# 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.

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<>();
```

# Add Elements to Vector

```
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);
  }
}
```

# Access Vector Elements

```
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(", ");
      }
   }
}
```

# Remove Vector Elements

```
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);
   }
}
```

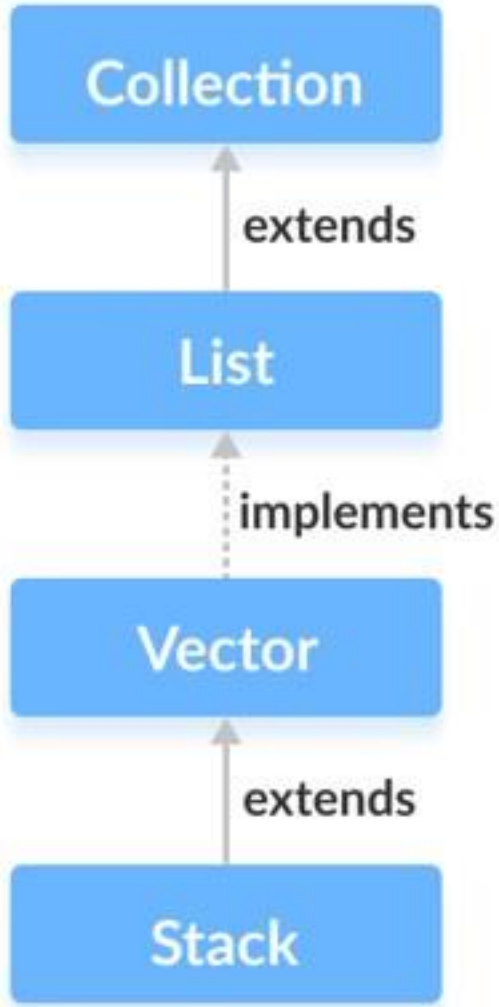
## Others Vector Methods

| 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 |

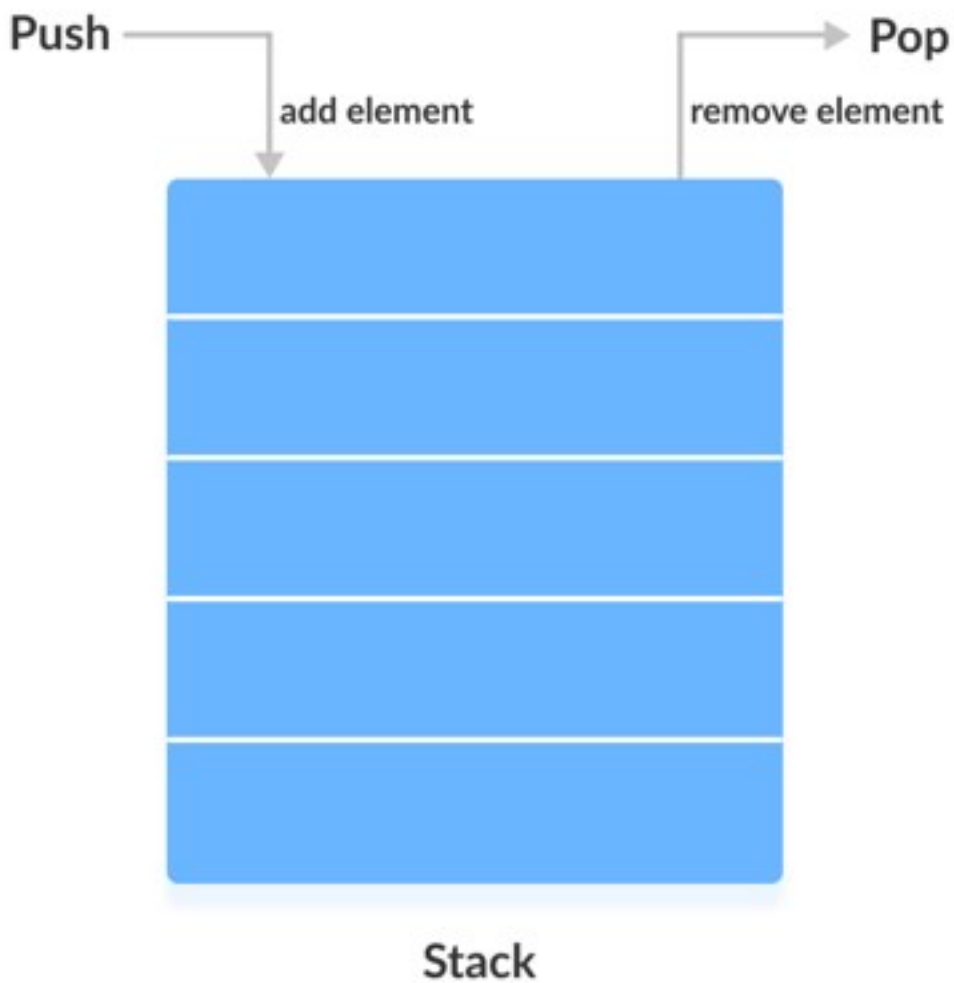
# Java Stack Class

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



## 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<>();
```

## 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);
   }
}
```

## 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);
   }
}
```

**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);

   }
}
```

**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);
   }
}
```

## 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);
   }
}
```

# Java Queue Interface

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**

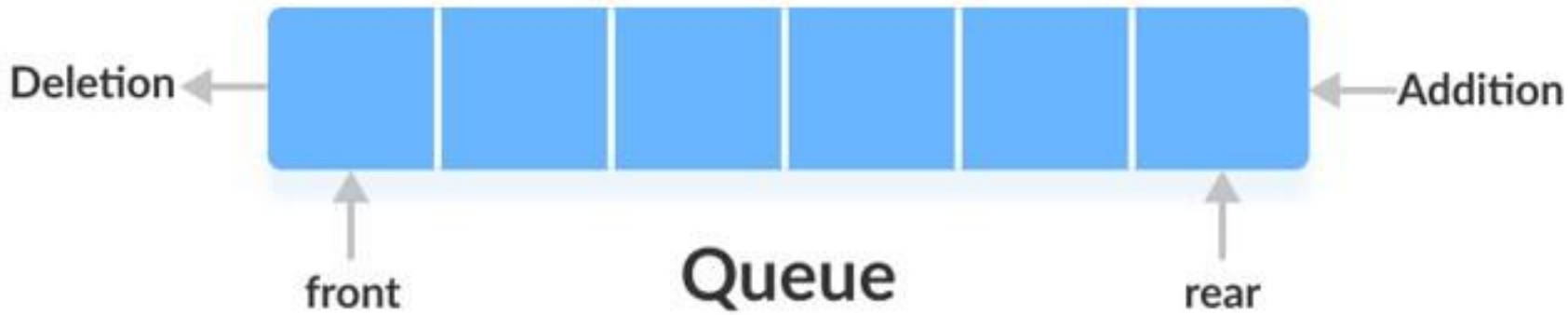
Since the Queue is an interface, we cannot provide the direct implementation of it.

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

- ArrayDeque
- LinkedList
- PriorityQueue

## 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<>();
```

**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.

# 1. Implementing the LinkedList Class

```
import java.util.Queue;
import java.util.LinkedList;
class Main {
    public static void main(String[] args) {
        // Creating Queue using the LinkedList class
        Queue<Integer> numbers = new LinkedList<>();
        // offer elements to the Queue
        numbers.offer(1);
        numbers.offer(2);
        numbers.offer(3);
        System.out.println("Queue: " + numbers);
        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " + accessedNumber);
       // Remove elements from the Queue
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " + removedNumber);
        System.out.println("Updated Queue: " + numbers);
    }
}
```

## 2. Implementing the PriorityQueue Class

```
import java.util.Queue;
import java.util.PriorityQueue;
class Main {
  public static void main(String[] args) {
    // Creating Queue using the PriorityQueue class
    Queue<Integer> numbers = new PriorityQueue<>();
    // offer elements to the Queue
    numbers.offer(5);
    numbers.offer(1);
    numbers.offer(2);
    System.out.println("Queue: " + numbers);
    // Access elements of the Queue
    int accessedNumber = numbers.peek();
    System.out.println("Accessed Element: " + accessedNumber);
    // Remove elements from the Queue
    int removedNumber = numbers.poll();
    System.out.println("Removed Element: " + removedNumber);
    System.out.println("Updated Queue: " + numbers);
  }
}
```

# Java Deque 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**.

**How to use Deque?**

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

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

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

Here, we have created objects animal1 and animal2 of classes ArrayDeque and LinkedList, respectively. These objects can use the functionalities of the Deque interface.

# Implementation of Deque in ArrayDeque Class

```
import java.util.Deque;
import java.util.ArrayDeque;
class Main {
   public static void main(String[] args) {
      // Creating Deque using the ArrayDeque class
      Deque<Integer> numbers = new ArrayDeque<>();
      // add elements to the Deque
      numbers.offer(1);
      numbers.offerLast(2);
      numbers.offerFirst(3);
      System.out.println("Deque: " + numbers);
      // Access elements of the Deque
      int firstElement = numbers.peekFirst();
      System.out.println("First Element: " + firstElement);
      int lastElement = numbers.peekLast();
      System.out.println("Last Element: " + lastElement);
      // Remove elements from the Deque
      int removedNumber1 = numbers.pollFirst();
      System.out.println("Removed First Element: " + removedNumber1);
      int removedNumber2 = numbers.pollLast();
      System.out.println("Removed Last Element: " + removedNumber2);
      System.out.println("Updated Deque: " + numbers);
   }
}
```

# Java LinkedList

The LinkedList class of the Java collections framework provides the functionality of the linked list data structure (doubly linkedlist).

## Creating a Java LinkedList

Here is how we can create linked lists in Java:

```
LinkedList<Type> linkedList = new LinkedList<>();
```

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

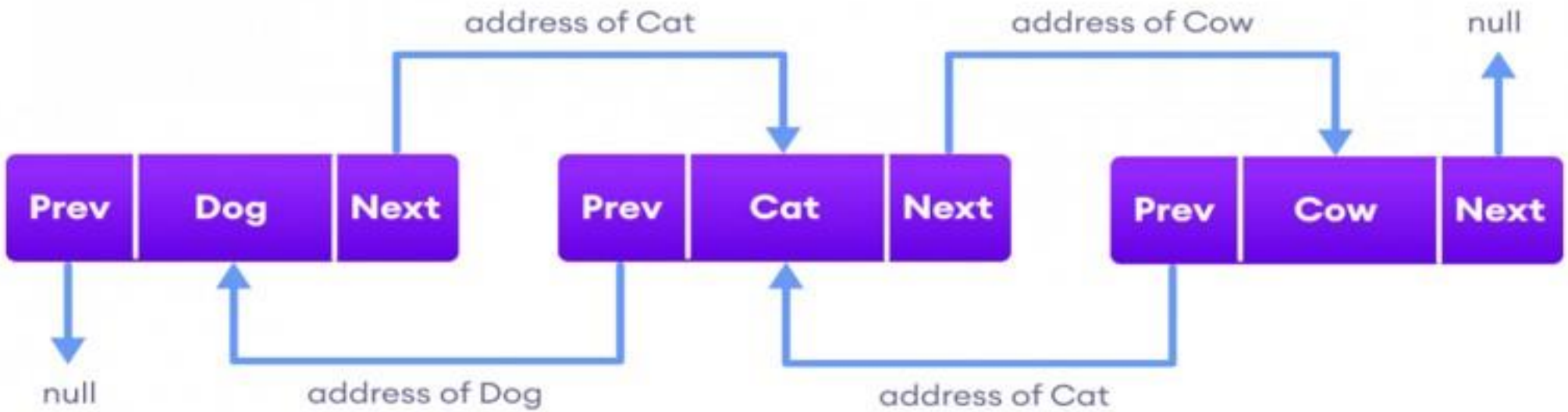
```
// create Integer type linked list
LinkedList<Integer> linkedList = new LinkedList<>();

// create String type linked list
LinkedList<String> linkedList = new LinkedList<>();
```

**Example: Create LinkedList in Java**

```
import java.util.LinkedList;

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

    // create linkedlist
    LinkedList<String> animals = new LinkedList<>();

    // Add elements to LinkedList
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Cow");
    System.out.println("LinkedList: " + animals);
  }
}
```

## Working of a Java LinkedList

Elements in linked lists are not stored in sequence. Instead, they are scattered and connected through links (**Prev** and **Next**).



## 1. Add elements to a LinkedList

We can use the add() method to add an element (node) at the end of the LinkedList. For example,

```
import java.util.LinkedList;
class Main {
  public static void main(String[] args){
    // create linkedlist
    LinkedList<String> animals = new LinkedList<>();
    // add() method without the index parameter
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Cow");
    System.out.println("LinkedList: " + animals);
    // add() method with the index parameter
    animals.add(1, "Horse");
    System.out.println("Updated LinkedList: " + animals);
  }
}
```

## 2. Access LinkedList elements

The get() method of the LinkedList class is used to access an element from the LinkedList. For example,

```
import java.util.LinkedList;
class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();
    // add elements in the linked list
    languages.add("Python");
    languages.add("Java");
    languages.add("JavaScript");
    System.out.println("LinkedList: " + languages);
    // get the element from the linked list
    String str = languages.get(1);
    System.out.print("Element at index 1: " + str);
  }
}
```

### 3. Change Elements of a LinkedList

The set() method of LinkedList class is used to change elements of the LinkedList. For example,

```
import java.util.LinkedList;
class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();
    // add elements in the linked list
    languages.add("Java");
    languages.add("Python");
    languages.add("JavaScript");
    languages.add("Java");
    System.out.println("LinkedList: " + languages);
    // change elements at index 3
    languages.set(3, "Kotlin");
    System.out.println("Updated LinkedList: " + languages);
  }
}
```

## 4. Remove element from a LinkedList

The remove() method of the LinkedList class is used to remove an element from the LinkedList.
For example,

```
import java.util.LinkedList;
class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();
    // add elements in LinkedList
    languages.add("Java");
    languages.add("Python");
    languages.add("JavaScript");
    languages.add("Kotlin");
    System.out.println("LinkedList: " + languages);
    // remove elements from index 1
    String str = languages.remove(1);
    System.out.println("Removed Element: " + str);
    System.out.println("Updated LinkedList: " + languages);
  }
}
```