Day-14 JA-111 Batch

Topics

- **The java.lang.Iterable<T> interface**
- **The java.util.Iterator<T> interface**
- **The java.util.Collection<E> interface**
- **The java.util.List<E> interface**
- **The java.util.ArrayList<E> class**
- **The java.util.Queue<E> & java.util.Dequeue<E> interface**
- **The java.util.LinkedList<E> class**
- **The java.util.Vector<E> & java.util.Stack<E> class**
- **The java.util.Set<E> interface**
- **The java.util.HashSet<E> & java.util.LinkedHashSet<E> class**

The java.lang.Iterable<T> interface

This interface contains only one abstract method

Method	Description
Iterator<T> iterator()	Returns an iterator over elements of type T.

Before talking about the collection interface, Let us talk about the ***java.util.Iterator<T> interface*** first because actually this interface provides the facility of iterating the elements in a *forward direction only*. This interface has following abstract methods

Method	Description
boolean hasNext()	Returns true if the iteration has more elements. (In other words, returns true if next() would return an element rather than throwing an exception.)
E next()	Returns the next element in the iteration.

One concrete method of this interface is

Method	Description
default void remove()	Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to next().

The java.util.Collection<E> interface

This interface is root interface in the collection hierarchy. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List.

This interface contains following abstract methods

Method	Description
boolean add(E e)	Ensures that this collection contains the specified element (optional

	operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)
boolean addAll(Collection<? extends E> c)	Adds all of the elements in the specified collection to this collection (optional operation). The behaviour of this operation is undefined if the specified collection is modified while the operation is in progress. (This implies that the behaviour of this call is undefined if the specified collection is this collection, and this collection is nonempty)
void clear()	Removes all of the elements from this collection (optional operation). The collection will be empty after this method returns
boolean contains(Object o)	Returns true if this collection contains the specified element. More formally, returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e))
boolean containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
boolean equals(Object o)	Compares the specified object with this collection for equality.
boolean isEmpty()	Returns true if this collection contains no elements
Iterator<E> iterator()	Returns an iterator over the elements in this collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee).
boolean remove(Object o)	Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element e such that (o==null ? e==null : o.equals(e)), if this collection contains one or more such elements. Returns true if this collection contained the specified element (or equivalently, if this collection changed as a result of the call)
boolean removeAll(Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation). After this call returns, this collection will contain no elements in common with the specified collection.
boolean retainAll(Collection<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation). In other words, removes from this collection all of its elements that are not contained in the specified collection
int size()	Returns the number of elements in this collection. If this collection contains more than Integer.MAX_VALUE elements, returns Integer.MAX_VALUE
Object[] toArray()	Returns an array containing all of the elements in this collection. If this collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order
<T> T[] toArray(T[] a)	Returns an array containing all of the elements in this collection; the

	runtime type of the returned array is that of the specified array. If the collection fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection

The java.util.List<E> interface

- List interface provides an ordered collection (elements are arranged in the list according to their insertion of order.) that allow to access element from a certain location using integer index. List is zero based means similar to array elements have starting index zero.
- Duplicate elements are allowed
- null values can also be inserted
- It has all methods of Collection interface and others are

Method	Description
void add(int index, E element)	Inserts the specified element at the specified position in this list (optional operation)
boolean addAll(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list at the specified position (optional operation)
E get(int index)	Returns the element at the specified position in this list
int indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element
int lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element
ListIterator<E> listIterator()	Returns a list iterator over the elements in this list (in proper sequence)
ListIterator<E> listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E remove(int index)	Removes the element at the specified position in this list (optional operation)
E set(int index, E element)	Replaces the element at the specified position in this list with the specified element (optional operation)
List<E> subList(intfromIndex, inttoIndex)	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive

A note about wrapper classes

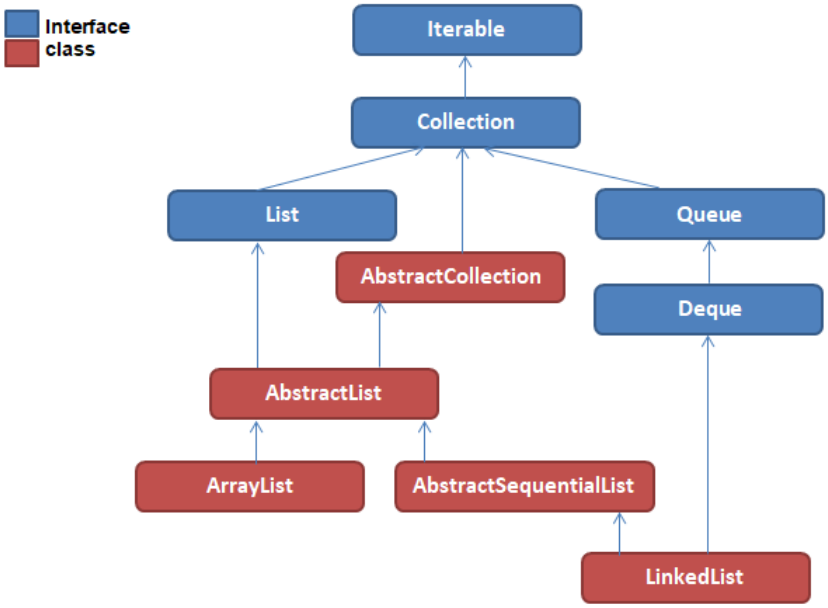
None of the collection classes work with the primitive types, they work only for objects and when a primitive value is added to collection then it is automatically converted to object of corresponding wrapper class. The automatic conversion of primitive type to object is called auto-boxing and vice versa is called auto-unboxing. Take a look at the example of auto-boxing and auto-unboxing-

An Example

```
Integer a = Integer.valueOf(10); //Manual boxing
Integer b = 20; //auto-boxing because no additional syntax by programmer
```

```
int c = a.intValue(); //Manual unboxing
int d = b; //auto-unboxing because no addition syntax by programmer
```

Hierarchy of List interface and implementing classes



ArrayList class

ArrayList class is provided by java to support implementation of dynamic array. Standard arrays are of fixed length i.e. they can't grow or shrink according to requirement at run time. Sometime it is not possible to know exact size of array in advanced. To handle such kind of situation java provides ArrayList class. An ArrayList's object presents a dynamic array. Addition of an element inside ArrayList object may cause expansion in size of array similarly deletion of an element from ArrayList may cause shrinking of array size.

ArrayList class stores element in sequential order hence inserting an element in middle of an ArrayList cause shifting of remaining element that Also, adding more elements than the capacity of the underlying array, a new array (twice the size) is allocated, and the old array is copied to the new one, so adding to an ArrayList is O(n) in the worst case but constant on average.

ArrayList allow random access of element in constant time.

Constructor	Description
ArrayList()	Constructs an empty list with an initial capacity of 10. Once the ArrayList reaches to the maximum capacity, then internally a new ArrayList object will be created in the memory automatically with formula newCapacity = (currentCapacity * 3/2)+1;
ArrayList(Collection<? extends E> c)	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
ArrayList(int initialCapacity)	Constructs an empty list with the specified initial capacity.

In addition to the methods of List interface, ArrayList class have following method

Method	Description
void ensureCapacity(int minCapacity)	Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void trimToSize()	Trims the capacity of this ArrayList instance to be the list's current size.

An Example:

```
List<String> ls = new ArrayList<String>();
ls.add("A");
ls.add("B");
ls.add(1,"W"); //now elements of the list are W A B
ls.add("D");
ls.add("D");   //duplicate element is allowed
ls.add("T");
ls.add(null);  //null can be added
ls.add(null);  //even for multiple times.

System.out.println(ls);  //[A, W, B, D, D, T, null, null]
```

Note: All the collection classes internally overrides the toString() method, so when we print the object of the collection classes, it will print the elements inside the [] square bracket, instead of printing the address.

Traversing elements of List using for loop

```
for(int i = 0; i < ls.size(); i++)
  System.out.print(ls.get(i) + " ");  //A W B D D T null null
```

Traversing elements of List using enhanced for loop/foreach loop

```
for(String element: ls)
  System.out.print(element + " ");  //A W B D D T null null
```

Traversing elements of List using Iterator

```
Iterator<String> itr = ls.iterator();
while(itr.hasNext())
  System.out.print(itr.next() + " ");  //A W B D D T null null
```

Using iterator to remove element from the list

```
itr = ls.iterator();
while(itr.hasNext()) {
  if(itr.next() == null)
    itr.remove();
}
System.out.println(ls);  //[A, W, B, D, D, T]
```

Finding elements in the List

```
System.out.println(ls.contains(null));  //false
System.out.println(ls.indexOf(null));  //-1
System.out.println(ls.contains("D"));  //true
System.out.println(ls.indexOf("D"));  //3
System.out.println(ls.lastIndexOf("D"));  //4
```

Changing element of the list

```
ls.set(4, null);  //now element of list are
System.out.println(ls);  //A, W, B, D, null, T
```

Remove elements from list

```
ls.remove(4);  //remove by index
ls.remove("T");  //remove by element
System.out.println(ls);  //A, W, B, D
```

Using ListIterator to iterate in forward direction and add elements

```
ListIterator<String> listItr = ls.listIterator();
while(listItr.hasNext()) {
  String element = listItr.next();
  System.out.print(element + " ");  //A W B D
  if(element.equals("A"))
    listItr.add("Z");
}
```

Using ListIterator to iterate in backward direction

```
while(listItr.hasPrevious()) {
  String element = listItr.previous();  //D B W Z A
  System.out.print(element + " ");
  if(element.equals("Z"))
    listItr.remove();
}
System.out.println("\n" + ls);  //[A, W, B, D]
```

Delete all elements from the list at once (but not list)

```
ls.clear();
System.out.println("Is list empty after calling clear method? " +
ls.isEmpty());
//Is list empty after calling clear method? true
```

Difference between ListIterator and Iterator

1. We can use Iterator to traverse Set and List and also Map type of Objects. But List Iterator can be used to traverse for List type Objects, but not for Set type of Objects.
2. By using Iterator we can retrieve the elements from Collection Object in forward direction only whereas List Iterator, which allows you to traverse in either directions using hasPrevious() and previous() methods.
3. ListIterator allows you modify the list using add() & remove() methods. Using Iterator you can't add, only remove the elements.

The java.util.Queue<T> interface

It is a subinterface of Collection that is used to hold elements prior to processing. Queue typically inserts elements FIFO manner (but not necessary), for priority queue order is decided according to order specified by comparator. Whatever ordering is used elements that is at head must be removed first, and elements are inserted into tail position in FIFO ordering it has following methods

Method	Description
boolean add(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available
E element()	Retrieves, but does not remove, the head of this queue throw exception if queue is empty.
E remove()	Retrieves and removes the head of this queue. If queue is empty then throw an exception
boolean offer(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty.

The java.util.Deque<T> interface

- The name deque is short for "double ended queue" and is usually pronounced "deck".
- It is a linear collection that supports element insertion and removal at both ends. This interface defines methods to access the elements at both ends of the deque.
- Apart from the methods of Queue and Collection interface we have following methods also.

Method	Description
void addFirst(E e)	Inserts the specified element at the front of this deque, throwing an IllegalStateException if no space is currently available.
void addLast(E e)	Inserts the specified element at the rear of this deque, throwing an IllegalStateException if no space is currently available.
E getFirst()	Retrieves, but does not remove, the first element of this deque. throws an exception if this deque is empty.
E getLast()	Retrieves, but does not remove, the last element of this deque. throws an exception if this deque is empty.
E removeFirst()	Retrieves and removes the first element of this deque, throws an exception if this deque is empty.
E removeLast()	Retrieves and removes the last element of this deque, throws an exception if this deque is empty.
boolean offerFirst(E e)	Inserts the specified element at the front of this deque, return false is no space is available

boolean offerLast(E e)	Inserts the specified element at the rear of this deque, return false is no space is available
E peekFirst()	Retrieves, but does not remove, the first element of this deque. throws null if this deque is empty.
E peekLast()	Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty.
E pollFirst()	Retrieves and removes the first element of this deque, return null if deque is empty.
E pollLast()	Retrieves and removes the last element of this deque, returns null if deque is empty.
Iterator<E> descendingIterator()	Returns an iterator over the elements in this deque in reverse sequential order. The elements will be returned in order from last (tail) to first (head).

The java.util.LinkedList<T> class

Doubly-linked list implementation of the List and Deque interfaces.

Difference between LinkedList and and ArrayList

1. LinkedList store elements within a doubly-linked list data structure. ArrayList store elements within a dynamically resizing array.
2. LinkedList allows for constant-time insertions or removals, but only sequential access of elements. In other words, you can walk the list forwards or backwards, but grabbing an element in the middle takes time proportional to the size of the list. ArrayLists, on the other hand, allow random access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap.
3. LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.
4. In a LinkedList, Finding the point of insertion/deletion is O(n) but Performing the insertion/deletion is O(1) and In Array List, Finding the point of insertion/deletion is O(1) but Performing the insertion/deletion is O(n)

An Example

```
//filename: Customer.java
package com.masai.sprint4;

public class Customer {
  private String customerId;
  private String customerName;
  private double orderAmount;

  public Customer(String customerId, String customerName, double
orderAmount) {
    this.customerId = customerId;
    this.customerName = customerName;
    this.orderAmount = orderAmount;
  }

  @Override
```

```
  public String toString() {
    return "Customer [customerId=" + customerId + ", customerName=" +
customerName + ", orderAmount=" + orderAmount
        + "]";
  }
}

//filename: LinkedListDemo.java
package com.masai.sprint4;

import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;

public class LinkedListDemo {
  public static void main(String[] args) {
    List<Customer> customerList = new LinkedList<>();
    Scanner sc = new Scanner(System.in);

    //code to take input details about the Customer details
    char addAnotherCustomer = 'y';
    do {
      System.out.print("Enter customer id, name and order amount ");
      String customerId = sc.next();
      String customerName = sc.next();
      double orderAmount = sc.nextDouble();
      Customer customer = new Customer(customerId, customerName,
orderAmount);
      customerList.add(customer);
      System.out.print("Do you want to add another customer (y/n) ");
      addAnotherCustomer = sc.next().charAt(0);
    }while(addAnotherCustomer == 'y');

    //convert the linked list to array
    Object customerArr[] = customerList.toArray();
    for(Object customer: customerArr)
      System.out.println(customer);

    sc.close();
  }
}
```

Output

```
Enter customer id, name and order amount C001 Aman 2541
Do you want to add another customer (y/n) y
Enter customer id, name and order amount C002 Bajrag 5858
Do you want to add another customer (y/n) y
Enter customer id, name and order amount C003 Cindrella 14526
Do you want to add another customer (y/n) n
Customer [customerId=C001, customerName=Aman, orderAmount=2541.0]
Customer [customerId=C002, customerName=Bajrag, orderAmount=5858.0]
Customer [customerId=C003, customerName=Cindrella, orderAmount=14526.0]
```

The java.util.Vector<T> class

- This class is also one of the implementation classes of List interface.
- This class is also same as the ArrayList class with following differences:

- The methods of ArrayList class are not synchronized, whereas most of the methods of the Vector class is synchronized. More on synchronized in the multithreading chapter.

The java.util.Stack<T> class

The stack is the subclass of the Vector class. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of the Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

An Example:

```
package com.masai.sprint4;
import java.util.Stack;

public class Main {
  public static void main(String args[]) {
    Stack<String> stack = new Stack<String>();
    stack.push("A");
    stack.push("B");
    stack.push("C");
    stack.push("D");
    stack.push("E");
    stack.peek();//return first object or throws exception if empty.
    stack.pop(); //remove the last element

    for(String s:stack){
      System.out.print(s + " ");
    }
  }
}
```

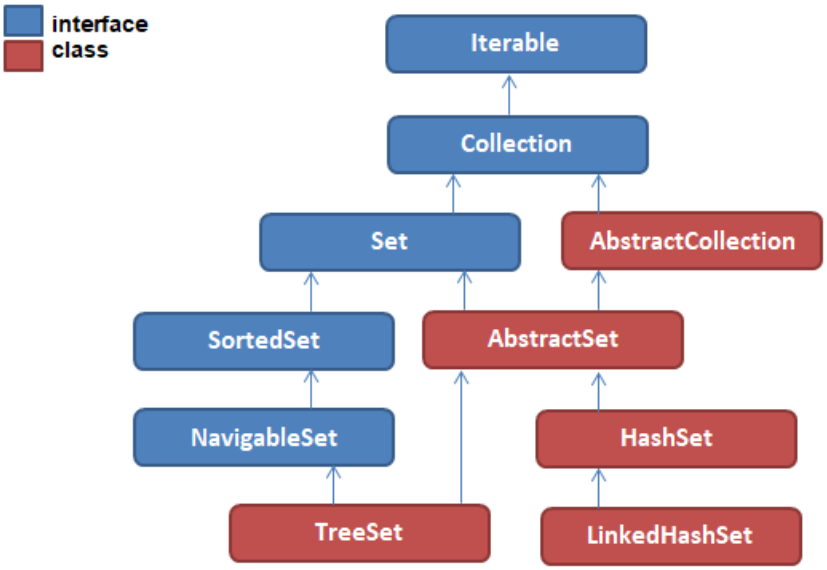
Output:

```
A B C D
```

The java.util.Set<T> Interface

- It is a subinterface of Collection that cannot contain duplicate elements.
- Only one null can be added yet some implementation like java.util.TreeSet<T> prohibits null value.
- The Set interface has implementation that are ordered as well as unordered both.
- It has all methods of Collection interface but no method of its own.

Hierarchy of Set interface and implementing classes



The java.util.HashSet<T> class

- It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.
- This class permits the null element.
- It stores information using mechanism called **hashing.** In that mechanism content of object is used to determine a unique that is known as hash code. Hash value is used as an index at which data associated with key is stored. Transformation of object content into the hash code is done automatically.

The HashSet has following constructors

Constructor	**Description**
HashSet()	Constructs a new, empty set; the backing HashMap instance has default initial capacity 16 and load factor 0.75
HashSet(Collection<? extends E> c)	Constructs a new set containing the elements in the specified collection.
HashSet(int initialCapacity)	Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and default load factor (0.75).
HashSet(int initialCapacity, float loadFactor)	Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor.

An Example

```
package com.masai.sprint4;
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo {
  public static void main(String[] args) {
    Set<String> hs = new HashSet<>();
    hs.add("W");
    hs.add("E");
    hs.add("Q");
    hs.add("D");
    hs.add("B");
    hs.add("P");

    System.out.println(hs);
    System.out.println(hs.size());
  }
}

Output
[P, Q, B, D, E, W]
6
```

To understand the internal storage mechanism of elements in HashSet, we have to understand HashMap first so this discussion will be done later.

The java.util.LinkedHashSet<T> class

- LinkedHashSet is subclass of HashSet class But it does not add any members of its own. LinkedHashSet class maintains elements in the order of their insertion.

- Hash table and linked list implementation of the Set interface, with predictable iteration order.

Just make one change in the above example

```
Set<String> hs = new LinkedHashSet<>();
```

```
Output
[W, E, Q, D, B, P]
6
```