Programming in Java

Dr. Nyein Aye Maung Maung
Dr. Eng (Ritsumeikan University, Japan)
Lecturer
Computer Engineering and Information Technology Dept.
Yangon Technological University

Course Schedule

Week	Topics
Week 1	Overview of JAVA, Data Types, Variables and Arrays
Week 2	Operators and Control Statements
Week 3	Classes and A closer look at Methods and Classes
Week 4	Inheritance, Polymorphism, Abstraction and Encapsulation
Week 5	Packages and Interfaces
Week 6	Exception Handling and Multi-threaded Programming
Week 7	String Handling
Week 8	Exploring java.lang and More utilities classes
Week 9	Java Collections Framework
Week 10	Java I/O
Week 11	Basic Graphical User Interface
Week 12	Event Handling
Week 13	Database Programming
Week 14	Applet and Networking

Lecture 9

Java Collections Framework

Outline of Class

- Java Collections Framework
  - Collections overview
  - The Collection Interfaces
  - The Collection Classes
    1. List
    2. Queue and priority queues
    3. Sets
    4. Maps

Lecture Objectives

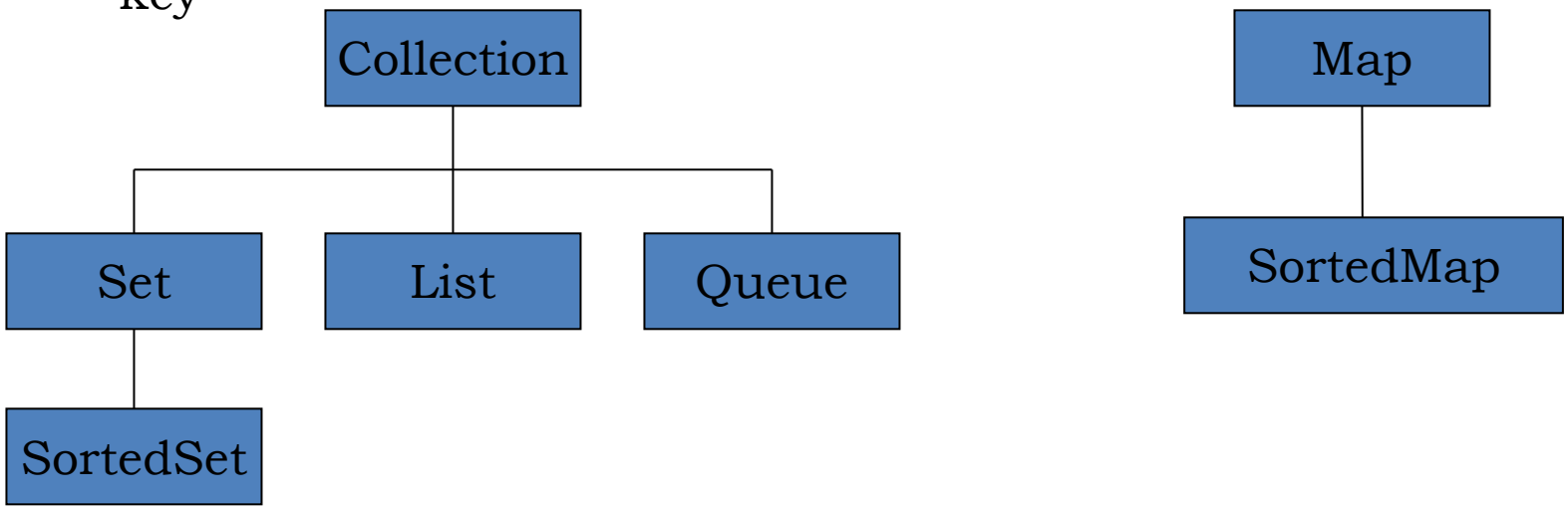
- To explore the relationship between interfaces and classes in the Java Collections Framework hierarchy
- To use the common methods defined in the **Collection** interface for operating collections
- To use the static utility methods in the **Collections** class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections
- To explore how and when to use different collections to store a set of elements

Overview

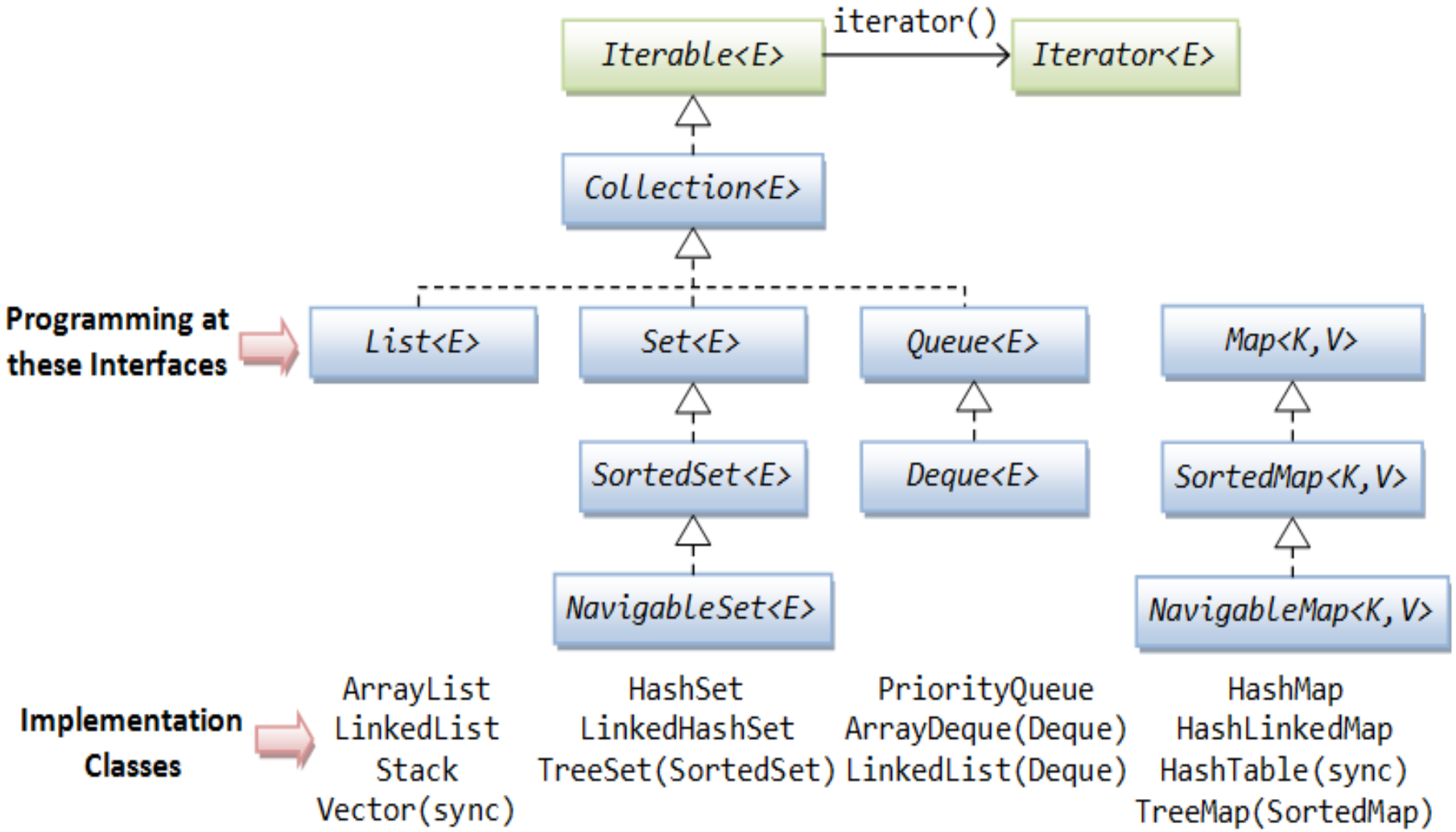
- A Collection is a container that groups similar elements into an entity.
- Examples would include a list of bank accounts, set of students, group of telephone numbers.
- The Collections framework in Java offers a unified approach to store, retrieve and manipulate a group of data
- The Collection interface defines the common operations for *lists, vectors, stacks, queues, priority queues, and sets*

Core Collections Framework

- The Java Collections Framework supports two types of containers:
  - One for storing a collection of elements is simply called a collection .
    - Sets store a group of **non-duplicate elements**.
    - Lists store **an ordered collection of elements**.
    - Queues store objects that are processed in first-in, first-out fashion.
  - The other, for storing key/value pairs, is called a map .
    - Efficient data structures for quickly searching an element using a key



Collection Interfaces



The Collection Classes

1. **List :** Duplicate elements are allowed
   - ArrayList
   - LinkedList
   - Vector
   - Stack

2. **Set:** Duplicate elements are not allowed
   - HashSet
   - LinkedHashSet (Insertion sort)
   - TreeSet (Sorted in ascending order)

3. **Map :** The collection is kept in key/value pairs. Any object can be a key or value. No duplicate keys allowed.
   - TreeMap
   - HashMap
   - LinkedHashMap

Operations of Collection Interface

- Basic Operations
  - The **add method** adds an element to the collection.
  - The **addAll** method adds all the elements in the specified collection to this collection. (Union)
  - The **remove** method removes an element from the collection.
  - The **removeAll** method removes the elements from this collection that are present in the specified collection.
  - The **retainAll** method retains the elements in this collection that are also present in the specified collection. (Intersection)
  - The **clear()** method simply removes all the elements from the collection.

Operations of Collection Interface

- Query Operations
  - The **size** method returns the number of elements in the collection.
  - The **contains** method checks whether the collection contains the specified element.
  - The **containsAll** method checks whether the collection contains all the elements in the specified collection.
  - The **isEmpty** method returns true if the collection is empty.
  - The Collection interface provides the **toArray()** method, which returns an array representation for the collection.

Accessing a Collection

a. Iterator

b. List Iterator

c. For-each

(a) Iterator

- Enable to cycle through a collection
- Provide sequential access to the elements in the collection using the **next()** method.
- Provide **hasNext()** method to check whether there are more elements in the iterator;
- Provide **remove()** method to remove the last element returned by the iterator
- Not allow modification of elements

```
Collection<String> words = new ArrayList<String>();
Iterator<String> iter = words.iterator();
  while (iter.hasNext ())
  {
      String word = iter.next ();
      < … process word >
  }
```

(b) List Iterator

- Extend Iterator to allow bidirectional transversal of a list and **modification of elements**
- Provide sequential access to the elements in the collection using the **next()** method.
- Provide **hasNext()** method to check whether there are more elements in the iterator;
- Provide **previous( )** to return the previous element
- Provide **hasPrevious()** for checking if there is a previous element
- **set (E obj)** assigns object to the current element

(c) For-each

- Convenience to use if it is **not necessary to modify contents** of a collection **or obtaining elements in reverse order**

```
Collection<String> words = new ArrayList<String>();
…
```

```
for (String word : words)
{
    < … process word >
}
```

```
Iterator<String> iter =
                words.iterator();
while (iter.hasNext ())
{
    String word = iter.next ();
    < … process word >
}
```

A “for each” loop is a syntactic shortcut that replaces an iterator

Topic 1

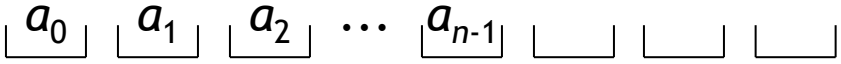
List

List

- A list defines a collection for storing elements in a sequential order.
  a. ArrayList
  b. LinkedList
  c. Vector
  d. Stack

List → ArrayList Class

- Represents a list as a dynamic array (array that is resized when full)
- Provides random access to the elements



- Implements all the methods of List<E>

Methods of List

`+add(index: int, element: Object): boolean`	Adds a new element at the specified index.
`+addAll(index: int, c: Collection<? extends E>) : boolean`	Adds all the elements in `c` to this list at the specified index.
`+get(index: int): E`	Returns the element in this list at the specified index.
`+indexOf(element: Object): int`	Returns the index of the first matching element.
`+lastIndexOf(element: Object): int`	Returns the index of the last matching element.
`+listIterator(): ListIterator<E>`	Returns the list iterator for the elements in this list.
`+listIterator(startIndex: int): ListIterator<E>`	Returns the iterator for the elements from `startIndex`.
`+remove(index: int): E`	Removes the element at the specified index.
`+set(index: int, element: Object): Object`	Sets the element at the specified index.

List → LinkedList Class

- Additional methods specific to LinkedList
  - addFirst( ) or offerFirst( ) : to add elements to the start of a list
  - addLast( ) or offerLast( ): To add elements to the end of the list
  - getFirst( ) or peekFirst( ): To obtain the first element
  - getLast( ) or peekLast( ): To obtain the last element
  - removeFirst( ) or pollFirst( ): To remove the first element, use.
  - removeLast( ) or pollLast( ): To remove the last element,

If you need to support random access through an index without inserting or removing elements at the beginning of the list, ArrayList offers the most efficient collection.

ArrayList is efficient for retrieving elements and LinkedList is efficient for inserting and removing elements at the beginning of the list.

Example 1: ArrayList and Operations of Collection Interface

```
import java.util.*;

public class TestCollection {
  public static void main(String[] args) {
    ArrayList<String> collection1 = new ArrayList<String>();
    collection1.add("New York");
    collection1.add("Atlanta");
    collection1.add("Dallas");
    collection1.add("Madison");

    System.out.println("A list of cities in collection1:");
    System.out.println(collection1);    // output: [New York, Atlanta, Dallas, Madison]

    System.out.println("\nIs Dallas in collection1? "
      + collection1.contains("Dallas"));  // output: True

    collection1.remove("Dallas"); //Remove Dallas from list
    System.out.println("\n" + collection1.size() + " cities are in collection1 now");
                                        // output: 3 cities are in collection1 now
```

```
    Collection<String> collection2 = new ArrayList<String>();
    collection2.add("Seattle");
    collection2.add("Portland");
    collection2.add("Los Angles");
    collection2.add("Atlanta");
    System.out.println("\nA list of cities in collection2:");
    System.out.println(collection2);   //output: [Seattle, Portland, Los Angeles, Atlanta]

    ArrayList<String> c1 = (ArrayList<String>)(collection1.clone());
    c1.addAll(collection2);
    System.out.println("\nCities in collection1 or collection2: ");
    System.out.println(c1); //output: [New York, Atlanta, Madison, Seattle, Portland, Los Angeles, Atlanta]
    c1 = (ArrayList<String>)(collection1.clone());
    c1.retainAll(collection2);
    System.out.print("\n Cities in collection1 and collection2: ");
    System.out.println(c1);  //Output: [Atlanta]

    c1 = (ArrayList<String>)(collection1.clone());
    c1.removeAll(collection2);
    System.out.print("\nCities in collection1, but not in 2: ");
    System.out.println(c1);  //Output: [New York, Madison]
  }
}
```

Example 2: Add data to ArrayList, then put the contents arrayList to LinkedList. After that displays the contents of LinkedList in both forward and backward directions.

```
import java.util.*;

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

    System.out.println("A list of integers in the array list:");
    System.out.println(arrayList); //Output: [10,1,2,30,2,1,4]
```

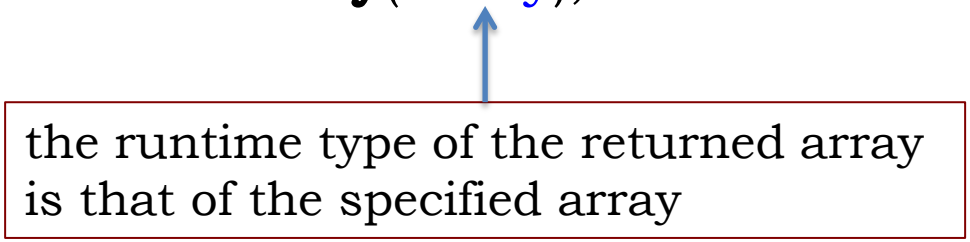
```
LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
linkedList.add(1, "red"); //Add red in index 1
linkedList.removeLast();
linkedList.addFirst("green"); //Add green in first index
System.out.println("Display the linked list forward:");
ListIterator<Object> listIterator = linkedList.listIterator();
while (listIterator.hasNext()) {
    System.out.print(listIterator.next() + " ");
}
System.out.println();
 //List iterator points on end of list after previous while loop
System.out.println("Display the linked list backward:");
 while (listIterator.hasPrevious()) {
    System.out.print(listIterator.previous() + " ");
}
}
}
```

List iterator points on head of list

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

List ←→ Array

- **asList**
  - For creating a list from a variable-length argument list of a generic type
    - List<String> list1 = Arrays.**asList**("red", "green", "blue");
    - List<Integer> list2 = Arrays.**asList**(10, 20, 30, 40, 50);
- **toArray**
  - For creating an object array from a list
    - Integer iarray[ ]=new Integer[list2.size()];
    - iarray=list1.**toArray**(iarray);



Example 3: Write a program that converts an arraylist to an Array and perform summing up the contents of the array.

```
import java.util.*;
class ArrayListToArray {
    public static void main(String args[]) {
        // Create an array list.
        ArrayList<Integer> al = new ArrayList<Integer>();
        // Add elements to the array list.
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);
        System.out.println("Contents of al: " + al);
        // Create an Integer Array of sized ArrayList
        Integer ia[] = new Integer[al.size()];
        ia = al.toArray(ia); //Covert the arraylist a1 to Array
        int sum = 0;
        // Sum the array.
        for(int i : ia) sum += i; //Use for-each iterator
        System.out.println("Sum is: " + sum);
    }
}
```

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

Example 4: Iterator and ListIterator Demo

```
import java.util.*;
class IteratorDemo {
    public static void main(String args[]) {
        // Create an array list.
        ArrayList<String> al = new ArrayList<String>();
        // Add elements to the array list.
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        // Use iterator to display contents of al.
        System.out.print("Original contents of al: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
        String element = itr.next();
        System.out.print(element + " ");
        }
        System.out.println();
```

```
// Modify objects being iterated.
ListIterator<String> litr = al.listIterator();
while(litr.hasNext()) {
String element = litr.next();
litr.set(element + "+"); //List iterator allows modifying the contents
}
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " "); //Display the modified contents forward
}
System.out.println();
// Now, display the list backwards.
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
String element = litr.previous();
System.out.print(element + " ");
}
System.out.println();
}
}
```

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

Example 5: Write a program that use for-each loop to iterate through an ArrayList to display the contents and to perform summing up the contents.

```
// Use the for-each for loop to cycle through a collection.
import java.util.*;
class ForEachDemo {
public static void main(String args[]) {
// Create an array list for integers.
ArrayList<Integer> vals = new ArrayList<Integer>();
// Add values to the array list.
vals.add(1);
vals.add(2);
vals.add(3);
vals.add(4);
vals.add(5);
```

```
// Use for-each loop to display the values.
System.out.print("Original contents of vals: ");
for(int v : vals)
System.out.print(v + " ");
System.out.println();
// Now, sum the values by using a for-each loop.
int sum = 0;
for(int v : vals)
sum += v;
System.out.println("Sum of values: " + sum);
}
}
```

```
Original contents of vals: 1 2 3 4 5
Sum of values: 15
```

Static Methods for Lists and Collections

- Collections Framework provides set of algorithms
  - Implemented as **static** methods
    - **List** algorithms
      - **sort**
      - **binarySearch**
      - **reverse**
      - **shuffle**
      - **fill**
      - **copy**
    - **Collection** algorithms
      - **min**
      - **max**
      - **disjoint**
      - **frequency**

List algorithms

1. Collections.sort(list l);
   - Collection.sort(list l) → natural ordering using comparable
   - Collection.sort(list l, Comparator comp) → Order using comparator
2. Collections.reverse(list l);
3. Collections.shuffle(list l);
4. Collections.copy(list destination, list source);
   - Copy to the same index (**replace existing**)
   - The destination list must be as long as the source list
     - If destination list is longer, remaining elements are not affected

List algorithms (cont')

5. Collections.fill(list l, obj o)
   - Fills the list with the object.
6. Binary Search
   - Collections.binarysearch(list l, key)
     - Search for a key in **a sorted list** (ascending order)
   - Collections.binarysearch(list l, key, Comparator comp)
     - Search for a key in the list **ordered according to comp**
   - Return – (insertion point+1) if the key is not found
     - [1 3 5 6] , search key: 4 → - (2+1) = -3

Collections algorithms

1. Collections.disjoint(c1: Collection, c2: Collection): boolean
   - Returns true if c1 and c2 have **no elements in common**
2. Collections.frequency(c: Collection, o: Object): int
   - Returns **the number of occurrences** of the specified element in the collection.
3. max(c: Collection): Object
   - Returns the **max object** in the collection.
4. min(c: Collection): Object
   - Returns the **min object** in the collection.

Example 6: Write a program that sorts a String ArrayList.

```
import java.util.*;
public class SortProgram {
      private static String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
      // display array elements
      public void printElements()
      {
            // create ArrayList
            ArrayList list = new ArrayList( Arrays.asList( suits ) );
            // output list
            System.out.println( "Unsorted array elements:\n" + list );
            // sort ArrayList
            Collections.sort( list );
            // output list
            System.out.println( "Sorted array elements:\n" + list );
            }
            // execute application
            public static void main( String args[] )
            {
            new SortProgram().printElements();
            }

} // end class Sort1
```

```
Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted array elements:
[Clubs, Diamonds, Hearts, Spades]
```

Example 7: Write a program that sorts a String ArrayList in reverse order.

```
import java.util.*;
public class ReverseSortProgram {
      private static String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
      // display array elements
      public void printElements()
      {
            // create ArrayList
            ArrayList list = new ArrayList( Arrays.asList( suits ) );
            // output list
            System.out.println( "Unsorted array elements:\n" + list );
            // sort ArrayList in descending oreder using a comparator
            Collections.sort( list, Collections.reverseOrder() );
            // output list
            System.out.println( "Sorted array elements:\n" + list );
            }
            // execute application
            public static void main( String args[] )
            {
            new ReversseSortProgram().printElements();
            }

} // end class
```

Method **sort** of class **Collections** can use a **Comparator** object to sort a **List**

```
Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted list elements:
[Spades, Hearts, Diamonds, Clubs]
```

Example 8: Write a program that shuffles a String ArrayList in random.

```
import java.util.*;
public class ReverseSortProgram {
      private static String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
      // display array elements
      public void printElements()
      {
            // create ArrayList
            ArrayList list = new ArrayList( Arrays.asList( suits ) );
            // output list
            System.out.println( "Unsorted array elements:\n" + list );
            // shuffle list elements in random
            Collections.shuffle( list);
            // output list
            System.out.println( "Sorted array elements:\n" + list );
            }
            // execute application
            public static void main( String args[] )
            {
            new ReversseSortProgram().printElements();
            }

} // end class
```

```
Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted list elements:
[Diamonds, Hearts, Clubs, Spades]
```

Example 9: A program that tests methods in Collections class.

```
import java.util.*;
public class test2 {
        private String letters[] = { "P", "C", "M","N"};
        private List list, copyList;
        // create a List and manipulate it with algorithms from class Collections
        public test2()
        {
        list = Arrays.asList( letters );     // get List
        String[] lettersCopy = new String[ 4 ];
        copyList = Arrays.asList( lettersCopy );

        System.out.println( "Initial List: " );
        System.out.println( list );

        Collections.reverse( list );         // reverse order
        System.out.println( "Reversed List: " );
        System.out.println( list );

        Collections.copy( copyList, list );  // copy List
        System.out.println( "Copied List: " );
        System.out.println( copyList );
```

```
Initial List:
[P, C, M, N]
```

```
Reversed List:
[N, M, C, P]
```

```
Copied List:
[N, M, C, P]
```

```
Collections.fill( list, "R" );
System.out.println( "List after calling fill: " );
System.out.println( list );

Collection<String> collection = Arrays.asList("red", "green", "blue");
System.out.println("Maximum object:"+Collections.max(collection) );
System.out.println("Minimum object:"+Collections.min(collection));

Collection<String> collection1 = Arrays.asList("red", "cyan");
Collection<String> collection2 = Arrays.asList("red", "blue");
Collection<String> collection3 = Arrays.asList("pink", "tan");
System.out.println(Collections.disjoint(collection1, collection2) );
System.out.println(Collections.disjoint(collection2, collection3));
System.out.println(Collections.frequency(collection, "red"));
        }
        // execute application
        public static void main( String args[] )
        {
        new test2();
        }

        } // end class
```

```
List after calling fill:
[R, R, R, R]
```

```
Maximum object:red
Minimum object:blue
```

```
false
true
1
```

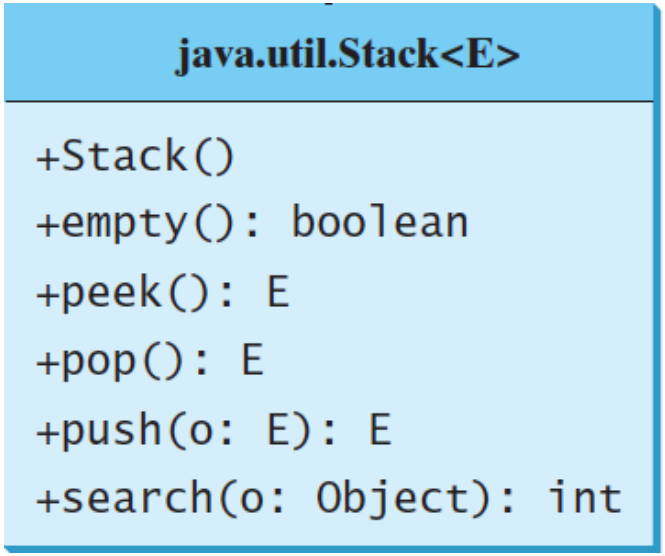
Example 9: Binary Search.

```
import java.util.*;
public class BinarySearchList {
public static void main( String args[] ){

List<Integer> list1 =Arrays.asList(5, 6,  7, 66,2, 4, 10,  45, 50, 11, 59, 60);
List<String> list2 = Arrays.asList("blue", "red", "green","cyan");

//Without sorting the list
System.out.println("Without sorting the Lists------");
System.out.println("Index of 2:"+Collections.binarySearch(list1, 2));  //-1
System.out.println("Index of 9: " + Collections.binarySearch(list1, 9)); //-7
System.out.println("Index of red: " +Collections.binarySearch(list2, "red")); //1
System.out.println("Index of cyan:" +Collections.binarySearch(list2, "cyan")); //-2

//After sorting the list
System.out.println("After sorting the Lists------");
Collections.sort(list1);
Collections.sort(list2);
System.out.println("Index of 2:"+Collections.binarySearch(list1, 2));   //0
System.out.println("Index of 9: " + Collections.binarySearch(list1, 9));   //-6
System.out.println("Index of red: " +Collections.binarySearch(list2, "red"));  //3
System.out.println("Index of cyan:" +Collections.binarySearch(list2, "cyan"));  //1
        }
        }
```

List → Vector

- The same as ArrayList , except that it contains synchronized methods for accessing and modifying the vector
- Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently(Multithreading and Parallel Programming)
- For the many applications that do not require synchronization, using **ArrayList is more efficient than using Vector**

List → Stack



Creates an empty stack.	
Returns true if this stack is empty.	
Returns the top element in this stack.	
Returns and removes the top element in this stack.	
Adds a new element to the top of this stack.	
Returns the position of the specified element in this stack.	

- Last-in, first-out

Example 10: Write a program that stores double values inputted from keyboard onto stack until user enters an invalid data format. Then, data on stack are displayed back to user.

```
import java.util.*;
        public class stackTest {
          static void showStackData(Stack st) {
                while(!st.isEmpty())
                {
                        Double d=(Double) st.pop();
                        System.out.println(d+" ");
                }
          }
          public static void main(String args[]) {
            Stack st = new Stack();
            Scanner input=new Scanner(System.in);
            while(input.hasNext())
            {
                if(input.hasNextDouble())
                        st.push(input.nextDouble());
                else
                {
                        System.out.println("Finished Entering Data!! ");
                        showStackData(st);
                        break;
                }
            } } }
```

Output

```
12
13
14
pp
Finished Entering Data!!
14.0
13.0
12.0
```

Quiz 1

1. Suppose that list1 is an arraylist that contains the strings red , yellow , and green , and that list2 is another arraylist that contains the strings red , yellow , and blue . Answer the following questions:

   a. What are list1 and list2 after executing list1.addAll(list2) ?
   b. What are list1 and list2 after executing list1.add(2,“cyan) ?
   c. What are list1 and list2 after executing list1.removeAll(list2) ?
   d. What are list1 and list2 after executing list1.remove(“green") ?
   e. What are list1 and list2 after executing list1.retainAll(list2) ?
   f. What is list1 after executing list1.clear() ?

2. Which list should you use to insert and delete elements at the beginning of a list?

3. Are all the methods in ArrayList also in LinkedList ? What methods are in LinkedList but not in ArrayList ?

4.How do you create a list from an array of objects?

5. Which method can you use to sort the elements in an ArrayList or a LinkedList ?

6.Which method can you use to perform binary search for elements in an ArrayList or a LinkedList ?

7. What are the output of the following statements.

```
import java.util.*;
public class test2 {
public static void main(String[] args) {
        List<String> list =Arrays.asList("yellow", "red", "green", "blu
        e");
        Collections.reverse(list);
        System.out.println(list);
        List<String> list1 =Arrays.asList("yellow", "red", "green", "bl
        ue");
        List<String> list2 = Arrays.asList("white", "black");
        Collections.copy(list1, list2);
        System.out.println(list1);
        Collection<String> c1 = Arrays.asList("red", "cyan");
        Collection<String> c2 = Arrays.asList("red", "blue");
        Collection<String> c3 = Arrays.asList("pink", "tan");
        System.out.println(Collections.disjoint(c1, c2));
        System.out.println(Collections.disjoint(c1, c3));
        Collection<String> collection =
        Arrays.asList("red", "cyan", "red");
        System.out.println(Collections.frequency(collection, "red"));
}
}
```

8. Which of the following static methods in the Collections class are for lists, and

- which are for collections?
- sort, binarySearch, reverse, shuffle, max, min, disjoint, frequency

9. Write a statement to find the largest element in an array of comparable objects. eg. objArray

Topic 2

QUEUE

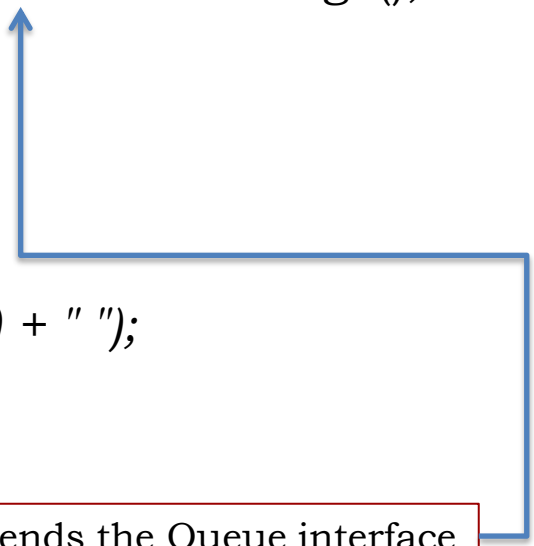
The Queue Interface

- ***Queue*** interface extends java.util.Collection with additional insertion, extraction, and inspection operations
  - offer (element: E)
    - Use to add an element to the queue
    - Similar to the add method in the Collection interface, but the offer method is preferred for queues.
  - poll( ) and remove( ) – retrieves and remove the first element of the queue
    - poll() returns null if the queue is empty, whereas remove() throws an exception.
  - peek( ) and element( ) – retrieves the first element without removing it
    - peek() returns null if the queue is empty, whereas element() throws an exception.

- **Deque (Double ended queue)**
  - support element insertion and removal at both ends
  - extends Queue with additional methods for inserting and removing elements from both ends
    - addFirst(e), removeFirst(), addLast(e), removeLast(), getFirst(), and getLast()

Example 11: Write a program that adds four cities on a queue and displays the contents back.

```
import java.util.*;
public class queueDemo {

        public static void main(String[] args) {

        Queue<String> queue =new LinkedList<String>();
        queue.offer("Oklahoma");
        queue.offer("Indiana");
        queue.offer("Georgia");
        queue.offer("Texas");
        while(queue.size() > 0)
        System.out.print(queue.remove() + " ");
        }
        }
```



LinkedList class implements the Deque interface, which extends the Queue interface

Output

```
Oklahoma Indiana Georgia Texas
```

Priority Queue

- The priority queue orders its elements according to their natural ordering using Comparable
- The element with the least value is assigned the highest priority and thus is removed from the queue first
- You can also specify an ordering using Comparator in the constructor

  PriorityQueue(initialCapacity, comparator)

Example 12: Write a program that adds four cities on a default priority queue and the priority queue with reverse order priority . Then, displays the contents back from both queues.

```
import java.util.*;

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

    System.out.println("Priority queue using Comparable:");
    while (queue1.size() > 0) {
      System.out.print(queue1.remove() + " ");
    }

    PriorityQueue<String> queue2 = new PriorityQueue<String>(
      4, Collections.reverseOrder());
    queue2.offer("Oklahoma");
    queue2.offer("Indiana");
    queue2.offer("Georgia");
    queue2.offer("Texas");

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

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

Quiz 2

1. How do you create a priority queue for integers? By default, how are elements ordered in a priority queue? Is the element with the least value assigned the lowest priority in a priority queue?

PriorityQueue<Integer> pq=new PriorityQueue<Integer>( );

Order its elements according to their natural ordering using Comparable

No. Least value is assigned the highest priority.

Topic 3

SET

Set

- A set is an efficient data structure for storing and processing non-duplicate elements
- Extends the Collection interface
  - HashSet,
  - LinkedHashSet
  - TreeSet

Set → Hashset

- Store information by using a mechanism called Hashing.
- A HashSet can be used to store duplicate-free elements
- Constructors
  - +HashSet()
  - +HashSet(c: Collection<? extends E>)
  - +HashSet(initialCapacity: int)
  - +HashSet(initialCapacity: int, loadFactor: float)
- Capacity, Load Factor
  - The capacity is the number of buckets in the hash table
  - The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.
    - When the number of elements exceeds the product of the capacity and load factor, the capacity is automatically doubled.
    - eg. capacity=16, load factor=0.75 →

    Capacity will be double when the size reaches 12 (16* 0.75)
- **Does not guarantee the order of its elements**

Example 13: Write a program that adds the following elements to a hash set and displays the contents.
{“London”, “Paris”,”New York”, “San Francisco”, “Beijing”, “NewYork”}

```
import java.util.*;

public class TestHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<String>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    System.out.println(set);

    // Display the elements in the hash set
    for (String s: set) {
      System.out.print(s.toUpperCase() + " ");
    }
  }
}
```

[San Francisco, New York, Paris, Beijing, London]
SAN FRANCISCO NEW YORK PARIS BEIJING LONDON

Example 14: Hash set Demo with duplicate contents

```
import java.util.HashSet;
import java.util.Set;
public class HashSetDemo {
   public static void main(String[] args) {
      String book1 = new String("Java for Dummies");
      String book1Dup = new String("Java for Dummies"); // same id as above
      String book2 = new String("Java for more Dummies");
      String book3 = new String("more Java for more Dummies");

      HashSet<String> set1 = new HashSet<String>();
      set1.add(book1);
      set1.add(book1Dup); // duplicate id, not added
      set1.add(book1);    // added twice, not added
      set1.add(book3);
      set1.add(null);     // Set can contain a null
      set1.add(book2);
      System.out.println(set1); // [nu   1: Java for Dummies,
                                //   2: Java for more Dumm  s, 3: more Java for
more Dummies]
```

```
        set1.remove(book1);
        set1.remove(book3);
        System.out.println(set1); // [null, __ Java for more Dummies]

        Set<String> set2 = new HashSet<String>();
        set2.add(book3);
        System.out.println(set2); // [__ more Java for more Dummies]
        set2.addAll(set1);        // "union" with set1
        System.out.println(set2); // [null, __ Java for more Dummies, __ more Java
for more Dummies]

        set2.remove(null);
        System.out.println(set2); // [__ Java for more Dummies, __ more Java for
more Dummies]
        set2.retainAll(set1);     // "intersection" with set1
        System.out.println(set2); // [__ Java for more Dummies]
    }
}
```

Set → LinkedHashSet

- LinkedHashSet **extends** HashSet with a linked-list implementation that **supports an ordering of the elements** in the set
  - Retrieved in the order in which they were inserted into the set

* Change HashSet in Example 14 to LinkedHashSet and varify the output sequence whether it is ordered upon their insertion or not.

Set → TreeSet

- Objects are stored in sorted, ascending order
- first() and last()
  - for returning the first and last elements in the set
- headSet(toElement) and tailSet(fromElement)
  - for returning a portion of the set whose elements are less than toElement and greater than or equal to fromElement
- pollFirst() and pollLast()
  - **remove and return** the first and last element in the tree set
- lower(e) : return largest element less than the specified element
- floor(e): return the largest element less than or equal to the specified element
- ceiling(e): return the smallest element greater than or equal to the specified element
- higher(e) : return the smallest element greater than a given element

Return null if there is no such element

Example 15: Tree set Demo

```
import java.util.*;

public class TestTreeSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<String>();
    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");
    //Create a tree set
    TreeSet<String> treeSet = new TreeSet<String>(set);
    System.out.println("Sorted tree set: " + treeSet);
    System.out.println("first(): " + treeSet.first());
    System.out.println("last(): " + treeSet.last());
    System.out.println("headSet(\"New York\"): " +
      treeSet.headSet("New York"));
    System.out.println("tailSet(\"New York\"): " +
      treeSet.tailSet("New York"));
    System.out.println("lower(\"P\"): " + treeSet.lower("P"));
    System.out.println("higher(\"P\"): " + treeSet.higher("P"));
    System.out.println("floor(\"P\"): " + treeSet.floor("P"));
    System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));
    System.out.println("pollFirst(): " + treeSet.pollFirst());
    System.out.println("pollLast(): " + treeSet.pollLast());
    System.out.println("New tree set: " + treeSet);
  }
}
```

```
Sorted tree set: [Beijing, London, New York,
Paris, San Francisco]
first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]
tailSet("New York"): [New York, Paris, San
Francisco]
lower("P"): New York
higher("P"): Paris
floor("P"): New York
ceiling("P"): Paris
pollFirst(): Beijing
pollLast(): San Francisco
New tree set: [London, New York, Paris]
```

Example 16: Write a program that counts the number of the keywords in a Java source file.

```
import java.util.*;
import java.io.*;

public class CountKeywords {
  public static void main(String[] args) throws Exception {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a Java source file: ");
    String filename = input.nextLine();

    File file = new File(filename);
    if (file.exists()) {
      System.out.println("The number of keywords in " + filename
        + " is " + countKeywords(file));
    }
    else {
      System.out.println("File " + filename + " does not exist");
    }
  }
```

```
 public static int countKeywords(File file) throws Exception {
    // Array of all Java keywords + true, false and null
    String[] keywordString = {"abstract", "assert", "boolean”,  "break", "byte", "case", "ca
tch", "char", "class", "const”, "continue", "default", "do", "double", "else", "enum”, "exte
nds", "for", "final", "finally", "float", "goto”, "if", "implements", "import", "instanceof", "i
nt", "interface", "long", "native", "new", "package", "private”, "protected", "public", "retu
rn", "short", "static", "strictfp", "super", "switch", "synchronized", "this”, "throw", "thro
ws", "transient", "try", "void", "volatile”, "while", "true", "false", "null"};

    Set<String> keywordSet =   new HashSet<String>(Arrays.asList(keywordString));
    int count = 0;
    Scanner input = new Scanner(file);
    while (input.hasNext()) {
      String word = input.next();
      if (keywordSet.contains(word))
        count++;
    }

    return count;
  }
}
```

Quiz 3

1. How do you create an instance of Set ? How do you insert a new element in a set? How do you remove an element from a set? How do you find the size of a set?

Set<Object obj> s=new HashSet<Object obj> ();

set.add(Element e)
set.remove(Element e)

set.size( ) method

2. What are the differences between HashSet , LinkedHashSet , and TreeSet ?

Hash Set	Linked Hash Set	Tree Set
unsorted	Insertion Sort	Sort upon comparator

3. How do you sort the elements in a set using the compareTo method in the Comparable interface? How do you sort the elements in a set using the Comparator interface?

Comparable → create a TreeSet using new TreeSet(), or create a TreeSet from a set using new TreeSet(set)
Comparator → create a TreeSet using new TreeSet(Comparator cmp)

4. Show the output of the following code:

```
import java.util.*;
public class Test {
public static void main(String[] args) {
LinkedHashSet<String> set1 = new LinkedHashSet<String>();
set1.add("New York");
LinkedHashSet<String> set2 = set1;
LinkedHashSet<String> set3 =
(LinkedHashSet<String>)(set1.clone());
set1.add("Atlanta");
System.out.println("set1 is " + set1);
System.out.println("set2 is " + set2);
System.out.println("set3 is " + set3);
}
}
```

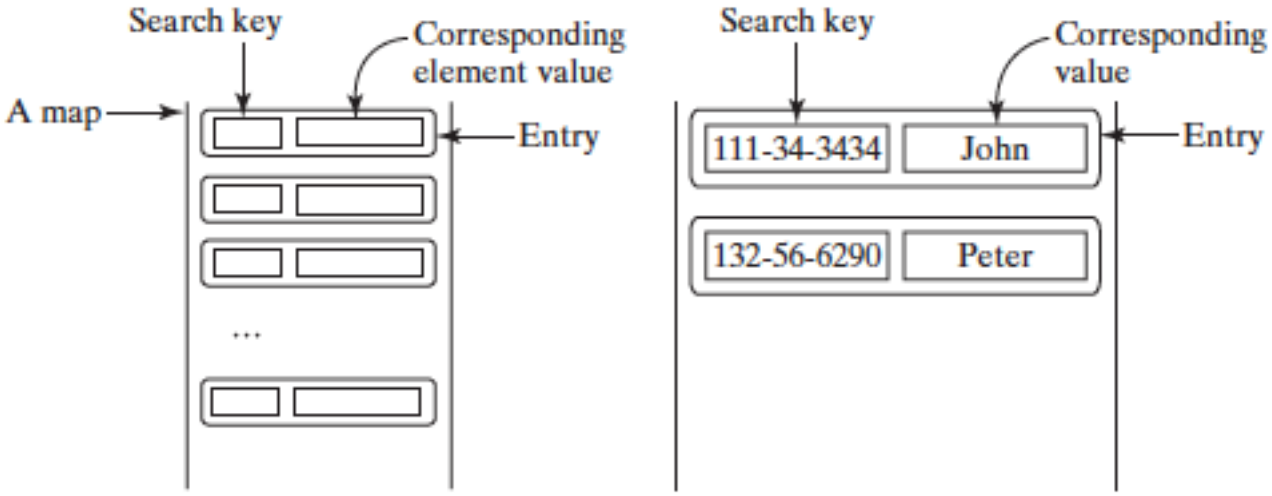
set1 is [New York, Atlanta]
set2 is [New York, Atlanta]
set3 is [New York]

Topic 4

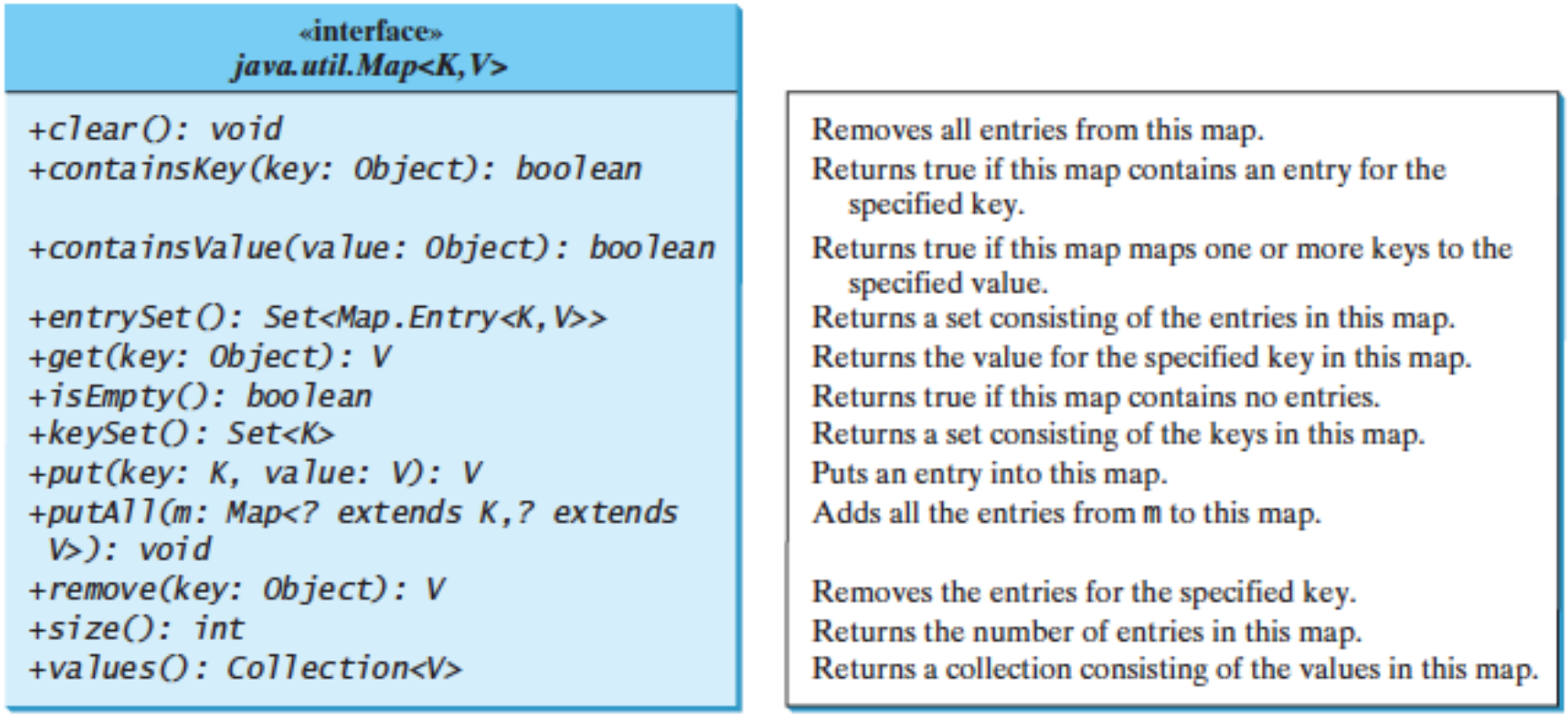
MAP

Map

- A map is a container object that stores a collection of key/value pairs
- A map stores the values along with the keys.
- The keys are like indexes.
- A map cannot contain duplicate keys.
- Each key maps to one value



Map Interface



The **entrySet( )** method declared by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

Types of Map

- HashMap
  - Random order
- LinkedMap
  - Insertion Sort (Default)
  - Access Sort (in the order in which they were last access)
    - LinkedMap (initialCapacity, loadFactor, true) //constructor
- TreeMap
  - Ascending order

Example 17: Map Demo.

```
import java.util.*;

public class TestMap {
  public static void main(String[] args) {
    // Create a HashMap
    Map<String, Integer> hashMap = new HashMap<String, Integer>();
    hashMap.put("Smith", 30);
    hashMap.put("Anderson", 31);
    hashMap.put("Lewis", 29);
    hashMap.put("Cook", 29);

    System.out.println("Display entries in HashMap");
    System.out.println(hashMap + "\n");

    // Create a TreeMap from the preceding HashMap
    Map<String, Integer> treeMap =
      new TreeMap<String, Integer>(hashMap);
    System.out.println("Display entries in ascending order of key");
    System.out.println(treeMap);
```

```
    // Create a LinkedHashMap
    Map<String, Integer> linkedHashMap =
      new LinkedHashMap<String, Integer>(16, 0.75f, true);
    linkedHashMap.put("Smith", 30);
    linkedHashMap.put("Anderson", 31);
    linkedHashMap.put("Lewis", 29);
    linkedHashMap.put("Cook", 29);

    // Display the age for Lewis
    System.out.println("\nThe age for " + "Lewis is " +
      linkedHashMap.get("Lewis"));

    System.out.println("Display entries in LinkedHashMap");
    System.out.println(linkedHashMap);
  }
}
```

```
Display entries in HashMap
{Smith=30, Lewis=29, Anderson=31, Cook=29}

Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}

The age for Lewis is 29
Display entries in LinkedHashMap
{Smith=30, Anderson=31, Cook=29, Lewis=29}
```

Example 18: Write a program that stores occurance of each word in a string in a map.

```
import java.util.*;

public class CountOccurrenceOfWords {
  public static void main(String[] args) {
    // Set text in a string
    String text = "Good morning. Have a good class. " +
      "Have a good visit. Have fun!";

    // Create a TreeMap to hold words as key and count as value
    Map<String, Integer> map = new TreeMap<String, Integer>();

    String[] words = text.split("[ \n\t\r.,;:!?(){}]");
    for (int i = 0; i < words.length; i++) {
      String key = words[i].toLowerCase();

      if (key.length() > 0) {
        if (!map.containsKey(key)) {
          map.put(key, 1);
        }
        else {
          int value = map.get(key);
          value++;
          map.put(key, value);
        }
      }
    }
```

Output

```
    // Get all entries into a set
    Set<Map.Entry<String, Integer>> entrySet = map.entrySet();

    // Get key and value from each entry
    for (Map.Entry<String, Integer> entry: entrySet)
      System.out.println(entry.getKey() + "\t" + entry.getValue());
  }
}
```

```
a       2
class   1
fun     1
good    3
have    3
morning         1
visit   1
```

Storing User-defined Objects in Collections

Example 19. Create a class named Address, which has name, street, city, state and code variables to hold the address of different peoples. Then, write a program that adds address of three different people into a linked list and displays all information in the list.

```
import java.util.*;
class Address {
        private String name;
        private String street;
        private String city;
        private String state;
        private String code;
        Address(String n, String s, String c,
        String st, String cd) {
        name = n;
        street = s;
        city = c;
        state = st;
        code = cd;
              }
```

```
    public String toString() {
    return name + "\n" + street + "\n" +
    city + " " + state + " " + code;
    }
}

class MailList {
public static void main(String args[]) {
    ArrayList<Address> ml = new ArrayList<Address>();
    // Add elements to the linked list.
    ml.add(new Address("J.W. West", "11 Oak Ave",
    "Urbana", "IL", "61801"));
    ml.add(new Address("Ralph Baker", "1142 Maple Lane",
    "Mahomet", "IL", "61853"));
    ml.add(new Address("Tom Carlton", "867 Elm St",
    "Champaign", "IL", "61820"));
    // Display the mailing list.
    for(Address element : ml)
    System.out.println(element + "\n");
    System.out.println();
}
}
```

Quiz 4

i. Suppose you need to write a program that stores non-duplicate elements , what data structure should you use?
hashset

ii. Suppose you need to write a program that stores non-duplicate elements in the order of insertion, what data structure should you use?
linked hash set

iii. Suppose you need to write a program that stores non-duplicate elements in increasing order of the element values, what data structure should yo u use?
tree set

iv. Suppose you need to write a program that stores a fixed number of the el ements (possibly duplicates), what data structure should you use?
Array

v. Suppose you need to write a program that stores the elements in a list wi th frequent operations to add and insert elements at the end of the list, w hat data structure should you use?
ArrayList

vi. Suppose you need to write a program that stores the elements in a list wi th frequent operations to add and insert elements at the beginning of the list, what data structure should you use?
linked list

Assignments

1. Write a program that lets the user enter numbers and stores them in an array list. Do not store duplicate numbers. Then, performs sorting, shuffling, and reversing the list and display the results.
2. Write a test program that stores 5 million integers in a linked list and test the time to traverse the list using an iterator vs. using the get(index) method.
3. Create two priority queues, {"George" , "Jim" , "John" , "Blake" , "Kevin" , "Michael" } and {"George" , "Katie" , "Kevin" , "Michelle" , "Ryan" }, and find their union, difference, and intersection.
4. Write a program that reads words from a text file and displays all the non-duplicate words in ascending order.

Next Week Lecture

- Java I/O