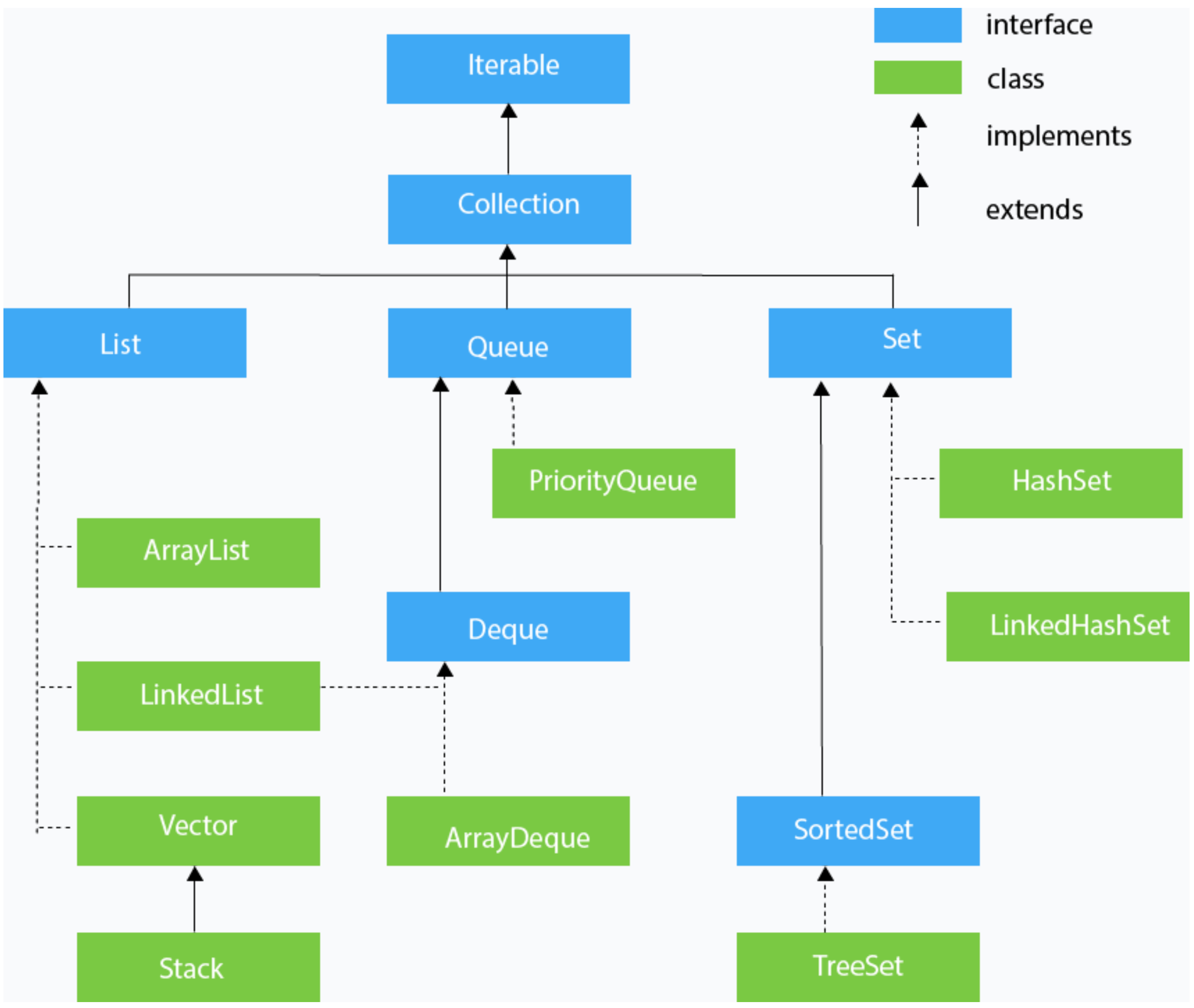
# Java Collections Framework

The Java **collections** framework provides a set of interfaces and classes to implement various data structures and algorithms.

For example, the LinkedList class of the collections framework provides the implementation of the doubly-linked list data structure.

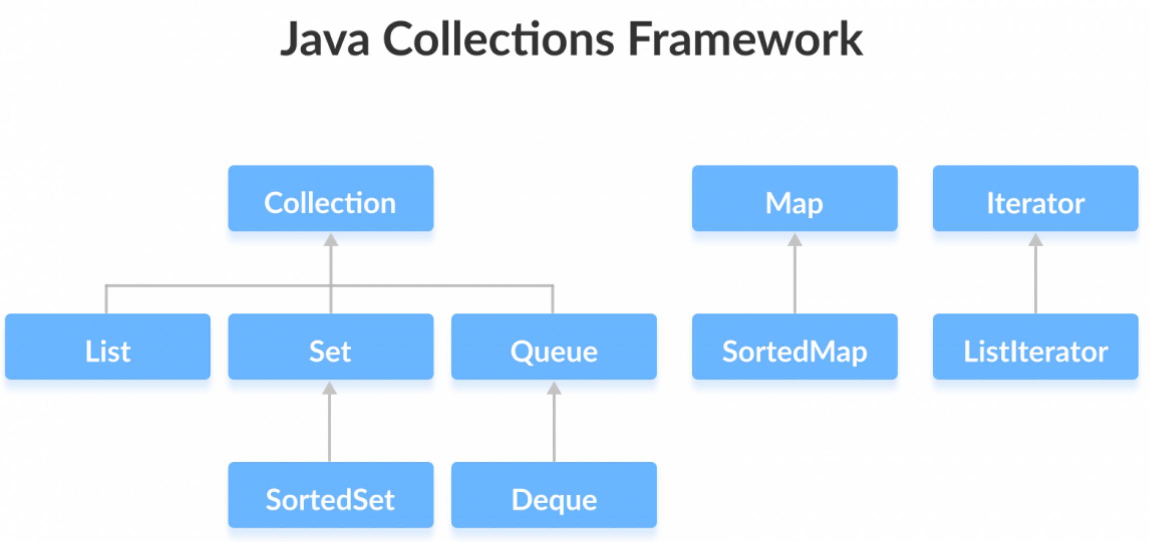
## Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the **classes** and **interfaces** for the Collection framework.



## Interfaces of Collections FrameWork

The Java collections framework provides various interfaces. These interfaces include several methods to perform different operations on collections.



### Java Collection Interface

The Collection interface is the root interface of the collections framework hierarchy. Java does not provide direct implementations of the Collection interface but provides implementations of its subinterfaces like List, Set, and Queue.

### Collections Framework Vs. Collection Interface

People often get confused between the collections framework and Collection Interface.

The Collection interface is the root interface of the collections framework. The framework includes other interfaces as well: Map and Iterator. These interfaces may also have subinterfaces.

### Subinterfaces of the Collection Interface

As mentioned earlier, the Collection interface includes subinterfaces that are implemented by Java classes.

All the methods of the Collection interface are also present in its subinterfaces.

Here are the subinterfaces of the Collection Interface:

### List Interface

The List interface is an ordered collection that allows us to add and remove elements like an array.

### Set Interface

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

### Queue Interface

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

### Methods of Collection

The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its subinterfaces.

- add() - inserts the specified element to the collection
- size() - returns the size of the collection
- remove() - removes the specified element from the collection
- iterator() - returns an iterator to access elements of the collection
- addAll() - adds all the elements of a specified collection to the collection
- removeAll() - removes all the elements of the specified collection from the collection
- clear() - removes all the elements of the collection

### Java 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.

### Java Iterator Interface

In Java, the Iterator interface provides methods that can be used to access elements of collections.

### 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.

### Example: ArrayList Class of Collections

The ArrayList class allows us to create resizable arrays. The class implements the List interface (which is a subinterface of the Collection interface).

```
// The Collections framework is defined in the java.util package
import java.util.ArrayList;
```

```
class Main {
    public static void main(String[] args){
        ArrayList<String> animals = new ArrayList<>();
        // Add elements
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");

        System.out.println("ArrayList: " + animals);
    }
}
```

**Output**:

```
ArrayList: [Dog, Cat, Horse]
```

### Access ArrayList Elements

To access an element from the arraylist, we use the get() method of the ArrayList class. For example,

```
import java.util.ArrayList;

class Main {
  public static void main(String[] args) {
    ArrayList<String> animals = new ArrayList<>();

    // add elements in the arraylist
    animals.add("Cat");
    animals.add("Dog");
    animals.add("Cow");
    System.out.println("ArrayList: " + animals);

    // get the element from the arraylist
    String str = animals.get(1);

    System.out.print("Element at index 1: " + str);
  }
}
```

**Output**

```
ArrayList: [Cat, Dog, Cow]
Element at index 1: Dog
```

In the above example, we have used the get() method with parameter 1. Here, the method returns the element at **index 1**.

### Java Vector

The Vector class is an implementation of the List interface that allows us to create resizable-arrays similar to the ArrayList class.

### Java Vector vs. ArrayList

In Java, both ArrayList and Vector implements the List interface and provides the same functionalities. However, there exist some differences between them.
The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.
It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called ConcurrentModificationException is generated. Hence, this continuous use of lock for each operation makes vectors less efficient.
However, in array lists, methods are not synchronized. Instead, it uses the Collections.synchronizedList() method that synchronizes the list as a whole.

**Note:** It is recommended to use ArrayList in place of Vector because vectors less efficient.

### Add Elements to Vector

- add(element) - adds an element to vectors
- add(index, element) - adds an element to the specified position

- addAll(vector) - adds all elements of a vector to another vector

For example,

```
import java.util.Vector;

class Main {
    public static void main(String[] args) {
        Vector<String> mammals= new Vector<>();

        // Using the add() method
        mammals.add("Dog");
        mammals.add("Horse");

        // Using index number
        mammals.add(2, "Cat");
        System.out.println("Vector: " + mammals);

        // Using addAll()
        Vector<String> animals = new Vector<>();
        animals.add("Crocodile");

        animals.addAll(mammals);
        System.out.println("New Vector: " + animals);
    }
}
```

**Output**

```
Vector: [Dog, Horse, Cat]
New Vector: [Crocodile, Dog, Horse, Cat]
```

### Remove Vector Elements

- remove(index) - removes an element from specified position
- removeAll() - removes all the elements
- clear() - removes all elements. It is more efficient than removeAll()

For example,

```
import java.util.Vector;

class Main {
    public static void main(String[] args) {
        Vector<String> animals= new Vector<>();
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");

        System.out.println("Initial Vector: " + animals);

        // Using remove()
        String element = animals.remove(1);
        System.out.println("Removed Element: " + element);
        System.out.println("New Vector: " + animals);

        // Using clear()
        animals.clear();
        System.out.println("Vector after clear(): " + animals);
    }
}
```

**Output**

```
Initial Vector: [Dog, Horse, Cat]
Removed Element: Horse
New Vector: [Dog, Cat]
Vector after clear(): []
```

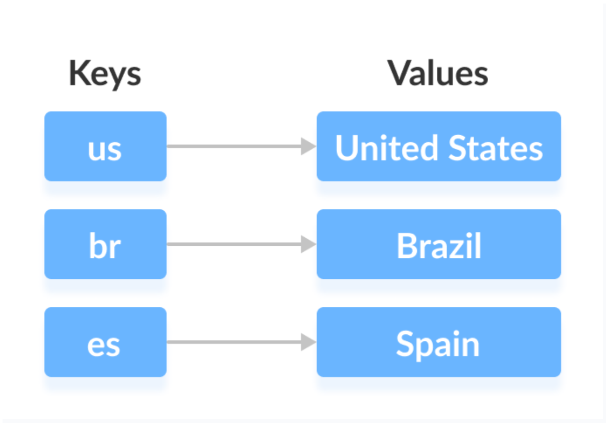
## Java Map Interface

The Map interface of the Java collections framework provides the functionality of the map data structure.

### Working of Map

In Java, elements of Map are stored in **key/value** pairs. **Keys** are unique values associated with individual **Values**.

A map cannot contain duplicate keys. And, each key is associated with a single value.



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).

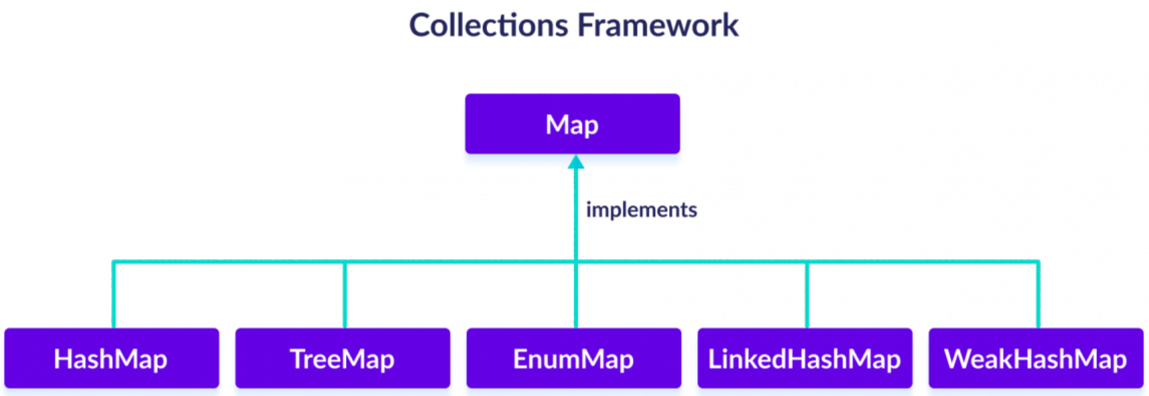
Hence we can access keys, values, and associations individually.

## Classes that implement Map

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

- HashMap
- EnumMap
- LinkedHashMap
- WeakHashMap
- TreeMap

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



**Java Map Subclasses**

### 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

### Implementation of the Map Interface

**1. Implementing HashMap Class**

```
import java.util.Map;
import java.util.HashMap;

class Main {

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

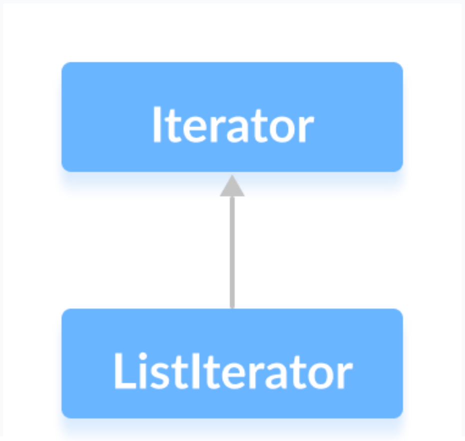
```
        System.out.println("Map: " + numbers);

        // 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);
    }
}
```

**Output**

```
Map: {One=1, Two=2}
Keys: [One, Two]
Values: [1, 2]
Entries: [One=1, Two=2]
Removed Value: 2
```

**Java Iterator Interface**

The Iterator interface of the Java collections framework allows us to access elements of a collection. It has a subinterface ListIterator.



All the Java collections include an iterator() method. This method returns an instance of iterator used to iterate over elements of collections.

**Methods of Iterator**

The Iterator interface provides 4 methods that can be used to perform various operations on elements of collections.

- hasNext() - returns true if there exists an element in the collection
- next() - returns the next element of the collection
- remove() - removes the last element returned by the next()
- forEachRemaining() - performs the specified action for each remaining element of the collection

**Example: Implementation of Iterator**

In the example below, we have implemented the hasNext(), next(), remove() and forEachRemining() methods of the Iterator interface in an array list.

```java
import java.util.ArrayList;
import java.util.Iterator;

class Main {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(3);
        numbers.add(2);
        System.out.println("ArrayList: " + numbers);

        // Creating an instance of Iterator
        Iterator<Integer> iterate = numbers.iterator();

        // Using the next() method
        int number = iterate.next();
        System.out.println("Accessed Element: " + number);

        // Using the remove() method
        iterate.remove();
        System.out.println("Removed Element: " + number);
        System.out.print("Updated ArrayList: ");

        // Using the hasNext() method
        while(iterate.hasNext()) {
            // Using the forEachRemaining() method
            iterate.forEachRemaining((value) -> System.out.print(value + ", "));
        }
    }
}
```

**Output**

```
ArrayList: [1, 3, 2]
Acessed Element: 1
Removed Element: 1
Updated ArrayList: 3, 2,
```

In the above example, notice the statement:

```
iterate.forEachRemaining((value) -> System.put.print(value + ", "));
```

Here, we have passed the lambda expression as an argument of the forEachRemaining() method.

Now the method will print all the remaining elements of the array list.

**Access Vector Elements**

- get(index) - returns an element specified by the index
- iterator() - returns an iterator object to sequentially access vector elements

For example,

```
import java.util.Iterator;
import java.util.Vector;

class Main {
    public static void main(String[] args) {
        Vector<String> animals= new Vector<>();
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");
        // Using get()
        String element = animals.get(2);
        System.out.println("Element at index 2: " + element);

        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        System.out.print("Vector: ");
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

**Output**

```
Element at index 2: Cat
Vector: Dog, Horse, Cat,
```