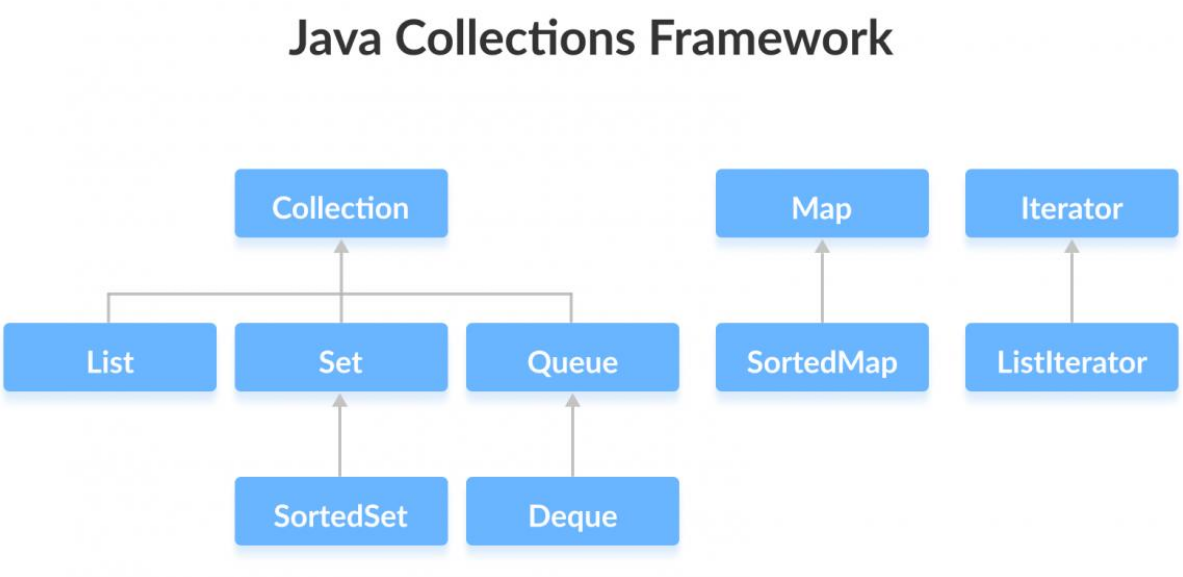
Java Collections Framework

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.

Interfaces of Collections FrameWork

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



We will learn about these interfaces, their subinterfaces, and implementation in various classes in detail in the later chapters.

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.

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

Before we wrap up this tutorial, let's take an example of the ArrayList class of the collections framework.

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:

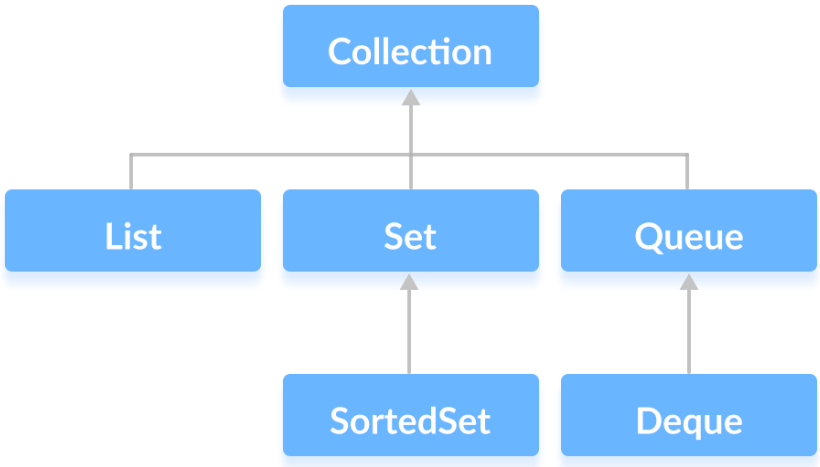
```
ArrayList: [Dog, Cat, Horse]
```

Java Collection Interface

The Collection interface is the root interface of the Java collections framework.

There is no direct implementation of this interface. However, it is implemented through its subinterfaces like List, Set, and Queue.

For example, the ArrayList class implements the List interface which is a subinterface of the Collection Interface.



Subinterfaces of Collection

As mentioned above, the Collection interface includes subinterfaces that are implemented by various classes in Java.

1. List Interface

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

2. Set Interface

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

3. Queue Interface

The `Queue` interface is used when we want to store and access elements in **First In, First Out(FIFO)** 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 List

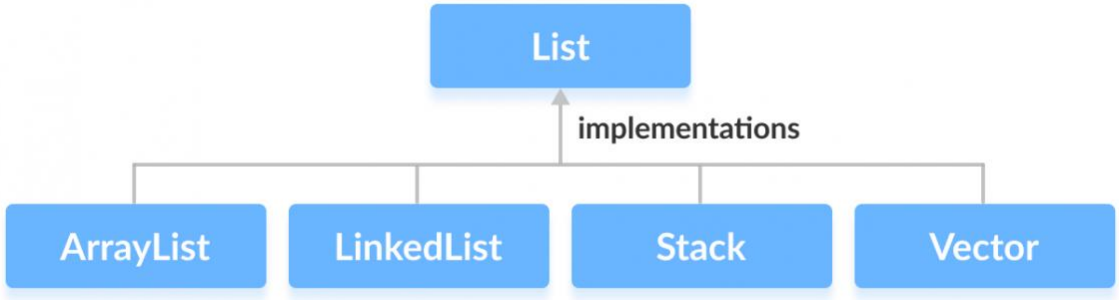
In Java, the List interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface.

Classes that Implement List

Since List is an interface, we cannot create objects from it.

In order to use functionalities of the List interface, we can use these classes:

- ArrayList
- LinkedList
- Vector
- Stack



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

How to use List?

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

```
// ArrayList implementation of List
List<String> list1 = new ArrayList<>();

// LinkedList implementation of List
List<String> list2 = new LinkedList<>();
```

Here, we have created objects *list1* and *list2* of classes ArrayList and LinkedList. These objects can use the functionalities of the List interface.

Methods of List

The List interface includes all the methods of the Collection interface. Its because Collection is a super interface of List.

Some of the commonly used methods of the Collection interface that's also available in the List interface are:

- `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 specified element

Implementation of the List Interface

1. Implementing the ArrayList Class

```
import java.util.List;
import java.util.ArrayList;

class Main {

    public static void main(String[] args) {
        // Creating list using the ArrayList class
        List<Integer> numbers = new ArrayList<>();

        // Add elements to the list
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("List: " + numbers);

        // Access element from the list
        int number = numbers.get(2);
        System.out.println("Accessed Element: " + number);

        // Remove element from the list
        int removedNumber = numbers.remove(1);
        System.out.println("Removed Element: " + removedNumber);
    }
}
```

Output

```
List: [1, 2, 3]
Accessed Element: 3
Removed Element: 2
```

2. Implementing the LinkedList Class

```
import java.util.List;
import java.util.LinkedList;

class Main {

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

```
        // Creating list using the LinkedList class
        List<Integer> numbers = new LinkedList<>();

        // Add elements to the list
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("List: " + numbers);

        // Access element from the list
        int number = numbers.get(2);
        System.out.println("Accessed Element: " + number);

        // Using the indexOf() method
        int index = numbers.indexOf(2);
        System.out.println("Position of 3 is " + index);

        // Remove element from the list
        int removedNumber = numbers.remove(1);
        System.out.println("Removed Element: " + removedNumber);
    }
}
```

Output

```
List: [1, 2, 3]
Accessed Element: 3
Position of 3 is 1
Removed Element: 2
```

Java List vs. Set

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

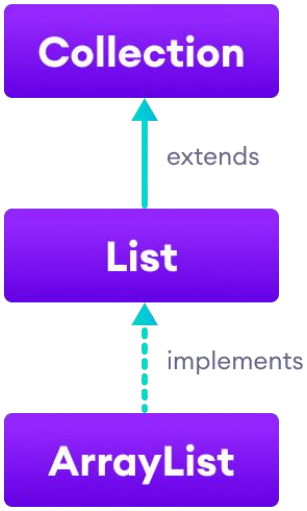
- Lists can include duplicate elements. However, sets cannot have duplicate elements.
- Elements in lists are stored in some order. However, elements in sets are stored in groups like sets in mathematics.

Now that we know what List is, we will see its implementations in ArrayList and LinkedList classes in detail in the next tutorials.

Java ArrayList Class

The ArrayList class of the Java collections framework provides the functionality of resizable-arrays.

It implements the List interface.



Java ArrayList Implementation

Java ArrayList Vs Array

In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it.

To handle this issue, we can use the `ArrayList` class. It allows us to create resizable arrays.

Unlike arrays, arraylists can automatically adjust its capacity when we add or remove elements from it. Hence, arraylists are also known as **dynamic arrays**.

Creating an ArrayList

Before using ArrayList, we need to import the java.util.ArrayList package first. Here is how we can create arraylists in Java:

```
ArrayList<Type> arrayList= new ArrayList<>();
```

Here, *Type* indicates the type of an arraylist. For example,

```
// create Integer type arraylist
ArrayList<Integer> arrayList = new ArrayList<>();

// create String type arraylist
ArrayList<String> arrayList = new ArrayList<>();
```

In the above program, we have used Integer not int. It is because we cannot use primitive types while creating an arraylist. Instead, we have to use the corresponding wrapper classes.

Here, Integer is the corresponding wrapper class of int.

Example: Create ArrayList in Java

```
import java.util.ArrayList;

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

    // create ArrayList
    ArrayList<String> languages = new ArrayList<>();

    // Add elements to ArrayList
    languages.add("Java");
    languages.add("Python");
    languages.add("Swift");
    System.out.println("ArrayList: " + languages);
  }
}
```

Output

```
ArrayList: [Java, Python, Swift]
```

In the above example, we have created an ArrayList named *languages*.

Here, we have used the add() method to add elements to the arraylist. We will learn more about the add() method later in this tutorial.

Note: We can also create an arraylist using the List interface. It's because the ArrayList class implements the List interface.

```
List<String> list = new ArrayList<>();
```

Basic Operations on ArrayList

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

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

1. Add Elements to an ArrayList

To add a single element to the arraylist, we use the `add()` method of the `ArrayList` class. For example,

```
import java.util.ArrayList;

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

    // add() method without the index parameter
    languages.add("Java");
    languages.add("C");
    languages.add("Python");
    System.out.println("ArrayList: " + languages);

    // add() method with the index parameter
    languages.add(1, "JavaScript");
    System.out.println("Updated ArrayList: " + languages);
  }
}
```

Output

```
ArrayList: [Java, C, Python]
Updated ArrayList: [Java, JavaScript, C, Python]
```

In the above example, we have created an ArrayList named *languages*. Here, we have used the add() method to add elements to *languages*.

Notice the statement,

```
languages.add(1, "JavaScript");
```

Here, we have used the **index number** parameter. It is an optional parameter that specifies the position where the new element is added.

2. 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**.

3. Change ArrayList Elements

To change element of the arraylist, we use the `set()` method of the `ArrayList` class. For example,

```
import java.util.ArrayList;

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

    // add elements in the array list
```

```
    languages.add("Java");
    languages.add("Kotlin");
    languages.add("C++");
    System.out.println("ArrayList: " + languages);

    // change the element of the array list
    languages.set(2, "JavaScript");
    System.out.println("Modified ArrayList: " + languages);
  }
}
```

Output

```
ArrayList: [Java, Kotlin, C++]
Modified ArrayList: [Java, Kotlin, JavaScript]
```

In the above example, we have created an ArrayList named *languages*. Notice the line,

```
language.set(2, "JavaScript");
```

Here, the set() method changes the element at **index 2** to *JavaScript*.

4. Remove ArrayList Elements

To remove an element from the arraylist, we can use the `remove()` 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 array list
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Horse");
    System.out.println("ArrayList: " + animals);

    // aemove element from index 2
    String str = animals.remove(2);
    System.out.println("Updated ArrayList: " + animals);
    System.out.println("Removed Element: " + str);
  }
}
```

Output

```
ArrayList: [Dog, Cat, Horse]
Updated ArrayList: [Dog, Cat]
Removed Element: Horse
```

Here, the remove() method takes the **index number** as the parameter. And, removes the element specified by the **index number**.

Methods of ArrayList Class

In previous section, we have learned about the add(), get(), set(), and remove() method of the ArrayList class.

Besides those basic methods, 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.

Iterate through an ArrayList

We can use the Java for-each loop to loop through each element of the arraylist. For example,

```
import java.util.ArrayList;

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

    // creating an array list
    ArrayList<String> animals = new ArrayList<>();
    animals.add("Cow");
    animals.add("Cat");
    animals.add("Dog");
    System.out.println("ArrayList: " + animals);

    // iterate using for-each loop
    System.out.println("Accessing individual elements:  ");

    for (String language : animals) {
      System.out.print(language);
      System.out.print(", ");
    }
  }
}
```

Output

```
ArrayList: [Cow, Cat, Dog]
Accessing individual elements:
Cow, Cat, Dog,
```

ArrayList To Array Conversion

We can convert the `ArrayList` into an array using the `toArray()` method. For example,

```
import java.util.ArrayList;

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

    // add elements in the array list
    languages.add("Java");
    languages.add("Python");
    languages.add("C++");
```

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

    // create a new array of String type
    String[] arr = new String[languages.size()];

    // convert ArrayList into an array
    languages.toArray(arr);
    System.out.print("Array: ");

    // access elements of the array
    for (String item : arr) {
      System.out.print(item + ", ");
    }
  }
}
```

Output

```
ArrayList: [Java, Python, C++]
Array: Java, Python, C++,
```

In the above example, we have created an arraylist named *languages*. Notice the statement,

```
languages.toArray(arr);
```

Here, the toArray() method converts the arraylist into an array and stores it in *arr*.

Java Array to ArrayList Conversion

We can also convert the array into an arraylist. For that, we use the asList() method of the Arrays class.

To use asList(), we must import the java.util.Arrays package first. For example,

```
import java.util.ArrayList;
import java.util.Arrays;

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

    // create an array of String type
    String[] arr = { "Java", "Python", "C++" };
    System.out.print("Array: ");

    // print array
    for (String str : arr) {
      System.out.print(str);
```

```
      System.out.print(" ");
    }

    // create an ArrayList from an array
    ArrayList<String> languages = new ArrayList<>(Arrays.asList(arr));
    System.out.println("\nArrayList: " + languages);
  }
}
```

Output

```
Array: Java Python C++
ArrayList: [Java, Python, C++]
```

In the above program, we first created an array arr of the String type. Notice the expression,

```
Arrays.asList(arr)
```

Here, the `asList()` method converts the array into an arraylist.

Note: We can also use the Arrays.asList() method to create and initialize the arraylist in a single line. For example,

```
 // create and initialize arraylist
ArrayList<String> animals = new ArrayList<>(Arrays.asList("Cat",
"Cow", "Dog"));
```

ArrayList to String Conversion

We can use the `toString()` method of the `ArrayList` class to convert an arraylist into a string. For example,

```
import java.util.ArrayList;

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

    // add elements in the ArrayList
    languages.add("Java");
    languages.add("Python");
    languages.add("Kotlin");
    System.out.println("ArrayList: " + languages);

    // convert ArrayList into a String
    String str = languages.toString();
```

```
    System.out.println("String: " + str);
  }
}
```

Output

```
ArrayList: [Java, Python, Kotlin]
String: [Java, Python, Kotlin]
```

Here, the toString() method converts the whole arraylist into a single string.

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 are not threadsafe and are less efficient.

Creating a Vector

Here is how we can create vectors in Java.

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

Here, *Type* indicates the type of a linked list. For example,

```
// create Integer type linked list
Vector<Integer> vector= new Vector<>();

// create String type linked list
Vector<String> vector= new Vector<>();
```

Methods of Vector

The `Vector` class also provides the resizable-array implementations of the `List` interface (similar to the `ArrayList` class). Some of the `Vector` methods are:

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]
```

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,
```

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(): []
```

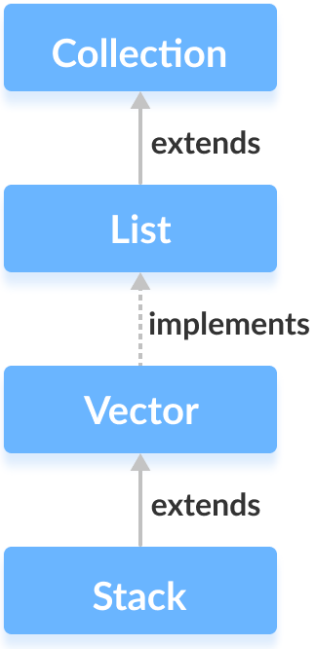
Others Vector Methods

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

Java Stack Class

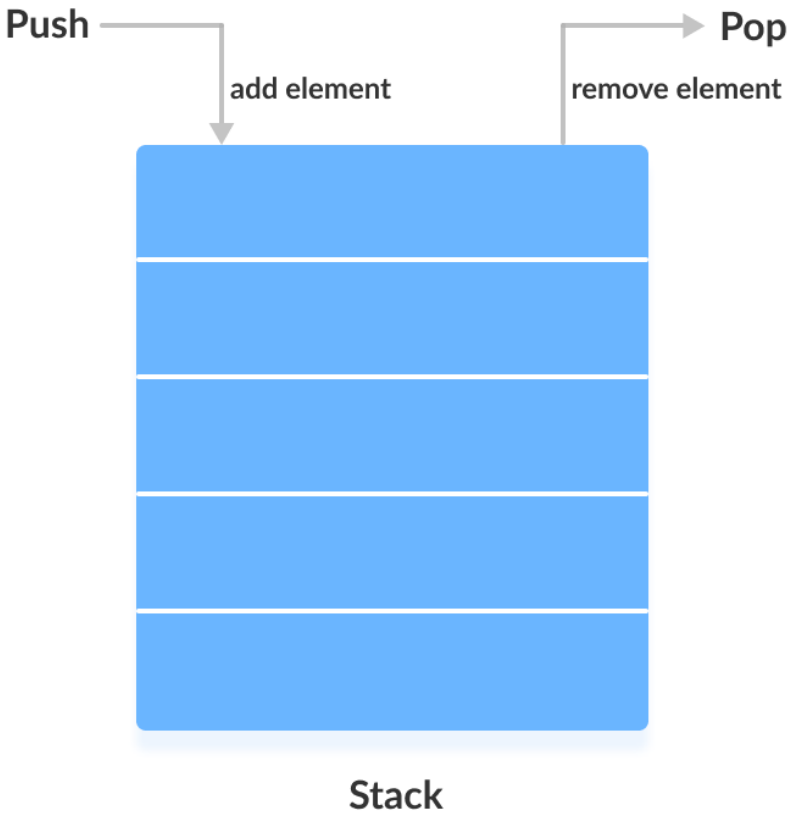
The Java collections framework has a class named `Stack` that provides the functionality of the stack data structure.

The `Stack` class extends the `Vector` class.



Stack Implementation

In stack, elements are stored and accessed in **Last In First Out** manner. That is, elements are added to the top of the stack and removed from the top of the stack.



Creating a Stack

In order to create a stack, we must import the `java.util.Stack` package first. Once we import the package, here is how we can create a stack in Java.

```
Stack<Type> stacks = new Stack<>();
```

Here, `Type` indicates the stack's type. For example,

```
// Create Integer type stack
Stack<Integer> stacks = new Stack<>();

// Create String type stack
Stack<String> stacks = new Stack<>();
```

Stack Methods

Since `Stack` extends the `Vector` class, it inherits all the methods `Vector`.

push() Method

To add an element to the top of the stack, we use the push() method. For example,

```
import java.util.Stack;

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

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");

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

Output

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

pop() Method

To remove an element from the top of the stack, we use the pop() method. For example,

```
import java.util.Stack;

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

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Initial Stack: " + animals);

        // Remove element stacks
        String element = animals.pop();
```

```
        System.out.println("Removed Element: " + element);
    }
}
```

Output

```
Initial Stack: [Dog, Horse, Cat]
Removed Element: Cat
```

peek() Method

The `peek()` method returns an object from the top of the stack. For example,

```
import java.util.Stack;

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

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Access element from the top
        String element = animals.peek();
        System.out.println("Element at top: " + element);

    }
}
```

Output

```
Stack: [Dog, Horse, Cat]
Element at top: Cat
```

search() Method

To search an element in the stack, we use the search() method. It returns the position of the element from the top of the stack. For example,

```
import java.util.Stack;

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

```
        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Search an element
        int position = animals.search("Horse");
        System.out.println("Position of Horse: " + position);
    }
}
```

Output

```
Stack: [Dog, Horse, Cat]
Position of Horse: 2
```

empty() Method

To check whether a stack is empty or not, we use the `empty()` method. For example,

```
import java.util.Stack;

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

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Check if stack is empty
        boolean result = animals.empty();
        System.out.println("Is the stack empty? " + result);
    }
}
```

Output

```
Stack: [Dog, Horse, Cat]
Is the stack empty? false
```

Use ArrayDeque Instead of Stack

The `Stack` class provides the direct implementation of the stack data structure. However, it is recommended not to use it. Instead, use the `ArrayDeque` class (implements the `Deque` interface) to implement the stack data structure in Java.