TASK

Java - Collections Framework

Visit our website

Introduction

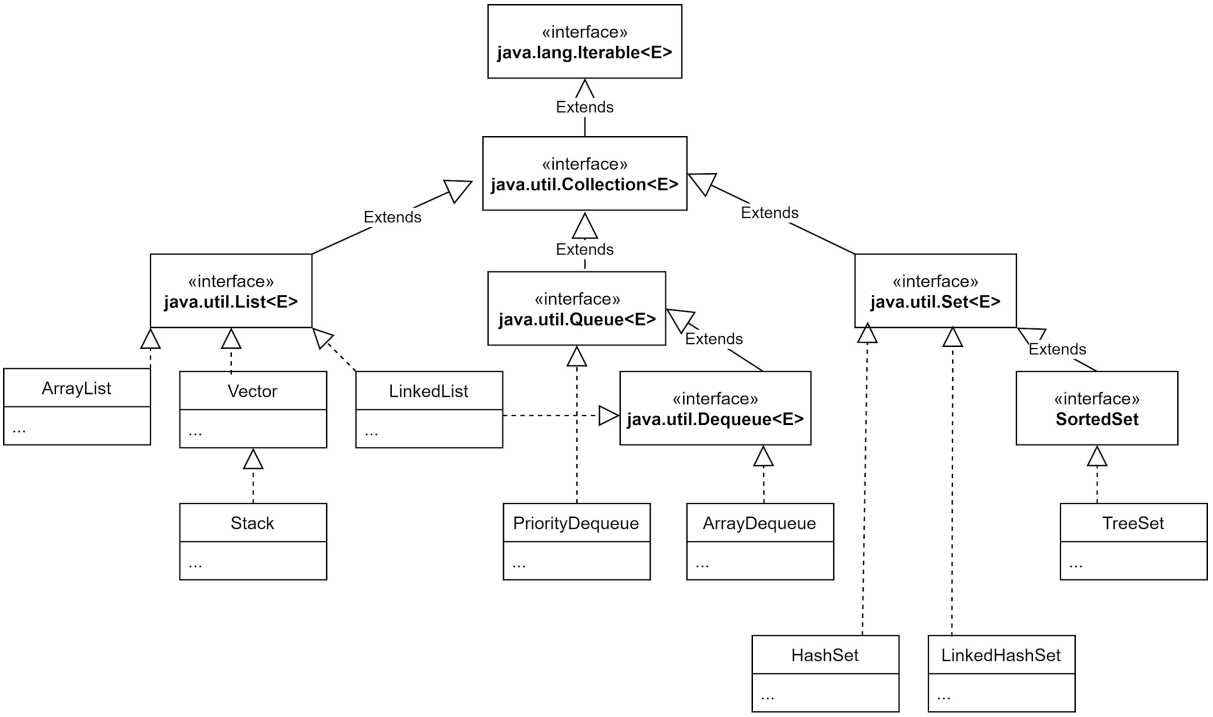
WELCOME TO THE JAVA - COLLECTIONS FRAMEWORK TASK!

In this task, we discuss the Java Collections Framework and how tools in this framework can be used to make your job as a software engineer easier and your programs better.

THE JAVA COLLECTIONS FRAMEWORK

To understand what Java's Collections Framework is, we must first understand what a **collection** is. As a child, you may have had a collection of stickers or marbles. A collection is a group of items that are stored together. In programming, a collection is a data structure that stores several items together. As you know, a data structure not only stores data but also supports operations for accessing and manipulating the data. Therefore, a collection is a data structure that allows you to store and manipulate several items. A data structure is implemented by creating a class. The class for a data structure should provide methods to support such operations as search, insertion, and deletion.

Oracle defines the Java Collections Framework as "a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details." In simple terms, this means that there can be many different types of collections. However, the Java Collections Framework provides a way of making sure that all these various collections act similarly. This makes it easier for programmers to learn and work with different collections. An essential part of the Java Collections Framework is the collection's interfaces. Interfaces provide the blueprint for how all collections in the Java Collections Framework will behave. Below is a flowchart of how the interfaces and classes are related. As you read this task, refer back to this diagram to help you make sense of how everything fits together *(Hint: it is a very common interview question to ask about this flowchart!)*.



JAVA COLLECTION INTERFACES

Remember that in OOP, an interface is a blueprint of classes to come. It is an abstract type that specifies the behaviour of future classes. The methods in an interface don't contain any logic. The logic for the methods specified in the interface is written in the classes that implement the interface. In essence, an interface shows a class what to do, and not how to do it.

The Java Collections Framework ensures that all collections can be used and manipulated in similar ways by using interfaces. The Java Collections Framework supports two types of collection interfaces:

1. The most basic interface is `java.util.Collection`.
2. The other collection interfaces are based on `java.util.Map`. Maps aren't true collections.

Maps are useful for quickly searching for an element using a key. However, in this task, we will only focus on true collections.

There are three different kinds of collections:

1. **Sets**, which store a group of non-duplicate elements.
2. **Lists**, which store an ordered collection of elements. Lists can include duplicate items.
3. **Queues**, which store objects that are processed in a first-in, first-out fashion.

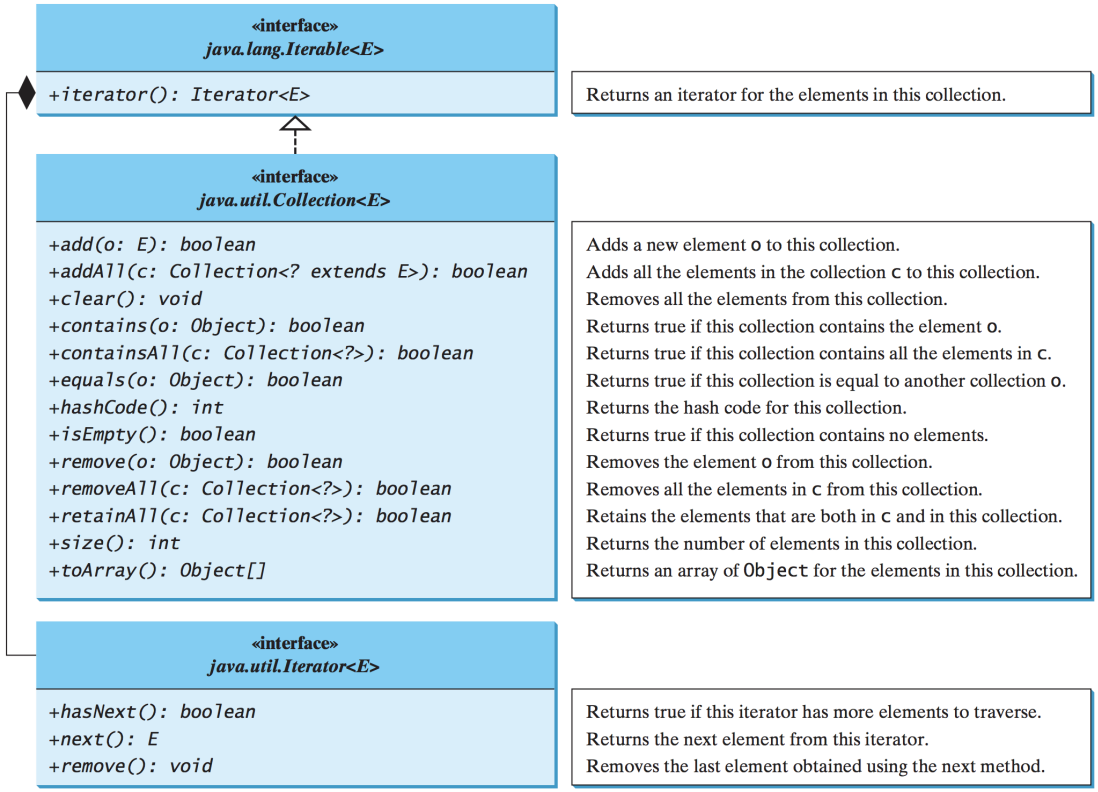
The `java.util.Collection` interface specifies that all classes that inherit from it should contain the following methods:

- `add()` — The *add* method adds an element to the collection.
- `addAll()` — The *addAll* method adds all the elements in the specified collection to this collection.
- `remove()` — The *remove* method removes an element from the collection.
- `removeAll()` — The *removeAll* method removes the elements from this collection that are present in the specified collection.
- `retainAll()` — The *retainAll* method retains the elements in this collection that are also present in the specified collection.

All these methods return a boolean. The return value is true if the collection is changed as a result of the method execution.

- `clear()` — The *clear* method simply removes all the elements from the collection.
- `size()` — The *size* method returns the number of elements in the collection.
- `contains()` — The *contains* method checks whether the collection contains the specified element.
- `containsAll()` — The *containsAll* method checks whether the collection contains all the elements in the specified collection.
- `isEmpty()` — The *isEmpty* method returns true if the collection is empty.
- `toArray()` — The *toArray* method returns an array representation for the collection.

Below is a UML diagram showing all the Collection interface's public methods:



Collection interface's public methods: (Liang, 2011, page 729)

ITERATORS

Besides the methods described in the previous section, the `java.util.Collection` interface also specifies an **iterator** object. Therefore, each collection has an iterator object. An iterator is used to traverse all the elements in the collection.

Iterator methods:

- `next()` — The iterator traverses through the elements in the collection using the `next()` method. See the code example below.
- `hasNext()` — The `hasNext()` method checks whether there are more elements in the iterator.
- `remove()` — The `remove()` method removes the last element returned by the iterator.

The code below shows how an iterator is used to traverse all the elements in an array list.

```
import java.util.*;

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

```
    Collection<String> collection = new ArrayList<String>();
    collection.add("New York");
    collection.add("Atlanta");
    collection.add("Dallas");
    collection.add("Madison");

    Iterator<String> iterator = collection.iterator();
    while (iterator.hasNext()) {
      System.out.print(iterator.next().toUpperCase() + " ");
    }
    System.out.println();
  }
}
```

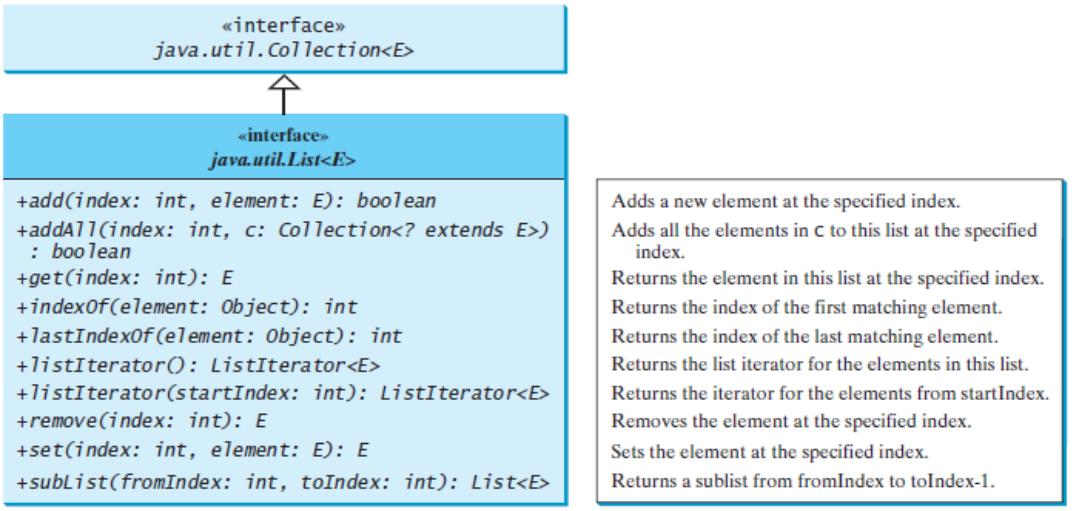
Output:

```
NEW YORK ATLANTA DALLAS MADISON
```

The program above creates a concrete collection object using `ArrayList` and adds four strings into the list. The program then obtains an iterator for the collection. The iterator traverses the strings in the list and displays the strings in uppercase.

LISTS

The **List** interface extends the Collection interface. Therefore, Lists include the methods described in the Collections interface. Lists store a collection of items in *sequential* order. Items can be inserted into or removed from specific locations in a list. Lists can contain duplicate items or null values.



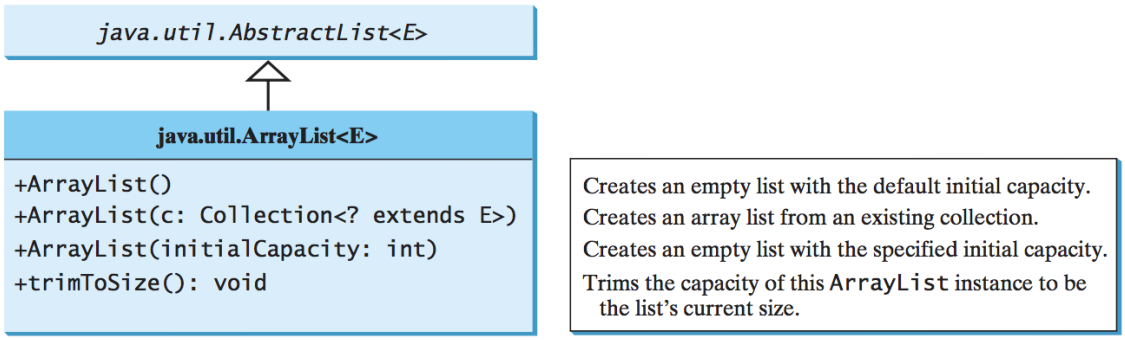
List interface's public methods (Liang, 2011, page 739)

There are two types of Lists: `ArrayList` or `LinkedList`.

ArrayLists

ArrayLists are very similar to arrays. Both ArrayLists and arrays allow you to store several items and access those items using indices. A key difference between arrays and ArrayLists is that arrays have a fixed size once they have been created. On the other hand, the size of an ArrayList can be dynamically changed. Therefore, when deciding whether to use an array or an ArrayList, determine whether you know how many items you need to store ahead of time or not. As elements are added to an ArrayList, its capacity grows automatically. An ArrayList does not automatically shrink.

Since ArrayLists implement the List interface, all the methods that are defined in the List class are used with ArrayLists. In addition, ArrayLists include the `trimToSize()` method. You can use the `trimToSize()` method to reduce the array capacity to the size of the list.



ArrayList's public methods (Liang, 2011, page 740)

In the example below, `arrayList.add(1);` adds the item, 1 to the `arrayList`. `arrayList.add(0, 10);` adds the item 10 to index 0 of the `arrayList`. Sorting lists is also easy. Notice how `arrayList` is sorted using the code: `Collections.sort(arrayList);`

```
import java.util.*;

public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<Integer>();
    arrayList.add(1); // 1 is autoboxed to new Integer(1)
    arrayList.add(0, 10);
    arrayList.add(3, 30);
```

```
    Collections.sort(arrayList);
  }
}
```

LinkedLists

A **LinkedList** is a data structure where each element in the list is a separate object that includes a reference to the object before and after it in the List (see the image below).

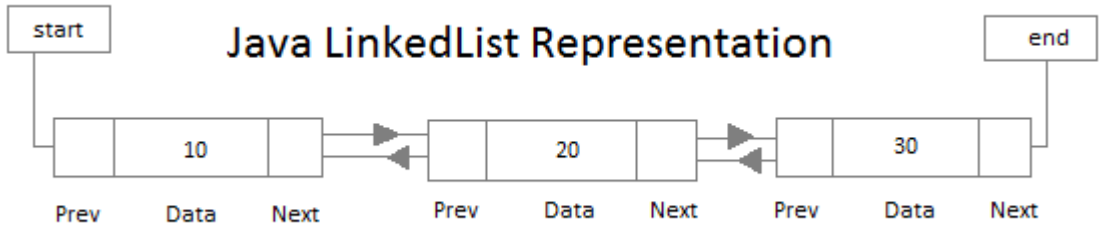
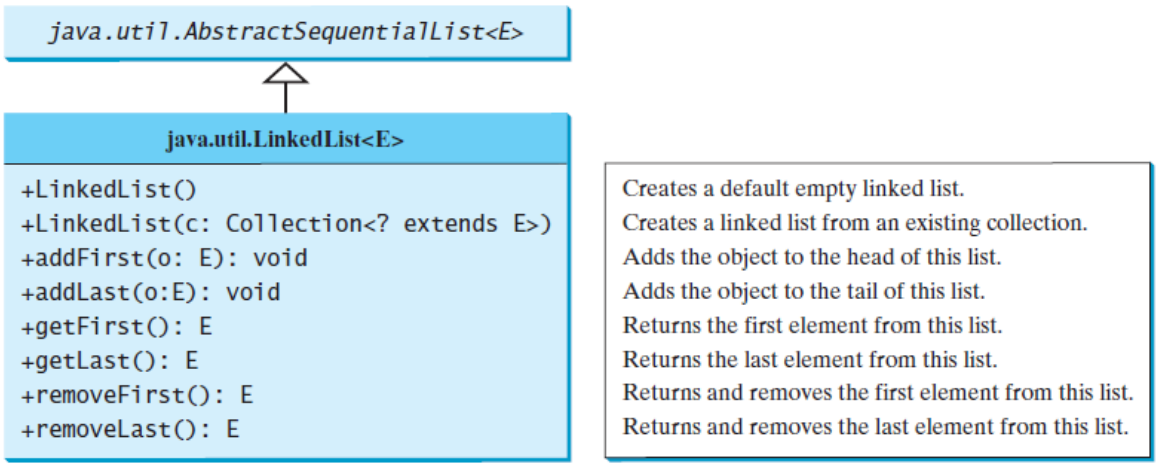


Image source: ***http://www.gkpedia.in/java-linkedlist-tutorial-with-example/***

Since LinkedLists also implement the List interface, all the methods that are defined in the List class are used in the LinkedList class. In addition to implementing the List interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list.



LinkedList's public methods (Liang, 2011, page 741)

The program by Liang (2011) below creates an ArrayList filled with numbers and inserts new elements into specified locations in the list. It also creates a LinkedList from the ArrayList and inserts and removes elements from the list. Finally, it traverses the list forward and backwards.

```
import java.util.*;

public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<Integer>();
    arrayList.add(1); // 1 is autoboxed to new Integer(1)
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(1);
    arrayList.add(4);
    arrayList.add(0, 10);
    arrayList.add(3, 30);

    System.out.println("A list of integers in the array list:");
    System.out.println(arrayList);

    LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
    linkedList.add(1, "red");
    linkedList.removeLast();
    linkedList.addFirst("green");

    System.out.println("Display the linked list forward:");
    ListIterator<Object> listIterator = linkedList.listIterator();
    while (listIterator.hasNext()) {
      System.out.print(listIterator.next() + " ");
    }
    System.out.println();

    System.out.println("Display the linked list backwards:");
    listIterator = linkedList.listIterator(linkedList.size());
    while (listIterator.hasPrevious()) {
      System.out.print(listIterator.previous() + " ");
    }
  }
}
```

Output:

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forward:
green 10 red 1 2 30 3 1
Display the linked list backwards:
1 3 30 2 1 red 10 green
```

A list can hold identical elements. Integer 1 is stored twice in the list.

ArrayLists vs LinkedLists

ArrayList and LinkedList operate similarly. The main difference between them is their internal implementation, which affects their performance. ArrayList is efficient for retrieving elements, while LinkedList is efficient for inserting and removing elements at the beginning of the list. Both have the same performance for inserting and removing elements in the middle or at the end of the list. LinkedLists use more memory than ArrayLists.

QUEUES AND PRIORITY QUEUES

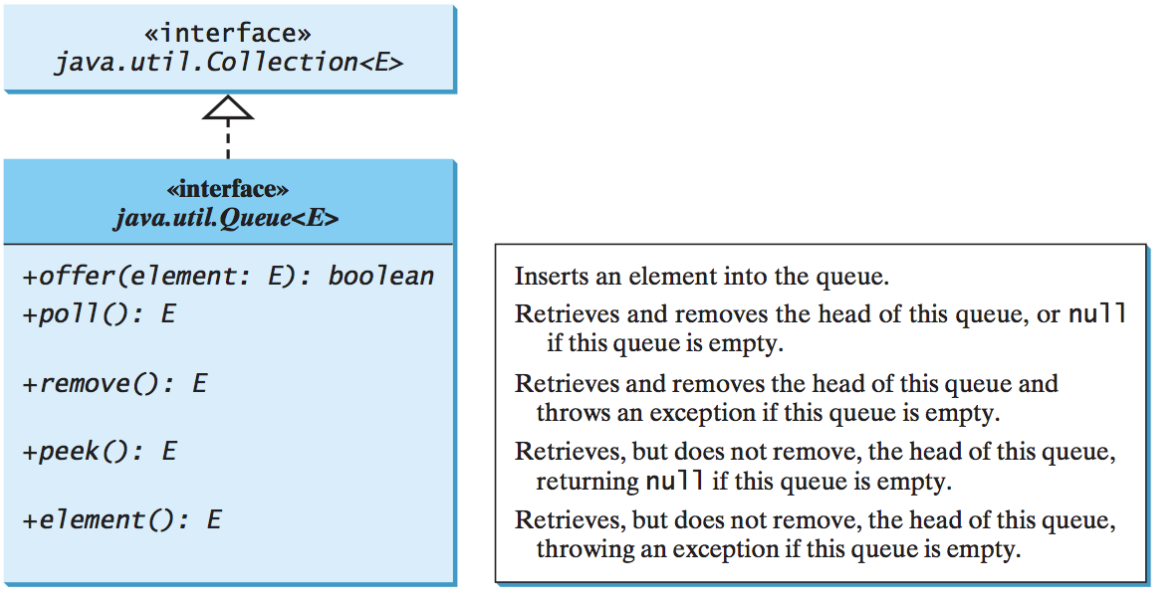
A queue is a concept with which we are all familiar. We have all waited in a queue at a bank, bathroom or checkout line at the grocery store. Queues are also often encountered in IT, e.g. print queues. In computer science, a queue is a simple data structure where data has to enter the queue at the back and leave the queue from the front. It is therefore known as a first-in, first-out (FIFO) data structure.

Queue terminology:

- **Enqueue** — is an operation that adds an item to the back of a queue.
- **Dequeue** — is an operation that removes an item from the front of a queue.

Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

The Queue interface extends `java.util.Collection` with additional insertion, extraction, and inspection operations. The diagram below shows all the methods in the Queue interface:



Queue's public methods (Liang, 2011, page 748)

The LinkedList class implements the Deque interface, which extends the Queue interface. Therefore, you can use LinkedList to create a queue. LinkedList is ideal for queue operations because it is efficient for inserting and removing elements from both ends of a list.

Deque supports element insertion and removal at both ends. The name *deque* is short for "double-ended queue" and is usually pronounced "deck." The Deque interface extends Queue with additional methods for inserting and removing elements from both ends of the queue. The methods `addFirst(e)`, `removeFirst()`, `addLast(e)`, `removeLast()`, `getFirst()`, and `getLast()` are defined in the Deque interface.

The following code uses a Queue to store strings.

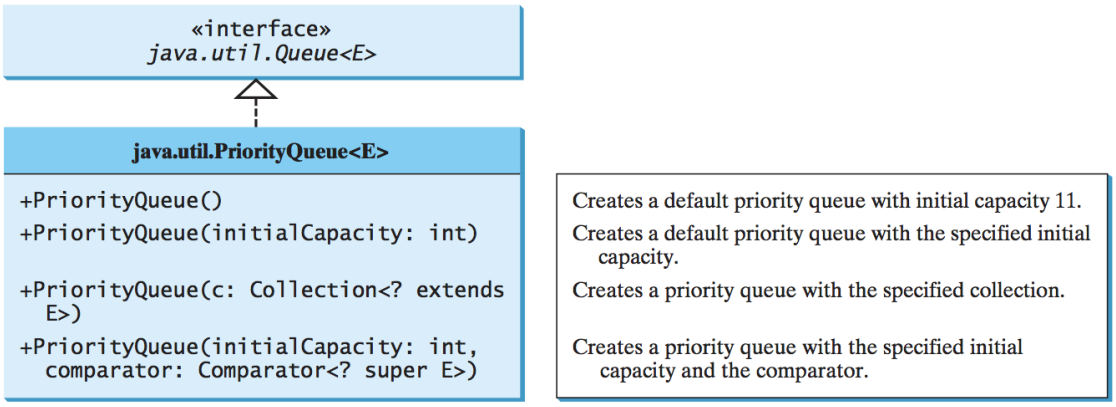
```
public class TestQueue {
    public static void main(String[] args) {
      java.util.Queue<String> queue = new java.util.LinkedList<String>();
      queue.offer("Collection");
      queue.offer("List");
      queue.offer("Queue");
      queue.offer("Set");

      while (queue.size() > 0) {
        System.out.print(queue.remove() + " ");
      }
   }
}
```

Output:

```
Collection List Queue Set
```

The code above creates a queue using LinkedList. It then adds four strings to the queue. The `size()` method defined in the Collection interface then returns the number of elements in the queue, and the `remove()` method retrieves and removes the element at the head of the queue.

The PriorityQueue class implements a priority queue. By default, the priority queue orders its elements according to their natural ordering. The element with the smallest value is assigned the highest priority and thus is removed from the queue first. If there are several elements with the same highest priority, the tie is broken arbitrarily. The diagram below shows all the methods in the PriorityQueue class:



PriorityQueue's public methods (Liang, 2011, page 750)

The code by Liang (2011) below uses a PriorityQueue to store strings.

```
import java.util.*;
public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<String> queue1 = new PriorityQueue<String>();
    queue1.offer("Oklahoma");
    queue1.offer("Indiana");
    queue1.offer("Georgia");
    queue1.offer("Texas");

    System.out.println("Priority queue using Comparable:");
    while (queue1.size() > 0) {
      System.out.print(queue1.remove() + " ");
    }
    PriorityQueue<String> queue2 = new PriorityQueue<String>(4,
Collections.reverseOrder());
    queue2.offer("Oklahoma");
```

```
    queue2.offer("Indiana");
    queue2.offer("Georgia");
    queue2.offer("Texas");

    System.out.println("\nPriority queue using Comparator:");
    while (queue2.size() > 0) {
      System.out.print(queue2.remove() + " ");
    }
  }
}
```

Output:

```
Priority queue using Comparable:
Georgia Indiana Oklahoma Texas
Priority queue using Comparator:
Texas Oklahoma Indiana Georgia
```

This program creates a PriorityQueue for strings using its no-arg constructor. This PriorityQueue orders the strings using their natural order, so the strings are removed from the queue in increasing order. It then creates a priority queue using the comparator obtained from `Collections.reverseOrder()`, which orders the elements in reverse order, so the strings are removed from the queue in decreasing order.

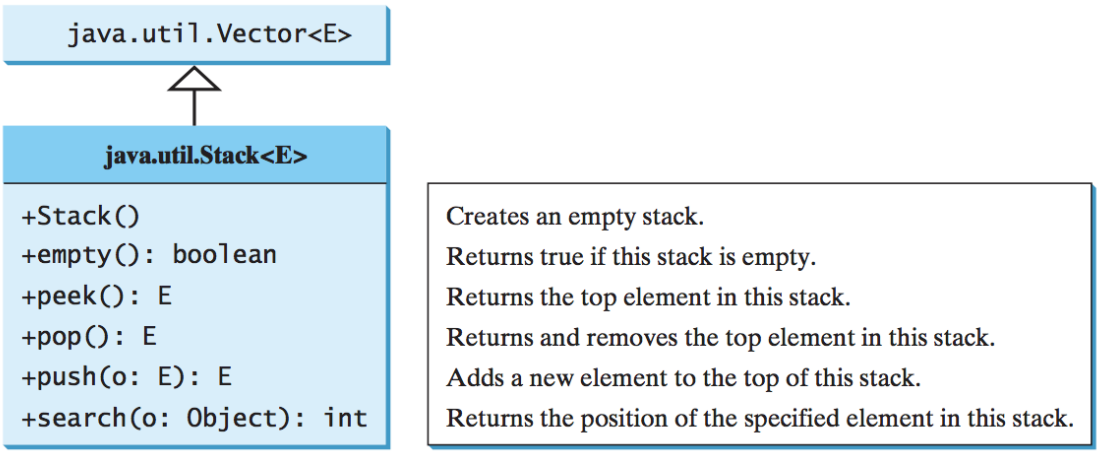
STACKS

A stack is a data structure where items are added to the top of the stack and removed from the top of the stack. It is therefore known as a last-in, first-out (LIFO) data structure.

Stack terminology:

- **Push** — is an operation that adds an item to the top of a stack.
- **Pop** — is an operation that removes an item from the top of a stack.

In the Java Collections Framework, Stack is implemented as an extension of Vector. Vector is the same as ArrayList, except that it contains synchronised methods for accessing and modifying the vector. Synchronised methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently. For the many applications that do not require synchronisation, using ArrayList is more efficient than using Vector. The Vector class extends the AbstractList class. The diagram below shows all the methods in the Stack class:



Stack's public methods (Liang, 2011, page 748)

Extra resource

There are many more classes in the Collections Framework than we have time to get into here! For more information on the different classes in the Java Collections Framework go to **https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html**.

Compulsory Task 1

Follow these steps:

- Create a Java file called **priorityQueue.java**. In this file, recreate a priority queue object without actually using PriorityQueue.
- Methods to recreate: `offer`, `remove`, `sort` and `size`.
- You are not allowed to use the built-in ArrayList methods to do these.
- Your program should be able to act like a Priority Queue when given an ArrayList of strings.
- Compile, save and run your file.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References

gkpedia. (n.d.). Java LinkedList tutorial with examples. Retrieved from gkpedia: http://www.gkpedia.in/java-linkedlist-tutorial-with-example/

Liang, Y. (2011). *Introduction to Java Programming Comprehensive Version* (Eighth ed.). New Jersey: Prentice Hall.

Oracle. (1993). Collections Framework Overview. Retrieved from Oracle Java Documentation: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html