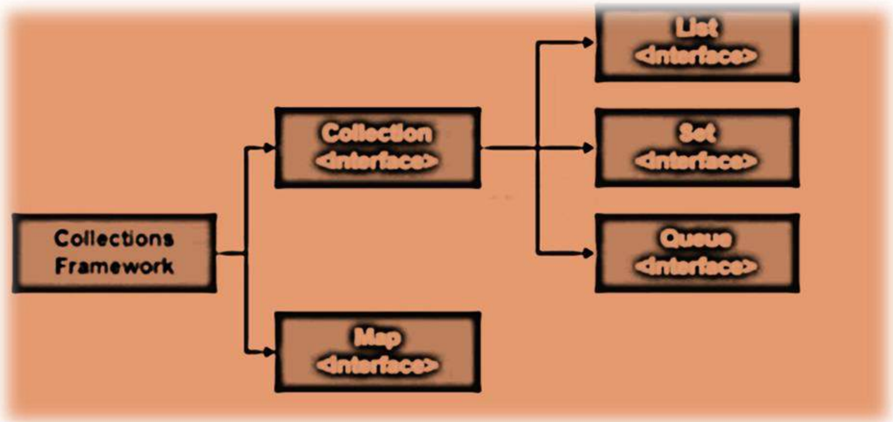
Collections

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. (Complete house plan)

Collection Framework has been defined in JDK 1.2 which holds all the Java Collection Classes and Interface in it.

- Collection interface (java.util.Collection)
- Map interface (java.util.Map)



Advantages

- Consistent API

  API has a basic set of interfaces like Collection, Set, List, or Map, all the classes. that implement these interfaces have some common set of methods.

- Reduces programming effort

  In built design in collection framework. So, focus on its best use in his program.

- Increases program speed and quality

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

Iterable Interface

It was introduced in JDK 1.5. It belongs to java.lang package. Iterable interface was introduced in JDK 1.5. It belongs to java.lang package.

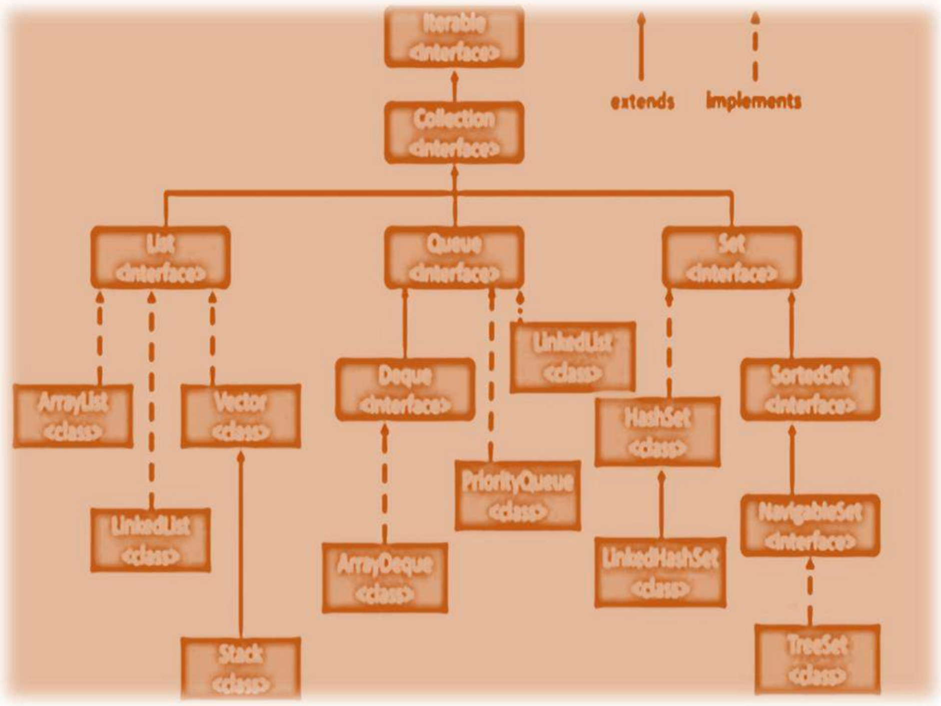
Iterator iterator();

An iterable interface allows an object to be the target of enhanced for loop (for-each loop)
--- Using enhanced for loop(for-each loop)
--- Using Iterable forEach loop
--- Using Iterator<T> interface

Collection Interface:

It extends the Iterable interface and is implemented by all the classes in the collection framework. It contains all the basic methods. It is implemented indirectly through its sub-interfaces like List, Queue, and Set.

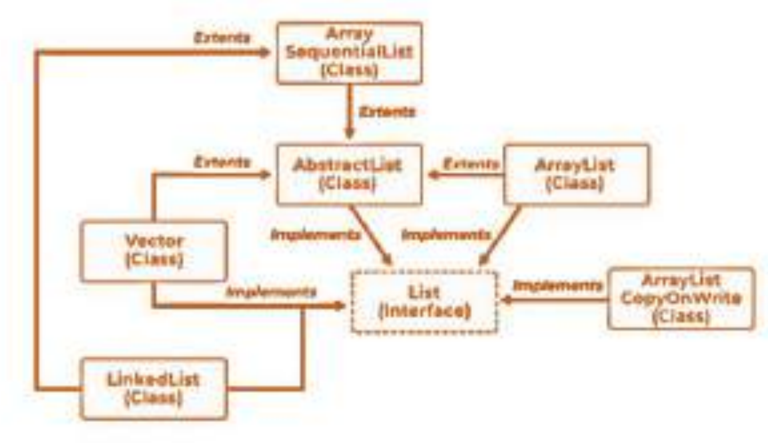
public interface Collection<E> extends Iterable<E>



1. List Interface:

It is extends the Collection Interface and is a part of java.util package. It is used to store the ordered collections of elements. So, in a Java List, you can organize and manage the data sequentially.

- Maintained the order of elements in which they are added.
- Allows the duplicate elements.
- Implementation classes are ArrayList, LinkedList, Stack, and Vector.
- Can add Null values that depend on the implementation.
- Access elements by their index and includes the listIterator() method, return
- Using ListIterator, we can traverse the list in both forward and backward dir.



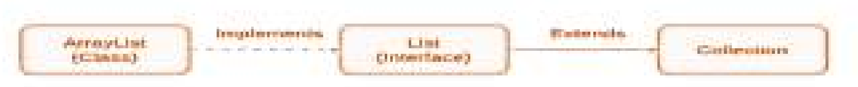
ArrayList

Java ArrayList is a part of the collections framework and it is a class of java.util package. It provides us with dynamic-sized arrays. since Java 1.2,

- Don't need to mention the size when creating ArrayList. It automatically adjusts its capacity as elements are added or removed.

- It may be slower than standard arrays, but it is helpful when the size is not known in advance.

```
ArrayList<Integer> arr = new ArrayList<Integer>();
```



- ArrayList inherits AbstractList and implements List.
- Size increases automatically as collection grows or shrinks.
- ArrayList allows random access to list elements.
- Wrapper class needed for primitive types like int.
- ArrayList is similar to C++ vector.
- ArrayList is not synchronized; use Vector for sync.

Constructors in ArrayList

Create an ArrayList class consists of various constructors which allow the possible creation of the array list.

- ArrayList(): used to build an empty array list. --- new ArrayList();
- ArrayList(Collection c): used to build an array list initialized ---- new ArrayList(c);
- ArrayList(int capacity): an array list with the initial capacity being specified. --- n

ArrayList Operations

ArrayList is a part of the Java Collections Framework and provides a dynamic array that can grow and shrink in size as elements are added or removed.

1. Creating an ArrayList: ArrayList<Type> list = new ArrayList<>();

2. Adding Elements:
   - end of the list: --- list.add(element);
   - At a specific index: --- list.add(index, element);

3. Accessing Elements
   - Get an element by index: --- Type element = list.get(index);
   - Get the size of the list: --- int size = list.size();

4. Removing Elements
   - Remove an element by index: --- list.remove(index);
   - Remove a specific element (1st): --- list.remove(Object element);
   - Remove all elements: --- list.clear();

5. Checking Elements
   - Check if an element exists: --- list.contains(element);
   - Check if the list is empty: --- list.isEmpty();
   - Check if the list contains: --- list.containsAll(Collection<?> c);

6. Modifying Elements
   - Set an element at a specific index: --- list.set(index, element);

7. Iterating through ArrayList --- Using a for loop:

8. Sorting the List:
   - Collections.sort();

9. Converting ArrayList to Array
   - Type[] array = list.toArray(new Type[0]);

10. Converting Array to ArrayList
    - ArrayList<> list = new ArrayList<>(Arrays.asList(array));

11. Sub list (Getting a part of the list)
    - ArrayList<> sublist = list.subList(fromIndex, toIndex);

12. Other Useful Methods
    - Check index of an element: --- int index = list.indexOf(element); // -1
    - Check if the list contains duplicates:

Complexity of ArrayList

Inserting Element in ArrayList	--- O(1)	O(1)
Removing Element from ArrayList	--- O(N)	O(N)
Traversing Elements in ArrayList	--- O(N)	O(1)
Replacing Elements in ArrayList	--- O(1)	O(N)

Vector

Vector class implements a growable array of objects. Vectors fall in legacy classes, but now it is fully compatible with collections. since Java 1.0

- All methods are synchronized, making it suitable for multi-threaded env
- Can store null elements.
- Provides backward compatibility with Enumeration, a legacy way of iterating over element.

```
public class Vector<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable
```

Constructors

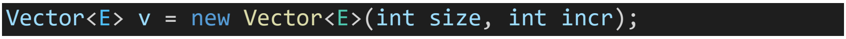
- Vector(): Creates a default vector of the initial capacity is 10.

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

- Vector(int size): Creates a vector whose initial capacity is specified by size.

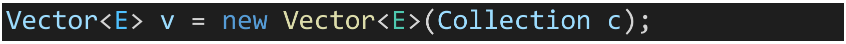
```
Vector<E> v = new Vector<E>(int size);
```

- Vector(int size, int incr): Creates a vector whose initial capacity is specified by size and increment is specified by increment.



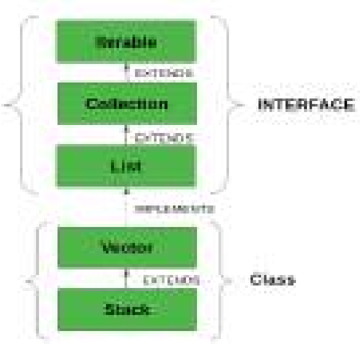
```
Vector<E> v = new Vector<E>(int size, int incr);
```

- Vector(Collection c): Creates a vector that contains the elements of collection c



```
Vector<E> v = new Vector<E>(Collection c);
```

All ArrayList operations work properly with Vectors because Vector was the first one. Therefore, it works properly with all operations.

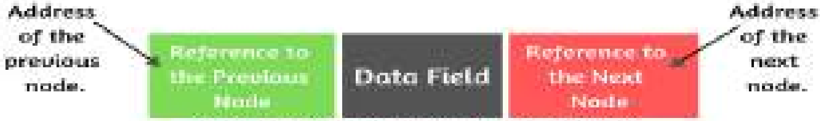


- Inserting Element in Vector o (1)
- Removing Element from Vector o (1)
- Accessing an element o (1)
- Inserting/deleting in the middle: o (n)
- Search (unsorted): O(n)
- Search (sorted): O (logn)

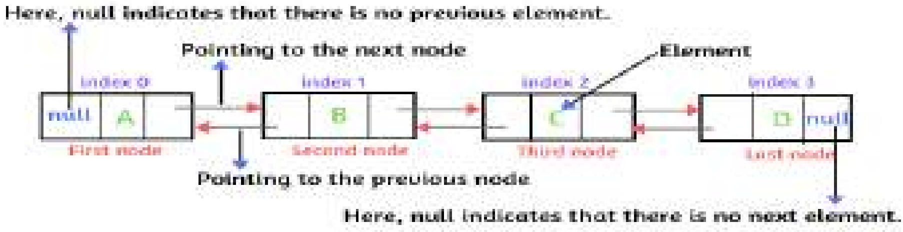
LinkedList

Linked List is a data structure that consists of a sequence of elements called nodes. Each node contains two parts:

- Data: The actual value or data that the node holds.
- Next: A reference (or pointer) to the next node in the list. ----- since Java 1.0


Address of the previous node.
Reference to the Previous Node
Data Field
Reference to the Next Node
Address of the next node.

Representation of Java LinkedList Node


Here, null indicates that there is no previous element.
Pointing to the next node
Element
index 0
index 1
index 2
index 3
null
A
B
C
D
null
First node
Second node
Third node
Last node
Pointing to the previous node
Here, null indicates that there is no next element.

A array representation of linear Doubly LinkedList in Java

The elements are linked using pointers and addresses and each element is known as a node.

Note: It also has a few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access.

A linked list can be of various types:

- Singly Linked List: Each node points to the next node in the list.
- Doubly Linked List: Each node has two references, one to the next node and one to the previous node.
- Circular Linked List: The last node points back to the first node, creating a circular structure.

Yes, most operations that work with ArrayList can be performed on a LinkedList because both ArrayList and LinkedList implement the List interface in Java. However, while both are part of the same interface and provide similar methods, the underlying implementation and performance characteristics differ.

Stack

Stack is non-primitive linear data structure. It is an ordered list in which addition of new data item and deletion of already exisiting data item is done from only one end Known as TOP OF STACK (TOP) or LIFO (last-in-first-out) Principle.

- The Stack class extends Vector and provides additional functionality specifically for stack operations, such as push, pop, peek, empty, and search.

- The Stack class can indeed be referred to as a subclass of Vector, inheriting its methods and properties.

Declaration of Stack

```
public class Stack<E> extends Vector<E>
Stack stack1 = new Stack();
```

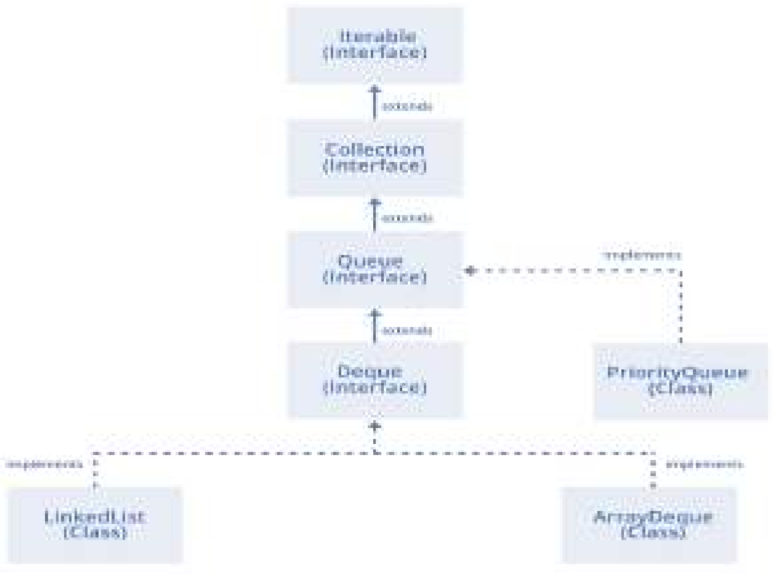
- Push: Adds elements 10, 20, 30, and 40 to the stack.
- Pop: Removes and returns the top element (40).
- Peek: Views the current top element (30).
- isEmpty: Checks if the stack has any elements left.
- size: Returns the current number of elements in the stack.

2. Queue Interface

Queue is non-primitive data structure. It is a homogenous collection of elements. In which stores and processes the data in FIFO (First In First Out) order. It is an ordered list of objects limited to inserting elements at the end of the list and deleting elements from the start of the list.

- Most implementations like PriorityQueue do not allow null elements.
- LinkedList, PriorityQueue, ArrayDeque, ConcurrentLinkedQueue (for thread-safe).
- Commonly used for Task scheduling, Message passing, and Buffer management.
- Supports iterating through elements. The order of iteration depends on the impl

```
Queue<Integer> q = new LinkedList<>();
```



PriorityQueue

PriorityQueue class are needed to be processed according to the priority, that's when the PriorityQueue comes into play.

- The PriorityQueue is based on the Priority Heap.
- The elements of the priority queue are ordered according to the natural ordering.
- No Null Elements are available in PriorityQueue it will throw NullPointerException.
- Size of the Priority Queue is dynamic, means will increase or decrease as required.
- Since PriorityQueue is not thread-safe. For PriorityBlockingQueue class

Deque Interface

Deque is related to the double-ended queue that supports adding or removing elements from either end of the data structure. It can either be used as a queue(first-in-first-out/FIFO) or as a stack(last-in-first-out/LIFO). Deque is the acronym for double-ended queue.

- Most implementations do not allow null elements, as null is used as a special return value to indicate the absence of elements.

- Use ConcurrentLinkedDeque or LinkedBlockingDeque for thread-safe operations and avoid ArrayDeque in concurrent environments as it is not thread-safe.

```
Deque<Obj> deque = new ArrayDeque<> ();
```

ArrayDeque class

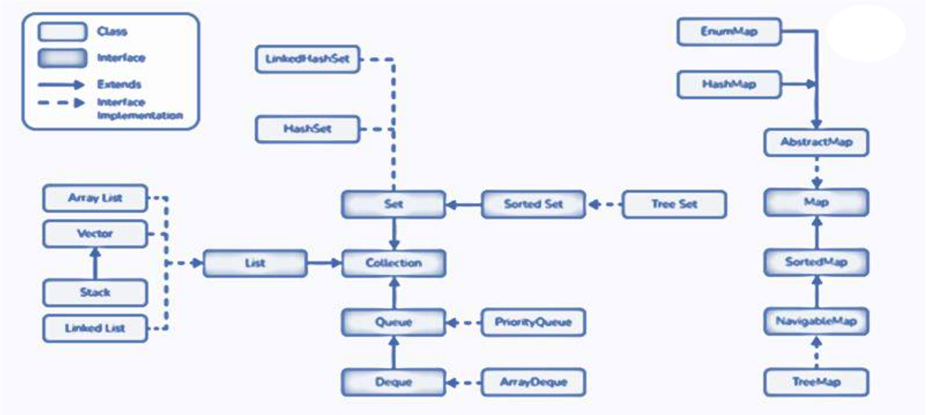
It allows users to add or remove an element from both sides of the queue. Array deques have no capacity restrictions and they grow as necessary to support usage.

3. Set Interface

Set Interface is an unordered collection of objects in which duplicate values cannot be stored. It is an interface that implements the mathematical set.

- No Specific Order: Does not maintain any specific order of elements (Exceptions: LinkedHashSet and TreeSet).
- Allows One Null Element: Most Set implementations allow a single null element.
- Implementation Classes: HashSet , LinkedHashSet and TreeSet.
- Thread-Safe Alternatives: For thread-safe operations, use ConcurrentSkipListSet or wrap a set using Collections.synchronizedSet()

Classes that implement the Set interface in Java Collections can be easily perceived from the image below as follows and are listed as follows:



HashSet

HashSet implements the Set interface of Collections Framework. It is used to store the unique elements and it doesn't maintain any specific order of elements. Since Java 1.2

- Can store the Null values.
- Uses HashMap (implementation of hash table data structure) internally.
- Also implements Serializable and Cloneable interfaces.
- HashSet is not thread-safe. So, to make it thread-safe, synchronization.

Declaring a HashSet

```
public class HashSet<E> extends AbstractSet<E>
          implements Set<E>, Cloneable, Serializable
```

Before storing an Object, HashSet checks whether there is an existing entry using hashCode() and equals() methods.

Internal Working of a HashSet

Set interface are internally backed up by Map. HashSet uses HashMap for storing its object internally. You must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet, we are passing only one value.

HashSet Methods:

@ add (E e): Adds the specified element to the set if it's not already present.

```
HashSet<String> set = new HashSet<>();
set.add("Apple");
set.add("Apple");
```

@ addAll(Collection<? extends E> c): Adds all elements of the specified collection to the set if they are not already present.

```
HashSet<String> set = new HashSet<>();
List<String> list = Arrays.asList("Apple", "Banana","Cherry");
set.addAll(list);
```

@ clear(): Removes all elements from the set.

```
set.clear();
```

@ contains(Object o): Checks if the set contains the specified element.

```
set.contains("Apple");  // returns true if "Apple"
```

@ containsAll(Collection<?> c): Checks if the set contains all elements of the specified collection.

```
set.containsAll(list);  // returns true if all elements
```

@ isEmpty(): Checks if the set is empty.

```
set.isEmpty();  // returns true if set has no elements
```

@ iterator(): Returns an iterator over the elements in the set.

```
Iterator<String> it = set.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}
```

@ remove(Object o): Removes the specified element from the set if it exists.

```
set.remove("Apple");  // returns true if "Apple" is removed
```

@ removeAll(Collection<?> c): Removes all elements of the specified collection from the set.

```
set.removeAll(list);  // removes all elements in list
```

@ retainAll(Collection<?> c): Retains only the elements in the set that are also contained in the specified collection.

```
set.retainAll(list);  // keeps only elements in set
```

@ size(): Returns the number of elements in the set.

```
set.size();  // returns the number of elements in the set
```

@ toArray(): Returns an array containing all the elements of the set.

```
Object[] arr = set.toArray();
```

@toArray(T[] a): Returns an array containing all elements of the set, with the runtime type of the array provided.

```
String[] arr = set.toArray(new String[0]);
```

@ clone(): Creates a shallow copy of the set.

```
HashSet<String> clonedSet = (HashSet<String>) set.clone();
```

@ forEach(Consumer<? super E> action): Performs the given action for each element of the set.

```
set.forEach(System.out::println);  // prints each element
```

@ spliterator(): Creates a Spliterator over the elements in the set.

```
Spliterator<String> spliterator = set.spliterator();
```

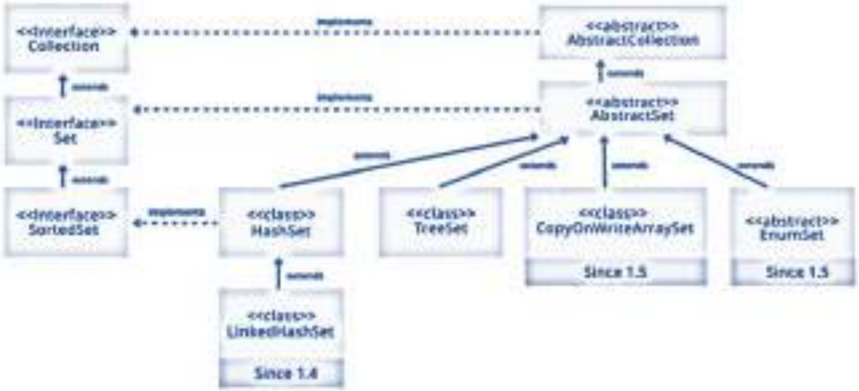
Tree Set

TreeSet is one of the most important implementations of the SortedSet interface. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. Since Java 1.2

- No duplicates: TreeSet ignores attempts to add duplicate elements.
- No null values: Null elements cause a NullPointerException.
- Navigable: Implements NavigableSet with methods like higher() and floor().
- Not thread-safe: Requires external synchronization for concurrent access.

Operations on the Set Interface

Set interface allows the users to perform the basic mathematical operation on the set.
Intersection: This operation returns all the common elements from the given two sets.
Union: This operation adds all the elements in one set with the other. For the above two
Difference: This operation removes all the values present in one set from the other set.



LinkedSet

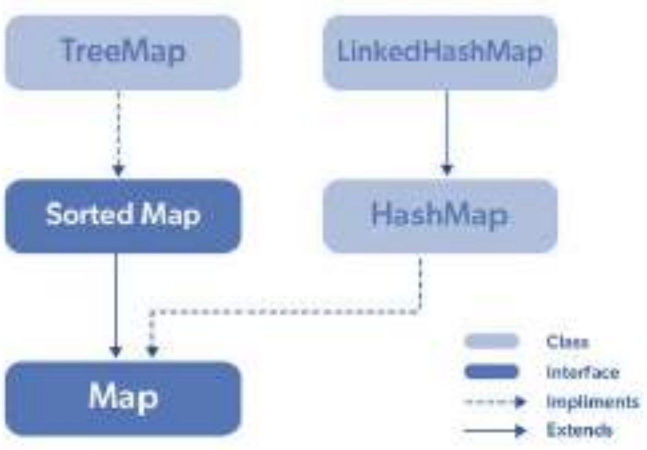
LinkedHashSet in Java implements the Set interface of the Collection Framework. It combines the functionality of a HashSet with a LinkedList to maintain the insertion order of elements.

- Stores unique elements only.
- Maintains insertion order.
- Provides faster iteration compared to HashSet.
- Allows null elements.

4. Map Interface

Map Interface represents a mapping between a key and a value. It is not a subtype of the Collection interface. Therefore, it behaves a bit differently from the rest of the collection types.

- Ensures that keys are unique. However, values can be duplicated.
- Allows one null key in implementations like HashMap and LinkedHashMap and allows multiple null values in most implementations.
- Use ConcurrentHashMap for thread-safe operations also, wrap an existing Map using Collections.synchronizedMap() for synchronized access.
- Order of a map depends on the specific implementations.
  --- TreeMap and LinkedHashMap have predictable orders.
  --- While HashMap does not.



HashMap

HashMap is a part of Java's collection since Java 1.2. To access a value, one must know its key. Hashing is a technique of converting a large String to a small String that represents the same String. A shorter value helps in indexing and faster searches.

```
public class HashMap<K,V> extends AbstractMap<K,V>
                      implements Map<K,V>, Cloneable, Serializable;
```

Creating Map Objects

```
Map hm = new HashMap();
```

LinkedHashMap

LinkedHashMap implements the Map interface and combines the features of a hash table and a linked list. It stores key-value pairs while maintaining the insertion order of the entries. It maintains the order in which elements are added. It just like

HashMap with the additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

- Stores unique key-value pairs.
- Maintains insertion order.
- Allows one null key and multiple null values.
- Fast performance for basic operations.

Declaration of LinkedHashMap

```
public class LinkedHashMap<K,V> extends HashMap<K,V>
                                        implements Map<K,V>
```

Data is stored in the form of nodes. The implementation of the LinkedHashMap is very similar to a doubly-linked list. LinkedHashMap class is not synchronized. For synchronization add Collections.synchronizedMap().

```
Map<K, V> synchronizedMap =
                Collections.synchronizedMap(new LinkedHashMap<>());
```

Advantages vs Disadvantages

✓ It maintains insertion order.	× Higher memory usage.
✓ Faster iteration.	× Slower insertion.
✓ Allows null values.	× Less efficient for large datasets.

Constructor:

--- LinkedHashMap (): default capacity (16) and load factor (0.75).

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<>();
```

--- LinkedHashMap (int initialCapacity): Creates a with a specified initial capacity.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<>(int capacity);
```

--- LinkedHashMap (int initialCapacity, float loadFactor): initial capacity and load factor.

```
LinkedHashMap<K, V> lhm = new
        LinkedHashMap<K, V>(int capacity, float fillRatio);
```

--- LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder): where the order is determined by access order if accessOrder is true.

```
        LinkedHashMap<K, V> lhm = new
            LinkedHashMap<K, V>(int capacity, float fillRatio, boolean Order);
```

Operatoins:

put(K key, V value)	--- Adds a key-value pair to the map. If the key exists
get(Object key)	--- Returns the value associated with the specified.
remove(Object key)	--- Removes the key-value pair associated.
containsKey(Obj key)	--- Checks if the map contains the specified key.
containsValue(Objval)	--- Checks if the map contains the specified value.
size()	--- Returns the number of key-value pairs in map
isEmpty()	--- Checks if the map is empty.
clear()	--- Removes all entries from the map.
keySet()	--- Returns a Set view of the keys in the map.
values()	--- Returns a Collection view of the values in map.
entrySet()	--- Returns a Set view of the key-value pairs in map.
putIfAbsent(K, V)	--- Adds the key-value pair only if the key is not
replace(K, V)	--- Replaces the value for the specified.
replace(K , oldV, newV)	--- Replaces the old value with the new value only

LinkedHashMap using a for-each loop. It used to iterate over the entries in a more functional style.

```
    for (Map.Entry<String, Integer> entry : map.entrySet()) {
      System.out.println(entry.getKey() + " = " + entry.getValue());
    }

    // Java 8 & above
    map.forEach((key, value) -> System.out.println(key + " = " + value));
```

Hashtable:

Hashtable class, implements a hash table that maps keys to values. Objects used as keys must implement the hashCode method and the equals method.

- It is similar to HashMap, but is synchronized.
- Hashtable stores key/value pair in hash table.
- Specify an object that is used as a key, and the value we want to ass to that key.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
- HashMap doesn't provide any Enum, while provides not fail-fast Enum.

it's recommended to use the Map interface or one of its implementations (such as HashMap or ConcurrentHashMap) instead of the Hashtable class.

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable

Advantages vs Disadvantages

- ✓ Thread-safe
- ✓ Simple to use

- × Obsolete
- × Limited functionality

HashTable Constructors:

--- Hashtable() creates an empty Hashtable with the default load factor of 0.75 and an initial capacity 11.

```
Hashtable<K, V> ht = new Hashtable<K, V>();
```

--- Hashtable(int initialCapacity) creates a hash table that has an initial size specified by initialCapacity and the default load factor is 0.75.

```
Hashtable<K, V> ht = new Hashtable<K, V>(int initialCapacity);
```

--- Hashtable(int size, float fillRatio) a hash table can be before it is resized upward and its Value lies between 0.0 to 1.0.

```
Hashtable<K, V> ht = new Hashtable<K, V>(int size, float fillRatio);
```

--- Hashtable(Map<? extends K,? extends V> m) creates a hash table that is initialized with the elements in m.

```
Hashtable<K, V> ht = new Hashtable<K, V>(Map m);
```

Operation

- put(K key, V value) --- Adds a key-value pair to the Hashtable.

```
table.put("name", "Kiran");
table.put(1, "One");
```

- get(Object key) --- Retrieves the value associated with the specified key.

```
String value = table.get("name");  // Returns "Kiran"
```

- remove(Obj key) --- Removes the key-value pair associated with the spec. key.

```
table.remove("name");
```

- containsKey(Obj key) --- Checks if a specified key exists in the Hashtable.

```
boolean contains = table.containsKey(1);  // true if key exists
```

- containsValue(Obj value) --- Checks if a specified value exists in the Hashtable.

```
boolean contains = table.containsValue("One"); //true if V exists
```

- keySet() --- Returns a Set of keys contained in the Hashtable.
  Set<K> keys = table.keySet();

- values() --- Returns a Collection of values contained in the Hashtable.
  Set<K> keys = table.keySet();

- size() --- Returns the number of key-value pairs in the Hashtable.
  Collection<V> values = table.values();

- isEmpty() --- Checks if the Hashtable is empty.
  boolean isEmpty = table.isEmpty(); // Returns true if empty

- clear() --- Removes all the key-value pairs from the Hashtable.
  table.clear();

- forEach(BiConsumer<? super K, ? super V> action): Performs the specified action for each entry in the Hashtable.

```
      table.forEach((key, value) -> {
         System.out.println(key + " = " + value);
});
```

Java Collections Framework (JCF) used to store, retrieve, manipulate, and communicate aggregate data. Instead of dealing with arrays (which have fixed sizes and limited capabilities), collections provide more flexible and powerful ways to handle groups of objects.

--- Storing Data Dynamically	: ArrayList, HashSet, and HashMap grow/shrink
--- Efficient Searching and Sorting	: TreeSet or TreeMap maintain sorted data.
--- Preventing Duplicates	: HashSet ensure all elements are unique.
--- Mapping Keys to Values	: Store key-value pairs, like a dictionary.
--- Concurrency and Thread Safety	: Thread-safe collection variants like CHM, COWA.
--- Queue and Stack Operations	: Queue, Deque, Stack, and PriorityQueue F/LIFO.
--- Custom Sorting and Filtering	: Comparator, Comparable, and streams.

Interface	Description	Common Implementations
List	Ordered, allows duplicates	ArrayList, LinkedList
Set	Unordered, no duplicates	HashSet, TreeSet, LinkedHashSet
Queue	FIFO operations	LinkedList, PriorityQueue
Map	Key-value pairs	HashMap, TreeMap, LinkedHashMap

Dictionary Class:
Dictionary class is an abstract class that represents a collection of key-value pairs, where keys are unique and used to access the values. It was part of the Java Collections Framework and it was introduced in Java 1.0 but has been largely replaced by the Map interface since Java 1.2.

- Stores key-value pairs, where keys are unique.
- Provides basic operations like insert, retrieve, and remove key-value pairs.
- Keys and values are stored as Object.
- Limited functionality compared to Map implementations.

Declaration: public abstract class Dictionary<K, V> extends Object

Pros & Cons

✓ Legacy Support
✓ Simple to use

× Obsolete
× Limited functionality

Abstract Class:
An abstract class in object-oriented programming is a class that cannot be instantiated directly and is meant to be inherited by other classes. It serves as a blueprint for other classes.

--- Cannot create objects
--- Can have abstract methods
--- Can have regular methods and properties
--- Enforces a contract

Abstract classes are useful helps a lot when designing clean, flexible, and scalable code.

Feature	Abstract Class	Interface
Can contain abstract methods?	Yes	Yes
Can contain impl methods?	Yes	Yes (since Java 8 using default/static)
Can contain variable?	Yes (with any access modifier)	Yes (but they are public static final by default)
Multiple Inheritance?	✕ No (Java supports only single)	☑ Yes (a class can impl)
Constructor allowed?	☑ Yes	✕ No
Access Modifiers supporte?	Yes (public, protected, private)	Only public or default
When to use?	When you need a base class with some shared logic	When you need to define a contract/structure

@ Abstract Class = Blueprint + Some parts already built

🏗 Think of it like a half-built house: the foundation is there; you need to finish it.

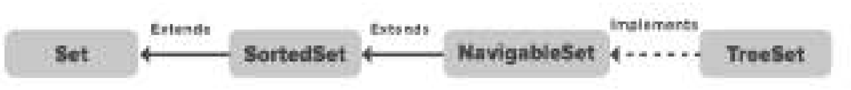
@ Interface = Pure contract, no implementation

📋 Think of it like a checklist: the class must fulfil all items on the list.

Some important interface

1. SortedSet Interface --- is present in java.util package extends the Set interface present in the collection framework.

   @ Automatic Sorting
   @ Unique Elements
   @ Null Handling



2. Comparator interface --- is used to sort the objects of user-defined classes. It allows us to define custom comparison logic outside of the class for which instances we want to sort.

   @ We need multiple sorting strategies for a class.
   @ When we want to keep the sorting logic separate from the class.

   public int compare(Object obj1, Object obj2):

   @ *When to Use Comparator*
   List of Student objects, containing fields like roll no, name, address, DOB, email... and we need to sort them based on roll no or name. We use Comparator here to define that logic separately.

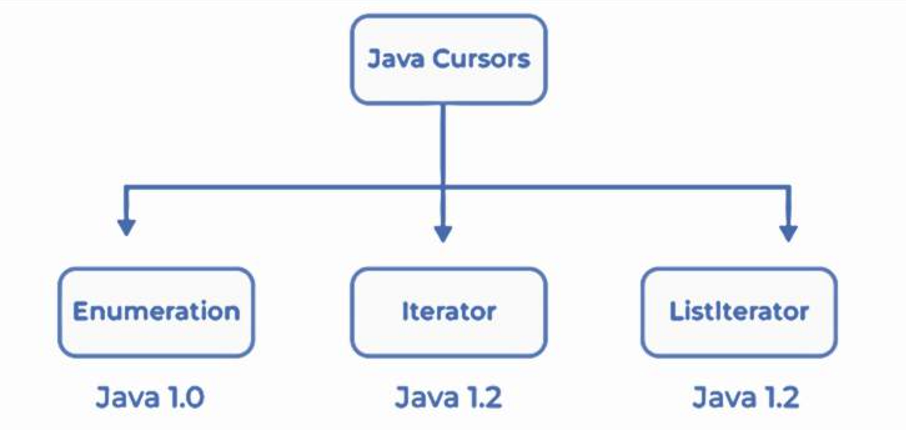
3. Comparable Interface --- is used to define the natural ordering of objects for a user-defined class. it provides a compareTo() method to compare instances of the class. A class has to implement a Comparable interface to define its natural ordering.

4. Iterator --- is an interface used to traverse elements in a Collection sequentially. It provides methods like hasNext(), next(), and remove() to loop through collections and perform manipulation.

   A Java Cursor is an Iterator, that is used to iterate or traverse or retrieve a Collection or Stream object's elements one by one.

Types of Cursors:
There are 3 cursors in Java as mentioned. SplitIterator can also be considered as a cursor as it is a type of Iterator only.

Important Point: Iterator is a concrete implementation of a cursor. The cursor refers to the general concept of traversing a collection but the iterator is one specific implementation of that concept in Java.



@ Collection is an object of any class implementing the Collection interface.

---> Iterator<E> itr = collection.iterator();

@ ListIterator extends Iterator and allows both forward and backward traversal. It also supports element modification during iteration.

---> *ListIterator<E> listItr = list.listIterator();*

@ Enumeration is an older interface from Java 1.0 used for iterating over collections. It only supports forward iteration and does not allow element removal.

---> *Enumeration<E> e = collection.elements();*

Feature / Interface	**Enumeration**	**Iterator**	**ListIterator**
Traversal Direction	Forward only	Forward only	Forward & Backward
Remove Elements?	✗ No	☑ Yes	☑ Yes
Add/Replace?	✗ No	✗ No	☑ Yes
Used with	Legacy classes	Collections	Lists only
Fail-fast?	✗ No	☑ Yes	☑ Yes