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

Unit 4

Lecture 29

Lecture 29

- Iterator Interface
- Collection Interface
- List Interface
- ArrayList
- LinkedList

Iterator Interface in Java

The Iterator interface in Java is a part of the Java Collections Framework, and it is used to traverse the elements in a collection. It provides a way to access and manipulate the elements of a collection in a sequential manner, without exposing the underlying implementation details of the collection. The Iterator is considered a universal iterator as it can be applied to any Collection object .

Key Features

It can traverse only in the forward direction .

Both read and remove operations can be performed.

It was included in Java JDK 1.2 .

Declaration

public interface Iterator<E>

Methods of Iterator

The Iterator interface provides 3 methods that can be used to perform various operations on elements of collections.

hasNext(): This method returns a boolean value indicating whether there are more elements to traverse in the collection.

next(): This method returns the next element in the collection's iteration. It throws a NoSuchElementException if there are no more elements to iterate over.

remove() (optional): This method removes the last element returned by the next() method from the underlying collection.

Collection Interface in Java

The Collection interface is defined in the java.util package, and it extends the Iterable interface. This means that all collections in Java are iterable, allowing them to be used in enhanced for-loops and with iterators. The Collection interface does not implement any methods directly; instead, it defines several methods that must be implemented by concrete classes that implement this interface.

Collection Interface in Java

These methods are:

add(E element): Adds the specified element to the collection.

addAll(Collection<? extends E> c): Adds all the elements from the specified collection to this collection.

clear(): Removes all elements from the collection.

contains(Object o): Returns true if the collection contains the specified element.

containsAll(Collection<?> c): Returns true if the collection contains all the elements in the specified collection.

Collection Interface in Java

equals(Object o): Compares the specified object with this collection for equality.

hashCode(): Returns the hash code value for this collection.

isEmpty(): Returns true if the collection is empty.

iterator(): Returns an iterator over the elements in this collection.

remove(Object o): Removes a single instance of the specified element from the collection.

Collection Interface in Java

removeAll(Collection<?> c): Removes from the collection all its elements that are also contained in the specified collection.

retainAll(Collection<?> c): Retains only the elements in this collection that are also contained in the specified collection.

size(): Returns the number of elements in the collection.

toArray(): Returns an array containing all the elements in this collection.

toArray(T[] a): Returns an array containing all the elements in this collection, using the specified array if it is big enough.

List Interface

- List in Java provides the facility to maintain the ordered collection.
- It contains the index-based methods to insert, update, delete and search the elements.
- It can have the duplicate elements also.
- We can also store the null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface.
- Through the ListIterator, we can iterate the list in forward and backward directions.
- The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector.

Example

```
public class CollectionsDemo {
    public static void main(String[] args)  {
        // ArrayList
        List<String> list = new ArrayList<>();
        list.add("Welcome");
        list.add("to");
        list.add("Java");
        System.out.println("The list is: " + list);
        Iterator<String> itr = list.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next()); }    }  }
```

In this example, we create an ArrayList and add elements to it. We then create an Iterator object and use it to iterate over the ArrayList. The hasNext() method checks if there are more elements, and the next() method retrieves the next element

ArrayList

An ArrayList in Java is a dynamic array that can grow or shrink in size dynamically. It's part of the Java Collections Framework and is present in the java.util package.

Some key points about ArrayList are:

Dynamic Sizing: Unlike arrays, ArrayList can dynamically resize itself.

Ordered Collection: ArrayList maintains the insertion order of elements.

Allows Duplicates: ArrayList can contain duplicate elements.

Random Access: You can access any element in ArrayList using its index, similar to arrays.

Not Synchronized: ArrayList is not synchronized, so it's not thread-safe by default.

Methods: ArrayList provides various methods like add(), get(), set(), remove(), size(), clear(), contains(), etc., for managing elements.

Declaration (Syntax):

Data structure declaration and initialization

DataStructureType<DataType> dataStructureVariable = new DataStructureType<>();

Manipulating data structure

dataStructureVariable.methodName();

Example:

// ArrayList declaration and initialization to store integers

ArrayList<Integer> arrayList = new ArrayList<>();

// Adding elements to ArrayList

arrayList.add(10);

Java ArrayList

- Java ArrayList class uses a dynamic array for storing the elements.
- It is like an array, but there is no size limit.
- We can add or remove elements anytime.
- So, it is much more flexible than the traditional array.
- It is found in the java.util package.

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
- We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

```
//Creating a List of type String using ArrayList
List<String> list=new ArrayList<String>();
//Creating a List of type Integer using ArrayList
List<Integer> list=new ArrayList<Integer>();
//Creating a List of type Book using ArrayList
List<Book> list=new ArrayList<Book>();
//Creating a List of type String using LinkedList
List<String> list=new LinkedList<String>();
```

The ArrayList and LinkedList classes provide the implementation of List interface.

```
import java.util.*;
public class ListExample1{
public static void main(String args[]){
 //Creating a List
 List<String> list=new ArrayList<String>();
 //Adding elements in the List
 list.add("Mango");
 list.add("Apple");
 list.add("Banana");
 list.add("Grapes");
 //Iterating the List element using for-each loop
 for(String fruit:list)
  System.out.println(fruit);
 } }
```

Get and Set Element in List

The *get() method* returns the element at the given index, whereas the *set() method* changes or replaces the element.

```
//accessing the element
 System.out.println("Returning element: "+list.get(1));
//it will return the 2nd element, because index starts
from 0

 //changing the element
 list.set(1,"Dates");
```

How to Sort List

```
//Creating a list of fruits
  List<String> list1=new ArrayList<String>();
  list1.add("Mango");
  list1.add("Apple");
  list1.add("Banana");
  list1.add("Grapes");
  //Sorting the list
  Collections.sort(list1);
```

Iterating ArrayList using Iterator

```
import java.util.*;
public class ArrayListExample2{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();
  list.add("Mango");
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
    Iterator itr=list.iterator();//getting the Iterator
  while(itr.hasNext()){//check if iterator has the elements
   System.out.println(itr.next());//printing the element and move to next
  } } }
```

LinkedList class

Java LinkedList class uses a doubly linked list to store the elements.

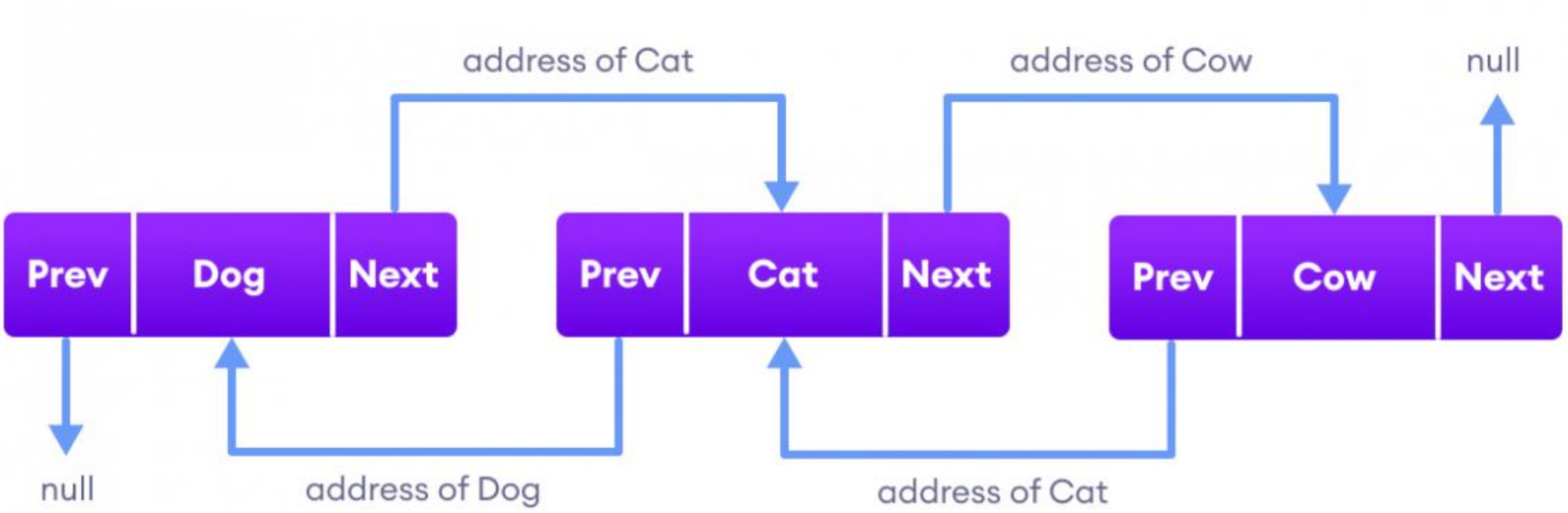
It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

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).



Methods of Java LinkedList

LinkedList provides various methods that allow us to perform different operations in linked lists. We will look at four commonly used LinkedList Operators in this tutorial:

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

LinkedList

The main advantages of using a LinkedList over an ArrayList are:

Efficient Insertion and Removal: Inserting and removing elements at the beginning or end of a LinkedList is highly efficient with a time complexity of O(1). This is because it only requires updating the references of the neighboring nodes.

Dynamic Size: Unlike arrays, LinkedList does not have a fixed size, and it can grow or shrink dynamically as elements are added or removed.

No Shift Required: When inserting or removing elements in the middle of a LinkedList, there is no need to shift the remaining elements, as they are simply linked through their references.

LinkedList

However, LinkedList has a few drawbacks compared to ArrayList:

Random Access: Accessing elements by index in a LinkedList is relatively slow, with a time complexity of O(n), as it requires traversing the list from the beginning or end.

Memory Overhead: Each node in a LinkedList requires additional memory to store the references to the next and previous nodes, resulting in higher memory overhead compared to an ArrayList.

Methods of Java LinkedList

Method	Description
boolean add(E e)	It is used to append the specified element to the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void addFirst(E e)	It is used to insert the given element at the beginning of a list.
void addLast(E e)	It is used to append the given element to the end of a list.
void clear()	It is used to remove all the elements from a list.
void addFirst(E e)	It is used to insert the given element at the beginning of a list.
Iterator<E> descendingIterator()	It is used to return an iterator over the elements in a deque in reverse sequential order.

Method	Description
E get(int index)	It is used to return the element at the specified position in a list.
E getFirst()	It is used to return the first element in a list.
E getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.
E remove()	It is used to retrieve and removes the first element of a list.
E remove(int index)	It is used to remove the element at the specified position in a list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element in a list.

LinkedList Example

```
import java.util.*;
public class LinkedList1{
 public static void main(String args[]){
  LinkedList<String> al=new LinkedList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

LinkedList Example to reverse a list of elements

```
import java.util.*;
public class LinkedList4{
 public static void main(String args[]){
  LinkedList<String> ll=new LinkedList<String>();
        ll.add("Ravi");
        ll.add("Vijay");
        ll.add("Ajay");
        //Traversing the list of elements in reverse order
        Iterator i=ll.descendingIterator();
        while(i.hasNext())
        {
           System.out.println(i.next());
        }
  }
}
```

Difference Between ArrayList and LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.