Java Unit - 6

Collections : Creating a collection by using generics, Implementing an ArrayList, Implementing TreeSet using Comparable and Comparator interfaces, Implementing a HashMap, Implementing a Deque.

Java Database Programming : Introduction to JDBC, JDBC Drivers, CRUD operation Using JDBC, Connecting to non-conventional Databases.

Collections

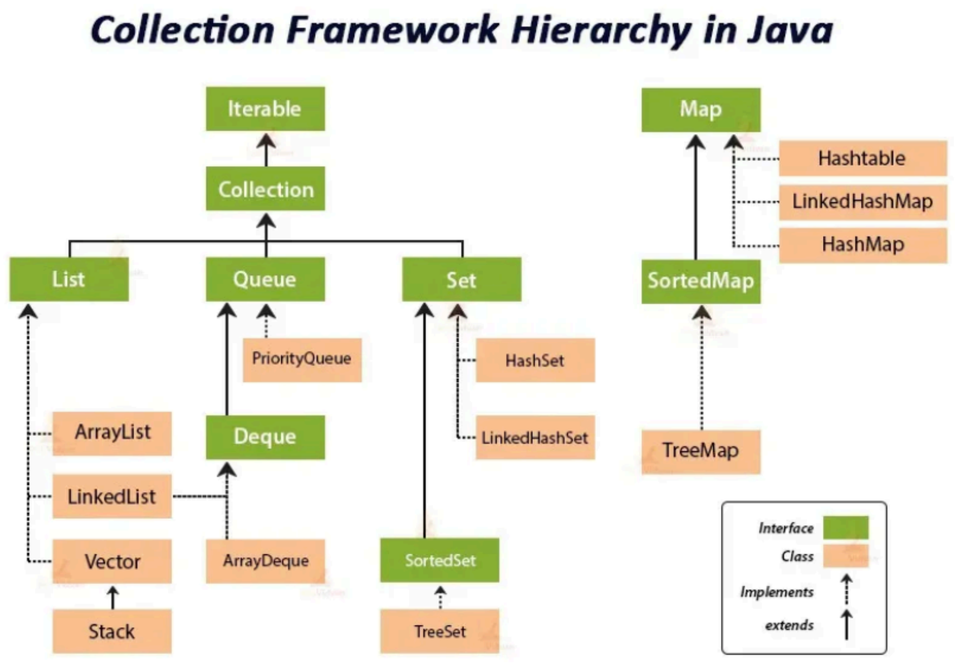
Java Collections Framework

The Java Collections Framework is a unified architecture that provides a set of **interfaces and classes** for storing and manipulating groups of data using various **data structures and algorithms**.

For example, the `LinkedList` class in this framework implements a **doubly-linked list** data structure, allowing efficient insertion and removal of elements.

Core Components

The framework includes a variety of **interfaces**, such as `List`, `Set`, `Queue`, and `Map`. These interfaces define common operations that can be performed on collections, like adding, removing, or accessing elements.



Why the Collections Framework?

The Java collections framework provides various data structures and algorithms that can be used directly. This has two main advantages:

- We do not have to write code to implement these data structures and algorithms manually.
- Our code will be much more efficient as the collections framework is highly optimized.

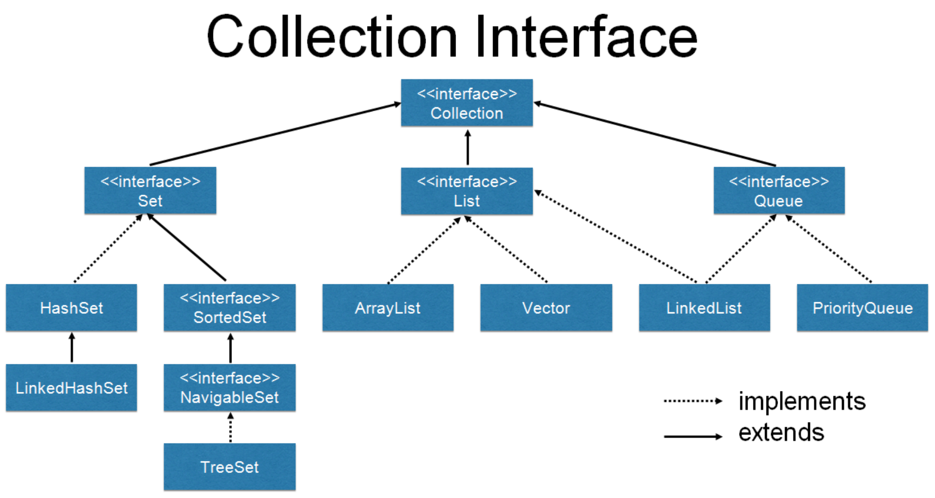
Moreover, the collections framework allows us to use a specific data structure for a particular type of data. Here are a few examples,

- If we want our data to be unique, then we can use the **Set** interface provided by the collections framework.
- To store data in key/value pairs, we can use the **Map** interface.
- The **ArrayList** class provides the functionality of resizable arrays.

Java Collection Interface

The `Collection` interface is the **root interface** in the Java Collections Framework hierarchy. It defines the basic operations that can be performed on a group of elements.

Java does not directly provide implementations of the `Collection` interface itself. Instead, it offers implementations for its **subinterfaces**, such as `List`, `Set`, and `Queue`, which define more specific behaviors for different types of collections.



Collections Framework vs. Collection Interface

The **Collections Framework** is a comprehensive set of **interfaces and classes** in Java used to store, manage, and manipulate groups of objects efficiently.

Within this framework, the **`Collection` interface** serves as the **root interface** for most collection types, such as `List`, `Set`, and `Queue`.

However, the framework also includes other important interfaces that are **not part of the `Collection` hierarchy**, such as:

- **Map –** for key-value pair data structures.
- **Iterator –** for traversing elements in a collection.

Subinterfaces of the Collection Interface

The Collection interface has several **subinterfaces**, each designed for specific types of data handling. These subinterfaces inherit all the methods defined in the Collection interface and may also introduce additional features tailored to their specific use.

Key Subinterfaces of the Collection Interface:

1. List Interface

The List interface represents an **ordered collection** of elements, similar to an array. It allows elements to be **added, accessed, or removed by index**, and it supports **duplicate elements**.

Classes that Implement the List Interface

Since `List` is an interface, you cannot create objects directly from it. Instead, Java provides several **concrete classes** that implement the `List` interface and offer its functionalities.

Here are the main classes that implement the `List` interface:

- ArrayList
- LinkedList
- Vector
- Stack

These classes are part of the **Java Collections Framework** and each has its own way of storing and managing elements, while still following the behavior defined by the `List` interface.

How to Use the List Interface in Java

To use the `List` interface in Java, you need to **import it from the `java.util` package**.

Example:

```
public static void main(String[] args) {          // Ummed Singh

       int[] arr1 = new int[10];

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

       arr.add(10);

       arr.add(20);

       arr.add(50);

       arr.add(20);

       arr.add(30);

//          Size + size/2 = 10 + 5 = 15;

       System.out.println(arr.get(1));

       System.out.println(arr.size());

       System.out.println(arr.remove(0));

       System.out.println(arr.get(0));

       System.out.println(arr.size());

   }
```

Methods of the List Interface

The List interface inherits all the methods from the Collection interface, as Collection is a superinterface of List.

In addition to the methods provided by the Collection interface, the List interface also includes methods that are specific to handling ordered collections and managing elements by index.

Some of the commonly used methods from the Collection interface, which are also available in the List interface, include:

Methods	**Description**
add()	adds an element to a list
addAll()	adds all elements of one list to another
get()	helps to randomly access elements from lists
iterator()	returns iterator object that can be used to sequentially access elements of lists
set()	changes elements of lists
remove()	removes an element from the list
removeAll()	removes all the elements from the list
clear()	removes all the elements from the list (more efficient than removeAll())
size()	returns the length of lists
toArray()	converts a list into an array
contains()	returns true if a list contains specific element

Implementation of the List Interface

1. Implementing the ArrayList Class

In Java, the `ArrayList` class is used to implement the functionality of **resizable arrays**. It provides a flexible and dynamic way to store elements, as it implements the `List` interface of the Collections Framework.

Java ArrayList vs. Array

In Java, when using an array, you must specify its size at the time of declaration. Once the size is set, it cannot be changed.

```
int[] numbers = new int[5];  // Fixed size array
```

However, this can be limiting if the number of elements is unknown or changes frequently. To solve this problem, Java provides the `ArrayList` class, which allows for **resizable arrays**.

Unlike arrays, `ArrayList` can automatically adjust its **capacity** when elements are added or removed. This makes `ArrayList` a **dynamic array**, offering more flexibility and ease of use in scenarios where the size of the collection may change over time.

Example

```
class CustomArrayList<E>{                              // Ummed Singh
   private static final int DEFAULT_CAPACITY = 2;
   private Object[] elements;
   private int size = 0;
   public CustomArrayList(){
       elements = new Object[DEFAULT_CAPACITY];
   }
   public void add(E ele){
       ensureCapacity();
       elements[size] = ele;
       size++;
   }
   public E get(int index){
       checkIndex(index);
       return (E) elements[index];
   }
//    {10, 20, 30, 40}; -> {10, 30, 40, 40} -> {10, 20, 30,
null} size--;
   public void remove(int index){
       checkIndex(index);
       for (int i = index; i<size -1; i++){
           elements[i] = elements[i+1];
       }
       elements[size-1] = null;
       size--;
   }

   void checkIndex(int index){
       if(index < 0 || index >= size){
           throw new IndexOutOfBoundsException("Index: " + index
+ " Size: " +size);
       }
   }

   void ensureCapacity(){
       if(size == elements.length){
           int newCapacity = elements.length + (elements.length
/2);
           elements = Arrays.copyOf(elements, newCapacity);
       }
   }
   int size(){
       return size;
   }

}
public class ArrayListImplementationDemo {

   public static void main(String[] args) {

       ArrayList<Integer> list = new ArrayList<>();
       CustomArrayList<Integer> cusList = new
CustomArrayList<>();
       cusList.add(10);
       cusList.add(20);
       cusList.add(30);

       System.out.println(cusList.size());
       System.out.println(cusList.get(4));
       cusList.remove(1);
       System.out.println(cusList.size());
   }
}
```

Methods of ArrayList Class

Here are some more ArrayList methods that are commonly used.

Methods	Descriptions
size()	Returns the length of the arraylist.
sort()	Sort the arraylist elements.
clone()	Creates a new arraylist with the same element, size, and capacity.
contains()	Searches the arraylist for the specified element and returns a boolean result.
ensureCapacity()	Specifies the total element the arraylist can contain.
isEmpty()	Checks if the arraylist is empty.
indexOf()	Searches a specified element in an arraylist and returns the index of the element.

Java Vector

The `Vector` class is another implementation of the **`List` interface** in Java that allows us to create **resizable arrays**, similar to the `ArrayList` class. However, there are some differences in how `Vector` behaves, particularly regarding **thread safety** and synchronization.

Java Vector vs. ArrayList

Both **`ArrayList`** and **`Vector`** implement the `List` interface and offer similar functionality, but there are key differences between them:

1. **Synchronization:**
    - **Vector:** Every operation in the `Vector` class is synchronized, meaning it automatically locks each operation to ensure that only one thread can access the vector at a time. This helps prevent issues when multiple threads are accessing the vector simultaneously, but it can make the `Vector` class **less efficient** because of the overhead caused by continuous locking.
    - **ArrayList:** Methods in the `ArrayList` class are **not synchronized**. If synchronization is needed, you can wrap an `ArrayList` with the `Collections.synchronizedList()` method to synchronize the list as a whole.
2. **Thread Safety:**
    - **Vector:** Built-in thread safety due to synchronization, but this comes at the cost of performance.
    - **ArrayList:** No built-in synchronization; better for single-threaded environments or when explicit synchronization is handled manually.

Note: It is generally recommended to use ArrayList instead of Vector, as Vector is less efficient due to its synchronization overhead.

Creating a Vector

To create a `Vector` in Java, you can use the following syntax:

```
Vector<Type> vector = new Vector<>();
```

Where `Type` represents the type of elements you want to store in the vector (e.g., `Integer`, `String`, etc.).

Other Vector Methods

Methods	Descriptions
set()	changes an element of the vector
size()	returns the size of the vector
toArray()	converts the vector into an array
toString()	converts the vector into a String
contains()	searches the vector for specified element and returns a boolean result

Implementing the LinkedList Class

Example:

```
public static void main(String[] args) {          // Ummed Singh
//          1. pre
//          2. next
//          3. data

       LinkedList<Integer> list = new LinkedList<>();
       list.add(10);
       System.out.println(list.get(1));

   }
```

Queue Interface

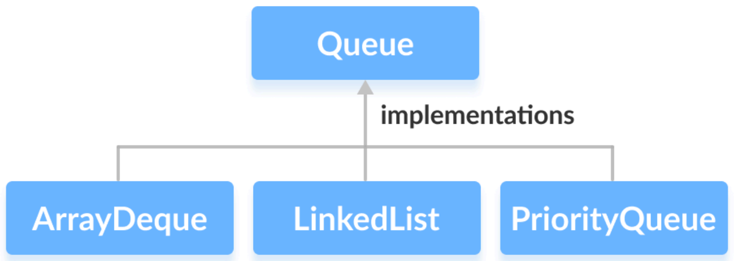
The Queue interface is used when we want to store and access elements in a First In, First Out manner.

The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface.

Classes that Implement Queue

Since the Queue is an interface, we cannot provide the direct implementation of it.
In order to use the functionalities of Queue, we need to use classes that implement it:

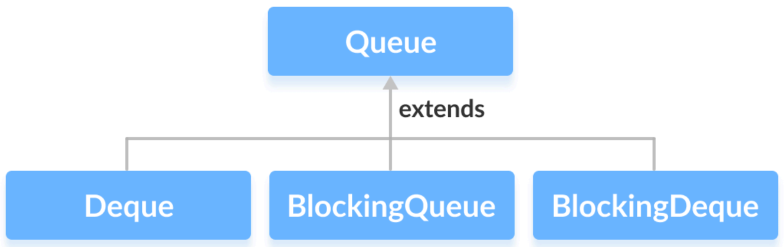
- ArrayDeque
- LinkedList
- PriorityQueue



Interfaces that extend Queue

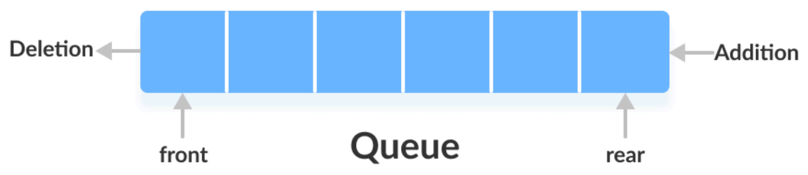
The Queue interface is also extended by various subinterfaces:

- Deque
- BlockingQueue
- BlockingDeque



Working of Queue Data Structure

In queues, elements are stored and accessed in First In, First Out manner. That is, elements are added from the behind and removed from the front.



How to use Queue?

In Java, we must ***import java.util.Queue*** package in order to use Queue.

```
// LinkedList implementation of Queue

Queue<String> animal1 = new LinkedList<>();

// Array implementation of Queue

Queue<String> animal2 = new ArrayDeque<>();

// Priority Queue implementation of Queue

Queue<String> animal3 = new PriorityQueue<>();
```

Methods of Queue

The Queue interface includes all the methods of the Collection interface. It is because Collection is the super interface of Queue.
Some of the commonly used methods of the Queue interface are:

- **add()** - Inserts the specified element into the queue. If the task is successful, **add()** returns true, if not it throws an exception.
- **offer()** - Inserts the specified element into the queue. If the task is successful, **offer()** returns true, if not it returns false.
- **element()** - Returns the head of the queue. Throws an exception if the queue is empty.

- **peek() -** Returns the head of the queue. Returns null if the queue is empty.
- **remove()** - Returns and removes the head of the queue. Throws an exception if the queue is empty.
- **poll()** - Returns and removes the head of the queue. Returns null if the queue is empty.

Implementation of the Queue Interface

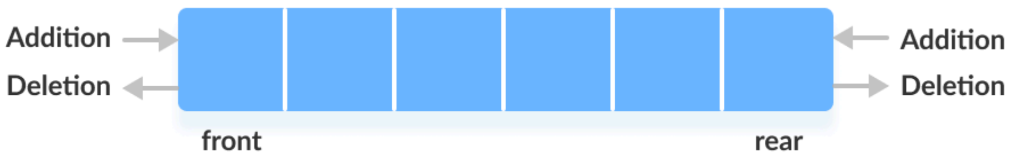
```
Queue<Integer> arr = new ArrayDeque<>();
Queue<Integer> queue = new LinkedList<>();
Queue<Integer> list = new PriorityQueue<>();
```

Java Deque Interface

The Deque interface of the Java collections framework provides the functionality of a double-ended queue. It extends the Queue interface.

Working of Deque

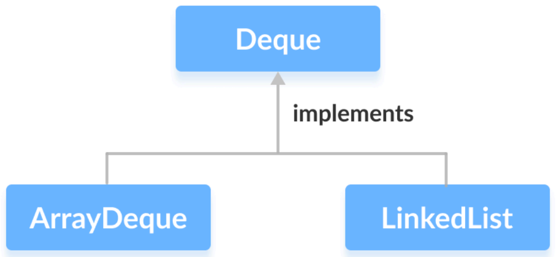
In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can insert and remove elements from both front and rear.



Classes that implement Deque

In order to use the functionalities of the Deque interface, we need to use classes that implement it:

- ArrayDeque
- LinkedList



How to use Deque?

In Java, we must import the java.util.Deque package to use Deque.

```
// Array implementation of Deque                          // Ummed Singh
Deque<String> animal1 = new ArrayDeque<>();

// LinkedList implementation of Deque
Deque<String> animal2 = new LinkedList<>();
```

Methods of Deque

Since Deque extends the Queue interface, it inherits all the methods of the Queue interface. Besides methods available in the Queue interface, the Deque interface also includes the following methods:

- **addFirst() -** Adds the specified element at the beginning of the deque. Throws an exception if the deque is full.
- **addLast() -** Adds the specified element at the end of the deque. Throws an exception if the deque is full.
- **offerFirst() -** Adds the specified element at the beginning of the deque. Returns false if the deque is full.
- **offerLast() -** Adds the specified element at the end of the deque. Returns false if the deque is full.
- **getFirst() -** Returns the first element of the deque. Throws an exception if the deque is empty.
- **getLast() -** Returns the last element of the deque. Throws an exception if the deque is empty.
- **peekFirst() -** Returns the first element of the deque. Returns null if the deque is empty.
- **peekLast() -** Returns the last element of the deque. Returns null if the deque is empty.
- **removeFirst() -** Returns and removes the first element of the deque. Throws an exception if the deque is empty.

- **removeLast() -** Returns and removes the last element of the deque. Throws an exception if the deque is empty.
- **pollFirst() -** Returns and removes the first element of the deque. Returns null if the deque is empty.
- **pollLast() -** Returns and removes the last element of the deque. Returns null if the deque is empty.

Deque as Stack Data Structure

The Stack class of the Java Collections framework provides the implementation of the stack. However, it is recommended to use Deque as a stack instead of the Stack class. It is because methods of Stack are synchronized.
Here are the methods the Deque interface provides to implement stack:

- **push() -** adds an element at the beginning of deque
- **pop() -** removes an element from the beginning of deque
- **peek() -** returns an element from the beginning of deque

---

Implementation of Deque in ArrayDeque Class

Example:

```
public static void main(String[] args) {           // Ummed Singh
       Deque<Integer> deque = new ArrayDeque<>(2);

       deque.addFirst(10);
       deque.addFirst(20);
       deque.addFirst(30);
       deque.addFirst(40);

       deque.offerFirst(10);
       deque.offerFirst(20);
       deque.offerFirst(30);

//        System.out.println(deque.peekLast());
       System.out.println(deque);
   }
```

Set Interface

The **Set** interface allows us to store elements in different sets similar to the set in mathematics. It cannot have duplicate elements.

Java List vs. Set

Both the List interface and the Set interface inherit the Collection interface. However, there exists some difference between them.

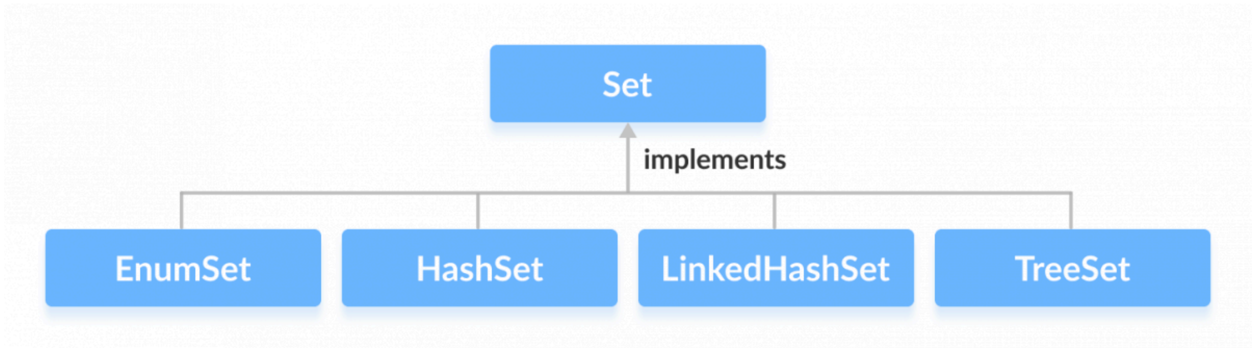
- Lists can include duplicate elements. However, sets cannot have duplicate elements.
- Elements in lists are stored in some order. However, elements in sets are stored in groups like sets in mathematics.

Classes that implement Set

Since Set is an interface, we cannot create objects from it.
In order to use functionalities of the Set interface, we can use these classes:

- HashSet
- LinkedHashSet
- EnumSet
- TreeSet

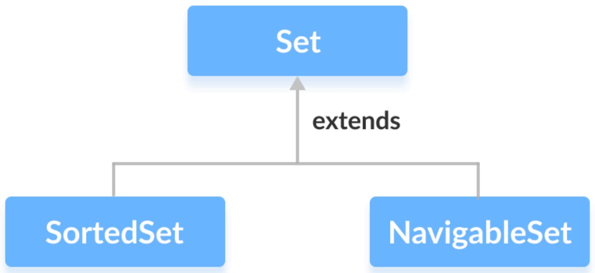
These classes are defined in the Collections framework and implement the Set interface.



Interfaces that extend Set

The Set interface is also extended by these subinterfaces:

- SortedSet
- NavigableSet



How to use Set?

In Java, we must ***import java.util.Set*** package in order to use Set.

```
// Set implementation using HashSet                        // Ummed Singh
Set<String> animals = new HashSet<>();
```

Here, we have created a Set called animals. We have used the HashSet class to implement the Set interface.

Methods of Set

The Set interface includes all the methods of the Collection interface. It's because Collection is a super interface of Set.
Some of the commonly used methods of the Collection interface that's also available in the Set interface are:

- **add() -** adds the specified element to the set
- **addAll() -** adds all the elements of the specified collection to the set
- **iterator() -** returns an iterator that can be used to access elements of the set sequentially
- **remove() -** removes the specified element from the set
- **removeAll() -** removes all the elements from the set that is present in another specified set
- **retainAll() -** retains all the elements in the set that are also present in another specified set
- **clear() -** removes all the elements from the set
- **size() -** returns the length (number of elements) of the set
- **toArray() -** returns an array containing all the elements of the set
- **contains() -** returns true if the set contains the specified element
- **containsAll() -** returns true if the set contains all the elements of the specified collection

- **hashCode() -** returns a hash code value (address of the element in the set)

Set Operations

The Java Set interface allows us to perform basic mathematical set operations like union, intersection, and subset.

- **Union -** to get the union of two sets x and y, we can use x.addAll(y)
- **Intersection -** to get the intersection of two sets x and y, we can use x.retainAll(y)
- **Subset -** to check if x is a subset of y, we can use y.containsAll(x)

Implementation of the Set Interface

Implementing HashSet Class

```
import java.util.Set;                                    // Ummed Singh

import java.util.HashSet;

public class HashSetDemo {

   public static void main(String[] args) {

       // Creating a set using the HashSet class

       Set<Integer> set1 = new HashSet<>();

       // Add elements to the set1

       set1.add(2);

       set1.add(3);

       System.out.println("Set1: " + set1);

       // Creating another set using the HashSet class

       Set<Integer> set2 = new HashSet<>();


       // Add elements
       set2.add(1);

       set2.add(2);

       System.out.println("Set2: " + set2);



       // Union of two sets

       set2.addAll(set1);

       System.out.println("Union is: " + set2);

   }

}
```

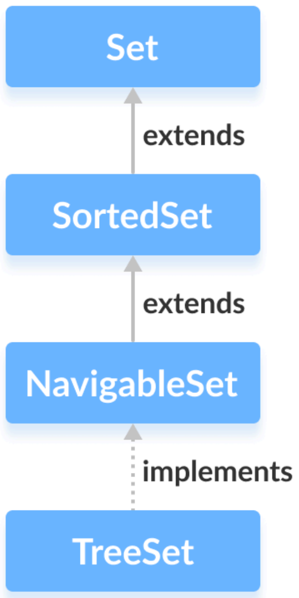
Implementing TreeSet Class

```
import java.util.Set;                                        // Ummed Singh
import java.util.TreeSet;
import java.util.Iterator;
public class TreeSetDemo {
   public static void main(String[] args) {
       // Creating a set using the TreeSet class
       Set<Integer> numbers = new TreeSet<>();

       // Add elements to the set
       numbers.add(2);
       numbers.add(3);
       numbers.add(1);
       System.out.println("Set using TreeSet: " + numbers);

       // Access Elements using iterator()
       System.out.print("Accessing elements using iterator():
");
       Iterator<Integer> iterate = numbers.iterator();
       while(iterate.hasNext()) {
           System.out.print(iterate.next());
           System.out.print(", ");
```

```
            }

        }
}
```

TreeSet

The TreeSet class of the Java collections framework provides the functionality of a tree data structure.
It extends the NavigableSet interface.



Creating a TreeSet

In order to create a tree set, we must import the java.util.TreeSet package first.
Once we import the package, here is how we can create a TreeSet in Java.

```
TreeSet<Integer> numbers = new TreeSet<>();
```

Here, we have created a TreeSet without any arguments. In this case, the elements in TreeSet are sorted naturally (ascending order).
However, we can customize the sorting of elements by using the Comparator interface.

Methods of TreeSet

The TreeSet class provides various methods that allow us to perform various operations on the set.

Insert Elements to TreeSet

- **add() -** inserts the specified element to the set
- **addAll() -** inserts all the elements of the specified collection to the set

---

Access TreeSet Elements

To access the elements of a tree set, we can use the iterator() method. In order to use this method, we must ***import java.util.Iterator*** package.

Remove Elements

- **remove() -** removes the specified element from the set
- **removeAll() -** removes all the elements from the set

Methods for Navigation

Since the TreeSet class implements NavigableSet, it provides various methods to navigate over the elements of the tree set.

1. first() and last() Methods

- **first() -** returns the first element of the set
- **last() -** returns the last element of the set

---

2. ceiling(), floor(), higher() and lower() Methods

- **higher(element) -** Returns the lowest element among those elements that are greater than the specified element.
- **lower(element) -** Returns the greatest element among those elements that are less than the specified element.
- **ceiling(element) -** Returns the lowest element among those elements that are greater than the specified element. If the element passed exists in a tree set, it returns the element passed as an argument.
- **floor(element) -** Returns the greatest element among those elements that are less than the specified element. If the element passed exists in a tree set, it returns the element passed as an argument.

---

3. pollfirst() and pollLast() Methods

- **pollFirst() -** returns and removes the first element from the set
- **pollLast() -** returns and removes the last element from the set

---

4. headSet(), tailSet() and subSet() Methods

headSet(element, booleanValue)

- The headSet() method returns all the elements of a tree set before the specified element (which is passed as an argument).
- The booleanValue parameter is optional. Its default value is false.
- If true is passed as a booleanValue, the method returns all the elements before the specified element including the specified element.

---

tailSet(element, booleanValue)

- The tailSet() method returns all the elements of a tree set after the specified element (which is passed as a parameter) including the specified element.
- The booleanValue parameter is optional. Its default value is true.
- If false is passed as a booleanValue, the method returns all the elements after the specified element without including the specified element.

---

subSet(e1, bv1, e2, bv2)

- The subSet() method returns all the elements between e1 and e2 including e1.
- The bv1 and bv2 are optional parameters. The default value of bv1 is true, and the default value of bv2 is false.
- If false is passed as bv1, the method returns all the elements between e1 and e2 without including e1.
- If true is passed as bv2, the method returns all the elements between e1 and e2, including e1.

Set Operations

The methods of the TreeSet class can also be used to perform various set operations.

Union of Sets

To perform the union between two sets, we use the addAll() method.

Intersection of Sets

To perform the intersection between two sets, we use the retainAll() method.

Difference of Sets

To calculate the difference between the two sets, we can use the removeAll() method.

Subset of a Set

To check if a set is a subset of another set or not, we use the containsAll() method.

```
public static void main(String[] args) {          // Ummed Singh

   HashSet<Integer> hashSet = new HashSet<>(Arrays.asList(10,
40, 20, 60, 30));

   System.out.println("Hashset: "+ hashSet);

   TreeSet<Integer> evenNumber = new TreeSet<>();
   evenNumber.add(2);
   evenNumber.add(8);
   System.out.println("Even number: " + evenNumber);
   // Creating a set using the TreeSet class
   TreeSet<Integer> numbers = new TreeSet<>();
   // Add elements to the set
   numbers.add(2);
   numbers.add(3);
   numbers.add(1);
   numbers.add(5);
   numbers.add(4);
   System.out.println("Set using TreeSet: " + numbers);

   numbers.addAll(evenNumber);
   System.out.println("Union: "+ numbers);

   numbers.retainAll(evenNumber);
   System.out.println("Intersection: "+ numbers);

   numbers.removeAll(evenNumber);
   System.out.println(numbers);

   System.out.println(numbers.contains(6));
}
```

Other Methods of TreeSet

Method	Description
clone()	Creates a copy of the TreeSet
contains()	Searches the TreeSet for the specified element and returns a boolean result
isEmpty()	Checks if the TreeSet is empty
size()	Returns the size of the TreeSet
clear()	Removes all the elements from the TreeSet

TreeSet Vs. HashSet

Both the TreeSet as well as the HashSet implements the Set interface. However, there exist some differences between them.

- Unlike HashSet, elements in TreeSet are stored in some order. It is because TreeSet implements the SortedSet interface as well.
- TreeSet provides some methods for easy navigation. For example, first(), last(), headSet(), tailSet(), etc. It is because TreeSet also implements the NavigableSet interface.
- HashSet is faster than the TreeSet for basic operations like add, remove, contains and size.

TreeSet Comparator

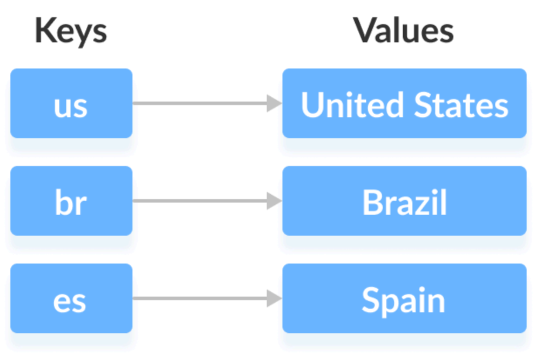
In all the examples above, tree set elements are sorted naturally. However, we can also customize the ordering of elements.
For this, we need to create our own comparator class based on which elements in a tree set are sorted. For example,

```
import java.util.TreeSet;                                 // Ummed Singh

import java.util.Comparator;

public class TreeSetComparatorDemo {

   public static void main(String[] args) {

       // Creating a tree set with a customized comparator

       TreeSet<String> animals = new TreeSet<>(new
CustomComparator());



       animals.add("Dog");

       animals.add("Zebra");

       animals.add("Cat");

       animals.add("Horse");

       System.out.println("TreeSet: " + animals);

   }
   // Creating a comparator class

   public static class CustomComparator implements
Comparator<String> {

       @Override

       public int compare(String animal1, String animal2) {

           int value =  animal1.compareTo(animal2);

           // elements are sorted in reverse order

           if (value > 0) {

               return -1;

           }

           else if (value < 0) {

               return 1;

           }

           else {

               return 0;

           }

       }

   }

}
```

In the above example, we have created a tree set passing the CustomComparator class as an argument.
The CustomComparator class implements the Comparator interface.
We then override the compare() method. The method will now sort elements in reverse order.

Map Interface

In Java, the **Map** interface allows elements to be stored in key/value pairs. Keys are unique names that can be used to access a particular element in a map. And, each key has a single value associated with it.



We can access and modify values using the keys associated with them.
In the above diagram, we have values: United States, Brazil, and Spain. And we have corresponding keys: us, br, and es.
Now, we can access those values using their corresponding keys.
Note: The Map interface maintains 3 different sets:

- the set of keys
- the set of values
- the set of key/value associations (mapping).

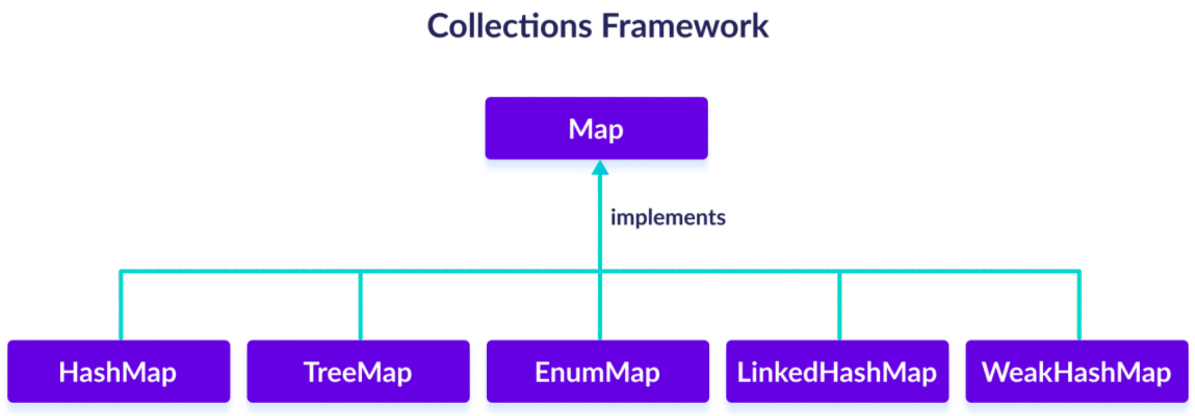
Hence we can access keys, values, and associations individually.

Classes that implement Map

Since Map is an interface, we cannot create objects from it.
In order to use the functionalities of the Map interface, we can use these classes:

- HashMap
- EnumMap
- LinkedHashMap
- WeakHashMap
- TreeMap

These classes are defined in the collections framework and implemented in the Map interface.

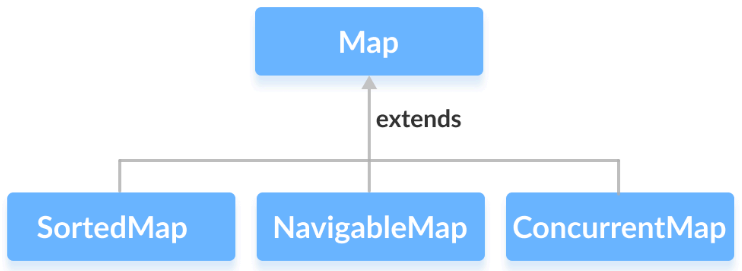


Java Map Subclasses

Interfaces that extend Map

The Map interface is also extended by these subinterfaces:

- SortedMap
- NavigableMap
- ConcurrentMap



How to use Map?

In Java, we must import the ***java.util.Map*** package in order to use Map. Once we import the package, here's how we can create a map.

```
// Map implementation using HashMap
Map<Key, Value> numbers = new HashMap<>();
```

In the above code, we have created a Map named numbers. We have used the HashMap class to implement the Map interface.

Here,

- **Key -** a unique identifier used to associate each element (value) in a map
- **Value -** elements associated by keys in a map

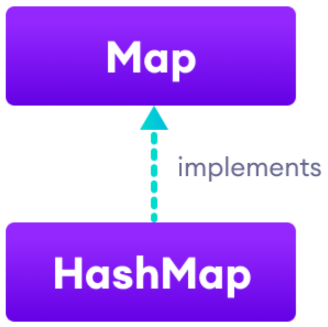
Methods of Map

The Map interface includes the following methods:

- **put(K, V) -** Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.
- **putAll() -** Inserts all the entries from the specified map to this map.
- **putIfAbsent(K, V) -** Inserts the association if the key K is not already associated with the value V.
- **get(K) -** Returns the value associated with the specified key K. If the key is not found, it returns null.
- **getOrDefault(K, defaultValue) -** Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.
- **containsKey(K) -** Checks if the specified key K is present in the map or not.
- containsValue(V) - Checks if the specified value V is present in the map or not.
- **replace(K, V) -** Replace the value of the key K with the new specified value V.
- **replace(K, oldValue, newValue) -** Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.
- **remove(K) -** Removes the entry from the map represented by the key K.
- **remove(K, V) -** Removes the entry from the map that has key K associated with value V.
- **keySet() -** Returns a set of all the keys present in a map.
- **values() -** Returns a set of all the values present in a map.
- **entrySet() -** Returns a set of all the key/value mapping present in a map.

Implementing HashMap Class

The HashMap class of the Java collections framework provides the functionality of the hash table data structure.
It stores elements in key/value pairs. Here, keys are unique identifiers used to associate each value on a map.
The HashMap class implements the Map interface.



Create a HashMap

In order to create a hash map, we must import the java.util.HashMap package first. Once we import the package, here is how we can create hashmaps in Java.

```
// hashMap creation with 8 capacity and 0.6 load factor
HashMap<K, V> numbers = new HashMap<>();
```

In the above code, we have created a hashmap named numbers. Here, K represents the key type and V represents the type of values. For example,

```
HashMap<String, Integer> numbers = new HashMap<>();
```

Here, the type of keys is String and the type of values is Integer.

Example 1: Create HashMap in Java

```
import java.util.HashMap;                                  // Ummed Singh
public class HashMapDemo {
   public static void main(String[] args) {
       // create a hashmap
       HashMap<String, Integer> languages = new HashMap<>();
       // add elements to hashmap
       languages.put("Java", 8);
       languages.put("JavaScript", 1);
       languages.put("Python", 3);
       System.out.println("HashMap: " + languages);
```

```
    }
}
```

In the above example, we have created a HashMap named languages.
Here, we have used the put() method to add elements to the hashmap.

Basic Operations on Java HashMap

The HashMap class provides various methods to perform different operations on hashmaps. We will look at some commonly used arraylist operations in this tutorial:

- Add elements
- Access elements
- Change elements
- Remove elements

```
public static void main(String[] args) {          // Ummed Singh

   // Creating a map using the HashMap

   Map<String, Integer> numbers = new HashMap<>();

   // Insert elements to the map

   numbers.put("One", 1);

   numbers.put("Two", 2);

   System.out.println("Map: " + numbers);

   // Access keys of the map

   System.out.println("Keys: " + numbers.keySet());


   // Access values of the map

   System.out.println("Values: " + numbers.values());

   // Access entries of the map

   System.out.println("Entries: " + numbers.entrySet());

   // Remove Elements from the map
```

```
    int value = numbers.remove("Two");

    System.out.println("Removed Value: " + value);

}
```

Other Methods of HashMap

Method	**Description**
clear()	removes all mappings from the HashMap
compute()	computes a new value for the specified key
computeIfAbsent()	computes value if a mapping for the key is not present
computeIfPresent()	computes a value for mapping if the key is present
merge()	merges the specified mapping to the HashMap
clone()	makes the copy of the HashMap
containsKey()	checks if the specified key is present in Hashmap
containsValue()	checks if Hashmap contains the specified value
size()	returns the number of items in HashMap
isEmpty()	checks if the Hashmap is empty

Iterate through a HashMap

To iterate through each entry of the hashmap, we can use Java for-each loop. We can iterate through keys only, values only, and key/value mapping. For example,

```
public class IterateHashMapDemo {                          // Ummed Singh

   public static void main(String[] args) {

       // create a HashMap

       HashMap<Integer, String> languages = new HashMap<>();

       languages.put(1, "Java");

       languages.put(2, "Python");

       languages.put(3, "JavaScript");

       System.out.println("HashMap: " + languages);


       // iterate through keys only

       System.out.print("Keys: ");

       for (Integer key : languages.keySet()) {

           System.out.print(key);

           System.out.print(", ");

       }


       // iterate through values only

       System.out.print("\nValues: ");

       for (String value : languages.values()) {

           System.out.print(value);

           System.out.print(", ");

       }
```

```
        // iterate through key/value entries

        System.out.print("\nEntries: ");

        for (Entry<Integer, String> entry : languages.entrySet())
{

            System.out.print(entry);

            System.out.print(", ");

        }

    }

}
```

Note that we have used the Map.Entry in the above example. It is the nested class of the Map interface that returns a view (elements) of the map.

We first need to import the ***java.util.Map.Entry*** package in order to use this class.

This nested class returns a view (elements) of the map.

Creating HashMap from Other Maps

In Java, we can also create a hashmap from other maps. For example,

```
import java.util.HashMap;                                  // Ummed Singh

import java.util.TreeMap;

public class OtherHashMapDemo {

    public static void main(String[] args) {



        // create a treemap
        TreeMap<String, Integer> evenNumbers = new TreeMap<>();

        evenNumbers.put("Two", 2);

        evenNumbers.put("Four", 4);

        System.out.println("TreeMap: " + evenNumbers);


        // create hashmap from the treemap

        HashMap<String, Integer> numbers = new
HashMap<>(evenNumbers);

        numbers.put("Three", 3);

        System.out.println("HashMap: " + numbers);

    }

}
```

In the above example, we have created a TreeMap named evenNumbers. Notice the expression,

```
numbers = new HashMap<>(evenNumbers)
```

Note: While creating a hashmap, we can include optional parameters: capacity and load factor. For example,

```
HashMap<K, V> numbers = new HashMap<>(8, 0.6f);
```

- 8(capacity is 8) - This means it can store 8 entries.
- 0.6f (load factor is 0.6) - This means whenever our hash table is filled by 60%, the entries are moved to a new hash table double the size of the original hash table.

If the optional parameters are not used, then the default capacity will be 16 and the default load factor will be 0.75.

Java Database Programming

Introduction to JDBC (Java Database Connectivity)

Java Database Connectivity (JDBC) is an **Application Programming Interface (API)** in Java that allows applications to communicate with relational databases. JDBC acts as a bridge between Java applications and database management systems (DBMS), enabling developers to perform operations such as **inserting, retrieving, updating, and deleting (CRUD operations)** data from databases using SQL queries.

JDBC provides a **standardized way** to interact with databases, ensuring that Java applications remain **portable** across different database systems without significant modifications.

Features of JDBC

1. Platform Independence

JDBC follows the principle of **“Write Once, Run Anywhere” (WORA)**, meaning that Java applications using JDBC can work on any platform that supports Java, such as **Windows, Linux, and macOS**. This is possible because JDBC is written in Java, which runs on the **Java Virtual Machine (JVM)**, making it platform-independent.

For example, if you develop a Java application using MySQL on Windows, the same code can be executed on a Linux or Mac system without modification.

---

2. Standard API

JDBC provides a **unified API** that works with different relational databases such as **MySQL, PostgreSQL, Oracle, SQL Server, and SQLite**.

This means that developers do not need to learn database-specific connectivity methods. Instead, they can use the **same JDBC API** to interact with multiple databases by simply changing the **database driver**.

For example, you can switch from MySQL to PostgreSQL by only modifying the **database URL** in your JDBC connection string, without rewriting your SQL queries or application logic.

3. Supports Multiple Database Drivers

JDBC supports different types of **database drivers**, allowing applications to connect to various databases seamlessly.

There are **four types of JDBC drivers**:

- **Type 1 (JDBC-ODBC Bridge Driver):** Uses ODBC drivers and is now deprecated.
- **Type 2 (Native-API Driver):** Uses database vendor-specific libraries.
- **Type 3 (Network Protocol Driver):** Uses middleware servers for database access.
- **Type 4 (Thin Driver):** A pure Java driver that directly communicates with the database over a network.

Most modern applications use the **Type 4 driver** because it is efficient, lightweight, and does not require additional software.

4. Enables Transaction Management

A **transaction** is a sequence of database operations that must be executed as a **single unit of work**. JDBC supports transaction management, which ensures **data integrity and consistency**.

JDBC transactions follow the **ACID properties**:

- **Atomicity** – All operations within a transaction succeed or none do.
- **Consistency** – The database remains in a valid state before and after a transaction.
- **Isolation** – Transactions execute independently without affecting each other.
- **Durability** – Once a transaction is committed, its changes are permanent.

Example: JDBC Transaction Management

```
import java.sql.Connection;                         //Ummed Singh
import java.sql.DriverManager;
```

```java
import java.sql.SQLException;
import java.sql.Statement;
public class TransactionExample {
   public static void main(String[] args) {
       try {
           Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/micro_h
otel", "root", "root");
           con.setAutoCommit(false); // Disable auto-commit


           Statement stmt = con.createStatement();
           stmt.executeUpdate("INSERT INTO hotel (hotel_id,
name, location) VALUES (1000, ‘City Palace’, ‘Udaipur’)");
           stmt.executeUpdate("INSERT INTO hotel (hotel_id,
name, location) VALUES (1001, ‘City Palace’, ‘Jaipur’)");


           con.commit(); // Commit transaction
           System.out.println("Transaction committed
successfully.");
       } catch (SQLException e) {
           e.printStackTrace();
       }
   }
}
```

Here, if **both INSERT queries succeed**, the transaction is **committed**; otherwise, **it is rolled back**.

---

5. Provides Connection Pooling

Connection pooling is a technique that **reuses database connections** instead of creating a new connection for every database request. This improves **performance** and **reduces system resource usage**.

Instead of establishing and closing database connections repeatedly, a **pool of connections** is maintained, and applications **reuse** these connections.

JDBC supports connection pooling through **DataSource API**, which allows third-party libraries like **Apache DBCP, HikariCP, and C3P0** to manage database connections efficiently.

```
public class ConnectionDemo {                    // Ummed Singh

   public static void main(String[] args) {

       String url = "jdbc:mysql://localhost:3306/mydb";

       String user = "root";

       String password = "password";


       try {

           Connection con = DriverManager.getConnection(url,
  user, password);

           System.out.println("Connected to the database!");

           con.close();

       } catch (SQLException e) {

           e.printStackTrace();

       }

   }

}
```

Introduction to JDBC Drivers

JDBC Drivers are **software components** that allow Java applications to connect to databases. They act as a bridge between a **Java application and a database management system (DBMS)** by converting Java calls into database-specific operations.

Each database (MySQL, PostgreSQL, Oracle, SQL Server, etc.) requires a compatible JDBC driver to communicate with it. JDBC drivers differ in how they handle this communication, which leads to different performance levels and use cases.

There are **four types of JDBC drivers**, each with its own advantages and limitations.

---

1. JDBC-ODBC Bridge Driver (Type 1)

The **JDBC-ODBC Bridge Driver** uses the **Open Database Connectivity (ODBC)** driver to interact with databases. ODBC is a standard API that allows applications to access database management systems (DBMSs) regardless of the database vendor.

How It Works

- The Java application sends a database request to the JDBC-ODBC Bridge Driver.
- The bridge translates the request into an ODBC call.
- The ODBC driver communicates with the database and retrieves the requested data.
- The data is then sent back to the Java application.

Characteristics

✅ Works with any database that has an **ODBC driver**.
✅ Can be used when there is no direct JDBC driver available for a database.

❌ **Requires ODBC Setup**, which can be complex.
❌ **Slower performance** due to multiple translation layers.
❌ **Not platform-independent** since it relies on native ODBC drivers.
❌ **Deprecated in Java 8** and removed in Java 9 due to security risks and inefficiency.

Example Usage (Before Deprecation)

```
import java.sql.Connection;                       // Ummed Singh

import java.sql.DriverManager;

import java.sql.SQLException;

public class Type1JDBCExample {
   public static void main(String[] args) {

       try {

           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //
   Load JDBC-ODBC Driver

           Connection con =
   DriverManager.getConnection("jdbc:odbc:myDSN", "user",
   "password");

           System.out.println("Connected using Type 1
   Driver");

           con.close();

       } catch (Exception e) {

           e.printStackTrace();

       }

   }

}
```

⚠ **This code will not work in Java 8+** because the JDBC-ODBC Bridge has been removed.

---

2. Native API Driver (Type 2)

The **Native API Driver** interacts with the database using **database vendor-specific native libraries**. It does not require an ODBC driver but depends on the **database's native client software** to function.

How It Works

- The Java application calls the Type 2 driver.
- The driver translates JDBC calls into **native API calls** provided by the database vendor.

- The database processes the request and returns the result via the **native API**.

Characteristics

✅ **Better performance than Type 1** since it avoids ODBC overhead.
✅ Works well when the native library is optimized for a particular database.

❌ **Requires native database libraries**, making it **platform-dependent**.
❌ Must be installed **on every client machine**, increasing complexity.
❌ Not suitable for web applications where clients may use different platforms.

Example Usage (Oracle Type 2 Driver)

```java
import java.sql.Connection;                         // Ummed Singh

import java.sql.DriverManager;

import java.sql.SQLException;

public class Type2JDBCExample {

   public static void main(String[] args) {

       try {

           Class.forName("oracle.jdbc.driver.OracleDriver"); //
Oracle Native Driver

           Connection con =
DriverManager.getConnection("jdbc:oracle:oci8:@localhost:1521:xe
", "user", "password");

           System.out.println("Connected using Type 2 Driver");

           con.close();

       } catch (Exception e) {

           e.printStackTrace();

       }

   }
}
```

📌 **Requires Oracle's native OCI (Oracle Call Interface) libraries installed on the system**.

---

3. Network Protocol Driver (Type 3)

The **Network Protocol Driver** (also called **Middleware Driver**) communicates with the database through a **middleware server**, which forwards the requests to the appropriate database.

How It Works

- The Java application sends JDBC calls to the Type 3 driver.
- The driver forwards these calls to a **middleware server** over the network.
- The middleware translates them into **database-specific calls** and communicates with the actual database.
- The database processes the request and sends the result back through the middleware.

Characteristics

✅ **More scalable** than Type 1 and Type 2.
✅ **Database-independent** – The middleware handles different database protocols.
✅ Suitable for **enterprise applications** with centralized database management.

❌ **Requires an additional middleware server**, adding overhead.
❌ Performance depends on **network latency and middleware efficiency**.

Example Usage

Middleware solutions like **IBM WebSphere** and **Sybase** use Type 3 drivers.

```
import java.sql.Connection;                    // Ummed Singh

import java.sql.DriverManager;

import java.sql.SQLException;
```

```
public class Type3JDBCExample {

   public static void main(String[] args) {

       try {

           Connection con =
  DriverManager.getConnection("jdbc:net://middleware-server:1
  521/mydb", "user", "password");

           System.out.println("Connected using Type 3
  Driver");

           con.close();

       } catch (Exception e) {

           e.printStackTrace();

       }

   }

}
```

📌 Rarely used today because **Type 4 drivers** provide better performance without middleware.

---

4. Thin Driver (Type 4)

The **Thin Driver**, also called the **Pure Java Driver**, is the most commonly used JDBC driver today. It directly communicates with the database using its **native protocol** over the network.

How It Works

- The Java application sends JDBC calls to the Type 4 driver.
- The driver directly interacts with the database over TCP/IP using the database's **native communication protocol**.

- The database processes the request and returns the result directly to the application.

Characteristics

✅ **Fastest and most efficient** since it eliminates extra translation layers.
✅ **Pure Java implementation**, making it **platform-independent**.
✅ **No additional software required**, making deployment easier.
✅ The **preferred choice** for most modern Java applications.

❌ Requires **different drivers for different databases** (e.g., MySQL, PostgreSQL, Oracle, etc.).

Example Usage (MySQL Type 4 Driver)

```java
import java.sql.Connection;                        // Ummed Singh

import java.sql.DriverManager;

public class Type4JDBCExample {

   public static void main(String[] args) {

       try {

           Class.forName("com.mysql.cj.jdbc.Driver"); // MySQL Thin Driver

           Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "root", "password");

           System.out.println("Connected using Type 4 Driver");

           con.close();

       } catch (Exception e) {

           e.printStackTrace();

       }

   }
}
```

📌 Most modern databases provide **Type 4 JDBC drivers**, such as **PostgreSQL, MySQL, Oracle, and SQL Server**.

CRUD Operations Using JDBC

CRUD stands for **Create, Read, Update, and Delete**. These operations are performed using JDBC with the following steps:

1. Establishing a Database Connection

```
import java.sql.Connection;                              // Ummed Singh
import java.sql.DriverManager;

public class DatabaseConnection {
   public static Connection getConnection() {
       Connection con = null;
       try {
           Class.forName("com.mysql.cj.jdbc.Driver");
           con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
"root", "password");
           System.out.println("Connection Established
Successfully");
       } catch (Exception e) {
           e.printStackTrace();
       }
       return con;
   }
}
```

2. Creating a Table in Database

```
import java.sql.Connection;                              // Ummed Singh
import java.sql.Statement;
public class CreateTable {                               // Ummed Singh
   public static void main(String[] args) {
       try {
           Connection con = DatabaseConnection.getConnection();
           Statement stmt = con.createStatement();
           String sql = "CREATE TABLE students (id INT PRIMARY
KEY, name VARCHAR(50), age INT)";
           stmt.executeUpdate(sql);
           System.out.println("Table Created Successfully");
           con.close();
       } catch (Exception e) {
           e.printStackTrace();
       }
   }
}
```

3. Insert Data into Database (Create Operation)

```
import java.sql.Connection;                             // Ummed Singh
import java.sql.PreparedStatement;
public class InsertData {
   public static void main(String[] args) {
       try {
           Connection con = DatabaseConnection.getConnection();
           String query = "INSERT INTO students (id, name, age)
VALUES (?, ?, ?)";
           PreparedStatement pstmt =
con.prepareStatement(query);
           pstmt.setInt(1, 1);
           pstmt.setString(2, "Ummed Singh");
           pstmt.setInt(3, 24);
           pstmt.executeUpdate();
           System.out.println("Data Inserted Successfully");
           con.close();
       } catch (Exception e) {
           e.printStackTrace();
       }
   }
}
```

4. Fetch Data from Database (Read Operation)

```
import java.sql.Connection;                             // Ummed Singh
import java.sql.ResultSet;
import java.sql.Statement;
public class ReadData {
   public static void main(String[] args) {
       try {
           Connection con = DatabaseConnection.getConnection();
           Statement stmt = con.createStatement();
           ResultSet rs = stmt.executeQuery("SELECT * FROM
students");
           while (rs.next()) {
               System.out.println("ID: " + rs.getInt("id") + ",
Name: " + rs.getString("name") + ", Age: " + rs.getInt("age"));
           }
           con.close();
       } catch (Exception e) {
           e.printStackTrace();
       }
   }
}
```

5. Update Data in Database (Update Operation)

```
import java.sql.Connection;                                // Ummed Singh
import java.sql.PreparedStatement;
public class UpdateData {
   public static void main(String[] args) {
       try {
           Connection con = DatabaseConnection.getConnection();
           String query = "UPDATE students SET age = ? WHERE id
= ?";
           PreparedStatement pstmt =
con.prepareStatement(query);
           pstmt.setInt(1, 22);
           pstmt.setInt(2, 1);
           pstmt.executeUpdate();
           System.out.println("Data Updated Successfully");
           con.close();
       } catch (Exception e) {
           e.printStackTrace();
       }
```

```
    }
}
```

6. Delete Data from Database (Delete Operation)

```
import java.sql.Connection;                          // Ummed Singh
import java.sql.PreparedStatement;

public class DeleteData {
    public static void main(String[] args) {
        try {
            Connection con = DatabaseConnection.getConnection();
            String query = "DELETE FROM students WHERE id = ?";
            PreparedStatement pstmt =
con.prepareStatement(query);
            pstmt.setInt(1, 1);
            pstmt.executeUpdate();
            System.out.println("Data Deleted Successfully");
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Non-Conventional Database

Non-conventional databases, also known as **NoSQL databases**, are different from traditional relational databases (like MySQL, Oracle, PostgreSQL). Instead of storing data in tables with rows and columns,they use **different data storage techniques** such as **document-based, key-value, column-family, or graph-based models**.

🔹 **Examples of Non-Conventional Databases:**
✅ **MongoDB** → Document-based (NoSQL).
✅ **Cassandra** → Column-family-based (NoSQL).
✅ **Firebase** → Cloud-based real-time database.
✅ **Redis** → Key-value store (often used for caching).

These databases are **schema-less**, **scalable**, and **flexible**, and are great for handling **unstructured** or **semi-structured** data.

JDBC (Java Database Connectivity) **cannot be used directly** with these databases. Instead, Java provides **custom drivers and APIs** for connecting to them.

🔹 Key Characteristics

Feature	Description
Schema-less	No fixed table structure – data can be flexible.
Scalable	Easy to scale horizontally across many servers.
High Performance	Optimized for fast read/write operations.
Supports Big Data	Handles huge amounts of unstructured/semi-structured data.

What Problems Do NoSQL Databases Solve?

Problem	Solution by NoSQL
Rigid schema	NoSQL is schema-less – different documents can have different structures.
Performance bottlenecks	Designed for high-speed read/write operations.
Scalability	Scales easily across multiple servers (horizontal scaling).
Handling Big Data	Excellent for storing and querying massive datasets.
Complex relationships	Graph databases handle relationships efficiently.

Introduction to MongoDB

MongoDB is a **NoSQL database** that stores data in **JSON-like documents** (BSON format). It is used for applications that require **scalability and flexibility**. A typical MongoDB document looks like this:

```
{
    "_id": "12345",
    "name": "Ummed Singh",
    "email": "Ummed@example.com",
    "age": 24
}
```

MongoDB collections are like tables, and documents are like rows, but there is no strict schema.

- **Structure**
  - **Database** → contains **collections**
  - **Collection** → contains **documents**
  - **Document** → stores the actual data (like a row in SQL)

Setting Up MongoDB

A. Installation

- Install MongoDB from https://www.mongodb.com/try/download/community

Start the MongoDB service:

- mongod

B. Tools

- **MongoDB Compass**: GUI for managing MongoDB
- **Mongo shell** or **command line**: For raw commands

Java Code – Connect and Perform CRUD

Step 1: Add MongoDB Java Driver Dependency

If using **Maven**, add this to `pom.xml`:

```
<!-- MongoDBDB Sync Driver -->
```

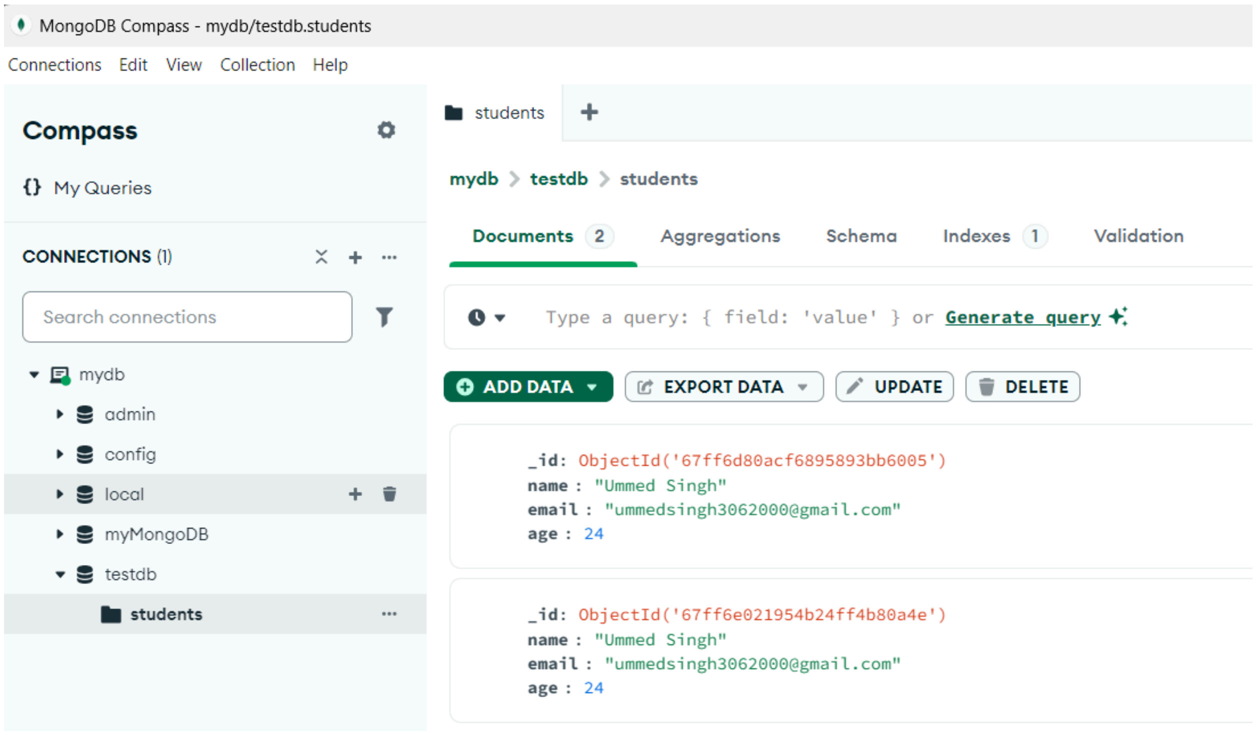
```
   <dependency>

       <groupId>org.mongodb</groupId>

       <artifactId>mongodb-driver-sync</artifactId>

       <version>4.11.1</version>

   </dependency>
```

Step 2: Java Code to Connect and perform CRUD Operations

```
public class MongoCRUDExample {                   // Ummed Singh

   public static void main(String[] args) {

       // 1. Connect to MongoDB

       String connectionString = "mongodb://localhost:27017";

       try (MongoClient mongoClient =
MongoClients.create(connectionString)) {



           // 2. Access the database and collection

           MongoDatabase database =
mongoClient.getDatabase("testdb");

           MongoCollection<Document> collection =
database.getCollection("students");



           // 3. Create - Insert a document

           Document student = new Document("name", "Ummed
Singh")

                   .append("email",
"ummedsingh3062000@gmail.com")
```

```
                   .append("age", 24);

           collection.insertOne(student);

           System.out.println("Inserted: " + student.toJson());


           // 4. Read - Find all documents

           System.out.println("\nAll students:");

           MongoCursor<Document> cursor =
collection.find().iterator();

           while (cursor.hasNext()) {

               System.out.println(cursor.next().toJson());

           }


           // 5. Update - Update student by name

           collection.updateOne(eq("name", "Ummed Singh"),

                   new Document("$set", new Document("age",
24)));

           System.out.println("\nUpdated Ummed's age to 25.");


           // 6. Read again to see update

           Document updatedStudent = collection.find(eq("name",
"Ummed Singh")).first();

           System.out.println("Updated document: " +
updatedStudent.toJson());


//             6. Delete
```

```
           collection.deleteOne(Filters.eq("name", "Ummed
Singh"));


       } catch (Exception e) {

           e.printStackTrace();

       }

   }

}
```



```
_id: ObjectId('67ff6d80acf6895893bb6005')
name : "Ummed Singh"
email : "ummedsingh3062000@gmail.com"
age : 24

_id: ObjectId('67ff6e021954b24ff4b80a4e')
name : "Ummed Singh"
email : "ummedsingh3062000@gmail.com"
age : 24
```

When to Use NoSQL?

Use NoSQL when:

- Data structure is not fixed (schema-less).

- You expect rapid growth in data volume (Big Data).
- You want fast read/write speeds.
- You need to scale easily.
- You're building modern applications like:
  - Social media platforms
  - Real-time analytics systems
  - Recommendation engines
  - Content management systems
  - E-commerce platforms

Real-World Example: E-Commerce Product Catalog (like Amazon)

🛍️ Scenario:

Let's say you are building an **online shopping website** (like Amazon or Flipkart). You need to store details about **different types of products**:

🧥 Product 1: A Jacket

```
{                                                           // Ummed Singh

       "productId": "JKT101",

       "name": "Winter Jacket",

       "brand": "Puma",

       "size": ["S", "M", "L", "XL"],

       "color": ["Black", "Blue"],

       "material": "Polyester",
       "category": "Clothing"
}
```

📱 Product 2: A Mobile Phone

```
{                                                          // Ummed Singh
       "productId": "MOB202",
       "name": "Samsung Galaxy S24",
       "brand": "Samsung",
       "ram": "8GB",
       "storage": "256GB",
       "battery": "5000mAh",
       "camera": "108MP",
       "category": "Electronics"
}
```

❓What’s the problem with MySQL?

In **MySQL**, you need to define a **fixed table structure**, like this:

```
CREATE TABLE products (                                    // Ummed Singh
     productId VARCHAR(10),
     name VARCHAR(100),
     brand VARCHAR(50),
     category VARCHAR(30),

     size VARCHAR(20),

     color VARCHAR(30),

     ram VARCHAR(10),

     storage VARCHAR(20),

     battery VARCHAR(20),

     camera VARCHAR(20)

);
```

❌ Issues:

- Wasted columns: Clothes don't need RAM or battery.
- Missing columns: Electronics don't have size or material.
- **Empty/null values everywhere**.
- **Hard to maintain and update** when new product types are added.
- You'll need **multiple tables and joins** = performance goes down.

✅ How MongoDB solves this:

In MongoDB, each product is stored as a **document**, and it can have its **own structure**. No need to worry about missing or extra fields.

📦 Jacket:

```
{                                                    // Ummed Singh

      "productId": "JKT101",
      "name": "Winter Jacket",

      "brand": "Puma",

      "size": ["S", "M", "L"],

      "color": ["Black", "Blue"]

}
```

📱 Phone:

```
{                                                          // Ummed Singh

        "productId": "MOB202",

        "name": "Samsung Galaxy S24",

        "ram": "8GB",

        "storage": "256GB",

        "battery": "5000mAh"

}
```

✅ Advantages of MongoDB here:

Feature	Benefit
Flexible structure	Each product can have different fields.
No wasted space	Only relevant data is stored.
Easy to add new types	Just insert a new document. No schema change needed.
Fast performance	No complex joins or constraints.