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

Unit 4

Lecture 28

Lecture 28

- Collection in Java
- Collection Framework in Java
- Hierarchy of Collection Framework

Collection in Java

- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that we perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects.
- Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Collections in Java

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

In Java, collections are part of the Java Collections Framework (JCF), which is a unified architecture for representing and manipulating groups of objects. The JCF provides several interfaces and classes that implement different types of collections, such as lists, sets, queues, and maps.

Why use collections instead of arrays?

Arrays in Java have several limitations:

Fixed Size: Once an array is created, its size cannot be changed. Collections, however, can grow or shrink dynamically as needed.

Type Safety: Collections can enforce type safety using generics, reducing runtime errors related to type casting.

Utility Methods: Collections provide various utility methods for common operations like searching, sorting, and manipulating data.

Performance: Collections are optimized for performance with various implementations for different use cases (e.g., fast random access, quick insertion/deletion, etc.).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

In Java, A Collection is a fundamental concept and a root interface in the Java Collections Framework (JCF). It represents a group of objects (known as elements) in a single unit including Interfaces and its implementations, i.e., classes and Algorithm.

The Collection interface is part of the java.util package and serves as the foundation for various data structures and algorithms that operate on collections of objects.

Benefits of using JCF

Reusability: Provides reusable data structures and algorithms.

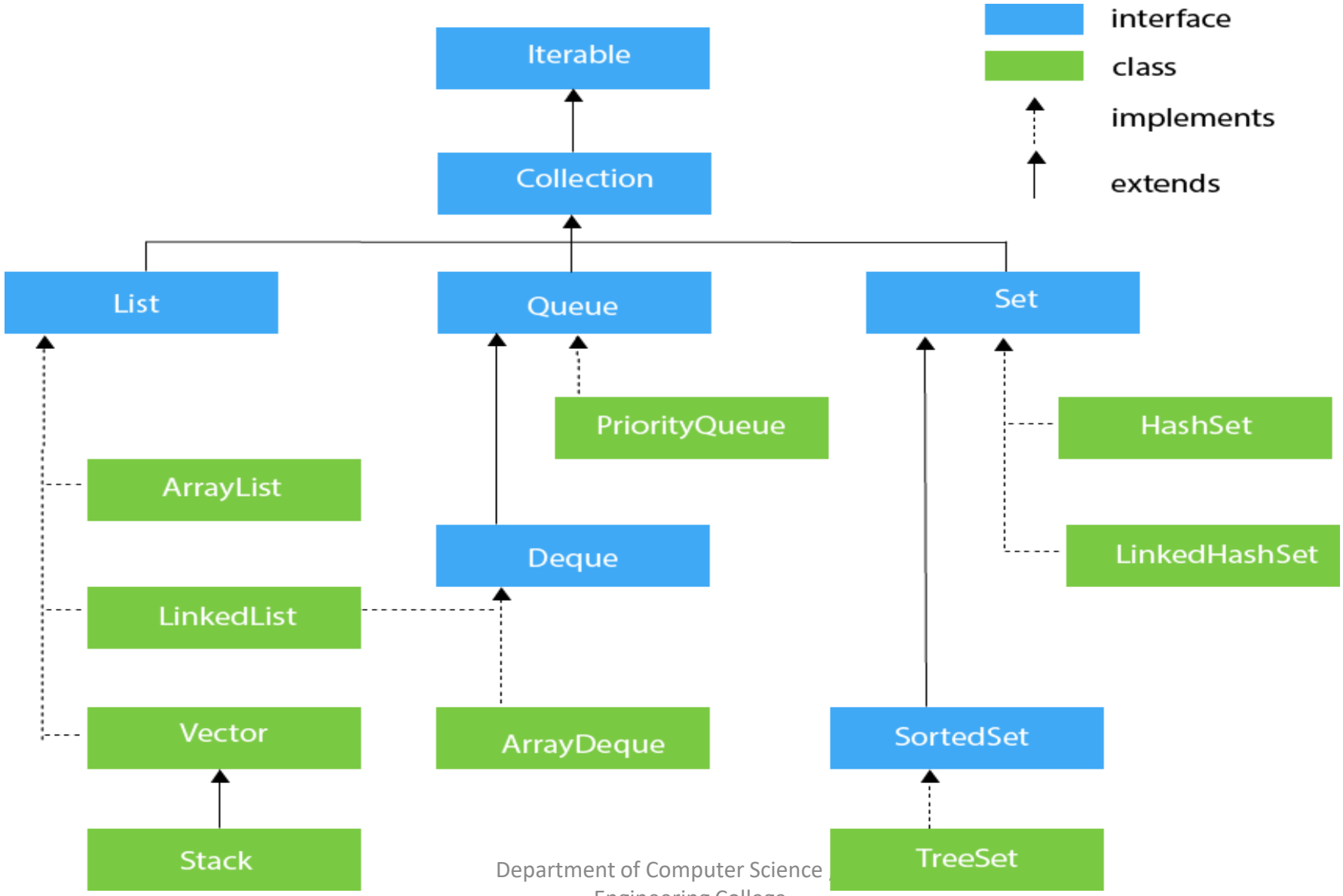
Interoperability: Allows different types of collections to work together seamlessly.

Performance: Offers various implementations optimized for different performance needs.

Flexibility: Supports dynamic resizing and allows collections to grow and shrink as needed.

Type Safety: Uses generics to ensure type safety at compile time.

Hierarchy of the Collection Framework in Java



Hierarchy of Collection Framework

- This image represents the hierarchy of interfaces and classes in the Java Collections Framework. It illustrates the relationships between different collection types and their implementations.
- At the top, we have the Iterable interface, which provides a way to iterate over elements in a collection. It is the root interface for all collection types.
- Moving down, we have the Collection interface, which is a child of Iterable and represents a group of objects. It defines the common methods and behaviors for all collection types.

Hierarchy of Collection Framework

Underneath Collection, we have three main interfaces:

List: An ordered collection that allows duplicate elements. Its implementations are ArrayList, LinkedList, Vector, and Stack.

Queue: A collection designed for holding elements prior to processing. Its implementations are PriorityQueue, Deque, and ArrayDeque.

Set: An unordered collection that does not allow duplicate elements. Its implementations are HashSet, LinkedHashSet, SortedSet, and TreeSet.

Hierarchy of Collection Framework

Each of these interfaces has its own concrete implementations, which are represented by the colored boxes in the image.

ArrayList, LinkedList, Vector, and Stack implement the List interface.

PriorityQueue, Deque, and ArrayDeque implement the Queue interface.

HashSet, LinkedHashSet, SortedSet, and TreeSet implement the Set interface.

These implementations provide different characteristics and performance trade-offs depending on the use case and requirements of the application.

Hierarchy of the Collection Framework in Java

- The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework.
- The collection framework contains an interface named an iterable interface which provides the iterator to iterate through all the collections.
- This interface is extended by the main collection interface which acts as a root for the collection framework.
- All the collections extend this collection interface thereby extending the properties of the iterator and the methods of this interface.

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes.

The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

Iterator<T> iterator()

It returns the iterator over the elements of type T.

Collection Interface

- The Collection interface is the interface which is implemented by all the classes in the collection framework.
- It declares the methods that every collection will have.

Methods of Collection interface

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.

No.	Method	Description
5	public int size()	It returns the total number of elements in the collection.
6	public void clear()	It removes the total number of elements from the collection.
7	public boolean contains(Object element)	It is used to search an element.
8	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.

List Interface

- This is a child interface of the collection interface.
- This interface is dedicated to the data of the list type in which we can store all the ordered collections of the objects.
- This also allows duplicate data to be present in it.
- This list interface is implemented by various classes like ArrayList, Vector, Stack, etc.

- List <data-type> list1= **new** ArrayList();
- List <data-type> list2 = **new** LinkedList();
- List <data-type> list3 = **new** Vector();
- List <data-type> list4 = **new** Stack();

Queue Interface

- Queue interface maintains the first-in-first-out order.
- It can be defined as an ordered list that is used to hold the elements which are about to be processed.
- There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

Set Interface

- Set Interface in Java is present in java.util package. It extends the Collection interface.
- It represents the unordered set of elements which doesn't allow us to store the duplicate items.
- We can store at most one null value in Set.
- Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

SortedSet Interface

- SortedSet is the alternate of Set interface that provides a total ordering on its elements.
- The elements of the SortedSet are arranged in the increasing (ascending) order.
- The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

SortedSet<data-type> set = new TreeSet();

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.

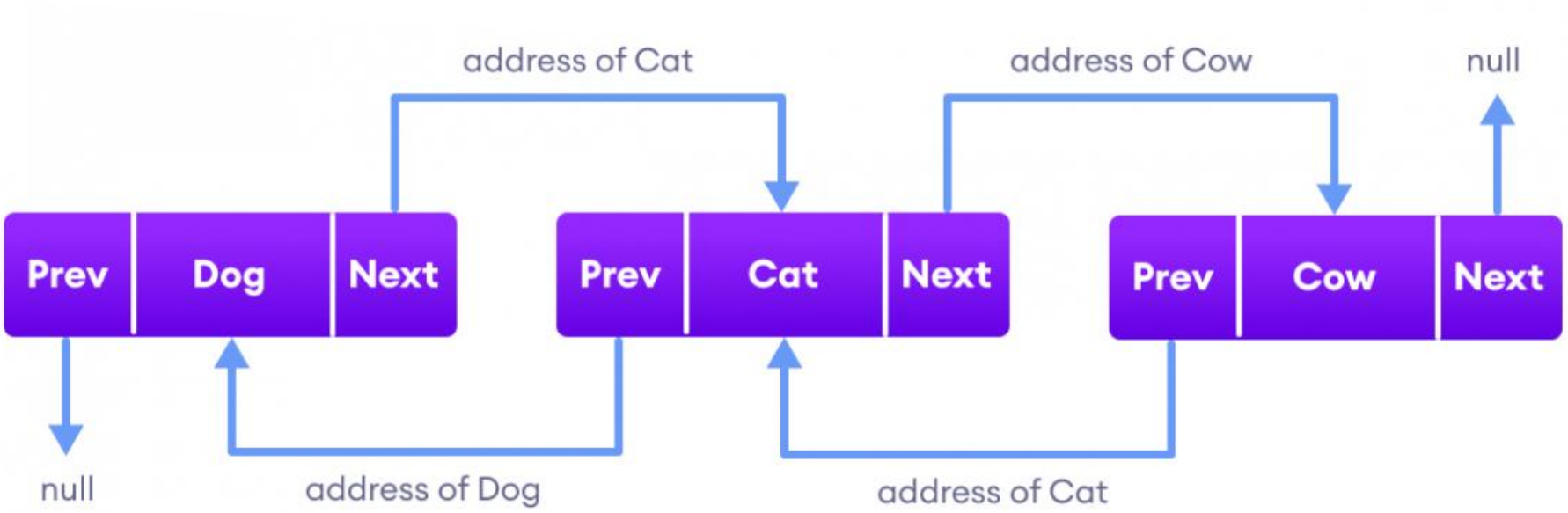
It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

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.

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.

ABES Engineering College
College Code 032

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.

ABES Engineering College
College Code 032

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.

```
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);
}
```

```
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

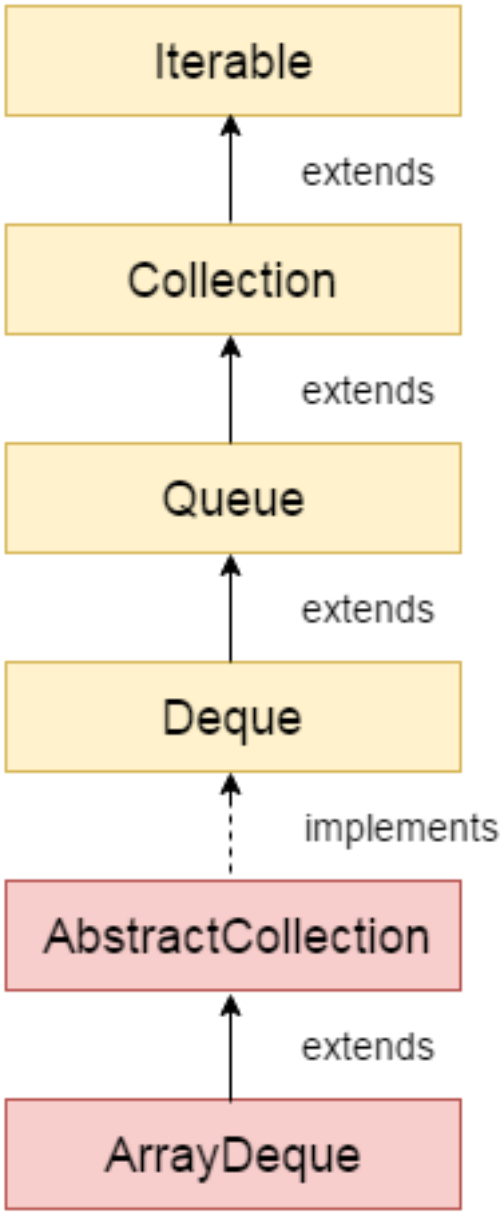
```
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        // Adding elements to the queue
        queue.add("Alice");
        queue.add("Bob");
        queue.add("Charlie");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

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.



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);
  }
  }
}
```

ABES Engineering College
College Code 032

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
```

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

Unit 4

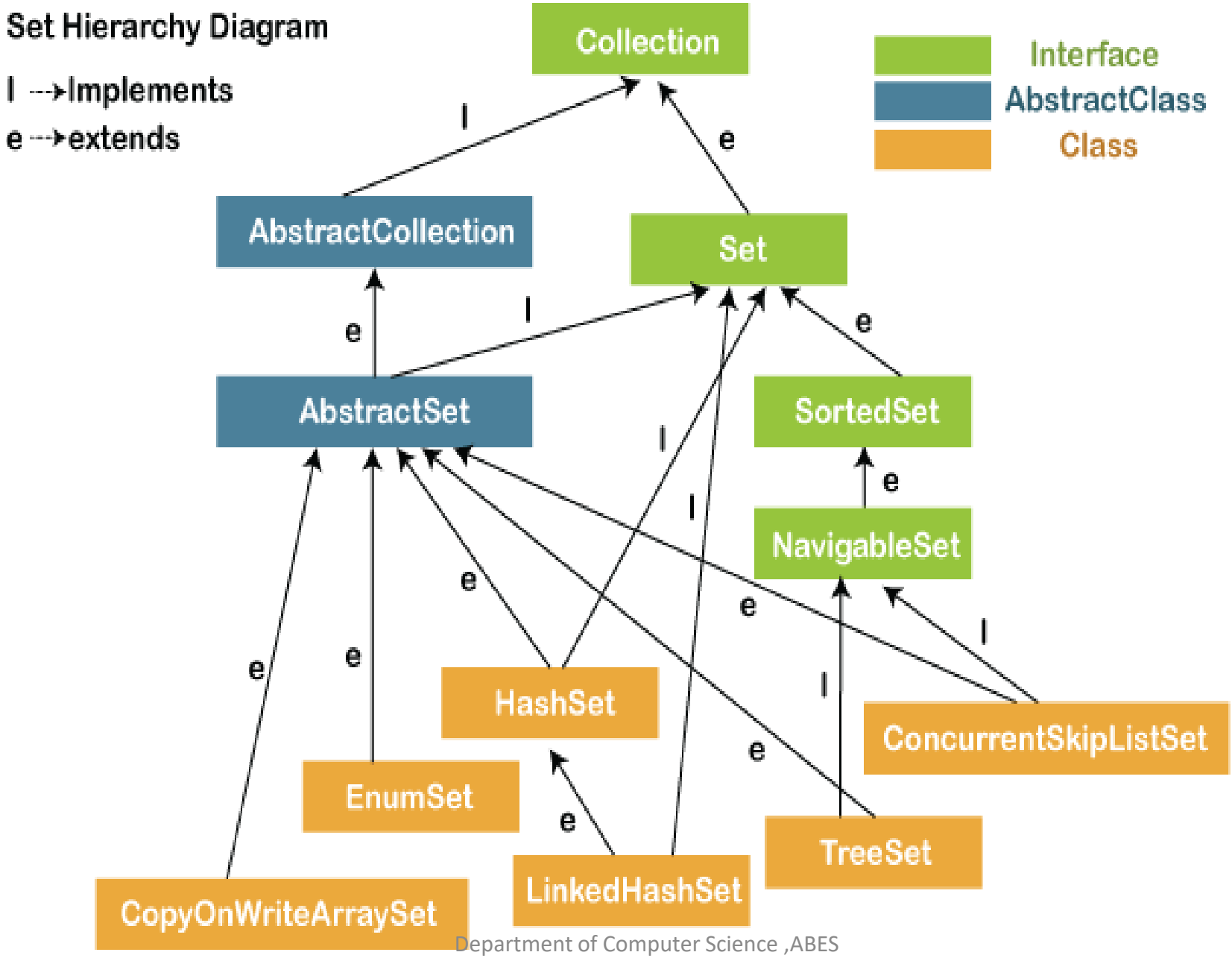
Lecture 31

Lecture 31

- **Set Interface**
- **HashSet**
- **LinkedHashSet**

Set Interface

- The set is an interface available in the java.util package.
- The set interface extends the Collection interface. An unordered collection or list in which duplicates are not allowed is referred to as a collection interface.
- The set interface is used to create the mathematical set.
- The set interface use collection interface's methods to avoid the insertion of the same elements. SortedSet and NavigableSet are two interfaces that extend the set implementation.


Set Hierarchy Diagram
I --▸Implements
e --▸extends
Collection
Interface
AbstractClass
Class
AbstractCollection
Set
AbstractSet
SortedSet
NavigableSet
HashSet
ConcurrentSkipListSet
EnumSet
TreeSet
LinkedHashSet
CopyOnWriteArraySet

Example

```
import java.util.*;
public class setExample{
   public static void main(String[] args)
   {
      // creating LinkedHashSet using the Set
Set<String> data = new LinkedHashSet<String>();
          data.add("Java");
      data.add("Set");
      data.add("Example");
      data.add("Set");
      System.out.println(data);
   }
}
```

Operations on the Set Interface

- On the Set, we can perform all the basic mathematical operations like intersection, union and difference.
- two sets, i.e.,

set1 = [22, 45, 33, 66, 55, 34, 77]

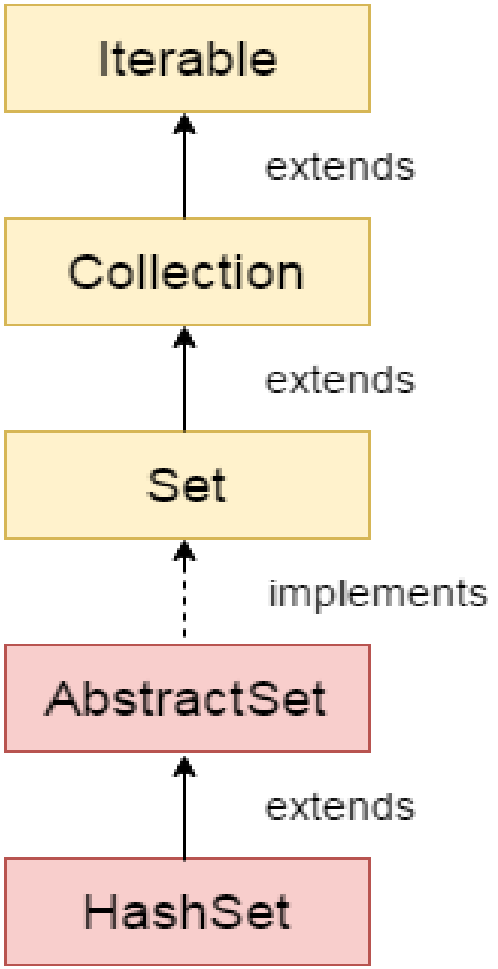
set2 = [33, 2, 83, 45, 3, 12, 55]

Intersection: The intersection operation returns all those elements which are present in both the set. The intersection of set1 and set2 will be [33, 45, 55].

- **Union:** The union operation returns all the elements of set1 and set2 in a single set, and that set can either be set1 or set2. The union of set1 and set2 will be [2, 3, 12, 22, 33, 34, 45, 55, 66, 77, 83].
- **Difference:** The difference operation deletes the values from the set which are present in another set. The difference of the set1 and set2 will be [66, 34, 22, 77].
- In set, addAll() method is used to perform the union, retainAll() method is used to perform the intersection and removeAll() method is used to perform difference.

Java HashSet

- Java HashSet class is used to create a collection that uses a hash table for storage.
- It inherits the Abstract Set class and implements Set interface.



important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called hashing.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16.

Constructors of Java HashSet class

1) HashSet() It is used to construct a default HashSet.
2) HashSet(int capacity) It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3) HashSet(int capacity, float loadFactor) It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.

Methods of Java HashSet class

add(E e)	It is used to add the specified element to this set if it is not already present.
clear()	It is used to remove all of the elements from the set.
contains(Object o)	It is used to return true if this set contains the specified element.
isEmpty()	It is used to return true if this set contains no elements.
iterator()	It is used to return an iterator over the elements in this set.
remove(Object o)	It is used to remove the specified element from this set if it is present.
size()	It is used to return the number of elements in the set.

HashSet Example

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
    HashSet<String> set=new HashSet();
          set.add("One");
          set.add("Two");
          set.add("Three");
          set.add("Four");
          set.add("Five");
          Iterator<String> i=set.iterator();
          while(i.hasNext())
          {
          System.out.println(i.next());
          }
 } }
```

HashSet example ignoring duplicate elements

```
HashSet<String> set=new HashSet<String>();
 set.add("Ravi");
 set.add("Vijay");
 set.add("Ravi");
 set.add("Ajay");
 //Traversing elements
 Iterator<String> itr=set.iterator();
 while(itr.hasNext()){
  System.out.println(itr.next());
 }
```

Ajay

Vijay

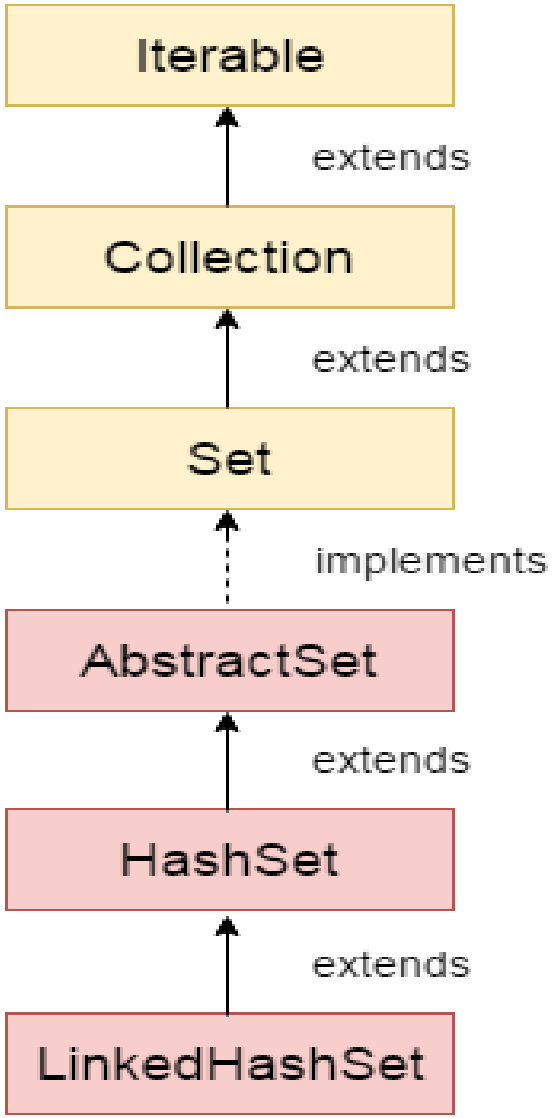
Ravi

HashSet example to remove elements

```
 set.remove("Ravi");
 //Removing all the elements from HashSet
  HashSet<String> set1=new HashSet<String>();
      set1.add("Ajay");
      set1.add("Gaurav");
      set.removeAll(set1);
//Removing elements on the basis of specified condition
      set.removeIf(str->str.contains("Vijay"));
      //Removing all the elements available in the set
      set.clear();
```

LinkedHashSet Class

- Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface.
- It inherits the HashSet class and implements the Set interface.



Java LinkedHashSet class are

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operations and permits null elements.
- Java LinkedHashSet class is non-synchronized.
- Java LinkedHashSet class maintains insertion order.

Note: Keeping the insertion order in the LinkedHashset has some additional costs, both in terms of extra memory and extra CPU cycles. Therefore, if it is not required to maintain the insertion order, go for the lighter-weight HashMap or the HashSet instead.

Constructors of Java LinkedHashSet Class

Constructor	Description
LinkedHashSet(int capacity, float fillRatio)	It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument.
LinkedHashSet(int capacity)	It is used to initialize the capacity of the linked hash set to the given integer value capacity.

```
import java.util.*;
class LinkedHashSet1{
 public static void main(String args[]){
 //Creating HashSet and adding elements
       LinkedHashSet<String> set=new LinkedHashSet();
            set.add("One");
            set.add("Two");
            set.add("Three");
            set.add("Four");
            set.add("Five");
            Iterator<String> i=set.iterator();
            while(i.hasNext())
            {
            System.out.println(i.next());
            }
 }
}
```

Output:

```
One
Two
Three
Four
Five
```

```
// Creating an empty LinekdhashSet of string type
LinkedHashSet<String> lhs = new LinkedHashSet<St
ring>();

// Adding elements to the above Set
// by invoking the add() method
lhs.add("Java");
lhs.add("T");
lhs.add("Point");
lhs.add("Good");
lhs.add("Website");
```

```
// displaying all the elements on the console
System.out.println("The hash set is: " + lhs);
 The hash set is: [Java, T, Point, Good, Website]
// Removing an element from the above linked Set
// since the element "Good" is present, therefore, the method remove
()
// returns true
System.out.println(lhs.remove("Good"));
 true
// After removing the element
System.out.println("After removing the element, the hash set is: " + lhs);
After removing the element, the hash set is: [Java, T, Point, Website]
// since the element "For" is not present, therefore, the method remo
ve()
// returns false
System.out.println(lhs.remove("For"));
false
```

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

Unit 4

Lecture 32

Lecture 32

- **SortedSet Interface**
- **TreeSet,**

SortedSet Interface

The SortedSet interface in Java extends the Set interface and provides a set that maintains its elements in ascending order. It is part of the java.util package and is primarily implemented by the TreeSet class.

Ordered Elements: The elements in a SortedSet are maintained in sorted order, either by their natural ordering (if they implement the Comparable interface) or by a Comparator provided at the time of set creation.

No Duplicates: Like other Set implementations, a SortedSet does not allow duplicate elements.

Null Elements: In general, a SortedSet does not allow null elements. However, some implementations like TreeSet allow one null element.

Implementation: The primary implementation of the SortedSet interface is the TreeSet class, which is backed by a self-balancing binary search tree (Red-Black Tree).

Sorting and Searching: Because the elements are stored in sorted order, operations like retrieving the smallest, largest, or n-th element, as well as searching for elements, can be performed efficiently.

Iterators and Subsets: The SortedSet interface provides methods for retrieving iterators that traverse the set in sorted order. It also allows you to retrieve subsets of the set based on a range of elements.

Example

```
import java.util.SortedSet;
import java.util.TreeSet;
 public class SortedSetExample {
public static void main(String[] args) {
// Creating a SortedSet (TreeSet)
 SortedSet<Integer> numbers = new TreeSet<>();
 // Adding elements to the SortedSet
 numbers.add(5);
numbers.add(2);
numbers.add(8);
numbers.add(1);
numbers.add(3);
 System.out.println("SortedSet of numbers: " + numbers);
// Output: SortedSet of numbers: [1, 2, 3, 5, 8]
```

Example

```
// Retrieving the first and last elements
System.out.println("First element: " + numbers.first());
// Output: First element: 1
System.out.println("Last element: " + numbers.last());
 // Output: Last element: 8
 // Retrieving a subset
SortedSet<Integer> subset = numbers.subSet(2, 5);
System.out.println("Subset from 2 to 5: " + subset);
 // Output: Subset from 2 to 5: [2, 3]
// Iterating over the SortedSet
System.out.print("Iterating over the SortedSet: ");
 for (Integer number : numbers)
{ System.out.print(number + " "); }
 System.out.println();
// Output: Iterating over the SortedSet: 1 2 3 5 8 } }
```

In the above example, we create a SortedSet (implemented by TreeSet) of Integer elements. We demonstrate adding elements to the set, retrieving the first and last elements, obtaining a subset based on a range, and iterating over the set.

Notice that the elements are automatically sorted in ascending order.The SortedSet interface is useful when you need to maintain a sorted collection of unique elements. It provides efficient methods for retrieving elements based on their sorted position, as well as for obtaining subsets of the set within a specific range.

One common use case for SortedSet is when you need to perform operations that rely on the sorted order of elements, such as finding the smallest or largest element, or retrieving elements within a specific range.

TreeSet

TreeSet is a concrete implementation of the SortedSet interface in Java. It is a self-balancing binary search tree implementation, specifically a Red-Black tree. Elements in a TreeSet are stored in ascending order according to their natural ordering or a Comparator provided at set creation time.

Ordering: TreeSet stores its elements in sorted ascending order. If the elements are of a class that implements the Comparable interface, they are sorted according to their natural ordering. If not, you can provide a custom Comparator at the time of set creation to define the sorting order.

No Duplicates: Like other Set implementations, TreeSet does not allow duplicate elements. Attempts to add a duplicate element will be ignored.

TreeSet

Null Handling: By default, TreeSet does not allow null elements. If you try to add a null element, it will throw a NullPointerException. However, it is possible to have one null element in a TreeSet by providing a custom Comparator that handles null values appropriately.

Self-Balancing: TreeSet is implemented as a self-balancing binary search tree (Red-Black tree). This means that after every insertion or deletion operation, the tree is rebalanced to maintain its height within reasonable limits, ensuring logarithmic time complexity for operations like add(), remove(), and contains().

TreeSet

Iterators and Subsets: TreeSet provides methods for retrieving iterators that traverse the set in sorted order. It also allows you to retrieve subsets of the set based on a range of elements.

Performance: The time complexity for basic operations like add(), remove(), and contains() is O(log n) on average, where n is the number of elements in the set. This makes TreeSet efficient for large datasets.

Example

```
import java.util.TreeSet;
 public class TreeSetExample {
public static void main(String[] args) {
// Creating a TreeSet
TreeSet<Integer> numbers = new TreeSet<>();
// Adding elements to the TreeSet numbers.add(5);
numbers.add(2);
 numbers.add(8);
numbers.add(1);
numbers.add(3); System.out.println("TreeSet of numbers: " +
numbers);
 // Output: TreeSet of numbers: [1, 2, 3, 5, 8]
```

Example

```
// Retrieving the first and last elements
System.out.println("First element: " + numbers.first());
 // Output: First element: 1
 System.out.println("Last element: " + numbers.last());
// Output: Last element: 8
// Retrieving a subset
TreeSet<Integer> subset = (TreeSet<Integer>) numbers.subSet(2, 5);
 System.out.println("Subset from 2 to 5: " + subset);
// Output: Subset from 2 to 5: [2, 3] //
Iterating over the TreeSet
System.out.print("Iterating over the TreeSet: ");
for (Integer number : numbers) { System.out.print(number + " "); }
 System.out.println(); // Output: Iterating over the TreeSet: 1 2 3 5 8 }
}
```

In this example, we create a TreeSet of Integer elements. We demonstrate adding elements to the set, retrieving the first and last elements, obtaining a subset based on a range, and iterating over the set. Notice that the elements are automatically sorted in ascending order.TreeSet is useful when you need to maintain a sorted collection of unique elements and perform operations that rely on the sorted order of elements, such as finding the smallest or largest element, or retrieving elements within a specific range.

It provides efficient operations due to its self-balancing binary search tree implementation.

However, keep in mind that the insertion, removal, and retrieval operations in a TreeSet are more expensive than those in a HashSet because of the overhead of maintaining the sorted order. If you don't need the elements to be sorted, a HashSet may be a better choice for better performance.

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

Unit 4

Lecture 33

Lecture 33

- **Map Interface**
- **HashMap Class**
- **LinkedHashMap Class**

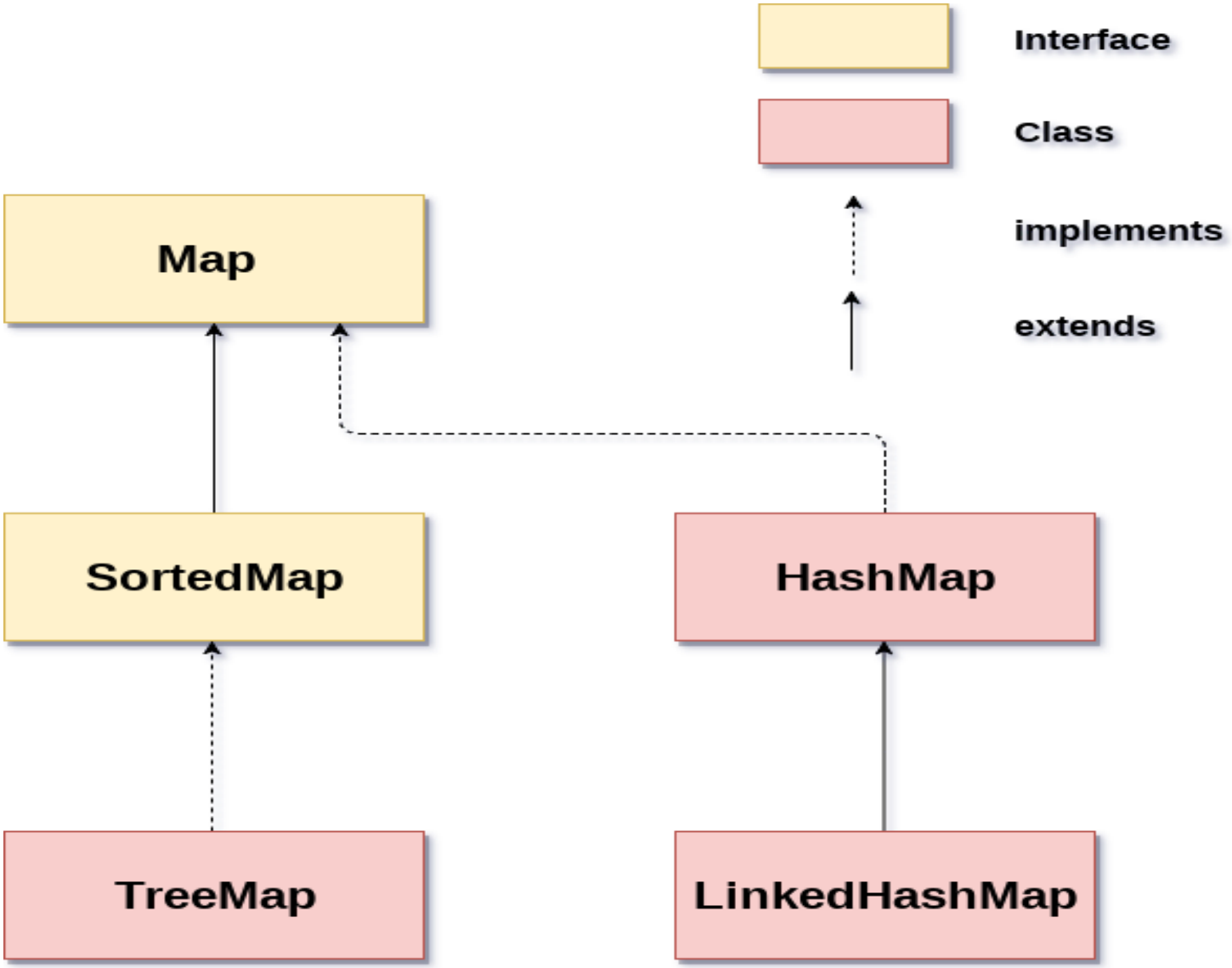
Map Interface

The Map interface in Java is part of the java.util package and represents a collection of key-value pairs. It provides a way to store and retrieve values using keys, similar to a dictionary or an associative array in other programming languages.

Key-Value Pairs: A Map stores data as key-value pairs, where each key is unique and maps to a specific value. Keys are used to access and retrieve the associated values.

Unique Keys: The keys in a Map must be unique. Attempting to insert a duplicate key will replace the value associated with the existing key.

Null Values: A Map can store null values, but the behavior for null keys depends on the specific implementation.


Interface
Class
implements
extends
Map
SortedMap
HashMap
TreeMap
LinkedHashMap

Map Interface

Implementations: The Map interface has several concrete implementations in Java, including HashMap, TreeMap, LinkedHashMap, and ConcurrentHashMap.

Ordering: Some Map implementations, like TreeMap, maintain an ordering of the keys, while others, like HashMap, do not guarantee any specific order.

Common Methods: The Map interface provides various methods for working with key-value pairs, such as put(), get(), containsKey(), containsValue(), remove(), size(), keySet(), values(), and entrySet().

```
import java.util.HashMap;
 import java.util.Map;
public class MapExample {
 public static void main(String[] args) {
// Creating a HashMap
Map<String, Integer> studentAges = new HashMap<>();
// Adding key-value pairs to the Map
studentAges.put("Alice", 20);
studentAges.put("Bob", 22);
 studentAges.put("Charlie", 19);
// Retrieving a value using its key
 int aliceAge = studentAges.get("Alice"); System.out.println("Alice's
age: " + aliceAge); // Output: Alice's age: 20
```

Example

```
// Checking if a key exists
boolean containsBob = studentAges.containsKey("Bob");
System.out.println("Contains 'Bob'? " + containsBob);
 // Output: Contains 'Bob'? true
// Iterating over the key-value pairs
System.out.println("Student Ages:");
for (Map.Entry<String, Integer> entry : studentAges.entrySet()) {
String name = entry.getKey();
int age = entry.getValue();
 System.out.println(name + ": " + age); }
 // Output: // Student Ages: // Alice: 20 // Bob: 22 // Charlie: 19
 }
 }
```

- In this example, we create a HashMap to store student names and their ages. We demonstrate adding key-value pairs to the Map, retrieving a value using its key, checking if a key exists, and iterating over the key-value pairs using the entrySet() method.
- Maps are widely used in scenarios where you need to associate keys with values and efficiently retrieve or manipulate those values based on their keys. Examples include caching systems, dictionaries, symbol tables, and various data structures that require efficient key-based lookups.
- Different Map implementations provide different performance characteristics and features. For example, HashMap offers constant-time performance for most operations but does not maintain any order of the key-value pairs. In contrast, TreeMap maintains the key-value pairs in sorted order based on the keys, but with a slightly higher overhead for certain operations.

Example

```
import java.util.HashMap;
 import java.util.Map;
public class MapExample {
 public static void main(String[] args) {
// Creating a HashMap
Map<String, Integer> studentAges = new
HashMap<>();
// Adding key-value pairs to the Map
studentAges.put("Alice", 20);
studentAges.put("Bob", 22);
studentAges.put("Charlie", 19);
// Retrieving a value using its key
 int aliceAge = studentAges.get("Alice");
System.out.println("Alice's age: " +
aliceAge); // Output: Alice's age: 20
```

```
// Checking if a key exists
boolean containsBob =
studentAges.containsKey("Bob");
System.out.println("Contains 'Bob'? " + containsBob);
 // Output: Contains 'Bob'? true
// Iterating over the key-value pairs
System.out.println("Student Ages:");
for (Map.Entry<String, Integer> entry :
studentAges.entrySet()) { String name =
entry.getKey();
int age = entry.getValue();
 System.out.println(name + ": " + age); }
 // Output: // Student Ages: // Alice: 20 // Bob: 22 //
Charlie: 19 }
 }
```

- In this example, we create a HashMap to store student names and their ages. We demonstrate adding key-value pairs to the Map, retrieving a value using its key, checking if a key exists, and iterating over the key-value pairs using the entrySet() method.
- Maps are widely used in scenarios where you need to associate keys with values and efficiently retrieve or manipulate those values based on their keys. Examples include caching systems, dictionaries, symbol tables, and various data structures that require efficient key-based lookups.
- Different Map implementations provide different performance characteristics and features. For example, HashMap offers constant-time performance for most operations but does not maintain any order of the key-value pairs. In contrast, TreeMap maintains the key-value pairs in sorted order based on the keys, but with a slightly higher overhead for certain operations.

HashMap Class

HashMap is one of the most commonly used implementations of the Map interface in Java. It stores key-value pairs in a hash table data structure, providing constant-time performance for basic operations like get(), put(), and remove() on average.

Hash Table Implementation: HashMap internally uses a hash table to store its key-value pairs. The hash table is an array of buckets, where each bucket can hold one or more key-value pairs.

Hashing: To store and retrieve values efficiently, HashMap uses a hashing function to map keys to specific indices (buckets) in the underlying array. This hashing function is implemented by the hashCode() method of the key objects.

Collision Handling: Since hashing can cause collisions (two or more keys mapping to the same bucket), HashMap handles collisions using separate chaining. Each bucket contains a linked list of key-value pairs that hash to that bucket.

HashMap Class

Null Keys and Values: HashMap allows one null key and any number of null values.

No Ordering: HashMap does not maintain any specific order of the key-value pairs. The order in which elements are returned by the iterator is not guaranteed to be the same as the insertion order.

Performance: Basic operations like get(), put(), and remove() have constant-time performance on average, assuming a good hash function and a properly sized hash table.

However, in the worst case (e.g., when all keys hash to the same bucket), the performance can degrade to linear time.Iterators: HashMap provides iterators to traverse the keys, values, or key-value pairs through the keySet(), values(), and entrySet() methods, respectively. The order of iteration is not guaranteed.

Example

```
import java.util.HashMap;
 import java.util.Map;
public class HashMapExample {
 public static void main(String[] args) {
 // Creating a HashMap
Map<String, Integer> studentAges = new
HashMap<>();
// Adding key-value pairs to the HashMap
studentAges.put("Alice", 20);
studentAges.put("Bob", 22);
studentAges.put("Charlie", 19);
 // Retrieving a value using its key
int aliceAge = studentAges.get("Alice");
System.out.println("Alice's age: " + aliceAge);
// Output: Alice's age: 20
```

```
// Checking if a key exists
boolean containsBob =
studentAges.containsKey("Bob");
System.out.println("Contains 'Bob'? " +
containsBob);
 // Output: Contains 'Bob'? true // Iterating over
the key-value pairs
System.out.println("Student Ages:");
 for (Map.Entry<String, Integer> entry :
studentAges.entrySet()) {
String name = entry.getKey();
int age = entry.getValue();
 System.out.println(name + ": " + age); }
// Output (order may vary): // Student Ages: //
Alice: 20 // Bob: 22 // Charlie: 19
}
```

In this example, we create a HashMap to store student names and their ages. We demonstrate adding key-value pairs to the HashMap, retrieving a value using its key, checking if a key exists, and iterating over the key-value pairs using the entrySet() method.

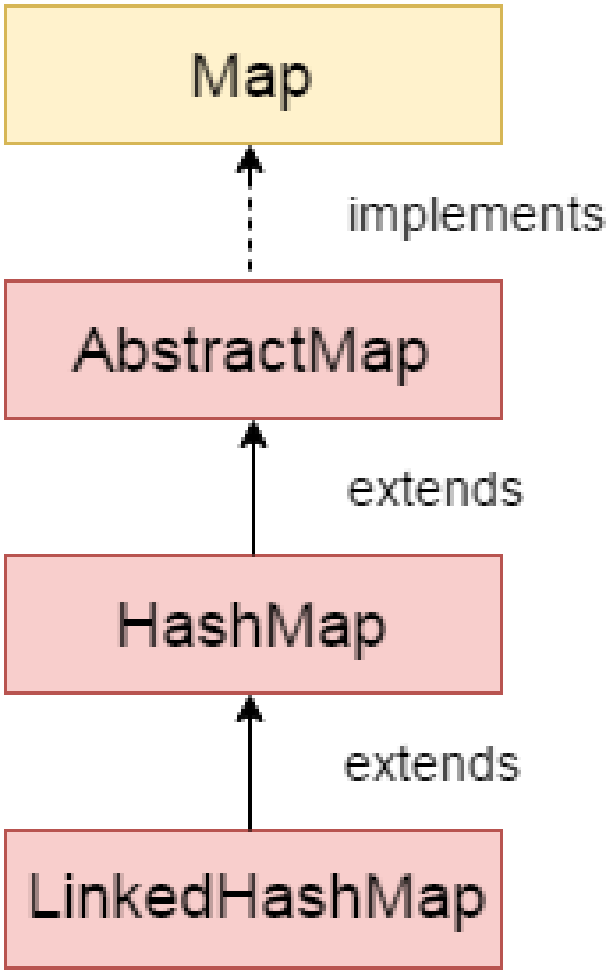
Note that the order of iteration is not guaranteed.

HashMap is widely used when you need to store and retrieve key-value pairs efficiently, and you don't need to maintain any specific order of the elements. It provides constant-time performance for basic operations on average, making it suitable for large datasets.

However, it's important to note that the performance of HashMap depends on the quality of the hash function and the load factor (the ratio of elements to the capacity of the hash table). If the hash function is not well-distributed or the load factor becomes too high, the performance can degrade due to an increase in collisions and the need for resizing the hash table.

LinkedHashMap Class

The LinkedHashMap class in Java is a subclass of the HashMap class and provides the additional functionality of maintaining the insertion order of the elements. It inherits the properties of the HashMap class and additionally maintains a doubly-linked list of the entries, allowing for insertion-order iteration.



- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

LinkedHashMap Class Features

Underlying Data Structure: LinkedHashMap uses a combination of a HashMap and a doubly-linked list to store the key-value pairs. The HashMap provides efficient key-value storage and retrieval, while the doubly-linked list maintains the insertion order of the elements.

Insertion Order: When elements are inserted into a LinkedHashMap, they are added to both the HashMap and the doubly-linked list. The order in which the elements are added to the LinkedHashMap is preserved in the doubly-linked list.

LinkedHashMap Class Features

Access Order: By default, LinkedHashMap maintains the insertion order of the elements. However, it can be configured to maintain the access order (the order in which the elements are accessed) by setting the accessOrder flag to true during the construction of the LinkedHashMap.

Performance: LinkedHashMap has the same time complexity as HashMap for basic operations like get(), put(), and remove(), which is O(1) on average. However, iterating over the elements in the LinkedHashMap takes O(n) time, where n is the number of elements.

```
import java.util.*;
class LinkedHashMap1{
 public static void main(String args[]){
  LinkedHashMap<Integer,String> hm=new
LinkedHashMap<Integer,String>();
   hm.put(100,"Amit");
   hm.put(101,"Vijay");
   hm.put(102,"Rahul");
   for(Map.Entry m:hm.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
 }
}
```

Output:

```
100 Amit
101 Vijay
102 Rahul
```

LinkedHashMap Example: Key-Value pair

```
import java.util.*;
class LinkedHashMap2{
 public static void main(String args[]){
   LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, String>()
    map.put(100,"Amit");
     map.put(101,"Vijay");
     map.put(102,"Rahul");
      //Fetching key
      System.out.println("Keys: "+map.keySet());
      //Fetching value
      System.out.println("Values: "+map.values());
      //Fetching key-value pair
      System.out.println("Key-Value pairs: "+map.entrySet());
 }
}
```

```
Keys: [100, 101, 102]
Values: [Amit, Vijay, Rahul]
Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]
```

LinkedHashMap Example:remove()

```
import java.util.*;
public class LinkedHashMap3 {
  public static void main(String args[]) {
   Map<Integer,String> map=new LinkedHashMap<Integer,String>();
    map.put(101,"Amit");
    map.put(102,"Vijay");
    map.put(103,"Rahul");
    System.out.println("Before invoking remove() method: "+map);
   map.remove(102);
   System.out.println("After invoking remove() method: "+map);
  }
}
```

```
Before invoking remove() method: {101=Amit, 102=Vijay, 103=Rahul}
After invoking remove() method: {101=Amit, 103=Rahul}
```

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

Unit 4

Lecture 34

Lecture 34

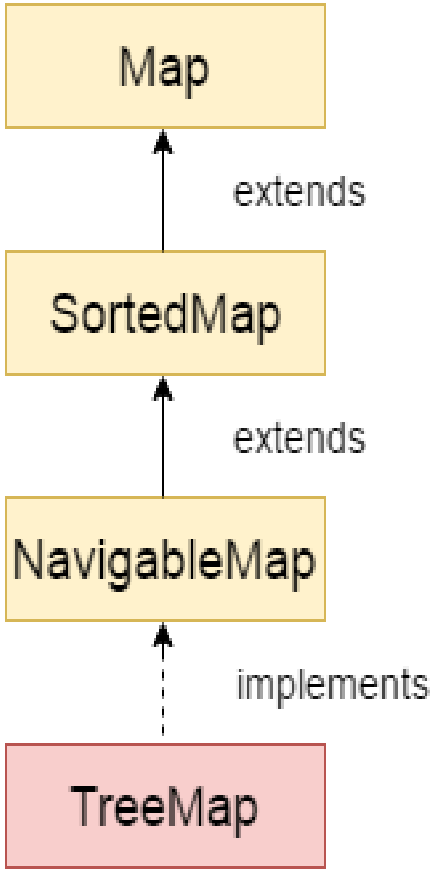
- TreeMap Class
- Hashtable Class

TreeMap Class

The TreeMap class in Java is a red-black tree-based implementation of the Map interface.

It provides an efficient way to store key-value pairs in a sorted order.

The keys in a TreeMap are sorted based on their natural ordering or by a custom Comparator provided during construction.



- Java TreeMap contains values based on the key.
- It implements the NavigableMap interface and extends AbstractMap class.
- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap is non synchronized.
- Java TreeMap maintains ascending order.

TreeMap Class Features

- **Underlying Data Structure:** TreeMap uses a self-balancing Red-Black tree data structure to store the key-value pairs. This data structure ensures that the tree remains balanced, providing logarithmic time complexity for most operations, such as insertion, deletion, and retrieval.
- **Sorting Order:** The keys in a TreeMap are sorted based on their natural ordering (if the keys implement the Comparable interface) or by a custom Comparator provided during construction. This sorting order is maintained automatically by the TreeMap.

TreeMap Class Features

Null Keys and Values: TreeMap allows for one null value, but it does not allow null keys, as keys must be comparable or provided with a Comparator.

Performance: Most operations in TreeMap, such as get(), put(), remove(), and containsKey(), have a time complexity of O(log n), where n is the number of elements in the map. This makes TreeMap an efficient choice when you need to perform frequent operations on sorted data.

Iteration Order: The elements in a TreeMap are traversed in ascending order of the keys during iteration.

Methods of Java TreeMap class

Method	Description
Set keySet()	It returns the collection of keys exist in the map.
V put(K key, V value)	It inserts the specified value with the specified key in the map.
void putAll(Map<? extends K,? extends V> map)	It is used to copy all the key-value pair from one map to another map.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean containsKey(Object key)	It returns true if the map contains a mapping for the specified key.
boolean containsValue(Object value)	It returns true if the map maps one or more keys to the specified value.

Methods of Java TreeMap class

K firstKey()	It is used to return the first (lowest) key currently in this sorted map.
V get(Object key)	It is used to return the value to which the map maps the specified key.
K lastKey()	It is used to return the last (highest) key currently in the sorted map.
V remove(Object key)	It removes the key-value pair of the specified key from the map.
Set<Map.Entry<K,V>> entrySet()	It returns a set view of the mappings contained in the map.
int size()	It returns the number of key-value pairs exists in the hashtable.
Collection values()	It returns a collection view of the values contained in the map.

Example

```
import java.util.*;
class TreeMap1{
 public static void main(String args[]){
   TreeMap<Integer,String> map=new TreeMap<Integer,String>();
     map.put(100,"Amit");
     map.put(102,"Ravi");
     map.put(101,"Vijay");
     map.put(103,"Rahul");

     for(Map.Entry m:map.entrySet()){
      System.out.println(m.getKey()+" "+m.getValue());
     }
 }
}
```

ABES Engineering College
College Code 032

Difference between HashMap and TreeMap

HashMap	TreeMap
1) HashMap can contain one null key.	TreeMap cannot contain any null key.
2) HashMap maintains no order.	TreeMap maintains ascending order.

Hashtable Class

The Hashtable class in Java is a thread-safe implementation of the Map interface.

It stores key-value pairs in a hash table data structure, providing constant-time performance for basic operations such as get() and put() on average.

Important Points

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable Class Features

- **Thread-Safety:** Hashtable is synchronized, which means that all methods are thread-safe and can be safely accessed by multiple threads concurrently. This thread-safety comes at the cost of reduced performance compared to non-synchronized collections like HashMap.
- **Null Keys and Values:** Hashtable does not allow null keys or null values. Attempting to insert a null key or value will result in a NullPointerException.
- **Underlying Data Structure:** Hashtable uses a hash table data structure to store the key-value pairs. It uses an array of buckets, where each bucket holds a linked list of key-value pairs that have the same hash code.

Hashtable Class Features

- **Load Factor and Rehashing:** Hashtable has a default load factor of 0.75, which means that when the number of elements in the Hashtable exceeds 75% of its capacity, it automatically increases its capacity and rehashes all the existing elements.
- **Iteration Order:** Hashtable does not maintain the insertion order of the elements. The elements are traversed in an arbitrary order during iteration.
- **Performance:** The get(), put(), and remove() operations in Hashtable have an average time complexity of O(1) when the hash function distributes the elements properly. However, in the worst-case scenario (when all elements hash to the same bucket), the time complexity degrades to O(n), where n is the number of elements.

Constructors of Java Hashtable class

Constructor	Description
Hashtable()	It creates an empty hashtable having the initial default capacity and load factor.
Hashtable(int capacity)	It accepts an integer parameter and creates a hash table that contains a specified initial capacity.
Hashtable(int capacity, float loadFactor)	It is used to create a hash table having the specified initial capacity and loadFactor.

Methods of Java Hashtable class

V put(K key, V value)	It inserts the specified value with the specified key in the hash table.
void putAll(Map<? extends K,? extends V> t))	It is used to copy all the key-value pair from map to hashtable.
V putIfAbsent(K key, V value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the hashtable.
V replace(K key, V value)	It replaces the specified value for a specified key.

Example 1

```
import java.util.*;
class Hashtable1{
 public static void main(String args[]){
  Hashtable<Integer,String> hm=new Hashtable<Integer,S
tring>();
  hm.put(100,"Amit");
  hm.put(102,"Ravi");
  hm.put(101,"Vijay");
  hm.put(103,"Rahul");
  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

```
103 Rahul
102 Ravi
101 Vijay
100 Amit
```

Example 2

```
import java.util.*;
public class Hashtable2 {
   public static void main(String args[]) {
  Hashtable<Integer,String> map=new Hashtable<Integer,String>();
     map.put(100,"Amit");
     map.put(102,"Ravi");
     map.put(101,"Vijay");
     map.put(103,"Rahul");
     System.out.println("Before remove: "+ map);
      // Remove value for key 102
      map.remove(102);
      System.out.println("After remove: "+ map);
   }
}
```

```
Before remove: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
After remove: {103=Rahul, 101=Vijay, 100=Amit}
```

Example 3

```
import java.util.*;
class Hashtable3{
 public static void main(String args[]){
    Hashtable<Integer,String> map=new Hashtable<Integer,String>();
     map.put(100,"Amit");
     map.put(102,"Ravi");
     map.put(101,"Vijay");
     map.put(103,"Rahul");
  //Here, we specify the if and else statement as arguments of the
method
     System.out.println(map.getOrDefault(101, "Not Found"));
     System.out.println(map.getOrDefault(105, "Not Found"));
 }
}
```

Vijay
Not Found

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

Unit 4

Lecture 35

Lecture 35

- Sorting

Sorting in Java Collection Framework

Sorting in Java's Collection Framework is a crucial aspect of organizing and manipulating data. Java provides various sorting algorithms and utilities to sort collections of elements based on different criteria.

The Java Collection Framework provides several methods for sorting collections, primarily through the Collections and Arrays classes.

These methods utilize Timsort, a highly efficient sorting algorithm.

Sorting Algorithms

Java's Collection Framework supports two main sorting algorithms:

Merge Sort: This is a stable, divide-and-conquer algorithm that sorts elements in O(n log n) time complexity, where n is the number of elements.

Timsort: This is a hybrid, stable sorting algorithm derived from Merge Sort and Insertion Sort. It has a time complexity of O(n log n) for most cases and performs well on partially sorted or nearly sorted collections.

The Collections.sort() method can be used to sort lists. By default, it sorts elements in their natural order (ascending).

Example

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class SortListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("banana");
        list.add("apple");
        list.add("orange");
        Collections.sort(list);
        System.out.println(list); // Output: [apple, banana, orange]
    }
}
```

sort a list using a custom order by providing a Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
public class CustomSortListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("banana");
        list.add("apple");
        list.add("orange");

        // Sort in reverse order
        Collections.sort(list, Comparator.reverseOrder());
        System.out.println(list); // Output: [orange, banana, apple]
    }
}
```

Sorting in reverse order

```
import java.util.Arrays;
import java.util.Comparator;
public class CustomSortArrayExample {
    public static void main(String[] args) {
        String[] array = {"banana", "apple", "orange"};

        // Sort in reverse order
        Arrays.sort(array, Comparator.reverseOrder());
        System.out.println(Arrays.toString(array));
// Output: [orange, banana, apple]
    }
}
```

- **Collections.sort()**: Sorts lists in natural order or with a custom comparator.
- **Arrays.sort()**: Sorts arrays in natural order or with a custom comparator.

These methods leverage Timsort for efficient sorting, ensuring good performance for a wide range of data types and structures.

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

Unit 4

Lecture 36

Lecture 36

- Comparable Interface
- Comparator Interface
- Properties Class in Java

Comparable and Comparator Interfaces

Comparable Interface: Objects that implement the Comparable interface can be sorted based on their natural order. The compareTo() method defines the sorting criteria.

Comparator Interface: The Comparator interface allows you to define a custom sorting order for objects that don't implement Comparable or when you need to sort based on different criteria.

Sorting Methods

Collections.sort(List): Sorts the elements in the specified List using the natural order defined by the elements' Comparable implementation.

Collections.sort(List, Comparator): Sorts the elements in the specified List using the provided Comparator.

Arrays.sort(array): Sorts the specified array using the natural order defined by the elements' Comparable implementation.

Arrays.sort(array, Comparator): Sorts the specified array using the provided Comparator.

- Java Comparable interface is used to order the objects of the user-defined class.
- This interface is found in java.lang package and contains only one method named compareTo(Object).
- It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only.
- For example, it may be rollno, name, age or anything else.

compareTo(Object obj

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

```
class Student implements Comparable<Student>{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}

public int compareTo(Student st){
if(age==st.age)
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
}
```

```
import java.util.*;
public class TestSort1{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

```
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Comparator interface

- **Java Comparator interface** is used to order the objects of a user-defined class.
- This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).
- It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Properties Class in Java

- The **properties** object contains key and value pair both as a string.
- The java.util.Properties class is the subclass of Hashtable.
- It can be used to get property value based on the property key.
- The Properties class provides methods to get data from the properties file and store data into the properties file.
- Moreover, it can be used to get the properties of a system.

Features of Properties Class in Java

Inheritance: Properties extends the Hashtable class, inheriting its thread-safe behavior and the ability to store key-value pairs. However, both keys and values in a Properties object are treated as Strings.

Loading Properties: Properties objects can be loaded from various sources, including files, input streams, and readers. The load() method reads properties from an input stream in a simple line-oriented format (key=value).

Storing Properties: The store() method can write the contents of a Properties object to an output stream or writer in a format suitable for loading into a Properties object.

Constructors of Properties class

Method	Description
Properties()	It creates an empty property list with no default values.
Properties(Properties defaults)	It creates an empty property list with the specified defaults.

Methods of Properties class

Method	Description
public void load(Reader r)	It loads data from the Reader object.
public void load(InputStream is)	It loads data from the InputStream object
public void loadFromXML(InputStream in)	It is used to load all of the properties represented by the XML document on the specified input stream into this properties table.
public String getProperty(String key)	It returns value based on the key.
public String getProperty(String key, String defaultValue)	It searches for the property with the specified key.
public void setProperty(String key, String value)	It calls the put method of Hashtable.
public void list(PrintStream out)	It is used to print the property list out to the specified output stream.
public void list(PrintWriter out))	It is used to print the property list out to the specified output stream.

Features of Properties Class in Java

Property File Format: The default format for property files is a simple line-oriented format with key-value pairs separated by an equal sign (=). Comments can be included by starting a line with a hash (#) or an exclamation mark (!).

Retrieving Values: Values can be retrieved from a Properties object using the getProperty() method, which takes a key as an argument and returns the corresponding value as a String. If the key is not found, it returns null or a specified default value.

Accessing Properties: In addition to accessing properties directly, Java provides a utility class called System that allows access to system properties, which are accessible through the System.getProperties() method.

ABES Engineering College
College Code 032

java class to read the data from the properties file.

```
import java.util.*;
import java.io.*;
public class Test {
public static void main(String[] args)throws Exception{
    FileReader reader=new FileReader("db.properties");
    Properties p=new Properties();
    p.load(reader);
    System.out.println(p.getProperty("user"));
    System.out.println(p.getProperty("password"));
}
}
```

db.properties

```
user=system
password=oracle
```

Output:system
oracle

Example of Properties class to get all the system properties

```
import java.util.*;
import java.io.*;
public class Test {
public static void main(String[] args)throws Exception{
Properties p=System.getProperties();
Set set=p.entrySet();
Iterator itr=set.iterator();
while(itr.hasNext()){
Map.Entry entry=(Map.Entry)itr.next();
System.out.println(entry.getKey()+" = "+entry.getValue());
}
}
}
```