Basics of the Object-Oriented Programming
Collections

List - ArrayList Class

- Uses a dynamic array for storing the elements
- It is like an array, but there is *no size limit*
- Can have the duplicate elements
- Maintains the insertion order internally
- Inherits the **AbstractList** class and implements List interface
- Provides methods to manipulate the size of the array that is used internally to store the list
- Permits null elements

Collections - ArrayList Class

- **Creation**
  - new ArrayList()
    - » Constructs an empty list with an initial capacity of ten
  - new ArrayList(int initialCapacity)
    - » Constructs an empty list with the specified initial capacity
- **Size**
  - int size()
    - » Returns the number of elements of the list

Collections - ArrayList Class

- **Storage**
  - boolean add(Object o)
    - » Add the specified element to the end of this list
  - boolean add(int index, Object element)
    - » Inserts the specified element at the specified position in this list
  - Object set(int index, Object element)
    - » Replaces the element at the specified position in this list with the specified element
- **Retrieval**
  - Object get(int index)
    - » Returns the element at the specified position in this list
  - Object remove(int index)
    - » Removes the element at the specified position in this list

Collections - ArrayList Class

- **Testing**
  - boolean isEmpty()
    - » Returns true if this list is empty
  - boolean contains(Object elem)
    - » Returns true if this list contains the specified element
- **Finding the position**(for fail= -1):
  - int indexOf(Object elem)
    - » Returns the index of the first occurrence of the specified element in this list, or -1 if the element is not in list
  - int lastIndexOf(Object elem)
    - » Returns the index of the last occurrence of the specified element in this list, or -1 if the element is not in list

Collections – Differences between Array List and Vector

- For most purposes, **ArrayList** and **Vector** are equivalent
  - The **Vector** class is older, and some methods have to be added to fit it into the java collection framework
  - The **ArrayList** class is newer and has been created as part of the Java collection framework
  - The **ArrayList** class is supposed to be even more efficient than the **Vector** class

Collections – Example of using ArrayList

```
import java.util.ArrayList;
import java.util.Date;
public class ArrayListExample {
   public static void main(String[] args) {
     ArrayList <Date>birthdays = new ArrayList<Date>( );
     birthdays.add(new Date(90, 1, 1));
     birthdays.add(new Date(90, 2, 2));
     birthdays.add(new Date(90, 3, 3));
     System.out.println("The elements of the list are:");
     for (Date d:birthdays)
          System.out.println(d);
     System.out.println("Change References");
     for (Date d:birthdays) {
               d.setDate(1);
               d.setMonth(3);
               d.setYear(90);
               }
     System.out.println("Now the elements of the list are:");
               for (Date d:birthdays)
                         System.out.println(d);
     birthdays.remove(0);
     System.out.println("The elements of the list after removing
     first element:");
               for (Date d:birthdays)
                         System.out.println(d);
                         }
}
```

>>outputs:

```
The elements of the list are:
Thu Feb 01 00:00:00 EET 1990
Fri Mar 02 00:00:00 EET 1990
Tue Apr 03 00:00:00 EEST 1990
Now the elements of the list are:
Sun Apr 01 00:00:00 EEST 1990
Sun Apr 01 00:00:00 EEST 1990
Sun Apr 01 00:00:00 EEST 1990
The elements of the list after removing first element:
Sun Apr 01 00:00:00 EEST 1990
Sun Apr 01 00:00:00 EEST 1990
```

Collections – LinkedList Class

- Is derived from the abstract class **AbstractSequentialList**
- It should be used when it is necessary to efficiently traverse a list
- Each element from list contains a reference to the **next** element and a reference to the **previous** element
- Linked lists behave well with **insertions** and **deletions** operations
- Iteration is slower
  - Can not search randomly, the list must be traversed element with element

Collections – Enumerators and Iterators

- There are objects used to traverse a collection
- Java has two variations:
  - **Enumeration** (old: from JDK 1.0)
  - **Iterator** (newer: from JDK 1.2)

Collections - Enumerators

- To get an enumerator *e* for container *v*:

  Enumeration e = v.elements();

  – *e* is initialized at the beginning of the list

- To get the first item and the following:

  someObject = e.nextElement()

- To check if we've gone through all of them:

  – e.hasMoreElements()

- Example:

```
for(Enumeration e=v.elements();e.hasMoreElements();)
   { System.out.println(e.nextElement()); }
```

- **Enumerations** do not allow for the modification of the collection, which is being traversed

  – **Iterators** are used if this is required

Collections – Enumerators (Example)

```
import java.util.Vector;
import java.util.Enumeration;
public class EnumerationTester {
  public static void main(String args[]) {
   Enumeration days;
   Vector dayNames = new Vector();
   dayNames.add(“Sunday”);
   dayNames.add(“Monday”);
   dayNames.add(“Tuesday”);
   dayNames.add(“Wednesday”);
   dayNames.add(“Thursday”);
   dayNames.add(“Friday”);
   dayNames.add(“Saturday”};
   days = dayNames.elements();
   while (days.hasMoreElements())
     System.out.println(days.nextElement());

  }

 }
```

Collections - Iterators

- An object used in a collection to provide sequential access to collection elements
  - This access allows for the examination and eventual modification of the elements
- The iterator requires an order of the elements of the collection even if the collection itself does not impose an order on the elements it contains
- If the collection requires an order on its elements, the iterator will use the same order

Collections - Iterator interface

- Iterators have three methods:
  - **hasNext**() - returns true if there is another element in the collection
  - **next**() - returns the next object
  - **remove**() - removes the last object taken using **next**()
- The iterator is associated with a collection object using the **iterator** method
  - Example: if *c* is an instance of a collection class (i.e., HashSet), the following code obtains an iterator for c:

```
HashSet c = new HashSet();
Iterator iteratorForC = c.iterator();
```

Collections- Using an iterator with a HashSet object

- A **HashSet** object does not require any order on the elements it contains
- However, an iterator will impose an order on the elements in the set
  - This will be the order in which the items are retrieved by **next**()
  - Although at each run of the program the order of the items produced may be the same, there is no requirement to impose this think

Collections - Using an iterator with a HashSet object

```
import java.util.HashSet;
import java.util.Iterator;
public class HashSetIteratorDemo
{
 public static void main(String[] args)
  {
   HashSet <String> s = new HashSet <String>( );
   s.add("health");
   s.add("love");
   s.add("money");
   System.out.println("The set contains:");
   Iterator i = s.iterator( );
    while (i.hasNext( ))
    System.out.println(i.next( ));
   i.remove( );
  System.out.println( );
  System.out.println("The set now contains:");
  i = s.iterator( );
  while (i.hasNext( ))
    System.out.println(i.next( ));
    System.out.println("End of program.");
  }
  }
```

>>output:

```
The set contains:
love
money
health

The set now contains:
love
money
End of program.
```

Collections- ListIterator

- List also provides a richer iterator, called a **ListIterator**, which allows
  - To traverse the list in either direction
  - To modify the list during iteration
  - To obtain the current position of the iterator
- The three methods that **ListIterator** inherits from **Iterator** (**hasNext**, **next**, and **remove**) do exactly the same thing in both interfaces
- The **hasPrevious** and the **previous** operations are exact analogues of **hasNext** and **next**
  - The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor
  - The **previous** operation moves the cursor backward, whereas **next** moves it forward

Collections – ListIterator (Example)

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
public class MyListIterator {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		 List <Integer> numbers = new ArrayList<Integer>();
		 numbers.add(new Integer(23));
		 numbers.add(new Integer(98));
		 numbers.add(new Integer(29));
		 numbers.add(new Integer(71));
		 numbers.add(new Integer(5));
	    ListIterator it= numbers.listIterator();
	    System.out.println(" elements in forward direction");
	    while(it.hasNext())
		System.out.println(it.next());
	    System.out.println(" elements in previous direction");
	    while(it.hasPrevious())
		System.out.println(it.previous());
	}
}
```

>>output:

```
 elements in forward direction
23
98
29
71
5
 elements in previous direction
5
71
29
98
23
```

Collections - Object Ordering

- A List **l** may be sorted as follows:

```
Collections.sort(l);
```

  - If the **List** consists of **String** elements, it will be sorted into alphabetical order
  - If it consists of **Date** elements, it will be sorted into chronological order
  - How does this happen?
    - **String** and **Date** both implement the **Comparable** interface

Collections - Object Ordering

- If you try to sort a list, and the class of elements of list do not implement **Comparable**, **Collections.sort**(*list*) will throw a **ClassCastException**
- Similarly, **Collections.sort**(*list*, *comparator*) will throw a **ClassCastException** if you try to sort a list whose elements cannot be compared to one another using the comparator

Collections - Object Ordering – Comparable interface

- Writing Your Own Comparable Types imply implementing the Comparable interface by the class whose instances are elements of the collection
  - **Comparable** interface declare compareTo() method

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

» Compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object

» If the specified object cannot be compared to the receiving object, the method throws a **ClassCastException**

Collections - Object Ordering – Comparable interface (Example)

```
public class Name implements Comparable<Name>{
 private String firstName, lastName;
 public Name(String first, String last) {
          this.firstName= first;
          this.lastName=last;
 }
@Override
public int compareTo(Name o) {
          // TODO Auto-generated method stub
          int lastCmp=lastName.compareTo(o.lastName);
          return(lastCmp!=0?lastCmp:firstName.compareTo(o.firstName));
}
public boolean equals(Object o) {
          if(!(o instanceof Name))      return false;
          Name n=(Name)o;
          return n.firstName.equals(firstName)&&n.lastName.equals(lastName);

}
 public String getFirstName()
    { return firstName;}
```

```
public String getLastName() {
          return lastName;
 }
 public String toString() {
          return firstName + " "+lastName;
 }
}
```

Collections - Object Ordering – Comparable interface(Example)

```
public class NameSorted {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
	    ArrayList <Name> names = new ArrayList<Name>( );
	    names.add(new Name("Anne", "Smith"));
	    names.add(new Name("John", "Smith"));
	    names.add(new Name("Tom", "Rich"));
	    names.add(new Name("Karl", "Ng"));
	    Collections.sort(names);
	    System.out.println(names);
	}

}
```

>> output:

```
[Karl Ng, Tom Rich, Anne Smith, John Smith]
```

Collections - Object Ordering – Comparator interface

- Comparators
  - What if you want to sort some objects in an order other than their natural ordering?
  - Or what if you want to sort some objects that don't implement **Comparable**?
  - To do either of these things, you'll need to provide a **Comparator —** an object that encapsulates an ordering
  - **Comparator** interface consists of the following method:

```
public interface Comparator<T> {

   int compare(T o1, T o2);

}
```

Collections - Object Ordering- Comparator interface (Example)

- Suppose you have a class called **Employee**,
  - And you want to list of employees in order of seniority

```
import java.util.Date;
public class Employee
{
 private Name name;
 private Date hireDate;
 Employee(Name n, Date h){
            name=n;
           hireDate=h;
 }
 public Name getName() {
            return name;
 }
 public Date hireDate() {
           return hireDate;
 }
public String toString() {
 return name.toString()+","+ hireDate.toString();
    }
}
```

Collections - Object Ordering- Comparator interface (Example)

```
import java.util.*;
public class EmpSort {
 static final Comparator<Employee> Senority_Order= new Comparator<Employee>() {
        @Override
        public int compare(Employee o1, Employee o2) {
                return o1.hireDate().compareTo(o2.hireDate());
        } };
        public static void main(String[] args) {
                List <Employee> e = new ArrayList<Employee>();
                Employee a1=  new Employee(new Name("ana", "rus"), new Date(1990,3,15));
                Employee a2=  new Employee(new Name("ion", "rus"), new Date(1980,3,15));
                Employee a3=  new Employee(new Name("george", "trif"), new Date(1995,3,15));
                e.add( a1);
                e.add( a2);
                e.add( a3);
                Collections.sort(e,  Senority_Order);
          for(Employee m:e)
               System.out.println(m.getName()+","+m.hireDate().getYear());
    }
}
```

```
>> output:
ion rus,1980
ana rus,1990
george trif,1995
```

Map Interface

- A **Map** is an object that maps **keys** to **values**
- A map cannot contain duplicate keys:
  - Each **key** can map to at most one **value**. It models the mathematical function abstraction.
  - The **Map** interface includes methods for
    - Basic operations (e.g., **get**, **remove**, **containsKey**, **containsValue**, **size**, and **empty**)
    - Bulk operations (e.g., **putAll** and **clear**)
    - Collection views (e.g., **keySet**, **entrySet**, and **values**)

Map

- JCF offers three implementations
  - **HashMap**
  - **TreeMap –** ordered by key
  - **LinkedHashMap**
- Their behavior/performance are analogous to **HashSet**, **TreeSet**, and **LinkedHashSet**
- **Map** allows to iterate over keys, values, or key-value pairs;
- **Map** provides a safe way to remove entries in the midst of iteration

Map

```
public interface Map {
    // Basic Operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // Bulk Operations
    void putAll(Map<? extends K,? extends V> t);
    void clear();
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
   // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    } }
```

Map - Basic operations

- **Object put(Object k, Object v)**
  - Puts an entry in the invoking map, overwriting any previous value associated with the key
    - The key and value are k and v, respectively
  - Returns **null** if the key did not already exist
  - Otherwise, the previous **value** linked to the **key** is returned
- **Object get(Object k)**
  - Returns the value associated with the key **k**
- **boolean containsKey(Object k)**
  - Returns **true** if the invoking map contains **k** as a key
  - Otherwise, returns **false**

Map - Basic operations

- **boolean containsValue(Object v)**
  - Returns **true** if the map contains **v** as a value
  - Otherwise, returns **false**
- **Size**
  - Returns the number of key/value pairs in the map
- **isEmpty**
  - Returns **true** if the invoking map is empty
  - Otherwise, returns **false**

Map - Basic operations (Example)

```
import java.util.HashMap;
import java.util.Map;
public class MapDemo {
 public static void main(String[] args) {
  // TODO Auto-generated method stub
   Map<String, String>m= new HashMap <String,
String>();
   m.put("zara", "8");
   m.put("daisy", "20");
   m.put("george", "18");
   System.out.println("map elements:"+m);
         }
}
```

```
>>output:
map elements:{george=18, daisy=20, zara=8}
```

Map - Basic operations (Example)

- Example of generating a frequency table of the words found in its argument list
  - The frequency table maps each word to the number of times it occurs in the argument list

```
import java.util.*;
public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String,
                                             Integer>();
        // Populate frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }}
```

```
>>input : java Freq if it is to be it is up to me to delegate

>>output :
8 distinct words:
   {to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Map - Basic operations (Example)

- To generate the frequency table in alphabetical order
  - Change HashMap to TreeMap
  - 8 distinct words: {be =1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
- To generate the frequency table in the order the words appear on command line
  - Change HashMap to LinkedHashMap
  - 8 distinct words: {if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}

Map - Basic operations

- Two **Map** instances are equal if they represent the same key-value mappings
- By convention, all general-purpose **Map** implementations provide constructors that take a **Map** object and initialize the new **Map** to contain all the key-value mappings in the specified **Map**
  - This standard Map conversion constructor is entirely analogous to the standard **Collection** constructor:
    - It allows create a **Map** of a desired implementation type that initially contains all of the mappings in another **Map**, regardless of the other **Map's** implementation type
  - For example, suppose you have a **Map**, named ***m***. The following code creates a new **HashMap** initially containing all of the same key-value mappings as ***m***

```
Map<K, V> copy = new HashMap<K, V>(m);
```

Map - Bulk operations

- **clear**
  - Removes all mappings
- **putAll**
  - Is analogue of the **Collection** interface's **addAll** operation

Map - Bulk operations

```
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo {
  public static void main(String[] args) {
      // TODO Auto-generated method stub
        Map<String, String>m= new HashMap <String,  String>();
        Map<String, String>m1= new HashMap <String, String>();
        m.put("zara", "8");
        m.put("daisy", "20");
        m.put("george", "18");
        System.out.println("elements in map 1:"+m);
        m1.putAll(m);
        System.out.println("elements in map 2:"+m1);
          }

}
```

```
>>output
elements in map 1:{george=18, daisy=20, zara=8}
elements in map 2:{george=18, daisy=20, zara=8}
```

Map - Collection views

- These methods allow a **Map** to be viewed as a **Collection**
- **keySet**
  - Return the set of keys contained in the **Map**
- **values**
  - Return the **Collection** (not Set) of values contained in the **Map**
- **entrySet**
  - Return the **Set** of key-value pairs of the **Map**

Map - Iterating Maps

- Iterating over **keys** using **for each** loop

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

- Iterating over keys using an **Iterator**

```
// Filter a map based on some property of its keys
// removes the associated entry from the backing map
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
    if (it.next().satifyASpecifiedCondition())
        it.remove();
```

Map - Iterating Maps - Example of iterating over keys using **for each** loop

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
class MapUtils
{	public static void main (String[] args)
	{
		Map<Integer, String> map = new HashMap<>();
		map.put(1, "One");
		map.put(2, "Two");
		map.put(3, " three");
		// 2. For-each Loop
		for (Integer key : map.keySet()) {
			System.out.println(key + "=" + map.get(key));
		}
}}
```

```
>> output :
1=One
2=Two
3= three
```

Map - Iterating Maps - Example of iterating over keys using **iterator**

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
class MapUtils
{       public static void main (String[] args)
        {
                Map<Integer, String> map = new HashMap<>();
                map.put(1, "One");
                map.put(2, "Two");
                map.put(3, " three");
                // 2. For-each Loop
                Iterator <Integer>it=map.keySet().iterator();
                while(it.hasNext()) {
                   Integer key=it.next();
                        System.out.println(key + "=" + map.get(key));
      }

}
}
```

```
>> output :
1=One
2=Two
3= three
```

Map - Iterating Maps - Iterating over **key-value pairs**

- A Map is in a sense a collection of Entry object
  - Map interface contains a nested interface Entry
    - » Map.Entry is encapsulated within Map

```
interface Map<K,V>{
    static interface Map.Entry<K,V>
    …….
    }
```

  - A map entry is a key-value pair

Map - Iterating Maps - Iterating over **key-value pairs**

- Iterating over **key-value pairs** using **for each** structure

```
for (Map.Entry<KeyType, ValType> e: m.entrySet())
    System.out.println(e.getKey() + ":" + e.getValue());
```

- Iterating over **key-value** pairs using iterator

```
for (Iterator<Map.Entry<KeyType, ValType>> it = m.entrySet().iterator(); it.hasNext(); )
    Map.Entry<KeyType, ValType> pair = Map.Entry<KeyType, ValType>it.next();
        System.out.println(pair.getKey()+":"+pair.getValue());
```

Map - Iterating Maps - Iterating over **key-value pairs**

- Example of iterating over key-value pairs using **for each** structure

```
package mapDemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
public class IterateHashMap {
   public static void main (String[] args)
          {
             Map<String, String> map = new HashMap<>();
             map.put("1", "One");
             map.put("2", "Two");
             map.put("3", " Three");
             // 2. For-each Loop
              for(Map.Entry<String, String>m:map.entrySet())
                    System.out.println(m.getKey() + "=" + m.getValue());
                 }}
```

Map - Iterating Maps - Iterating over **key-value pairs**

- Example of iterating over key-value pairs using **iterator** structure

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
class IterationDemo
{
  public static void main(String[] arg)
  {
      Map<String,String> map = new
                                  HashMap<String,String>();
      map.put(“Ana", “Alba-Iulia");
      map.put(“Victor", “Cluj");
      map.put(“Horia", “Alba-Iulia");
      map.put(“Natalia", “Borsa");
```

```
// using iterators
 Iterator<Map.Entry<String, String>> itr = map.entrySet().iterator();
       while(itr.hasNext())
          {
             Map.Entry<String, String> entry = itr.next();
             System.out.println( "Key = " + entry.getKey() +
                                          ", Value = " + entry.getValue());
                    }
           }
      }
```

Maps - Complex operations

- Non-destructive operations (i.e., they don't modify the backing Map)
  - Testing if map m_2 is submap of m_1 (i.e., m_1 contains all (key-value) pairs of m_2)

```
if (m1.entrySet().containsAll(m2.entrySet())) {
  ...
}
```

  - Testing if two **Map** objects contain mappings for all of the same keys

```
if (m1.keySet().equals(m2.keySet())) { ... }
```

  - All keys common to two **Map** objects

```
Set<KeyType>commonKeys = new HashSet<KeyType>(m1.keySet());
commonKeys.retainAll(m2.keySet());
```

Maps - Destructive operations (Example)

```
import java.util.*;
public class GfG {
   // Driver code
    public static void main(String[] args)
    {
        // Initializing a Map of type HashMap
        Map<String, String> map = new HashMap<>();
        map.put("1", "One");
        map.put("3", "Three");
        map.put("5", "Five");
        map.put("7", "Seven");
        map.put("9", "Nine");
        System.out.println(map);
        map.remove("3");
        System.out.println(map);
    } }
```

Output:

```
{1=One, 3=Three, 5=Five, 7=Seven, 9=Nine}
{1=One, 5=Five, 7=Seven, 9=Nine}
```

Maps - Multimaps

- Multimap
  - Maps each key to multiple values
  - How to implement mutimaps
    - values as **List** instances
- Example – AnagramFinder (from java.sun.com site)
  - Read a word
  - Find anagram for it (in a given dictionary - one word per line)

Maps – Multimaps (Example)

```
import java.util.*;
import java.io.*;
public class AnagramFinder {
    public static void main(String[] args) {
      String myWord = alphabetize(args[0]);
      Map<String, List<String>> m =
             new HashMap<String, List<String>>();
       try {
            Scanner s = new Scanner(new
                                        File("dictionary.txt"));
            while (s.hasNext()) {
              String word = s.next();
              String alpha = alphabetize(word);
              List<string> l = m.get(alpha);
              if (l == null)
                  m.put(alpha, l=new ArrayList<String>());
              l.add(word);
            }
       }
```

```
catch (IOException e) {
            System.err.println(e);
            System.exit(1);
       }

for (String myKey: m.keySet()) {
             if (myWord.equals(myKey)) {
                  System.out.println("\nFound a match for
                                              argument " +  args[0]);
                  System.out.println(m.get(myKey));
             }
         }
    }
    private static String alphabetize(String s) {
         char[] a = s.toCharArray();
         Arrays.sort(a);
         return new String(a);
    }
}
```

Sorted Maps

- Map that maintains its entries sorted
- Order
  - Key natural order
  - Comparator provided at construction time
- Additional Operations (there are analog with **SortedSet** operations)

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Sorted Maps

- comparator()
  - Returns the invoking sorted map's comparator
  - If the natural ordering is used for the invoking map, null is returned
- subMap(K fromKey, K toKey)
  - Returns a map containing those entries with keys that are greater than or equal to start and less than end
- headMap(K toKey)
  - Returns a sorted map for those map entries with keys that are less than toKey

Sorted Maps

- tailMap(K fromKey)
  - Returns a map containing those entries with keys that are greater than or equal to fromKey
- firstKey()
  - Returns the first key in the invoking map
- K lastKey()
  - Returns the last key in the invoking map

Sorted Maps

- **Inherited Operations**
  - Similar behavior with two exceptions
    - »Iterator returned by the **iterator** operation on any of the sorted map's Collection views traverse the collections in order
    - »The arrays returned by the Collection views' toArray operations contain the keys, values, or entries in order
- **Constructors**
  - All implementations of Map define a standard conversion constructor that takes a Map parameter
  - **SortedMap** implementation (e.g., TreeMap class) should also provide a constructor that takes a **Comparator** parameter and returns an empty sorted map
    - »If **null** is passed as **Comparator**, the mapping keys will be sorted according to natural ordering

Sorted Maps

- SortedMap has its implementation in various classes like TreeMap

```
public class TreeMapDemo {
   public static void main(String args[]) {
    TreeMap tm = new TreeMap();
    tm.put(„milk", new Double(34.34));
    tm.put(„potatoes", new Double(12.22));
    tm.put(„bread", new Double(13.00));
    // Get a set of the entries
    Set set = tm.entrySet();
    // Get an iterator
    Iterator i = set.iterator();
    // Display elements
    while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();
```

```
>> output
bread: 13.0
milk: 34.34
potatoes: 12.22
```

Stack class

- Is the subclass of Vector
- Implements the last-in-first-out data structure (a stack structure)
- Contains all the methods of Vector class and also provides its methods:
  - public E peek()
    - » Returns the object at the top of the stack without removing it from the stack
  - Public E pop()
    - » Removes the object at the top of this stack and returns that object as the value of this function
  - boolean push(E item)
    - » Pushes an item onto the top of this stack

Stack - Example

```
import java.util.*;
 public class StackDemo{
 public static void main(String args[]) {
 Stack<String >st= new Stack<String>();
 st.push("ana");
 st.push("dana");
 st.push("luca");
 st.push("victor");
 st.pop();
 Iterator <String>it =st.iterator();
 while(it.hasNext())
        System.out.println(it.next());
 }
  }
```

```
>>output:
ana
dana
luca
```

Queue Interface

- Queue interface maintains the first-in-first-out order
- Classes which implements the Queue interface
  - PriorityQueue
  - Deque
  - ArrayDeque

Queue Interface

- Some methods defined in the interface:
  - E element()
    - » Retrieves, but does not remove, the head of this queue
  - E peek()
    - » Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty
  - E poll()
    - » Retrieves and removes the head of this queue, or returns null if this queue is empty
  - E remove()
    - » Retrieves and removes the head of this queue

Queue Interface

- Queue interface can be instantiated as:
  - Queue<String> q1 = **new** PriorityQueue();
  - Queue<String> q2 = **new** ArrayDeque();
- PriorityQueue class
  - Implements the Queue interface
  - Holds the elements or objects which are processed by their priorities
    - The PriorityQueue is based on the priority heap
    - The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used
  - Doesn't allow null values to be stored in the queue

PriorityQueue class -Example

```
import java.util.*;
public class PriorityQueueDemo {
 public static void main(String[] args) {
  // TODO Auto-generated method stub
  PriorityQueue<String> queue=new PriorityQueue<String>();
  queue.add("Ana Dragan");
  queue.add("Dan Marginean");
  queue.add("Alex Pop");
  queue.add("George Ionescu");
  System.out.println("head:"+queue.element());
  System.out.println("head:"+queue.peek());
  System.out.println("iterating the queue elements:");
  Iterator itr=queue.iterator();
      while(itr.hasNext())
         System.out.println(itr.next());
      queue.remove();
      queue.poll();
      System.out.println("after removing elements:");
      Iterator itr1=queue.iterator();
         while(itr1.hasNext())
        System.out.println(itr1.next());         }}
```

```
>>output:
head:Alex Pop
head:Alex Pop
iterating the queue elements:
Alex Pop
Dan Marginean
Ana Dragan
George Ionescu
after removing elements:
Dan Marginean
George Ionescu
```

Deque Interface

- Deque interface extends the Queue interface
- In Deque, we can remove and add the elements from both the side
- Deque can be instantiated as:
  - Deque d = **new** ArrayDeque();
- ArrayDeque class
  - Implements the Deque interface
  - Unlike queue, we can add or delete the elements from both the ends
  - Is faster than ArrayList and Stack and has no capacity restrictions

ArrayDeque - Example

```
import java.util.ArrayDeque;
import java.util.Deque;
public class ArrayDequeDemo {
 public static void main(String[] args) {
  // TODO Auto-generated method stub
 Deque<String> deque = new ArrayDeque<String>();
 deque.add("ana");
 deque.add("dan");
 deque.add("george");
 //Traversing elements
 for(String str :deque)
  System.out.println(str);
}
}
```

```
>> output
ana
dan
george
```