayList<String>(); // Correct

ArrayList<Object> l4= new ArrayList<String>(); // Compile Time Error :
Imcompatible Types
```

Generic Classes

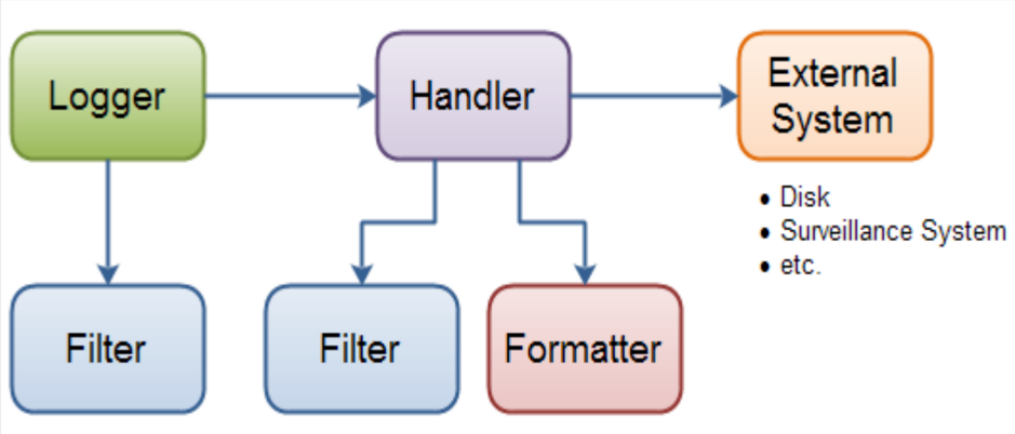
Logger

```
private final static Logger LOGGER = Logger.getLogger(MyClass.class.getName());
logger.addHandler(new ConsoleHandler());
```

log4j has three main components:

- loggers: captures the logging information.
- appenders: publishes the logging information to various preferred destinations.
- layouts: formats the logging information in different styles.

- How Java Logging API works?



Garbage Collection

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.
- The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects). - `Using newInstance(), clone()`
- There are 3 ways to make an object unreferenced :

1. `By nulling a reference`

```
Employee e=new Employee();
e=null;
```

2. `By assigning a reference to another`

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3. `By anonymous object`

```
new Employee();
```

finalize() method

- The finalize() method is invoked each time before the object is garbage collected.
- This method can be used to perform cleanup processing.
- This method is defined in Object class as: `protected void finalize(){}`

gc() method

- The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.
- `public static void gc(){}`
- Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

```
public class TestGarbage1{
 public void finalize(){
    System.out.println("object is garbage collected");
 }

 public static void main(String args[]){
    TestGarbage1 s1=new TestGarbage1();
```

```
    TestGarbage1 s2=new TestGarbage1();
    s1=null;
    s2=null;

    System.gc();
 }
}

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

---

20. JAVA 8 Features

1. lambda expression

- A lambda expression is an `anonymous function without a declaration`.
- Replacement of Anonymous inner class : If work of class was just to implement a method of Functional interface
- Only for Functional interface
- The `function which doesn't have the name,return type and access modifiers.`
- Lambda Expression also known as `anonymous functions or closures`.

```
public String str(String str) {
 return str;
}
// TO
(String str) -> return str;
// TO
str -> str;
```

- Anonymous inner class! = Lambda Expression
- Anonymous inner class can extend concrete class, can extend abstract class,can implement interface with any number of methods but Lambda expression can implement an interface with only single abstract method(FunctionalInterface).

```
Thread t = new Thread(new Runnable() {
    public void run() {
        for(int i=0; i<10; i++) {
            System.out.println("Child Thread");
        }
    }
});
// TO
Thread t = new Thread(() -> {
```

```
        for(int i=0; i<10; i++) {
            System.out.println("Child Thread");
        }
    });
```

Differences between anonymous inner classes and Lambda expression

Anonymous Inner class	Lambda Expression
It's a class without name	It's a method without name(anonymous function)
Anonymous inner class can extend Abstract and concreateclasses	lambda expression can't extend Abstract and concreate classes
Anonymous inner class can implement An interface that contains any number of Abstract methods	lambda expression can implement an Interface which contains single abstract method (FunctionalInterface)
Inside anonymous inner class we can Declare instance variables.	Inside lambda expression we can't Declare instance variables,whater the variables declare are simply acts as local variables.
Anonymous inner classes can be Instantiated	lambda expressions can't be instantiated
Inside anonymous inner class "this" Always refers current anonymous Inner class object but not outer class Object.	Inside lambda expression "this" Always refers current outer class object.that is enclosing class object.
Anonymous inner class is the best choice If we want to handle multiple methods.	Lambda expression is the best Choice if we want to handle interface With single abstract method (FuntionalInterface).
In the case of anonymous inner class At the time of compilation a separate Dot class file will be generated (outerclass$1.class)	At the time of compilation no dot Class file will be generated for Lambda expression.it simply convert in to private method outer class.
Memory allocated on demand Whenever we are creating an object	Reside in permanent memory of JVM (Method Area).

2. Functional Interface

- Only 1 abstract method
- Any number of default or static methods
- @FunctionalInterface annotation
- java.util.function pacakge

1. Predicate

`test() | 1 arg | boolean Output`

```
Predicate<String> checkLength = str->str.legth()>5;
sout(checkLength.test("abcd")) /// False
```

- Predicate Joining
- and(), or(), negate()

```
public static void m1(predicate<integer>p, int[] x) {
    for(int x1:x) {
        if(p.test(x1))
            System.out.println(x1);
    } }
psvm(){
int[] x = {0, 5, 10, 15, 20, 25, 30};
predicate<integer> p1 = i->i>10;
predicate<integer> p2=i -> i%2==0;

System.out.println("The Numbers Greater Than 10:");
m1(p1, x);
System.out.println("The Even Numbers Are:");
m1(p2, x);
System.out.println("The Numbers Not Greater Than 10:");
m1(p1.negate(), x);
System.out.println("The Numbers Greater Than 10 And Even Are:");
m1(p1.and(p2), x);
System.out.println("The Numbers Greater Than 10 OR Even:" );
m1(p1.or(p2), x);
}
```

- Predicate is a boolean valued function and(), or(), negate() are default methods present inside Predicate interface.

2. Consumer

`accept() | 1 arg | NO Output`

- Modifies data

```
class Person{
    private String name;
    public String getName(){return name;}
    public void setName(String name){this.name = name;}
}

psvm{
    Person p = new Person();
    Consumer<Person> setName = t->t.setName("Ayushi");
    setName.accept(p)
    sout(p.getName()); /// Ayushi
}
```

3. Function

`apply(arg, res) | 1 arg | Output`

```
Function<Integer, String> getInt = t->t*10+" data";
sout(getInt.apply(2)) ///20 data
```

4. Supplier

`get() | NO Input | Supply`

```
Supplier<Double> getRandomDouble = () -> Math.random();
sout(getRandomDouble.get()) /// 0.400
```

1. BiFunction : Represents a function that accepts two argument and produces a result.
2. BiConsumer : Represents an operation that accepts two input argument and returns no result.
3. BiPredicate : Represents a predicate (boolean-valued function) of two arguments.

3. Default method / Defender method / Virtual extension method and Static Method

- Implementation of methods is possible in interface now
- `default void show() {...}`
- we can declare default concrete methods also inside interface,which are also known as Defender methods.
- Problem of `multiple inheritance` : Two interfaces can contain default method with same signature then there may be a chance of ambiguity problem(diamond problem) to the implementation class. To overcome this problem `compulsory we should override default method in the implementation class` otherwise we get compiletime error.
- Override Default() : Interface default methods are by-default available to all implementation classes.Based on requirement implementation class can use these default methods directly or can override.
- Override Object class methods as default - CTError : object class methods are by-default available to every java class hence it's not required to bring through default methods.

Differences between interface with default methods and abstract class

Eventhough we can add concrete methods in the form of default methods to the interface , it wont be equal to abstract class.

Interface with Default Methods	Abstract Class
Inside interface every variable is Always public static final and there is No chance of instance variables	Inside abstract class there may be a Chance of instance variables which Are required to the child class.
Interface never talks about state of Object.	Abstract class can talk about state of Object.
Inside interface we can't declare Constructors.	Inside abstract class we can declare Constructors.
Inside interface we can't declare Instance and static blocks.	Inside abstract class we can declare Instance and static blocks.
Functional interface with default Methods Can refer lambda expression.	Abstract class can't refer lambda Expressions.
Inside interface we can't override Object class methods.	Inside abstract class we can override Object class methods.

Interface with default method != abstract class

- Interface - `static method support` - public static abstract now
- `public static void m1() {....}`
- direct now - I.show()
- Overriding :
- As interface static methods by default not available to the implementation class,overriding concept is not applicable.
- Based on our requirement we can define exactly same method in the implementation class,it's valid but not overriding.

Lambda Expression Vs Method Reference

Following points need to be considered while selecting lambda expression and method reference

- Lambda expression allows to provide **inline function definition**.
- **Allows** to pass the parameters at the time of invocation.
- Useful when a particular method from a functional interface is **not frequently required to be invoked.**
- Method reference allows to **invoke predefined method.**
- **Does not allow** to pass parameters.
- Useful when a particular method from a functional interface is **frequently required to be invoked.**

3. Method Reference

- Provides a compact and `more readable form of a lambda expression` to `call a method that is already defined`.

- Allows to make use of a `method as an argument for a matching functional interface` - Should have same signature

1. Reference to a static method : `ContainingClass::staticMethodName`
2. Reference to a Instance method : `containingObject::instanceMethodName`
3. Reference to a constructor : `ClassName::new`

- `Passing method to method`
- Method as parameter in Method - Call by method
- `lst.forEach(System.out::println)`
- Create getters setters as optional
- `functional Interface method can be mapped to our specified method` by using :: (double colon)operator. This is called method reference.
- Our specified method can be either static method or instance method.
- FunctionalInterface method and our specified method `should have same argument types` ,except this the remaining things like returntype,methodname,modifiers etc. are not required to match.

```
List<String> lst = Arrays.asList("A", "B", "C", "D")
lst.forEach(str -> System.out.println(str)) // [A,B,C,D]
~
lst.forEach(System.out::println) // Call by Method
```

Constructor References

- We can use :: (double colon)operator to refer constructors also
- Syntax: `classname :: new`

```
Interf f = sample :: new;
functional interface f referring sample class constructor
```

- In method and constructor references compulsory the argument types must be matched.

4. Stream API

- Processing objects from the collection
- allows functional-style operations on the elements
- `use only once`
- Create a stream object to the collection by using `stream()` method of Collection interface. stream() method is a default method added to the Collection in 1.8 version.
- `lst.forEach(i->sout(i))` // Normal list
- `lst.stream().forEach(i->sout(i))` // Stream list
- `lst.parallelStream().forEach(i->sout(i))` // Parllel stream list - create threads as per core
- We can process the objects in the following two phases : 1.configuration 2.processing

1. configuration:

- We can configure either by using filter mechanism or by using map mechanism.

2. Filtering:

- We can configure a filter to filter elements from the collection based on some boolean condition by using filter()method of Stream interface.
- `public Stream filter(Predicate<T> t)`

1. Intermediate methods - filter(), map()
2. Terminate Methods - findFirst(), forEach()

```
lst.stream().filter(i->
        {sout("hi")
        return true;
        }).findFirst(); // hi (lst has 100 values);

lst.stream().filter(i->
        {sout("hi")
        return true;
        }).findFirst().orElse(0);
```

Terminal Operations

- `List<Integer> list = Arrays.asList(1,2,3,4)`

1. toArray() : List to Array | `list.stream().toArray()`
2. collect() : Convert to other form | `list.stream().collect(Collectors.toSet()`
3. count() : Take count of elements | `list.stream().count()`
4. reduce() : Change one form to another | `list.stream().reduce(0, (x,y)=>(x+y))`
5. forEach() : call one by one | `list.stream().forEach(System.out::print)`
6. forEachOrdered() : call one by one in order | `list.stream().forEachOrdered(System.out::print)`
7. min() : get minimum element | `list.stream().min((a,b)->a-b)`
8. max() : get maximum element | `list.stream().min((a,b)->a-b)`
9. anyMatch() : tells if there's any match | `list.stream().anyMatch(w->e==2)`
10. allMatch() : tells if all matches | `list.stream().allMatch(w->e%2==0)`
11. noneMatch() : tells if none matches | `list.stream().noneMatch(w->e%2==0)`
12. findAny() : find any element | `list.stream().findAny().get()`
13. findFirst() : finds first element | `list.stream().findFirst().get()`

Intermediate Operations

- Intermediate operations `return a new stream`.
- They are always `lazy` : executing an intermediate operation such as filter () does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.
- `List<Integer> list = Arrays.asList(1,2,3,4)`
- `List<String> listStr = Arrays.asList("a1", "b2b", "c3cc")`

1. filter() : filters based on condition | `list.stream().filter(e->e%2==0)` ///[2,4]
2. map() : transform into another form | `list.stream().map(e -> e*2)` /// [2,,4,6,8]
3. flatMap() : transforms + fetch data as well | `listStr.stream().flatMap(a->Stream.of(a.charAt(1)))` /// [1,2,3]
4. distinct() : gives distinct data | `list.stream().distinct()`
5. sorted() : sorts data | `list.stream().sorted()`
6. peek() : like forEach but needs terminal operation | `list.stream().peek(System.out::println).findFirst()`
7. limit() : it limits ele based on number passed | `list.stream().limit()` ///[1,2]
8. skip() : skips some ele | `list.stream().skip(1)` /// [2,3,4]

5. New Date Time API - Immutable

- `import java.time.LocalDate` - same for Time

```
LocalDate d = LocalDate.now() // 2023-03-31
LocalDate d = LocalDate.of(2023, Month.MARCH, 31) // 2023-03-31 - gives errpor for leap yr
LocalDate d = LocalDate.now(ZoneId.of("Asia/Kuwait")) // 2023-03-31
Instant i = Instant.now() // // 2023-03-31T11:41:14.81Z : machine readable time
LocalDateTime dt = LocalDateTime.now() // 2023-03-31T11:41:14.800
.isBefore()
```

6. Optional

- `Avoids Null Pointer Exception` if(str==null){str.length()}
- import java.util.Optional
- public final class Optional extends Object
- Optional class - A container object may or may not contain a non-null value.
- Create Optional :

1. of()
2. ofNullable()
3. empty()

- isPresent() - true if value is present
- get() - returns value else error
- orElse() - gives value if present, or default value

```
String[] str = new String[5];
s[0] = "ram";
s[1] = "sam"; // s[2,3,5] = null

// Problem
sout(s[2].toUpperCase()) // Null Pointer Exception
```

```
// Solution
Optional<String> o = Optional.ofNullable(s[2]);

if(o.isPresent()){
   sout(o.get().toUpperCase())
}else{
   sout("Value not exist");
}
```
