Core Java:

**Load Factor**

The Load Factor is a threshold, if the ratio of the current element by initial capacity crosses this threshold then the capacity increases so that the operational complexity of the HashMap remains O(1). The meaning of operational complexity of O(1) means the retrieval and insertion operations take constant time.  
*The default load factor of a HashMap is****0.75f****.*

*13th element, load factor = 13/16 = 0.8125 is greater than 0.75, at the insertion of the 13th element we double the capacity.*

*Now the capacity is 32.*

**Fail Fast And Fail Safe Iterators in Java:**

**Fail Fast :**

[Iterators](https://contribute.geeksforgeeks.org/iterators-in-java/) in java are used to iterate over the Collection objects.Fail-Fast iterators immediately throw *ConcurrentModificationException* if there is **structural modification** of the collection. Structural modification means adding, removing any element from collection while a thread is iterating over that collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.

**Fail-Safe:**

iterators don’t throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that’s why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.

When iterating, on each next() call, the current value of modCount gets compared with the initial value. If there's a mismatch, it throws ConcurrentModificationException which aborts the entire operation.

**Factory method design pattern in Java:**

It is a creational design pattern which talks about the creation of an object. The factory design pattern says that define an interface ( A java interface or an abstract class) and let the subclasses decide which object to instantiate.

The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.

e.g

**public** **interface** Notification {

**void** notifyUser();

}

**public** **class** SMSNotification **implements** Notification{

@Override

**public** **void** notifyUser() {

// **TODO** Auto-generated method stub

System.***out***.println("got sms nofification");

}

}

**public** **class** EmailNotification **implements** Notification{

@Override

**public** **void** notifyUser() {

// **TODO** Auto-generated method stub

System.***out***.println("got email nofification");

}

}

**public** **class** NotificationFactory {

**public** **static** Notification getNotificationFactory(String notify) {

**if**("email".equals(notify)) {

**return** **new** EmailNotification();

}**else** **if**("sms".equals(notify)) {

**return** **new** SMSNotification();

}

**return** **null**;

}

**public** **class** FactoryClient {

**public** **static** **void** main(String args[]) {

NotificationFactory obj=**new** NotificationFactory();

Notification notify=obj.*getNotificationFactory*("email");

notify.notifyUser();

}

}

Now what if we introduce a new type of loan named PersonalLoan. We don’t need to change anything in the client end. We just need to change in the factory class. Let us create a new class like this**.**

**Singleton Pattern on the basis of Lazy intilization,double check locking : p**ublic static Singleton getInstance1(){

if (instance == null) {

        instance = new Singleton();

    }

    return instance;

}

* The above code will create multiple instances of Singleton class if called by more than one thread in parallel(known as multithreading).
* The primary solution to the current problem will be to make[**getInstance()**](https://www.geeksforgeeks.org/java-signature-getinstance-method-with-examples/#:~:text=The%20getInstance()%20method%20of,with%20the%20most%20preferred%20Provider.)method synchronized.

**p**ublic static synchronized Singleton getInstance1(){

if (instance == null) {

        instance = new Singleton();

    }

    return instance;

}

**public** **static** Singleton getInstance2(){ // Single Checked

**if** (instance == **null**) {

**synchronized** (Singleton.**class**)

{

// Double checked

**if** (instance == **null**) {

instance = **new** Singleton();

}

}

}

**return** instance;

}

**And another is enum singleton design patter to overcome the problem of double check locking**

The **singleton**pattern restricts the instantiation of a class to one object. **INSTANCE**is a **public static final field** that represents the **enum**instance.

* Guaranteed one instance (Cannot instantiate more than one enum even through reflection).
* Thread-safe.
* Serialization.

**Differences between TreeMap, HashMap and LinkedHashMap in Java**

**1)HashMap: public class HashMap extends AbstractMap**

**implements Map,Cloneable, Serializable**

* A HashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains **no order**.

**2)LinkedHashMap:** LinkedHashMap offers **0(1)** lookup and insertion. Keys are ordered by their insertion order. It is implemented by doubly-linked buckets.  
**Syntax:**

**public class LinkedHashMap extends HashMap**

**0implements Map**

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead **maintains insertion order**.

LinkedHashMap<String, String> linked=**new** LinkedHashMap<>();

linked.put("1", "teju");

linked.put("5", "Vihu");

linked.put("2", "Sandip");

linked.put("7", "Anil");

System.***out***.println("linked Map:"+linked);

o/p:linked Map:{1=teju, 5=Vihu, 2=Sandip, 7=Anil} but these insertion order not maintained in HashMap

**3)TreeMap: Syntax:**

**public class TreeMap extends AbstractMap implements**

**NavigableMap, Cloneable, Serializable**

* A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead **maintains ascending order(Sorted using the natural order of its key**)

**4)Hashtable:**“Hashtable” is the generic name for hash-based maps.  
**Syntax:**

**public class Hashtable extends Dictionary implements**

**Map, Cloneable, Serializable**

* A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* It contains only unique elements.
* It may have not have any null key or value.
* It is synchronized.
* It is a legacy class.

**Collection vs Collections:**

| **Collection** | **Collections** |
| --- | --- |
| It is an interface. | It is a utility class. |
| It is used to represent a group of individual objects as a single unit. | It defines several utility methods that are used to operate on collection. |
| The Collection is an interface that contains a static method since java8. The Interface can also contain abstract and default methods. | It contains only static methods. |

**ArrayList Internal working:**

### ****How the size of ArrayList grows dynamically?****

If we try to add an element using the add() method in the array list Internally then it checks for the capacity to store the new element or not, If not then the new capacity is calculated as shown in the internal code of the add() method.

public boolean add(E e) {

ensureCapacityInternal(size + 1); // Size Increments

elementData[size++] = e;

return true;

}

The new capacity is calculated which is 50% more than the old capacity and the array is increased by that capacity. It uses Arrays.copyOf which gives the array increased to the new length by right shift operator also it will grow by 50% of old capacity.

int newCapacity = oldCapacity + (oldCapacity >> 1);

**e.g newcapacity=10+(1010>>1)=10+(101)=10+5=15**

**When you call default constructor then it creates emply list so default capacity will be 10**

**When you pass capacity to List<> list = new ArrayList<>(20); then it will be 20 capacity**

## **Runtime performance of ArrayList**

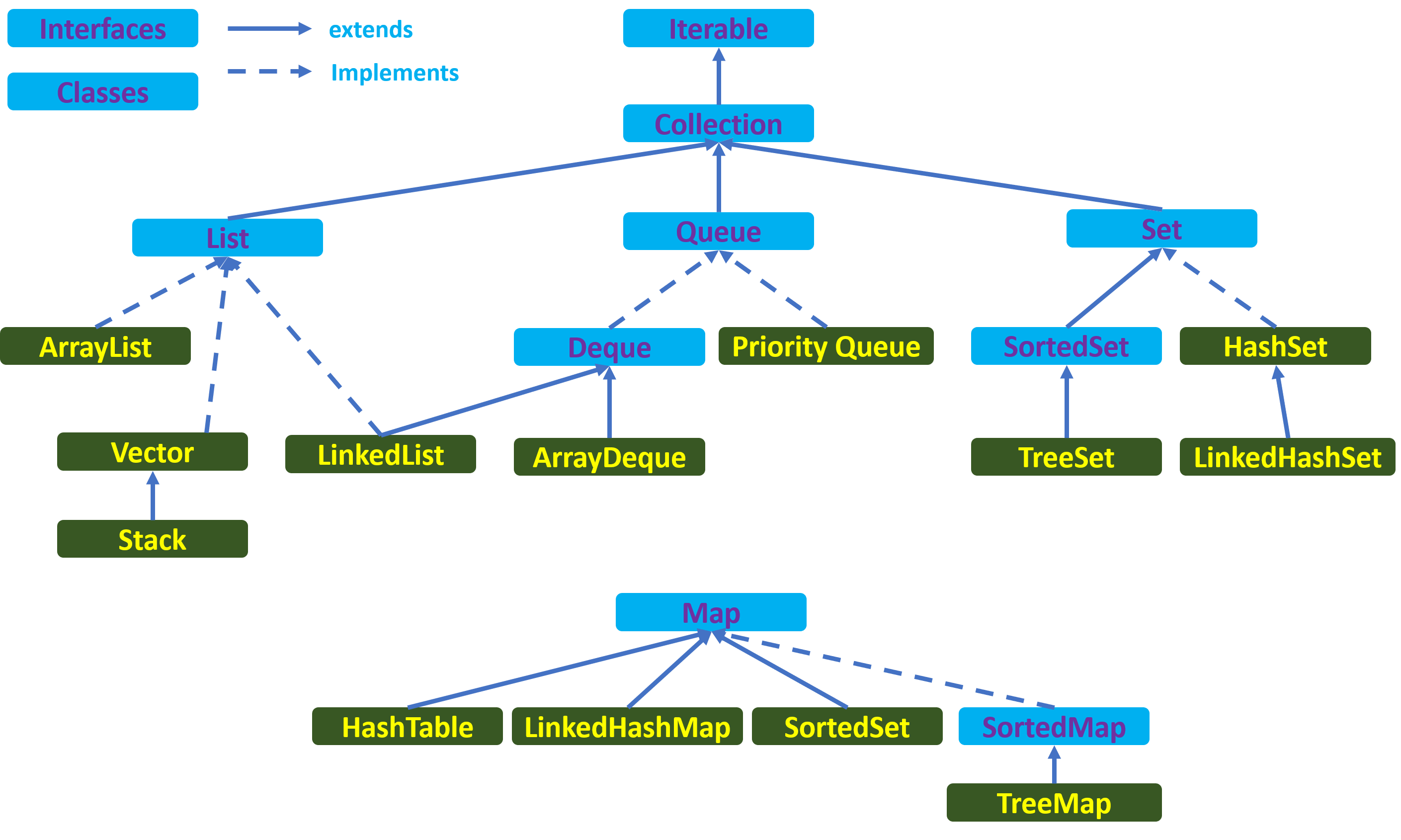
The size, isEmpty, get, set, iterator, and listIterator operations run in constant time **O(1)**. The add operation runs in amortized constant time, that is, adding n elements requires **O(n)** time.

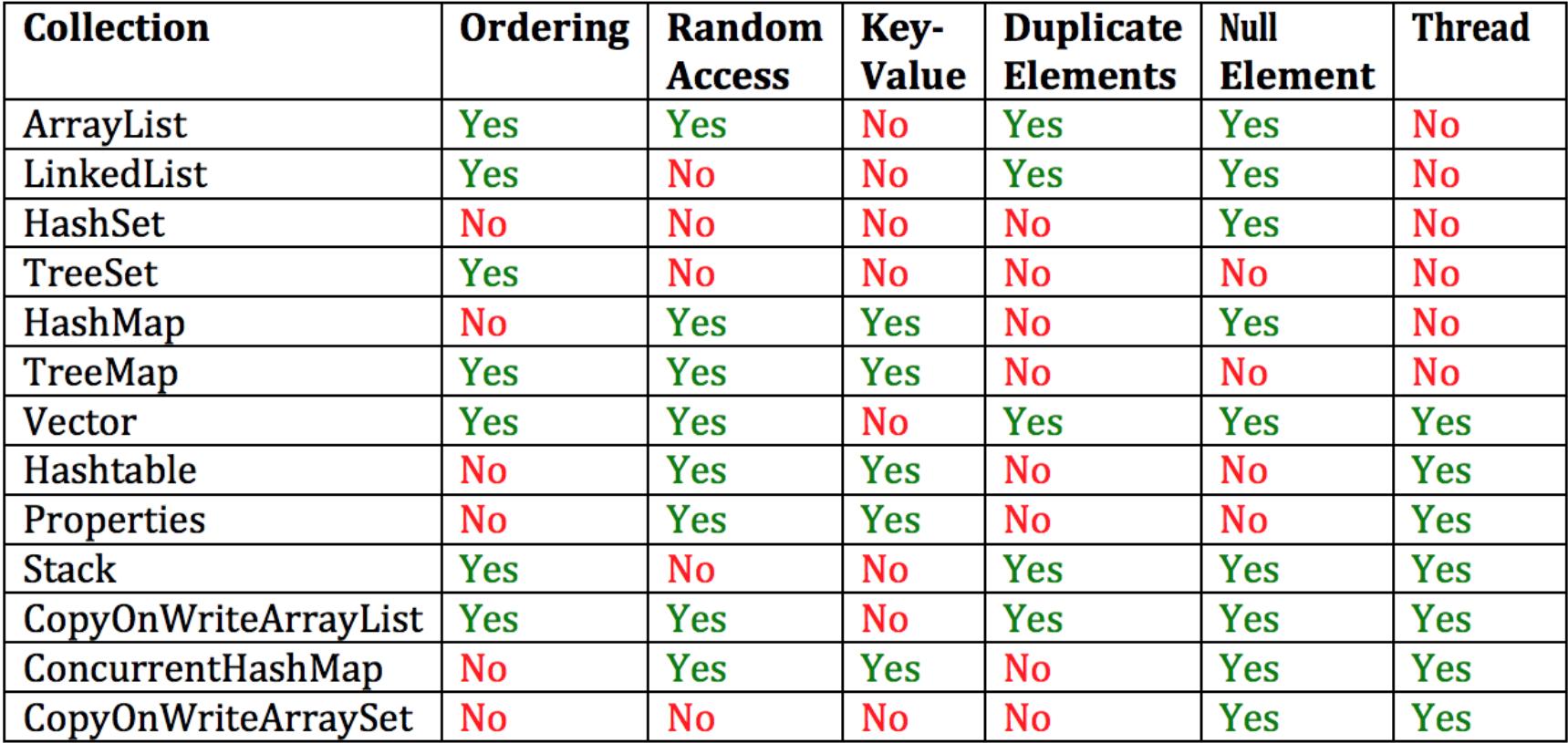
**Threshold**

Threshold = (Current Capacity) \* (Load Factor)

The load factor is the measure that decides when to increase the capacity of the ArrayList. The default load factor of an ArrayList is 0.75f.  For example, current capacity is 10. So, loadfactor = 10\*0.75=7 while adding the 7th element array size will increase.

Whenever we create an ArrayList and it reaches its threshold, Internally creates a new ArrayList object with a new capacity and copies all old elements from the old ArrayList to a new object.







### Java 8 Durga course:

### Lambda Expression:

### 1) To enable the functional programming.

### 2) Write more readable, maintable .

### a->a.length(); This is equivalent to

### int getLenth(String s){ return s.length()

### }

### Functional Interface:

### 1) It has single abstract method e.g. comparable and Runnable

### 2) We have to extend function interface with void method1 (); in second interface method .but interface which has already abstract method with same name then second interface has diff method() then in this case it will give an error.

### 3) It is used to invoke the lambda expressions or lambda expression provides functional interface method implementation.

### Q: why functional interface has only one abstract method?

### ANS: When we implement the lambda expression for function interface, compiler always check type and parameter due to that Functional interface should have only one abstract method.

### It has more than one abstract method and when we implement abstract method implementation using lambda then compiler gets confused which method should I call hence function interface has only one method.

### \*Collection sorting using lambda:

### Comparable Interface: It provides default sort ordering.

### 1) When we do collections.sort(list) it maintains the natural ordering

### Comparator: If you want to sort the element in desc order or your own sorting order then you have to use Comparator Interface

### When compare () gets called and below implementation to sort the element using Descending order like,

### Compare (Object obj1,Object obj2){

### If (obj1>obj2)// if (obj1<obj2)//second for ascending order

### Return -1;//this will return greater than element

### Else if(obj1<obj2)// if(obj1>obj2)

### Return 1//this will return small element

### Else

### Return 0;

### }

### Default method:

### If two interfaces has same default method then compile gives the compiler error when class implement both the interfaces.

### In this scenario you can implement default method in your class or in implemented class , you can call default method like below

### Class A implement Left{

### Public void m1(){

### Left.super.m1()// in this scenario left interface method will get called

### }

### }

### What are the various categories of pre-defined function interfaces?

**Function:** To transform arguments in returnable value. It takes any kind of parameter and return any type of value. It has apply ();

**Predicate:** To perform a test and return a Boolean value. It has

**Consumer:** Accept arguments but do not return any values.

**Supplier:** Do not accept any arguments but return a value.

**Operator:** Perform a reduction type operation that accepts the same input types.

### What is Method Reference?

Method reference is a compact way of referring to a method of functional interface. It is used to refer to a method without invoking it. :: (double colon) is used for describing the method reference. The syntax is class::methodName

For e.g.:

Integer::parseInt(str) \\ method reference

str -> Integer.ParseInt(str); \\ equivalent lambda

### What are Java 8 streams?

A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.

**What are the main components of a Stream?**

Components of the stream are:

* A data source
* Set of Intermediate Operations to process the data source
* Single Terminal Operation that produces the result

Stream has two types of operation

**1) Intermediate Operations:** Process the stream elements

**2) Terminal Operations:** Used to collect the processed Stream data.

**Java Stream:**

**Below are the in-built function interfaces in java 8:**

**1) Filter(Predicate Functional Interface)**: Filter function uses Predicate functional interface with argument and return the true result. It is intermediately operation. We use it into true and false scenario and argument can generic. It has test () method. It is in java.util.function interface.

i)Test():it is abstract method

ii) negate(), p1.or(p2),p1.and(p2): these all are default method.

iii)isEqual () is static method: Predicate.isEqual(“teju”):it check the object or equal or not

Predicate<Integer> i=value->value>10;

System.***out***.println(i.test(20));

System.***out***.println(i.test(10));

Predicate<String> str1=str->str.length()>5;

System.***out***.println("Check str length:"+str1.test("Vihaan"));

**2)For Each(Consumer Functional Interface):**

ForEach uses Consumer functional interface with argument but does not return anything. It is terminal operation. We can print the value. It has void accept () method which does not return anything

**3) Supplier Functional interface:**

This interface can be used when there is no input but o/p should be expected. This interface has get() method . This interface in such cases you do some operation and u want a result

Lambda Expression: It is an anonymous function and it is used only for functional interface method to implement the method

**What is Map() and FlatMap();**

Both these API are intermediate methods and returns another stream as part of the output

Map():It is used for data transformation and it is mapper function produces single value for each input so it is called 1-1 mapping .

FlatMap(): It is used for data transformation as well flattering . It is Map()+Flattering. It is mapper function produces multiple value for each input so it is called one to many mapping.

**4)Function Functional interfaces Java 8:**

It takes any kind of parameter and return any kind of value. It has below methods and It is in java.util.Function package

1)apply(): abstract method

2)andThen() and compose(): are default methods

3)identify (): it is static method. It always return same o/p which is provided in input

Function<String, Integer> func=s->s.length();//String is i/p and Integer is o/p

System.***out***.println(func.apply("Teju"));o/p : 4

Functional Chaining:

Function<String, String> f1=f->f.toUpperCase();

Function<String, String> f2=s->s.substring(0,9);

String result=f1.andThen(f2).apply("Vihaanbankar");

System.***out***.println("Result:"+result);

o/p Result:VIHAANBAN

What is differenece between andThen() and Compose()

Function<Integer, Integer> f3=i->i+i;

Function<Integer, Integer> f4=j->j\*j\*j;

Integer res=f3.compose(f4).apply(2);

System.***out***.println("The result of compose is:"+res);

Function<Integer, Integer> f5=i->i+i;

Function<Integer, Integer> f6=j->j\*j\*j;

Integer rest=f5.andThen(f6).apply(2);

System.***out***.println("The result of andThen is:"+rest);

o/p: The result of compose is:16// here f4 gets performed with 8 o/p then calls f3 function i.e 8+8=16

The result of andThen is:64// here f5 function gets performed and then f6 gets executed

System.***out***.println("The result of andThen is:"+rest);

**What is different between Predicate and Function interface:**

**Predicate**: 1)it is used for conditional check and when we want to return Boolean value.

2)Predicate<T> it has only one input type bz already it return Boolean value which is defined by java 8

**Function:** 1)It return any kind of value. Function<T,R> T is for i/p and R is for return type2)It is used in scenario where you want to return any kind of value e.g is Employee average salary.

**BiPredicate ,BiFunction ,BiConsumer:**

**->** when we want two 2 input values then we should go for this predicate ,function and consumer

BiPredicate<Integer, Integer> biPre=(a,b)->(a+b)%2==0;

Boolean test=biPre.test(2, 2);

System.***out***.println("Test:"+test);

BiFunction<String,String,Boolean> biFunt=(i,j)->i.startsWith("T") && j.startsWith("V");

System.***out***.println("BiFunction::"+biFunt.apply("Teju", "Vihaan"));

BiConsumer<Integer, Integer> biConsumer=(c1,c2)->{

Integer re=c1\*c2;

System.***out***.println(re);

};

System.***out***.println("The Biconsumer is:");

biConsumer.accept(3,4);

O/P: Test:true

BiFunction::true

The Biconsumer is:

12

**Note:**

**1)when we use Predicate<T> , it accepts only wrapper type of class due to that performance goes down so to overcome this problem below interface came into picture.**

**->** IntPredicate

-> DoublePredicate

-> LongPredicate

**int** []x= {7,10,40,4,6};

IntPredicate p=s->s%2==0;

**for**(**int** re:x) {

System.***out***.println("To check even no:"+p.test(re));

**All 15 primitive types of functiona interfaces:**

**1)IntFunction<R> 2)DoubleFunction<R> 3)LongFunction<R>: this all are has apply ()**

**2)ToIntFunction<T> 2)ToLongfunction : it has applyAsInt() and applyAsDouble()**

**3)IntSupplier 2)LongSupplier3)DoubleSupplier 4)BooleanSupplier : it has method like getAsInt(), getAsDouble() and getAsLong() instead of get method of Supplier()**

**Q:Can we use two abstract method in functional interface?**

TestInterface i = () -> System.out.println("Hans");

You give an implementation to the void hans() method of the TestInterface.

If you could assign a lambda expression to an interface having more than one abstract method (i.e. a non functional interface), the lambda expression could only implement one of the methods, leaving the other methods unimplemented.

You can't solve it by assigning two lambda expressions having different signatures to the same variable (Just like you can't assign references of two objects to a single variable and expect that variable to refer to both objects at once).

**More explanation for above Q**

Whenever you are using the [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/)for the[**Function interface**](https://javagoal.com/functional-interface-in-java-8/), the compiler strictly ensures the interface has only one [**abstract method**](https://javagoal.com/abstract-method-in-java/). If the interface contains more than one [**abstract method**](https://javagoal.com/abstract-method-in-java/) the program shows an error. Because the [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/) provides body to the only a **single abstract method.** When you write [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/) the compiler assumes the statements of [**lambda expression**](https://javagoal.com/java-8-lambda-expressions/) will be the body of the abstract method.

\***Method and Constructor References:**

**1)** We can call method references and constructor reference through: :operator**.** We can use:: for code reusability**.**

**2)** Method ref is the alternative ways of lambda expression and restriction of method ref is method parameter will be same.

3) we can do method ref for static method and instance method

The method references can only be used to replace a single method of the lambda expression. A code is more clear and short if one uses a lambda expression rather than using an anonymous class and one can use method reference rather than using a single function lambda expression to achieve the same.

**Stream:**

**What is diff bet collection stream and io stream?**

**Java.util.stream process the operations on collection of objects and produce the desired result.**

 Filter(): it is used when u want to filter the data which will return the result in Boolean formate.

Map(): Map will

**\*Sorting in Java 8:**

**->sorted (): by default natural sorting order.**

**->sorted(Comparator c) for customized sorting order**

**Stream to Array: need to use Stream.toArray() method**

**Max() and MIN():**

**Min(Comparator c): return mini value according to specified comparator.**

**Max(Comparator c): return max value as per comparator.**

**There 3 conditions below is the snapshot of descending order**

studList.stream().sorted((f1,f2)->f2.getRollno()<f1.getRollno()?-1:f1.getRollno()>f2.getRollno()?1:0)

o/p

Student [rollno=21, name=Manushree]

Student [rollno=20, name=Tanushree]

Student [rollno=10, name=Vihaan]

Student [rollno=4, name=Sandip]

**Below is the snapshot of ascending order**

studList.stream().sorted((f1,f2)->f1.getRollno()<f2.getRollno()?-1:f1.getRollno()>f2.getRollno()?1:0)

o/p:Student [rollno=4, name=Sandip]

Student [rollno=10, name=Vihaan]

Student [rollno=20, name=Tanushree]

Student [rollno=21, name=Manushree]

**We can use sorted() method to sorting in stream.**

emp.stream().sorted((emp1, emp2) ->(**int**) (emp1.getSalary()-emp2.getSalary())).forEach(e->System.***out***.println(e));// this is using old way

**Stream Distinct: When we use distinct on custom classes, we need to override the hashCode() and equal() method from that class**

emp.stream().distinct().forEach(t->System.***out***.println("::"+t));

\*\*\*Stream to array conversion: we can use stream.of(Array.asList(1,2))

**Terminal Operations**:

1)findAny() and findFirst():

findFirst return the first elements of the stream but findAny is free to select any element in the stream.

Optional<Integer> i=list.stream().filter(t->(t%4==0)).findAny();

System.***out***.println(i.get());

Stream.reduce():Many times, we need to perform operations where a stream reduces to single resultant value. The reduction operation combines all elements of the stream into a single result.

E.g sum of number , 1+2=3

**\*Parallel vs Sequential Stream in Java**

### Sequential Stream

Sequential Streams are non-parallel streams that use a single thread to process the pipelining. Sequential stream performs operation one by one. **stream()** method returns a sequential stream in Java.

**Parallel streams**

Using parallel streams, our code gets divide into multiple streams which can be executed parallelly on separate cores of the system and the final result is shown as the combination of all the individual core’s outcomes.

**Stream aggregate function:**

**Max**: Employee e3=emp.stream().max(Comparator.*comparing*(sal->sal.getSalary())).get();

**Min:** Integer min= list.stream().min(Integer::*compare*).get();

**Sum: int** sum= list.stream().mapToInt(i->i).sum();

**Method References:**

**Ans:** method reference is the shorthand syntax for a lambda expression that contains just one method call.

**Time and Date API:**

**There are various classes in java.time.**

**1)LocalDate 2)LocalTime 3)LocalDateTime**

LocalDate ld=LocalDate.*now*();

System.***out***.println("Local Date:"+ld);

LocalTime lt=LocalTime.*now*();

System.***out***.println("The local time "+lt);

**o/p:**

Local Date:2021-06-05

The local time 19:07:51.426049300

**Below are the advantage of java 8 over java 7**

1. **Not thread safe :** Unlike old java.util.Date which is not thread safe the new date-time API is *immutable* and doesn’t have setter methods.
2. **Less operations :** In old API there are only few date operations but the new API provides us with many date operations.

**Summary of java 8 features:**

1. lambda expression : it is used to invoke the functional interface method.
2. Default method: It is used for backward functionality when there are multiple classes and they are implementing the interface in this scenario, u can create default method or u can override that method into implemented class.

**What is MetaSpace in Java 8**

The area where all the classes that are stored in Java 8 are called MetaSpace. MetaSpace has replaced the PermGen. Since MetaSpace is dynamic as it can grow dynamically and it does not have any size limitation,

**Why wait(), notify() and notifyAll() are in Object class and not in Thread class in java**

Ans:  
**Every Object has a monitor**, acquiring that monitors allow thread to hold lock on object. But **Thread class does not have any monitors**.

wait- wait method tells the current thread to give up monitor and go to sleep.

notify- Wakes up a single thread that is waiting on this object's monitor.

So you see wait() and notify() methods work at the monitor level, thread which is currently holding the monitor is asked to give up that monitor through wait() method and through notify() method (or notifyAll) threads which are waiting on the object's monitor are notified that threads can wake up.

Important point to note here is that monitor is assigned to an object not to a particular thread. That's one reason why these methods are in Object class.

**Object level locking:**

Object level locking means you want to synchronize non static method or block so that it can be accessed by only one thread at a time for that instance.

**Class level locking:**

Class level locking means you want to synchronize static method or block so that it can be accessed by only one thread for whole class. If you have 10 instances of class, only one thread will be able to access only one method or block of any one instance at a time. It is used if you want to protect static data.

**SOLID principles**:

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle
3. Liskov’s Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

**1)SRP**: a class should have only one reason to change which means every class should have a single responsibility or single job or single purpose.

Most of the time it happens that when programmers have to add features or new behavior they implement everything into the existing class which is completely wrong. It makes their code lengthy, complex and consumes time when later something needs to be modified

2) **Open/Closed Principle:**

This principle states that “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” which means you should be able to extend a class behavior, without modifying it.

**3. Liskov’s Substitution Principle:**

“Derived or child classes must be substitutable for their base or parent classes“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

Lets take a farmer example , a farmer’s son should inherit farming skills from his father and should be able to replace his father if needed. If the son wants to become a farmer then he can replace his father. Design level e.g is rectangle has four sides with width and height value and square class is inheriting the rectangle features. In order to do that we can swap child (square) with parent(rectangle).

**4. Interface Segregation Principle:**

The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.

e.g there is one interface

interface Athlete{

void highJump();

void longJump();

void swim();

}

There is JhonDoe is swimming Athlete but by implementing the interface Athlete we have to implement all other method which JhonDoe will never use.

We will follow the interface segregation principle and we will refactor the original interface and create two other interfaces one for Jumping athletes and one for Swimming athletes.

So JhonDoe will not have to implement actions that he is not capable of performing.

**5. Dependency Inversion Principle:**

* High-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions.
* Abstractions should not depend upon details. Details should depend upon abstractions.

public class BackEndDeveloper {

    public void writeJava() {   }

}

public class FrontEndDeveloper {

    public void writeJavascript() {

    }

Public class Project {

    private BackEndDeveloper backEndDeveloper = new BackEndDeveloper();

    private FrontEndDeveloper frontEndDeveloper = new FrontEndDeveloper();

    public void implement() {

        backEndDeveloper.writeJava();

        frontEndDeveloper.writeJavascript();

    } }

}

So as we can see the Project class is a high level module and it depends on low level modules such as BackEndDeveloper and FrontEndDeveloper. We are actually violating the first part of the dependency inversion principle

In order to tackle this problem we shall implement and interface called the Developer interface and BackEndDeveloper and FrontEndDeveloper class will implement the Developer interface

public interface Developer {

    void develop();

}

public class FrontEndDeveloper implements Developer {

    @Override

    public void develop() {

        writeJavascript();

    }

    public void writeJavascript() {

    }

}

//there will be same for BackEndDeveloper

public class Project {

    private List<Developer> developers;

    public Project(List<Developer> developers) {

        this.developers = developers;

    }

    public void implement() {

        developers.forEach(d->d.develop());

    }

}

* **In which scenario , we can use Thread class or Runnable interface?**

class Task implements Runnable{  
 @Override  
 public void run (){  
 System.out.println ("Runnable interface");  
 }  
}class ThreadDemo extends Thread{  
 @Override  
 public void run(){  
 System.out.println( "Thread class ");  
 }  
}class RunnableDemo {  
 public static void main (String ... args){new ThreadDemo().start();  
   
 new Thread (new Task(), "Thread 1").start();  
   
 new Thread (new Task(), "Thread 2").start();  
 }  
}

In the above example Task implements Runnable and this Task is executed by two different threads. The Runnable objects could be shared by many threads. Another way is to extend Thread class as explained in ThreadDemo. As ThreadDemo had already extended Thread class, it can not extend some another class because Java does not support multiple inheritance. There are many reasons to prefer Runnable over Thread. Some of them are as below:

Java does not support multiple inheritance. Hence if your class had already extended some another class, it cannot extend Thread class anymore.

By implementing Runnable we can reuse the task to execute it on different threads

**Java Memory management Overview:**

**Java Heap:**

When a Java program started Java Virtual Machine gets some memory from the Operating System. Java Virtual Machine or JVM uses this memory for all its needs and part of this memory is call java heap memory.  
  
**When objects goes for garbage collection:**

**1) when references is NULL**

**2) when one object ref is assigned to other refe.**

**Heap Generations for Garbage collections:**

**Mark and Sweep Algorithm:**

JDK, JRE and JVM:

JDK=JRE+development tools

JRE=Set of libraries + JVM

**How does JRE works? : Below diagram show what is done at compile time**



How it works at runtime:

**1) Class Loader: it loads necessary classes**

**2) Byte code verifier: it verifies all the codes that means it should not violate the system integrity**

**3) Interpreter:** At runtime the Byte Code is loaded, checked and run by the interpreter. The interpreter has the following two functions:

* Execute the Byte Code
* Make appropriate calls to the underlying hardware

**What is difference between git fetch and git pull.**

Ans: Git pull : 1)is what you would do to bring a local branch up-to-date with its remote version, while also updating your other remote-tracking branches.

2) on the other hand does that AND brings (copy) those changes from the remote repository.

Git fetch: is the command that tells your local git to retrieve the latest meta-data info from the original. It does not modify ur local changes.