* Constructor chaining using this keyword:

⟹ new Temp(8,10) // invokes parameterized constructor 3.

```
Temp(int x, int y){
    this(5);            // invokes parameterized ctor 2.
    sysout(x * y) :
}

    → Temp(x,y){
        this();         // invoke default ctor
        sysout(x);
    }

        → Temp(){
            sysout("default");
        }
```

* Rules of constructor chaining:

① The this() expression should always be the first line of the constructor.

② There should be at-least ~~be~~ one constructor without the this() keyword.

③ Constructor chaining can be achieved in any order.

* What happens if we change the order of ctors?

→ Nothing, ctor chaining can be achieved in any order.

* Constructor chaining to other class using super() keyword:

⟹ Similar to constructor chaining in same class, super() should be the first line of the constructor as super class's constructor are invoked before the sub class's constructor.

Alternative Method : Using Init block

Initializer block contains the code that is always executed whenever an instance is created. It is used to declare/initiate the common part of various constructors of the class.

- The contents of initializer block are executed whenever any constructor is invoked.

- The order of initialization constructors and initializer block doesn't matter, Initializer block always executed before constructor.

When we want certain common resources to be executed with every constructor we can put the code in the Init block.

* Private constructor:

Like any method we can provide access specifier to the constructor.
If it's made private, then it can only be accessed inside the class and that constructor is called private constructor.

Private constructors allow us to restrict the instantiation of a class.
They prevent the creation of class instances in any place other than the class itself.

Public and private constructors used together, ~~show~~ allow control over how ~~to~~ we wish to instantiate our classes. This is known as "constructor delegation"

* The major scenarios where we can use private constructors:

① Internal Constructor Chaining.
② Singleton Class design pattern.

* Singleton Class:

As the name implies, a class is said to be singleton if it limits the number of objects of that class to one

- We can't have more than a single object for such classes.
- Singleton classes are employed extensively in concepts like Networking and Database Connectivity.

* Design Pattern of Singleton Classes:

- The constructor of singleton class would be private so there must be another way to get the instance of that class.
- This problem is resolved using a class member instance and a factory method to return ~~a~~ the class member.

* Static Block:

Whenever we use a static keyword and associate it ~~with~~ to a block then that block is referred to as a static block. (also called as static clause) that can be used for static initialization of a class.

This code inside the static block is executed only once. when the first time the class loaded in the memory.

Note: Static blocks can also be executed befor constructors.

* Calling static block in Java:

In order to call any static block, there is no specified way as static block executes automatically when class is loaded in the memory.

Imp * Can we print something on the console without creating main() method?

⟹ Yes,

We can print if we are using JDK version 1.6 or previous and after that it will thrown an error.

Note:- A class can have any number of static initialization blocks and they can appear anywhere in the class body.

- The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

- We use Initialiser Block in Java if we want to execute a fragment of code for every object ~~created~~ which is seen widely in enterprising industries in development.

* Nested Interface:

We can declare interfaces as a member of a class or another interface. Such an interface is called as member interface or nested interface.

* Interface in a class:

- Interfaces (or classes) can only have public and default access specifiers when it declared outside any other class.

- This interface declared in a class can either be default, public, protected not private.

- While implementing the interface, we mention the interface as c_name.i_name where c_name is the name of the class in which it is nested and i_name as the name of itself (interface).

* Interface in another Interface:

- An interface can be declared inside another interface also.

- We mention the interface as i_name1.i_name2 where i_name1 is the interface in which it is nested and i_name2 is the name of the interface to be implemented.

* Marker Interface:

It is an empty interface which does not contains any fields or any methods.

Note: Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces.

Ⓐ Cloneable Interface:

A cloneable interface is present in java.lang package.

- There is a method ~~in~~ clone() in object class.

A class that implements the cloneable interface indicates that it is legal for clone() method to make a field-for-field copy of instances of that class.

Invoking Object's clone method on a instance of the class that does not implement the cloneable interface results an exception CloneNotSupported-Exception being thrown.

By convention, classes that implement this interface should override Object.clone() method.

Ⓑ Serializable interface:

- Serializable interface is present in java.io package
- It is used to make an object eligible for saving it's state into a file. This is called 'Serialization'.
- Classes that do not implement this interface will not have any of their state serialized or deserialized.

Note: All subtypes of a serilizable class are themselves serializable.

Ⓒ Remote Interface:

- Remote interface is present in java.rmi. package. A remote object is an object which is stored at one machine and accessed from another machine. So, to make an object a remote object, we need to flag it with Remote Interface.
- Remote interface serves ~~as~~ to identify interfaces whose methods may be invoked from a non-local virtual ~~memory~~ m/c. Any object that is a remote object must directly or indirectly implement this interface. RMI provides some convenience classes that remote object implementations can extend which facilates remote object creation.

* Functional Interface:

A functional interface is an Interface that contains only one abstract method.

From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

Note: A functional interface can have any number of default methods.

- Also known as Single Abstract Method Interfaces.

- Functional interfaces are used and executed by representing the interface with an annotation called @FunctionalInterface.

- By default, the method define inside the interface is abstract only.

- We can also called Lambda expressions as the instance of functional interface.

* Some Built-In Functional Interfaces:

① Runnable: only contains the run() method.
② Comparable: only contains the compareTo() method.
③ ActionListener: only contains actionPerformed() method.
④ Callable: only contains the call() method.

* Java SE 8 included 4 main kinds of functional interfaces:

① Consumer:

Consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument.

It returns nothing. There are also functional variants of the consumer - "DoubleConsumer", 'IntConsumer' and 'LongConsumer'. These varients accept primitive values as arguments.

Bi-Consumer:

This varient of consumer interface takes two arguments and returns nothing.

It is used ~~to~~ in iterating through the ~~map~~ entries of the map.

```
Consumer<Integer> consumer = (value) ->
        sysout(value);
```

② Predicate:

In scientific logic, a function that accepts an argument and in return generates a ~~bole~~ boolean as an answer is known as predicate.

Bi-Predicate: predicate takes one argument but Bi-predicate two arguments and return boolean just like predicate

③ Function:

A function is a type of functional interface in java that receives only a single arguments and returns a value after the required processing.

Ⓐ Bi-function:

It takes two arguments.

Ⓑ Unary Operator:

Unary operator extends function.

Ⓒ Binary Operator:

Binary operator extends Bi-function.

Note: In both the input and output values must be identical and of the same type.

④ Supplier:

A type of functional interface that does not take any input or argument and yet returns a single output.

The diffrent extensions of the Supplier functional Interface:

BooleanSupplier, DoubleSupplier, LongSupplier and IntSupplier.

* Comparator Interface:

A comparator interface is used to order of user defined classes.

A comparator object is capable of comparing two objects of the same class.

public int compare (Object obj1, Object obj2):

Method 1:

Write our own sort() function using one of the standard algorithm.

Method 2:

Using comparator interface which is present in java.util package and contains 2 methods compare (Object obj1, Object obj2) and second method is equals (Object element). Using a comparator, we can sort the elements based on data members.

public void sort (List list, ComparatorClass c)

To sort given List, Comparator class must implement a comparator ~~class~~ interface.

Sort() method working:

sort() calls compar() of the classes it is sorting. to compare two elements it ask "which is greater"? compare method returns -1, 0, 1.

* Exceptions Handing :

Exception Handling is a mechanism to handle runtime errors so that regular flow of the application can be preserved (maintain).

* What is an exception ?

→ An Exception is an unwanted or unexpected event, which occures during the execution of a program. i.e. at run time, that disrupts the normal flow of the program's instructions.

Exception can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object.

It contains the information about the exception such as the name and description of the exception and the state of the program when the exception occured.

* An Exception can occur for many reason. some of them:

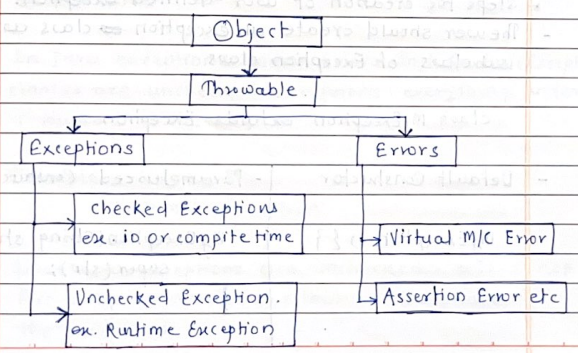
- Invalid user input
- Device failure
- Loss of network connection
- Code errors
- Opening an unavailable file
- Physical limitations (out of disk memory)

* What is an Error ?

→ Error refers to an illegal operation performed by the user which results the abnormal working of the program.

* Exception Hierarchy :

- All exceptions and errors types are subclasses of class Throwable. One branch is headed by Exception.
- ex. NullPointerException
- Another branch, Error is used by the Java run-time system (JVM) to indicate errors having to do with runtime environment itself (JRE).
- StackOverFlow Error.



* Types of Exceptions:

(A) Built-In Exceptions :

- Buil-In exceptions are the exceptions which are available in Java libraries.

Examples :
Arithmetic Exceptions, ArrayIndexOutOfBoundsException, ClassNotFoundException, IOException etc.

(B) User-Defined Exception :

- In Java, Sometimes, the built-In exceptions are not able to describe a certain situation. In such cases, user can also create exceptions which are called 'user-defined exceptions'.

* Steps for creation of user-defined exception.
- The user should create an exception class as a subclass of Exception class

class MyException extends Exception

- Default Constructor	- Parameterized Constructor
MyException () { }	MyException(String str){ super(str); }

* There are two types of Exceptions :

(A) Checked Exceptions :

- This are the exceptions that are checked at compile time.

- If ~~a method~~ some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the throws keyword.

- Examples :
FileNotFoundException, IOException etc

(B) Unchecked Exceptions :

- This are the exceptions that are not checked at compile time.

- In Java exceptions under Errors and RuntimeException classes are unchecked exceptions, everything under throwable is checked.

- Example :
Arithmetic Exception

Note : In c++, all exceptions are unchecked, so it is not forced by the compile to either handle or specify the exceptions.

* Blocks and Keywords used for Exception handling:

① try:

The try block contains set of statements where an exception can occur.

```
try {
    // statements that might cause exception.
}
```

② catch:

Catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception that occurs in associated try block.

```
catch {
    // statements that handle exception.
}
```

③ throw:

Throw keyword is used to transfer control from try block to catch block.

The throw keyword in Java is used for explicitly throw an exception from a method or any block of code.

④ throws:

Throws keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to ~~be~~ the caller and does not handle itself.

⑤ finally:

It is executed after catch block. We basically use it to put some common code when there are multiple catch block.

* Flow control in try, catch, finally:

1. Control flow in try-catch clause OR try-catch-finally clause:

Case 1: Exceptions occurs in try block and handled in catch block.

Case 2: Exceptions occurs in try block and is not handled in catch block.

Case 3: Exceptions doesn't occur in try-block.

2. try-finally clause:

Case 1: Exception occurs in try block.

Case 2: Exception doesn't occure in try-block.

* Control Flow in try-catch OR try-catch-finally:

Case 1: Exception occurs in try block & handled in catch block:

- If a statement in ~~any~~ ~~t~~ try block raised an exception, then the rest of the try block doesn't execute and control passes to the corresponding catch block.

- After executing the catch block, the control will be transferred to finally block (if present) and then the rest program will be executed.

Case 2: Exception occurred in try-block is not handled in catch block:

- In this case default handling mechanism is followed. If finally block is present, it will be executed followed by default handling mechanism.

Case 3: Exception doesn't occur in try-block:

- In this case catch block never runs as they are only meant to be run when an exception occurs.

- If finally block is present, will be executed followed by the rest of the program.

* Control Flow in try-finally:

In this case, no matter whether an exception occur in try-block or not, "finally will always be executed!" But control flow will depend upon whether exception has occured in try-block or not.

Case 1: Exception raised:

- If an exception has occured in try-block then control flow will be finally block ~~flow~~ followed by default exception handling mechanism.

Case 2: Exception not raised:

- If an exception does not occur in try-block then control flow will be finally block followed by rest of the program.

* throw and throws in Java:

① throw:

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.

We can either throw checked or unchecked exceptions.

Note: The throw keyword is mainly used to throw custom exceptions.

Syntax:

throw Instance

ex:

throw new Arithmetic Exception ("/ by zero");

- This exception, i.e. Instance must be of type Throwable or a Subclass of Throwable.

- For example, Exception is a sub-class of throwable and user-defined exceptions typically extend Exception class.

- Unlike c++, data types such as int, char, floats or non-throwable classes cannot be used as exceptions.

The flow of the execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception.

If it finds match, controlled is transferred to that statement otherwise next enclosing try block is checked and so on.

If no matching catch is found then the default exception handler will halt the program.

② throws:

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions.

The caller to these methods has to handle the exception using a try-catch block.

- In a program, if there is a chance of raising an exception then compiler always warn us about it and compulsorily we should handle that checked exceptions, otherwise we will get compile time error saying "unreported exception xxx must be caught or declared to be thrown".

- To prevent this compile time error, we can handle the exception in two ways:

A. By using try-catch.

B. By using throws keyword.

We can use throws keyword to delegate the responsibility of exception handling to the caller (it may be a method or JVM) then caller method is responsible to handle that exception.

Note:- throws keyword is required only for checked exceptions and usage of throws keyword for unchecked exceptions is meaningless.

- throws keyword is ~~used for~~ (Required) only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.

- By the help of throws keyword we can provide information to the caller of the method about the exception.

* Why use custom exceptions?

→ - To catch and provide specific treatment to a subset of existing Java exceptions.

- Business Logic exceptions:
  These are the exceptions related to business logic and workflow. It's usefull for the application users or the developers to understand the exact problem.

Note: In order to create a custom exception, we need to extend the Exception class that belongs to "java.lang" package.

* Types of Errors?

→ ① Compile-time.
② Run-time.
③ Logical.

* Multithreading in Java:

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
Each part of such program is called a thread.

- So, threads are light-weight processes within a process.

"Multithreading in Java is a process of executing two or more threads simultaneously to maximum utilization of CPU."

* Thread:

Thread is basically a lightweight sub-process, a smallest unit of processing.

- Java uses threads by using a "Thread Class".

- There are two types of threads.
Ⓐ user thread.
Ⓑ daemon thread.

* Advantages of single thread:

- Reduces overhead in the application as single thread execute in the system.
- Also, it reduces the maintenance cost of the application.

* Threads can be created by using two mechanisms:

(A) Extending the Thread class:

- We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class.
- A thread begins it's life inside run() method. We create an object of our new class and call start() method to start the execution of a thread.
- Start() invokes the run() method on the Thread object.

(B) Thread Creation by implementing the Runnable Interface:

- We create a new class which implements java.lang.Runnable interface and override run() method.
- Then we instantiate a Thread object and call start() method on this object.

* Thread Class vs Runnable Interface:

- If you extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
- Thread class provides some methods like yeild(), interrupt() etc which not available in Runnable.
- Using Runnable will give you an object can be shared amongst multiple threads.

* Lifecycle and States of a Thread in Java:

A Java thread, at any point of time exist in any one of the following states:

(1) New:

In this phase, the thread is creating using class "Thread class". It remains in this state till the program starts the thread. It is also known as born thread.

(2) Runnable:

In this phase, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the schedular whether to run the thread.

(3) Running:

When the thread starts executing, then the state is changed to "running" state. The schedular selects one thread from the thread pool, and it starts executing in the application.

(4) Waiting:

This is the state when a thread has to wait. As there multiple threads are running in the application, there is a need for synconization bet" threads, Hence one thread has to wait, till the other thread gets executed. Therefore, this state is reffered as waiting state.

⑤ Dead:

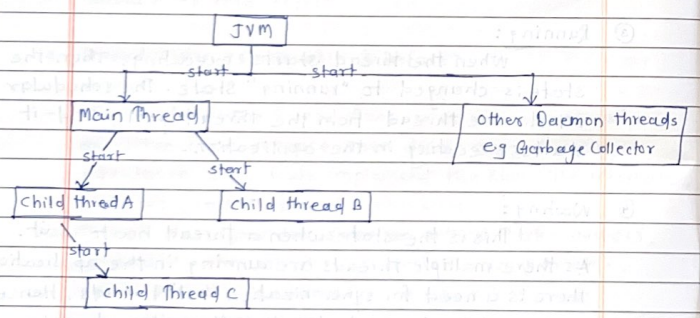
This is the state when the thread is terminated. The thread is in running state and as soon as it completed processing it is in "dead state".

* Main thread :

When a Java program starts up, one thread begin running immediately. This is wally called the main thread of our program.

* Certain properties associated with main thread:

- It is the thread from which other "child" thread will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.



* How to control Main Thread:

- To control it we must obtain a reference to it. This can be done by calling the method currentThread() which is present in Thread Class.
- This method returns a reference to the thread on which it is called. The default priority of main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

* Relationship between main() method and main thread :

For each Java program, a Main thread is created by JVM. The "Main" thread first verifies the existance of the main() method, and then it initialize the class.

* Java Thread Priority in Multithreading :

Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly.

The default priority is set to 5 as expected.
Minimum priority is set to 1.
Maximum priority is set to 10.

* What is ThreadPool in Java?

→ A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource trashing.

Since, the thread already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

* Risks in using Thread Pools:

- Deadlock.
- Thread Leakage.
- Resource Thrashing.

* Synchronization in Java:

- In multithreading, there is the asynchronous behavior of the program. If one thread is ~~waiting~~ writing some data and another thread which is reading data at the same time, might create inconsistency in the application.

- When there is a need to access the shared resources by two or more threads, then synchronization ~~appro~~ approach is ~~uti~~ utilized.

- Java has provided synchronized methods to implement synchronized behaviour.

- In this approach, once the thread reaches inside the synchronized block, then no ~~one~~ other thread can call that method on the same object. All threads have to wait till that thread finishes the synchronized block and comes out of that.

In this way, the synchronization helps in a multithreaded application. One thread has to wait till other thread finishes it's execution only then the other threads are allowed for execution.

```
Synchronized (Object){
    // Block of statement to be synchronized.
}
```

* There are 4 ways to achieve thread safety:

① Using Synchronization

② Using Volatile Keyword.

③ Using Automic Variable.

④ Using final keyword.

* Collection Framework In Java :

Any group of individual objects which are represented as a single unit is known as the collection of objects.

In Java, a seprate framework named the "Collection Framework" has been defined in JDK 1.2 which holds all the collection classes and interfaces in it.

java.util.Collection and java.util.Map are the two main root interfaces of Java collection classes.

* What is a framework :

A framework is a set of classes and interfaces which provide a ready-made architecture.

In order to implement a new feature or a class, there is no need to define a framework.

However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

* Need for a separate collection framework :

Before the collection framework was introduced, the standard methods for grouping Java Objects were Arrays or Vectors or Hashtables.

All these collections had no common interface. Therefore, through the main aim of all the collections is the same, the implementations of all these collections was defined independently and had no correlation among them.

And also, it is very difficult for the users to remember all the different methods, syntax and constructors present in every collection class.

* Advantages :

1) Consistent API :

The API has a basic set of interfaces like Collection, Set, List, Map and all the classes (ArrayList, LinkedList, Vector etc) that implement these interfaces have some common set of methods.

2) Reduces Programming Efforts :

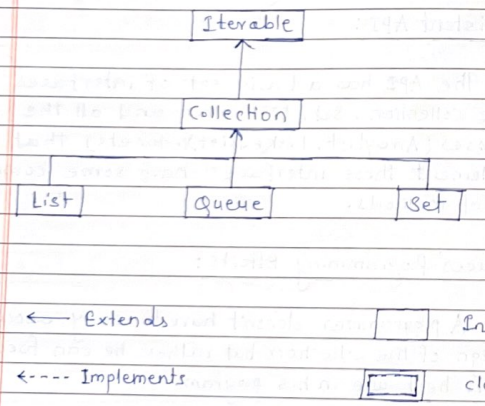
A programmer doesn't have to worry about the design of the collection but rather he can focus on its best use in his program.

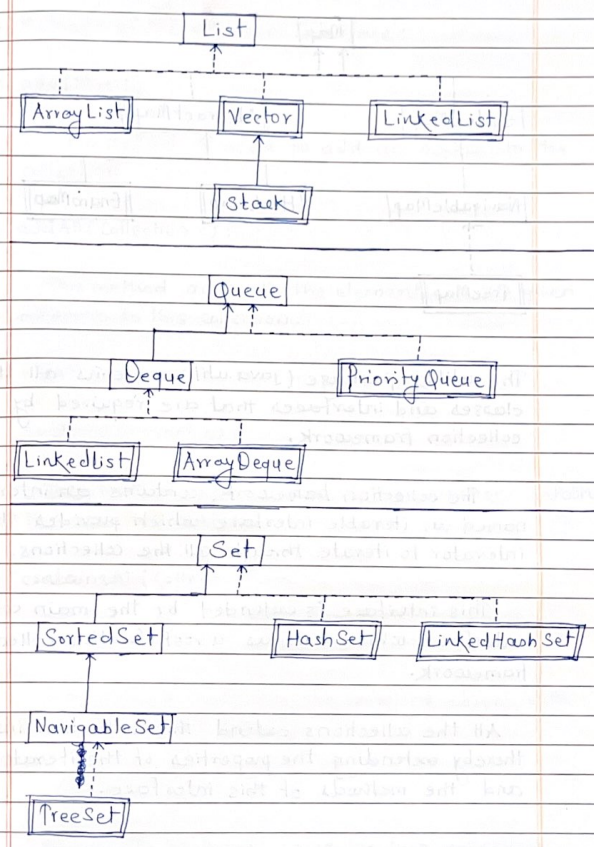
Therefore, the basic concept of object oriented programming, abstraction has been successfully implemented.

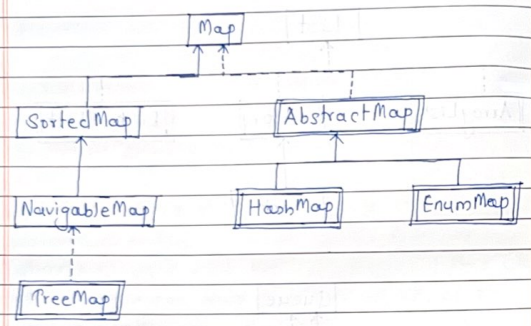
3) Increase Program Speed And Quality:

Increase performance by providing high-performance implementations of ~~that~~ useful data structures and algorithms because in this case, the programmer ~~and~~ need not think of the best implementation of a specific data structure.

He can simply use the best implementation to drastically boost the performance of his algorithm/program.







The utility package (java.util) contains all the classes and interfaces that are required by the collection framework.

The collection framework contains an interface named an iterable interface which provides the interator to iterate through all the collections.

This interface is extended by the main collection interface which acts as a root for the collection framework.

All the collections extend this collection interface thereby extending the properties of the iterator and the methods of this interface.

* Methods of the collection Interface :

① add(Object) :

This method is used to add an object to the collection.

② addAll(Collection c) :

This method adds all the elements in the given collection to this collection.

③ clear() :

④ contains(Object o) :

This method returns true if the collection contains the specified element.

⑤ containsAll(Collection c) :

⑥ equals(Object o) :

This method compares the specified object with this collection for equality.

⑦ hashCode() :

Returns the hash code value for this collection.

⑧ isEmpty():

Returns true if this collection contains no elements

⑨ iterator():

Returns an iterator over the elements in this collection

⑩ max():

Returns maximum value present in the collection.

⑪ parallelStream():

Returns a parallel Stream with this collection as it's source.

⑫ remove(object o):

Remove the given object from the collection. If there are duplicate values, then this method removes the first occurence of the object.

⑬ removeAll(Collection c):

⑭ removeIf(Predicate filter):

Remove all the elements of this collection that satisfy the given predicate.

⑮ retainAll(Collection c):

Retain only the ~~collection~~ elements in this collections that are contained in the specified collection.

⑯ size():

Returns the number of elements in the collection.

⑰ Spliterator():

Create a spliterator over the elements in this collection.

⑱ stream():

Return a sequential Stream with this collection as it's source.

⑲ toArray():

Return an array containing all the elements in this collection.

* Stream.distinct():

Returns a stream consisting of distinct elements in a stream.

① Iterable Interface:

This is a root interface for the entire collection framework. The collection interface extends the iterable interface.

Therefore, inherently, all the interfaces and classes implement this interface.

The main functionality of this interface is to provide an iterator for the collection.

Therefore it contains only one abstract method which is the iterator. It returns the

Iterator iterator();

② Collection ~~Framework~~ Interface:

This interface extends the iterable interface and is implemented by all the classes in the collection framework.

This interface contains all the basic methods which every collection has. All these methods implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation.

This interface ensures that the names of the methods are universal for all the collections.

This interface builds a foundation on which the collection classes are implemented.

③ List Interface:

- child interface of collection framework.

- The data of the List type in which we can store all the ordered collection of the objects. This allows duplicate data to be present.

- Implemented by ~~A~~ classes like Arraylist, Vector, Stack etc.

- We can instantiate a list object with any of these classes. for example.

```
List<T> al = new ArrayList<>();
List<T> ll = new LinkedList<>();
List<T> v = new Vector<>();
```

Where T is the type of the object.

* The classes which implement the List interface:

Ⓐ Arraylist Class:

- Arraylist provides dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

- The size of an Arraylist is increased automatically if the collection grows or shrinks if the objects are removed from the collection.

- Java Arraylist allows us to randomly access the list.

- Arraylist can not be used for primitive types, like int, char, etc. For such cases, we need wrapper class.

(B) LinkedList class:

- LinkedList class is an implementation of the LinkedList data structure.
  Which is a linear data structure where the elements are not stored in contigous locations and every element is a separate object with a data part and address part.

- The elements are linked using pointers and addresses.

- Each element is known as node.

ex
```
LinkedList <Integer> ll = new LinkedList <Integer>();
for (int i = 1 ; i<=5 ; i++)
    ll.add(i);
sysout (ll);
```

(C) Vector class:

- A vector provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helful in programs where lots of manipulation in the array is needed.

- This is identical to ArrayList in terms of implementation.

- The primary diffrence between a Vector and ArrayList is that Vector is synchronized and an ArrayList is non-synchronised.

```
Vector <Integer> v = new Vector <Integer>();
```

(D) Stack class:

- Stack class models and implements the "stack data structure".

- The class is based on the basic principle of LIFO.

- In addition to the basic push and pop operations, the class provides three more functions, empty, search and peek.

- The class can be referred as the subclass of vector.

Note: Stack is a subclass of vector and a legacy class. It is thread-safe which might be overhead in an environment where thread safety is not needed.

An alternate to stack is to use ArrayDequeue which is not thread-safe and has faster array implementation.

④ Queue Interface:

- A queue interface maintains the FIFO order simillar to a real-life (world) queue line.
- This interface is dedicated to storing all the elements where the order of the elements matter.
- There are various classes like PriorityQueue, ArrayDeque, etc.
- Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes.

```
Queue <T> pq = new PriorityQueue<>();
Queue <T> ad = new ArrayDeque<>();
```

Where T is the type of the object.

Ⓐ PriorityQueue Class: Implements Queue

- A priorityQueue is used when the objects are supposed to be processed based on the priority.
- Sometimes the elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
- When priority of elements matters this class is used.

⑤ Deque Interface:

- This is a very slight variation of the queue data structure.
- Dequeu also known as a double-ended queue, is a data structure where we can add and remove the elements from both ends of the queue.
- This interface extends the queue Interface.
- ArrayDeque class implements the Deque interface, we can instantiate a deque object with this class.

```
Deque <T> ad = new ArrayDeque<>();
```

where T is a type of object.

Ⓐ ArrayDeque class: ~~extends~~ Imp. Deque

- Provide us with a way to apply resizable - Array.
- This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue.
- ArrayDeque have no capacity restrictions and they grow as necessary to support usage.

⑥ Set Interface:

- A set is an unordered collection of objects in which duplicates values are not / can not be stored.
- Used when to avoid duplication and store only unique values.
- This interface is implemented by HashSet, TreeSet, LinkedHashSet etc.

```
Set<T> hs = new HashSet<>();
Set<T> lhs = new LinkedHashSet<>();
Set<T> ts = new TreeSet<>();
```

where T is the type of the object.

Ⓐ HashSet class: Implements Set

- The HashSet class is an inherent implementation of the hash table data structure.
- The objects that we insert into the HashSet do not guarentee to inserted in the same order.
- The objects are inserted based on their hashcode. This class also allows the insertion of null elements.

Ⓑ LinkedHashSet class: Implements Set

- LinkHashSet is very simillar to a Hashset.
- The diffrence is that this uses a doubly linked list to store the data and retain the ordering of all the elements.

⑦ SortedSet Interface: Extends Set

- Simillar to set Interface. The only diffrence is that this interface has extra methods that maintain the ordering of the elements.
- Used to handle the data which needs to be sorted.

```
SortedSet<T> ts = new TreeSet<>();
```

wher T is type of a object.

Ⓐ Treeset Class: implements SortedSet

- The TreeSet class uses a Tree for storage.
- The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided.
- This must be consistent with equals if it is to correctly implement the set interface.
- It can also be ordered by a Comparator provided at a set creation time, depending on which constructor is used.

Ⓑ Map Interface:

- A map is a data structure that supports the key-value pair mapping for the data.
- This interface doesn't support duplicate keys because the same key cannot have multiple mappings.
- A map is useful if there is data and we wish to perform operations on the basis of the key.
- Implemented by HashMap, TreeMap, etc.

~~<Map>~~ Map <T> hm = new ~~Map~~ HashMap <>();

Map <T> tm = new TreeMap <>();

where T is object.

Ⓐ HashMap class: Implements Map

- It stores the data in (key, Value) pairs. To access a value in a HashMap, we must know it's key. "HashMap uses a technique called Hashing."
- Hashing is a technique of converting a large string to small string that represents the same string so that the indexing and search operations are faster.
- HashMap is used internally by HashSet.