Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 4

Lecture 30

Lecture 30

- Vector
- Stack
- Queue Interface

Vector

- Vector is a class that implements a dynamic array.
- It is similar to ArrayList, but it is synchronized, making it thread-safe.
- However, this synchronization can impact performance, so ArrayList is generally preferred unless synchronization is required.

- Vector is like the dynamic array which can grow or shrink its size.
- Unlike array, we can store n-number of elements in it as there is no size limit.
- It is a part of Java Collection framework since Java 1.2.
- It is found in the java.util package and implements the List interface, so we can use all the methods of List interface here.
- It is recommended to use the Vector class in the thread-safe implementation only.

Vector is similar to the ArrayList, but with two differences-

1. Vector is synchronized.
2. Java Vector contains many legacy methods that are not the part of a collections framework.

Java Vector Constructors

SN	Constructor	Description
1)	vector()	It constructs an empty vector with the default size as 10.
2)	vector(int initialCapacity)	It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
3)	vector(int initialCapacity, int capacityIncrement)	It constructs an empty vector with the specified initial capacity and capacity increment.

Java Vector Methods

SN	Method	Description
1)	add()	It is used to append the specified element in the given vector.
2)	addAll()	It is used to append all of the elements in the specified collection to the end of this Vector.
3)	addElement()	It is used to append the specified component to the end of this vector. It increases the vector size by one.
4)	capacity()	It is used to get the current capacity of this vector.
5)	clear()	It is used to delete all of the elements from this vector.

```java
import java.util.*;
public class VectorExample {
       public static void main(String args[]) {
         //Create a vector
         Vector<String> vec = new Vector<String>();
         //Adding elements using add() method of List
         vec.add("Tiger");
         vec.add("Lion");
         vec.add("Dog");
         vec.add("Elephant");
         //Adding elements using addElement() method of Vector
         vec.addElement("Rat");
         vec.addElement("Cat");
         vec.addElement("Deer");
        System.out.println("Elements are: "+vec);
       }
}
```

Feature	ArrayList	Vector
Synchronization	Not synchronized; not thread-safe	Synchronized; thread-safe
Performance	Faster due to lack of synchronization overhead	Slower due to synchronization overhead
Growth Policy	Increases capacity by 50% when full	Doubles its size when full
Legacy Status	Part of the Java Collections Framework (JCF)	Considered a legacy class in Java
Usage Recommendation	Preferred for single-threaded or low-concurrency applications	Use in multi-threaded applications where thread safety is required

Iterator	Fail-fast (throws ConcurrentModificationException if modified while iterating)	Fail-fast (throws ConcurrentModificationException if modified while iterating)
Initial Capacity	Can be specified; default is 10	Can be specified; default is 10
Capacity Increment	Can be adjusted manually	Can be specified; doubles if not specified
Enumeration	Not available	Supports both Enumeration and Iterator
Vector-specific Methods	No specific methods; uses List methods	Contains legacy methods such as addElement, removeElement, elements
Legacy Methods	No legacy methods; adheres to the List interface	Contains methods from earlier versions of Java (before Collections Framework)
Thread Safety Implementation	External synchronization needed if thread safety is required	Built-in synchronization; all methods are synchronized
API Consistency	More consistent with other Java Collections classes	Less consistent due to legacy methods and API

Methods:

- **`add(E element)`:** Adds an element to the end of the vector.
- **`add(int index, E element)`:** Inserts an element at the specified index.
- **`remove(int index)`:** Removes the element at the specified index.
- **`get(int index)`**: Returns the element at the specified index.
- **`size()`:** Returns the number of elements in the vector.
- **`capacity()`:** Returns the current capacity of the vector.
- **`elements()`:** Returns an enumeration of the components of the vector.
- **`firstElement()`:** Returns the first component of the vector.
- **`lastElement()`:** Returns the last component of the vector.
- **`setSize(int newSize)`:** Sets the size of the vector.

In this example, we create a Vector of integers, add elements to it, access an element using its index, remove an element, and iterate over the vector.

```
import java.util.Vector;
public class VectorExample {
   public static void main(String[] args) {
      Vector<Integer> vector = new Vector<>();
      // Adding elements to the Vector
      vector.add(10);
      vector.add(20);
      vector.add(30);
```

```
   // Accessing elements using index
      int element = vector.get(1);
      System.out.println("Element at index 1: " + element);
      // Removing an element
      vector.remove(0);
      // Iterating over the Vector
      for (Integer num : vector) {
         System.out.println(num);
      }
   }
}
```

Stack Class:

The Stack in Java represents a last-in, first-out (LIFO) stack of objects. It is based on the basic principle of stack data structure where elements are added to the top of the stack and removed from the top. It extends the Vector class with five operations that allow a vector to be treated as a stack.

The five operations are:

push(E item): Pushes an item onto the top of the stack.

pop(): Removes the object at the top of the stack and returns that object.

peek(): Looks at the object at the top of the stack without removing it from the stack.

empty(): Tests if the stack is empty.

search(Object o): Searches for the specified object in the stack and returns its position relative to the top of the stack (1-based index).

Stack Class

- Stack is a subclass of Vector and a legacy class in Java that extends `Vector`
- It implements the List interface.
- It is synchronized, making it thread-safe.
- It allows null elements.

Limitations:

- Stack is a subclass of Vector and a legacy class in Java that extends `Vector`. It has been part of Java since version 1.0. This means it inherits all the methods of `Vector`, which can be both unnecessary and confusing.
- The Stack class is not as efficient as other implementations of the stack data structure (like ArrayDeque) for certain operations because it is based on an array (via Vector) that may need to be resized, which can be costly in terms of performance.

Methods of the Stack Class

Method	Method Description
empty()	The method checks the stack is empty or not.
push(E item)	The method pushes (insert) an element onto the top of the stack.
pop()	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	The method looks at the top element of the stack without removing it.
search(Object o)	The method searches the specified object and returns the position of the object.

```
//creating an object of Stack class
Stack stk = new Stack();
//pushing elements into stack
stk.push("BMW");
stk.push("Audi");
stk.push("Ferrari");
stk.push("Bugatti");
stk.push("Jaguar");
//iteration over the stack
Iterator iterator = stk.iterator();
while(iterator.hasNext())
{
Object values = iterator.next();
System.out.println(values);
}
```

```
//creating an instance of Stack class
Stack <Integer> stk = new Stack<>();
//pushing elements into stack
stk.push(119);
stk.push(203);
stk.push(988);
System.out.println("Iteration over the stack using forEach()
Method:");
//invoking forEach() method for iteration over the stack
stk.forEach(n ->
{
System.out.println(n);
});
```

```
//creating an object of Stack class
Stack stk = new Stack();
//pushing elements into stack
stk.push("BMW");
stk.push("Audi");
stk.push("Ferrari");
stk.push("Bugatti");
stk.push("Jaguar");
//iteration over the stack
Iterator iterator = stk.iterator();
while(iterator.hasNext())
{
Object values = iterator.next();
System.out.println(values);
}
```

ABES Engineering College
College Code 032
Estd.2000

```
import java.util.Stack;
public class StackExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new
Stack<>();
// Pushing elements onto the stack
        stack.push(1);
        stack.push(2);
        stack.push(3);
```

```
  // Popping elements from the stack
        System.out.println(stack.pop());
 // Output: 3
        System.out.println(stack.pop());
 // Output: 2
        // Peeking at the top element of the
stack
        System.out.println(stack.peek());
 // Output: 1
        // Checking if the stack is empty
        System.out.println(stack.isEmpty())
;
 // Output: false
    }
}
```

Example: Part: 1

```
import java.util.Stack;
public class StackExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        // Pushing elements onto the stack
        stack.push("Java");
        stack.push("Python");
        stack.push("C++");
        // Peeking at the top element of the stack
```

Example: Part: 2

```
// Peeking at the top element of the stack
System.out.println("Top element: " + stack.peek());
 // Popping elements from the stack
System.out.println("Popped element: " + stack.pop());
System.out.println("Popped element: " + stack.pop());
 // Checking the size of the stack
 System.out.println("Stack size: " + stack.size());
// Checking if the stack is empty
 System.out.println("Is stack empty? " + stack.isEmpty());
 // Popping the last element
System.out.println("Popped element: " + stack.pop());
// Checking if the stack is empty after popping all elements
System.out.println("Is stack empty? " + stack.isEmpty()); } }
```

Queue Interface:

- The Queue interface in Java represents a first-in, first-out (FIFO) queue of objects.
- The `Queue` interface is a part of the Java Collections Framework and is used to represent a collection designed for holding elements prior to processing.
- It is primarily used to model data structures that provide FIFO (First-In-First-Out) access.
- The `Queue` interface extends the `Collection` interface.
- It extends the Collection interface and adds the insertion, removal, and inspection operations of a queue.
- The Queue interface provides several methods for adding, removing, and inspecting elements.

Queue Interface:

- Some of the key methods include:
- **`add(E e)`**: Adds the specified element to the queue. Throws an exception if the queue is full (not applicable for `LinkedList`).
- **`offer(E e)`**: Adds the specified element to the queue. Returns `true` if successful, `false` otherwise.
- **`remove()`**: Retrieves and removes the head of the queue. Throws an exception if the queue is empty.
- **`poll()`**: Retrieves and removes the head of the queue. Returns `null` if the queue is empty.
- **`element()`**: Retrieves, but does not remove, the head of the queue. Throws an exception if the queue is empty.
- **`peek()`**: Retrieves, but does not remove, the head of the queue. Returns `null` if the queue is empty.
- **`contains(Object o)`**: Returns `true` if the queue contains the specified element.

How to create a `Queue`?

In Java, you can create a `Queue` in several ways, depending on your requirements and the specific implementation you want to use. Here are the common ways to create a `Queue`:

Using LinkedList

```
Queue<Integer> queue1 = new LinkedList<>();
```

Using ArrayDeque

```
Queue<Integer> queue3 = new ArrayDeque<>();
```

An example demonstrating the use of the Queue interface in Java Part:1

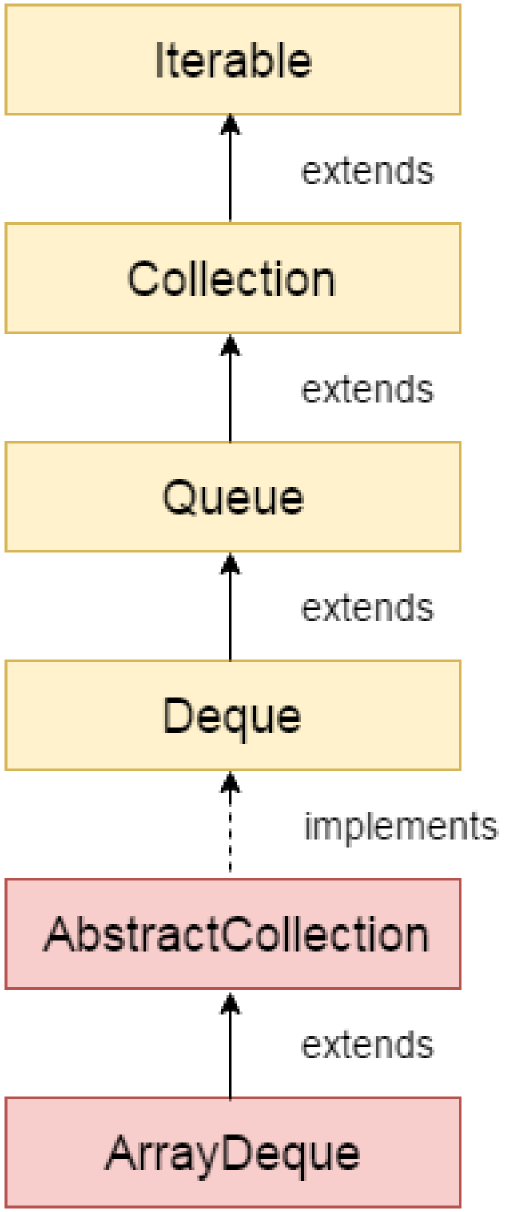
```
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        // Adding elements to the queue
        queue.add("Alice");
        queue.add("Bob");
        queue.add("Charlie");
```

An example demonstrating the use of the Queue interface in Java Part:2

```
    // Removing and returning the head of the queue
    System.out.println("Removed element: " + queue.remove()); // Output: Alice
      // Returning the head of the queue without removing it
      System.out.println("Peeked element: " + queue.peek()); // Output: Bob
      // Checking if the queue is empty
      System.out.println("Is queue empty? " + queue.isEmpty()); // Output: false
   }
}
```

ArrayDeque class

we need a class that implements the Deque interface, and that class is ArrayDeque. It grows and shrinks as per usage. It also inherits the AbstractCollection class.



ABES Engineering College
College Code 032
Estd.2000

ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

```
import java.util.*;
public class ArrayDequeExample {
  public static void main(String[] args) {
  //Creating Deque and adding elements
  Deque<String> deque = new ArrayDeque<String>();
  deque.add("Ravi");
  deque.add("Vijay");
  deque.add("Ajay");
  //Traversing elements
  for (String str : deque) {
  System.out.println(str);
  }
  }
}
```

Methods of Java Deque Interface

Method	Description
boolean add(object)	It is used to insert the specified element into this deque and return true upon success.
boolean offer(object)	It is used to insert the specified element into this deque.
Object remove()	It is used to retrieve and removes the head of this deque.
Object poll()	It is used to retrieve and removes the head of this deque, or returns null if this deque is empty.
Object element()	It is used to retrieve, but does not remove, the head of this deque.
Object peek()	It is used to retrieve, but does not remove, the head of this deque, or returns null if this deque is empty.

bject peekFirst()	The method returns the head element of the deque. The method does not remove any element from the deque. Null is returned by this method, when the deque is empty.
Object peekLast()	The method returns the last element of the deque. The method does not remove any element from the deque. Null is returned by this method, when the deque is empty.
Boolean offerFirst(e)	Inserts the element e at the front of the queue. If the insertion is successful, true is returned; otherwise, false.
Object offerLast(e)	Inserts the element e at the tail of the queue. If the insertion is successful, true is returned; otherwise, false.

```
Deque<String> deque=new ArrayDeque<String>();
   deque.offer("arvind");
   deque.offer("vimal");
   deque.add("mukul");
   deque.offerFirst("jai");
   System.out.println("After offerFirst Traversal...");
   for(String s:deque)
  {
      System.out.println(s);
   }
```

Output

```
After offerFirst Traversal...
jai
arvind
vimal
mukul
```