

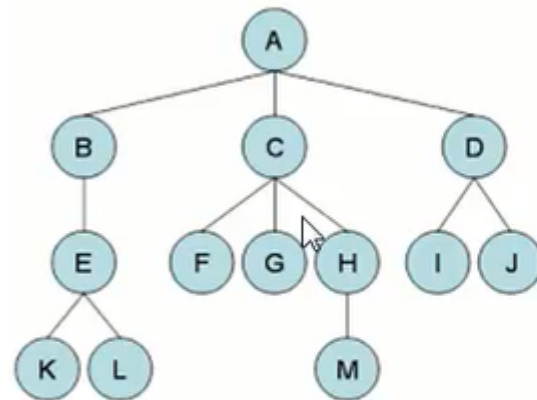
Java Collections

- **A data structure**

- specialized format for organizing and storing **data**.

- **General data structure**

- array, the file, the record, the table, the tree, and so on.

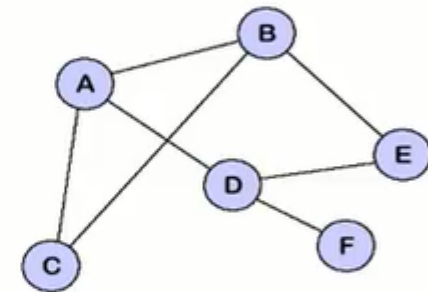
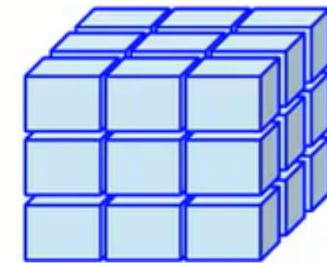


Linked list

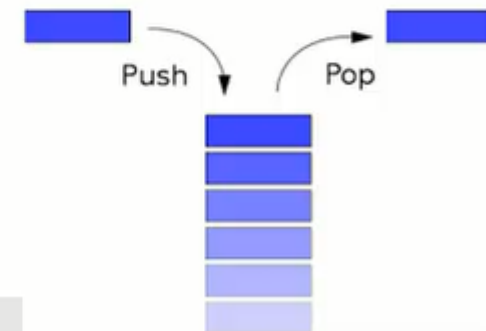
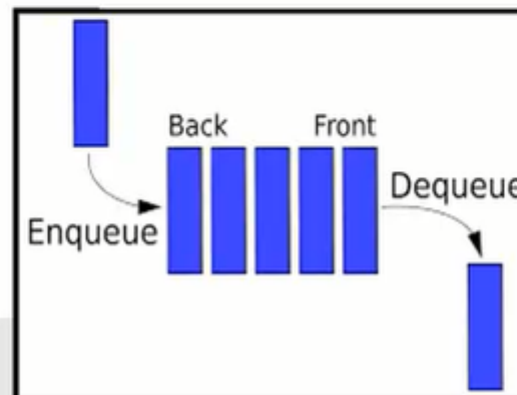
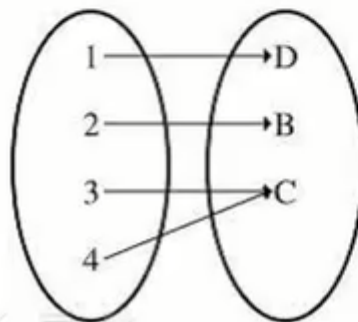


data format

Array



Key Value



Java Collections Framework

- A collection (sometimes called a *container*) is an object that groups multiple elements into a single unit.
- A collection framework is a unified architecture for representing and manipulating collections.
- Collection let you *store, organize* and *access* objects in an efficient manner.
- The Collections Framework is a sophisticated hierarchy of *interfaces, classes and algorithms* that provide state-of-the-art technology for managing groups of objects.

Java Collections Framework

- **Interfaces:**

- Interface in Java refers to the abstract data types.
- They allow Java collections to be manipulated independently from the details of their representation.

- **Classes:**

- Classes in Java are the implementation of the collection interface. It basically represents data structures that are used again and again.

- **Algorithm:**

- Algorithm refers to the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces.

Collection

Activity using Java collection framework-defined in collection interface

- Add objects to collection
- Remove objects from collection
- Search for an object in collection
- Retrieve/get object from collection
- Iterate through the collection

The Collection Interface-common methods

- The Collection interface provides the basis for List-like collections in Java. The interface includes:

boolean add(Object)

boolean addAll(Collection)

void clear()

boolean contains(Object)

boolean containsAll(Collection)

boolean equals(Object)

boolean isEmpty()

Iterator iterator()

boolean remove(Object)

boolean removeAll(Collection)

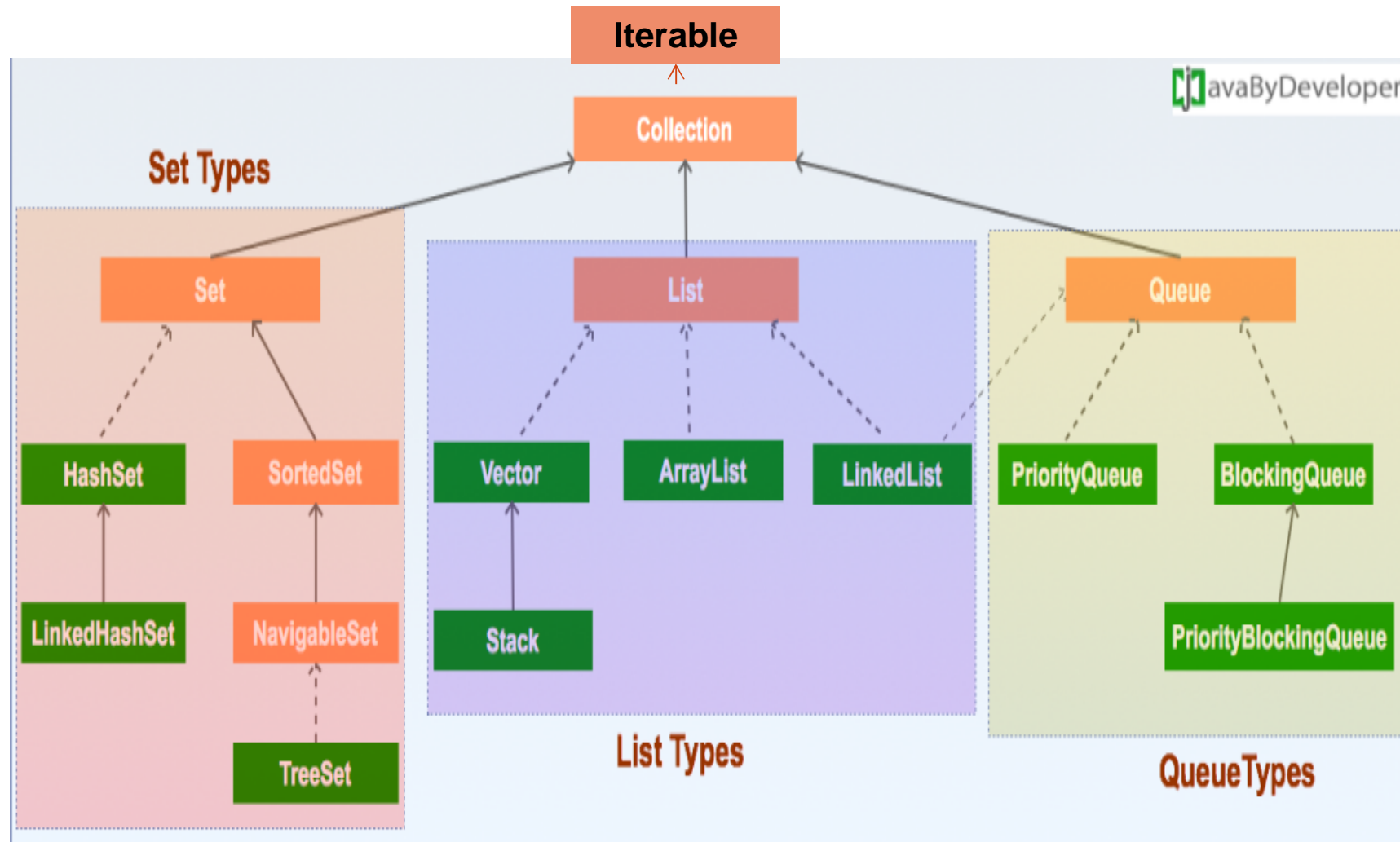
boolean retainAll(Collection)

int size()

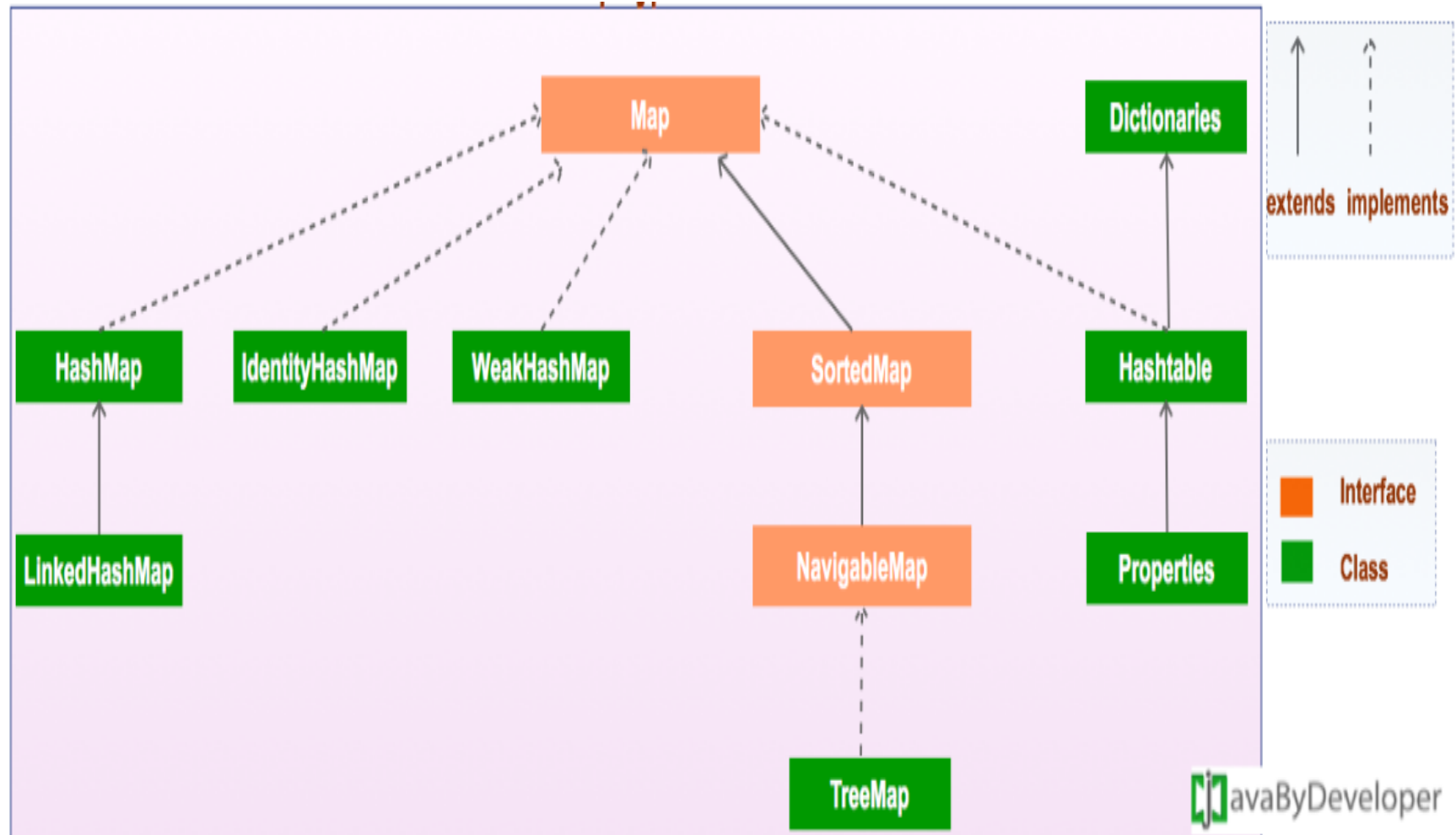
Object[] toArray()

Object[] toArray(Object[])

Collection Interface



Map Interface



List, Set , Queue , Map

- List allows duplicates.
- Set can contain unique items.
- Set can have ordered items.
- Map allows data to be stored as a key value pair.

Iterator

- An **Iterator** is an object that enables traversing through a **collection** and to remove elements from the **collection** selectively, if required
- **Iterator** for a **collection** can be obtained by calling its **iterator** method.
- All collections interfaces inherit from `Iterable` enabling the object to target of for-each loop.
- Each collection class provides specific implementation of iterators.
- Example
 - `ListIterator`
 - `DescendingIterator` etc..

Iterator Interface

```
public interface Iterator<T> {  
    public boolean hasNext( );  
    public T next( );  
    public void remove( ); // (optional operation)  
}
```

It is in the `java.util` package of the Java API

Iterator Interface Methods

- The Iterator interface 3 methods we will use:

```
boolean hasNext( )
```

```
//returns true if this iteration has more elements
```

```
E next( )
```

```
//returns the next element in this iteration
```

```
//pre: hasNext()
```

```
void remove( )
```

```
/*Removes from the underlying collection the last  
element returned by the iterator.
```

```
pre: This method can be called only once per call to  
next. After calling, must call next again before calling  
remove again.
```

CS314

```
*/
```

Iterators

A note on ListIterators

- 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.
- The nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous
- The set method overwrites the last element returned by next or previous with the specified element.
- The add method inserts a new element into the list immediately before the current cursor position.

```
public interface ListIterator<E> extends Iterator<E> {  
    boolean hasNext();  
    E next();  
    boolean hasPrevious();  
    E previous();  
    int nextIndex();  
    int previousIndex();  
    void remove(); //optional  
    void set(E e); //optional  
    void add(E e); //optional  
}
```



Collections Classes

- **Collections** class is one of the utility classes in Java Collections Framework.
- The java.util package contains the Collections class. Collections class is basically used with the static methods that operate on the collections or return the collection.
- All the methods of this class throw the **NullPointerException** if the collection or object passed to the methods is null.

Algorithms

- **Algorithms** are another important part of the collection mechanism.
- Algorithms operate on collections and are defined as **static methods** within the Collections class.
- Thus, they are available for all collections
- An **iterator** gives you a general-purpose, standardized way of accessing the elements within a collection, one at a time.
- Thus, an iterator provides a means of enumerating the **contents of a collection**.
- Because each collection implements Iterator, the elements of any collection class can be accessed through the methods defined by Iterator.

algorithms

- The collections framework also provides polymorphic versions of algorithms you can run on collections.
 - Sorting
 - Shuffling
 - Routine Data Manipulation
 - Reverse
 - Fill copy
 - etc.
 - Searching
 - Binary Search
 - Composition
 - Frequency
 - Disjoint
 - Finding extreme values
 - Min
 - Max

```
Collections.shuffle( list );  
Collections.sort( list );  
Collections.copy( copyList, list );  
result = Collections.binarySearch( list, key );  
Collections.max( listRef )  
Collections.fill( list, "R" );
```

Generics

- Concept of any collection is same irrespective of its types stored.
 - A list of names (string)
 - A set of numbers (int)
 - A list of employees (object).
- Till jdk 1.5 Collection framework provided classes whose methods accepted only parameters of type `java.lang.Object`.
- This required casting data back to respective type when extracting values from collection.
 - The process was error prone and there was a performance overhead.
- Jdk 1.5 introduced generics. It allowed collections to be created for specific type of values.
- Generics allows to abstract over types.

Generics are described as follows:

- Provide compile-time type safety
- Eliminate the need for casts
- Provide the ability to create compiler-checked homogeneous collections

Collections Overview-without Generics

1st Example:

```
static public void main(String[] args) {  
    ArrayList argsList = new ArrayList();  
    for(String str : args) {  
        argsList.add(str);  
    }  
    if(argsList.contains("Koko") {  
        System.out.println("We have Koko");  
    }  
    String first = (String)argsList.get(0);  
    System.out.println("First: " + first);  
}
```

Collections Overview-with Generics

2nd Example – now with Generics:

```
static public void main(String[] args) {  
    ArrayList<String> argsList =  
        new ArrayList<String>();  
    for(String str : args) {  
        argsList.add(str); // argsList.add(7) would fail  
    }  
    if(argsList.contains("Koko") {  
        System.out.println("We have Koko");  
    }  
    String first = argsList.get(0); // no casting!  
    System.out.println("First: " + first);  
}
```

Generics

Generics are a way to define which types are allowed in your class or function

// old way

```
List myIntList1 = new LinkedList(); // 1
```

```
myIntList1.add(new Integer(0)); // 2
```

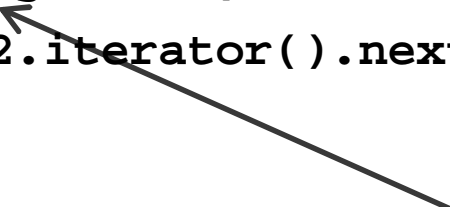
```
Integer x1 = (Integer) myIntList1.iterator().next(); // 3
```

// with generics

```
List<Integer> myIntList2 = new LinkedList<Integer>(); // 1'
```

```
myIntList2.add(new Integer(0)); // 2'
```

```
Integer x2 = myIntList2.iterator().next(); // 3'
```



Can put here just 0,
using autoboxing

Generics

Example 1 – Defining Generic Types:

```
public interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}
```

```
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

```
public interface Map<K,V> {  
    V put(K key, V value);  
}
```

Generics

Example 2 – Defining (our own) Generic Types:

```
public class GenericClass<T> {  
    private T obj;  
    public void setObj(T t) {obj = t;}  
    public T getObj() {return obj;}  
    public void print() {  
        System.out.println(obj);  
    }  
}
```

Main:

```
GenericClass<Integer> g = new GenericClass<Integer>();  
g.setObj(5); // auto-boxing  
int i = g.getObj(); // auto-unboxing  
g.print();
```

Lists

- List
 - Ordered Collection that can contain duplicate elements
 - Sometimes called a *sequence*
 - Implemented via interface `List`
 - `ArrayList` (resizable array)
 - `LinkedList`
 - `Vector`
 - `List` method `Iterator` is a bidirectional iterator
 - `Iterator` parameter (if used) tells where to start iterating

public interface **List**<E>
extends Collection<E>

- List — an ordered collection (sometimes called a *sequence*).
- Lists can contain **duplicate elements**.
- List class maintains **insertion order**.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

public interface **List** extends Collection

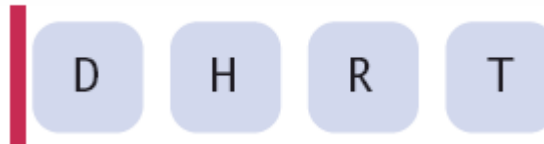
```
public interface List<E> extends Collection<E> {  
    // Positional access  
    E get(int index);  
    E set(int index, E element);           //optional  
    boolean add(E element);               //optional  
    void add(int index, E element);        //optional  
    E remove(int index);                  //optional  
    boolean addAll(int index,  
        Collection<? extends E> c); //optional  
  
    // Search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    // Iteration  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
  
    // Range-view  
    List<E> subList(int from, int to);  
}
```

List Iterators

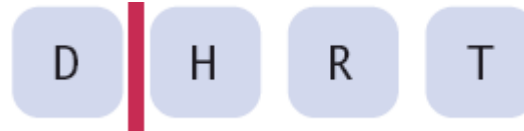
- Think of an iterator as pointing **between** two elements (think of cursor in word processor)

```
ListIterator<String> iterator = myList.listIterator()
```

Initial ListIterator position



```
iterator.next();
```



```
iterator.add("J");
```



Iterators and Loops

- Iterators are often used in **while** and “**for-each**” loops
hasNext returns true if **there is a next element**
next returns a **reference** to the value of the **next element**

```
while (iterator.hasNext())  
{  
    String name = iterator.next();  
    // Do something with name  
}
```

```
for (String name : employeeNames)  
{  
    // Do something with name  
}
```

```
while (iterator.hasPrevious())  
{  
    String name =  
        iterator.previous();  
    // Do something with name  
}
```

Adding and Removing with Iterators

- **Adding** `iterator.add("Juliet");`
 - A new node is added **AFTER** the Iterator
 - The Iterator is **moved past the new node**
- **Removing**
 - Removes the object that was returned with **the last call** to **next** or **previous**
 - It can be **called only once** after **next** or **previous**
 - You **cannot call it immediately after a call to add**

If you call the remove method improperly, it throws an `IllegalStateException`.

```
while (iterator.hasNext())
{
    String name = iterator.next();
    if (condition is true for name)
    {
        iterator.remove();
    }
}
```

public interface **Set**<E> extends Collection<E>

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);           //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                               //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Adding and Removing with Iterators

- **Adding** `iterator.add("Juliet");`
 - A new node is added **AFTER** the Iterator
 - The Iterator is **moved past the new node**
- **Removing**
 - Removes the object that was returned with **the last call** to **next** or **previous**
 - It can be **called only once** after **next** or **previous**
 - You **cannot call it immediately after a call to add.**(why?)

If you call the remove method improperly, it throws an `IllegalStateException`.

```
while (iterator.hasNext())
{
    String name = iterator.next();
    if (condition is true for name)
    {
        iterator.remove();
    }
}
```

Page 31

```
import java.awt.Color;
4  import java.util.*;
5
6  public class CollectionTest {
7      private static final String colors[] = { "red", "white", "blue" };
8
9      // create ArrayList, add objects to it and manipulate it
10     public CollectionTest()
11     {
12         List list = new ArrayList();
13
14         // add objects to list
15         list.add( Color.MAGENTA );    // add a color object
16
17         for ( int count = 0; count < colors.length; count++ )
18             list.add( colors[ count ] );
19
20         list.add( Color.CYAN );        // add a color object
21
22         // output list contents
23         System.out.println( "\nArrayList: " );
24
25         for ( int count = 0; count < list.size(); count++ )
26             System.out.print( list.get( count ) + " " );
27
```



```

•// remove all String objects
•29     removeStrings( list );
•31     // output list contents
•32     System.out.println( "\n\nArrayList after calling removeStrings: " );
•33
•34     for ( int count = 0; count < list.size(); count++ )
•35         System.out.print( list.get( count ) + " " );
•36
•37 } // end constructor CollectionTest
•39 // remove String objects from Collection
•40 private void removeStrings( Collection collection )
•41 {
•42     Iterator iterator = collection.iterator(); // get iterator
•43
•44     // loop while collection has items
•45     while ( iterator.hasNext() )
•46
•47         if ( iterator.next() instanceof String )
•48             iterator.remove(); // remove String object
•49 }
• public static void main( String args[] )
• 52     {
• 53         new CollectionTest();
• 54     } } //

```

ArrayList within ArrayList

```
// Java code to demonstrate the concept of  
// array of ArrayList
```

```
import java.util.*;  
public class Arraylist {  
    public static void main(String[] args)  
    {  
        int n = 3;  
  
        // Here aList is an ArrayList of ArrayLists  
        ArrayList<ArrayList<Integer> > aList =  
            new ArrayList<ArrayList<Integer> >(n);  
  
        // Create n lists one by one and append to the  
        // master list (ArrayList of ArrayList)  
        ArrayList<Integer> a1 = new ArrayList<Integer>();  
        a1.add(1);  
        a1.add(2);  
        aList.add(a1);  
  
        ArrayList<Integer> a2 = new ArrayList<Integer>();  
        a2.add(5);  
        aList.add(a2);  
    }  
}
```

```
ArrayList<Integer> a3 = new ArrayList<Integer>();
    a3.add(10);
    a3.add(20);
    a3.add(30);
    aList.add(a3);

    for (int i = 0; i < aList.size(); i++) {
        for (int j = 0; j < aList.get(i).size(); j++) {
            System.out.print(aList.get(i).get(j) + "
");
        }
        System.out.println();
    }
}
```

Problem

Input

5

5 41 77 74 22 44

1 12

4 37 34 36 52

0

3 20 22 33

5

1 3

3 4

3 1

4 3

5 5 // end class SetTest

Sample Output

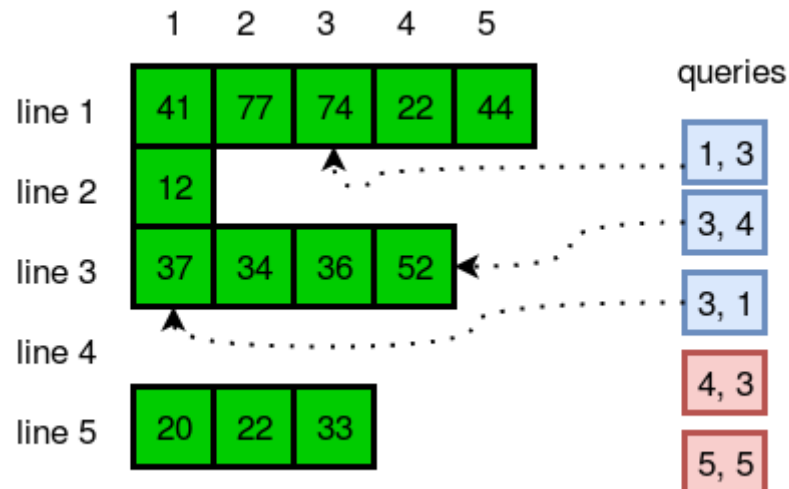
74

52

37

ERROR!

ERROR!



Problem

- **Sample Input 1:**
- 2
- 5 9
- 1 3 3 10 2
- 6 20
- 2 4 6 3 1 1
- **Sample Output 1:**
- Yes
- No

Problem

- Given an array “A” of positive integers. Your task is to make the largest number possible formed by concatenating each of the array elements exactly once.
- **Example:**
- Let's say the given array is [9, 98, 7]. All the possible numbers formed by concatenating each of the array elements are 7989,7998,9879,9897,9987,9798. Among the six numbers, 9987 is the greatest number. Hence the answer is 9987.
- Given an integer array ‘ARR’ of length ‘N’, return the number of longest increasing subsequences in it.
- The longest increasing subsequence(LIS) is the longest subsequence of the given sequence such that all elements of the subsequence are in increasing order.
- Let ‘ARR’ = [50, 3, 90, 60, 80]. In this array the longest increasing subsequences are [50, 60, 80] and [3, 60, 80].

Linked List

- **Java LinkedList** class is doubly-linked list implementation of the List
- Permits all elements including duplicates and NULL.
- LinkedList maintains the **insertion order** of the elements.

Linked List Methods

- **boolean add(Object o)** : appends the specified element to the end of a list.
- **void add(int index, Object element)** : inserts the specified element at the specified position index in a list.
- **void addFirst(Object o)** : inserts the given element at the beginning of a list.
- **void addLast(Object o)** : appends the given element to the end of a list.
- **int size()** : returns the number of elements in a list
- **boolean contains(Object o)** : return true if the list contains a specified element, else false.
- **boolean remove(Object o)** : removes the first occurrence of the specified element in a list.
- **Object getFirst()** : returns the first element in a list.
- **Object getLast()** : returns the last element in a list.
- **int indexOf(Object o)** : returns the index in a list of the first occurrence of the specified element, or -1 if the list does not contain specified element.
- **lastIndexOf(Object o)** : returns the index in a list of the last occurrence of the specified element, or -1 if the list does not contain specified element.
- **Iterator iterator()** : returns an iterator over the elements in this list in proper sequence.
- **Object[] toArray()** : returns an array containing all of the elements in this list in proper sequence.
- **List subList(int fromIndex, int toIndex)** : returns a view of the portion of this list between the specified fromIndex (inclusive) and toIndex (exclusive).


```
•3  import java.util.*;
•4
•5  public class ListTest {
•6      private static final String colors[] = { "black", "yellow",
•7          "green", "blue", "violet", "silver" };
•8      private static final String colors2[] = { "gold", "white",
•9          "brown", "blue", "gray", "silver" };
•10
•11      // set up and manipulate LinkedList objects
•12      public ListTest()
•13      {
•14          List link = new LinkedList();
•15          List link2 = new LinkedList();
•16
•17          // add elements to each list
•18          for ( int count = 0; count < colors.length; count++ ) {
•19              link.add( colors[ count ] );
•20              link2.add( colors2[ count ] );
•21          }
•22
•23          link.addAll( link2 );          // concatenate lists
•24          link2 = null;                 // release resources
```

```
printList( link );
27
28     uppercaseStrings( link );
29
30     printList( link );
31
32     System.out.print( "\nDeleting elements 4 to 6..." );
33     removeItems( link, 4, 7 );
34
35     printList( link );
36
37     printReversedList( link );
38
39 } // end constructor ListTest
40
41 // output List contents
42 public void printList( List list )
43 {
44     System.out.println( "\nlist: " );
45
46     for ( int count = 0; count < list.size(); count++ )
47 System.out.print( list.get( count ) + " " );
48 System.out.println();
49 }
50 }
```

```
•// locate String objects and convert to uppercase
•53     private void uppercaseStrings( List list )
•54     {
•55         ListIterator iterator = list.listIterator();
•56
•57         while ( iterator.hasNext() ) {
•58             Object object = iterator.next(); // get item
•59
•60             if ( object instanceof String ) // check for String
•61                 iterator.set( ( ( String ) object ).toUpperCase() );
•62         }
•63     }
•64
•65     // obtain sublist and use clear method to delete sublist items
•66     private void removeItems( List list, int start, int end )
•67     {
•68         list.subList( start, end ).clear(); // remove items
•69     }
•70
•71     // print reversed list
•72     private void printReversedList( List list )
•73     {
74         ListIterator iterator = list.listIterator( list.size() );
```

- `System.out.println("\nReversed List:");`
- `77`
- `78 // print list in reverse order`
- `79 while(iterator.hasPrevious())`
- `80 System.out.print(iterator.previous() + " ");`
- `81 }`
- `83 public static void main(String args[])`
- `84 {`
- `85 new ListTest();`
- `86 }`
- `88 } // end class ListTest`

```
list:
black yellow green blue violet silver gold white brown blue gray silver
```

```
list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER
```

```
Deleting elements 4 to 6...
```

```
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER
```

```
Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

```

•import java.util.*;
public class UsingToArray {
    // create LinkedList, add elements and convert to array
    public UsingToArray() {
•10        String colors[] = { "black", "blue", "yellow" };
•12        LinkedList links = new LinkedList( Arrays.asList( colors ) );
•14        links.addLast( "red" ); // add as last item
•15        links.add( "pink" ); // add to the end
•16        links.add( 3, "green" ); // add at 3rd index
•17        links.addFirst( "cyan" ); // add as first item
•19        // get LinkedList elements as an array
•20        colors = ( String [] ) links.toArray( new String[ links.size() ] );
•22        System.out.println( "colors: " );
•    for ( int count = 0; count < colors.length; count++ )
•        25            System.out.println( colors[ count ] );
•        26    }
•        28    public static void main( String args[] )
•        29    {
•        30        new UsingToArray();
•        31    } colors:
•
•        33    } // end class UsingToArray

```

```

cyan
black
blue
yellow
green
red
pink

```

```

•// locate String objects and convert to uppercase
•53     private void uppercaseStrings( List list )
•54     {
•55         ListIterator iterator = list.listIterator();
•56
•57         while ( iterator.hasNext() ) {
•58             Object object = iterator.next(); // get item
•59
•60             if ( object instanceof String ) // check for String
•61                 iterator.set( ( ( String ) object ).toUpperCase() );
•62         }
•63     }
•64
•65     // obtain sublist and use clear method to delete sublist items
•66     private void removeItems( List list, int start, int end )
•67     {
•68         list.subList( start, end ).clear(); // remove items
•69     }
•70
•71     // print reversed list
•72     private void printReversedList( List list )
•73     {
74         ListIterator iterator = list.listIterator( list.size() );

```

Problem

- Detect a loop in linked list.

Synchronized Collections

- Except for **Vector and Hashtable**, the collections in the collections framework are unsynchronized by default, so they can operate efficiently when multithreading is not required.
- Because they're unsynchronized, however, concurrent access to a Collection by multiple threads could cause **indeterminate results**

The Vector class

- In java.util
 - Must put “import java.util.*;” in the java file
- Probably the most useful class in the library (in my opinion)
- The Vector class implements a growable array of objects.
- A Vector is a collection of “things” (objects)
 - It has nothing to do with the geometric vector
 - it is rarely used in a non-thread environment as it is **synchronized**, and due to this, it gives a poor performance in adding, searching, deleting, and updating its elements.

Vector methods

- Constructor: `Vector()`
- Adding objects: `add (Object o);`
- Removing objects: `remove (int which)`
- Number of elements: `size()`
- Element access: `elementAt()`
- Removing all elements: `clear()`

Vector code example

```
Vector v = new Vector();  
System.out.println (v.size() + " " + v);      0 []  
  
v.add ("first");  
System.out.println (v.size() + " " + v);      1 [first]  
  
v.add ("second");  
v.add ("third");  
System.out.println (v.size() + " " + v);      3 [first, second, third]  
  
String s = (String) v.elementAt (2);  
System.out.println (s);                       third  
  
String t = (String) v.elementAt (3);         (Exception)  
System.out.println (t);  
  
v.remove (1);  
System.out.println (v.size() + " " + v);      2 [first, third]  
  
v.clear();  
System.out.println (v.size() + " " + v);      0 []
```

Vector within vector

- `import java.util.*;`
- `public class Main {`
- `public static void main(String args[]) {`
- `//define and initialize a vector`
- `Vector inner_vec = new Vector();`
- `inner_vec.add("Software");`
- `inner_vec.add("Testing");`
- `inner_vec.add("Java");`
- `inner_vec.add("Tutorials");`
-
- `//define another vector and add first vector to it.`
- `Vector outer_vec = new Vector();`
- `outer_vec.add(inner_vec);`
- `String str;`
- `//display the contents of vector of vectors`
- `System.out.println("Contents of vector of vectors:");`
- `for(int i=0;i<inner_vec.size();i++){`
- `str = (String) ((Vector) outer_vec.get(0)).get(i);`
- `System.out.print(str + " ");`
- `} }`

Stack

- The **Stack<E>** data type and its four methods:
 - **push(E)**, **pop()**, **peek()**, and **empty()**
- How the Java libraries implement **Stack**
- A stack allows access only to the top element
- A stack's storage policy is Last-In, First-Out
- How to implement **Stack** using:
 - An array
 - A linked list
- Using **Stack** in applications-
 - Finding palindromes
 - Testing for balanced (properly nested) parentheses
 - Evaluating arithmetic expressions

```
import java.util.Stack;
import java.util.EmptyStackException;

public class StackTest
{
    public static void main(String[] args)
    {
        Stack<Number> stack = new Stack<>(); // create a Stack

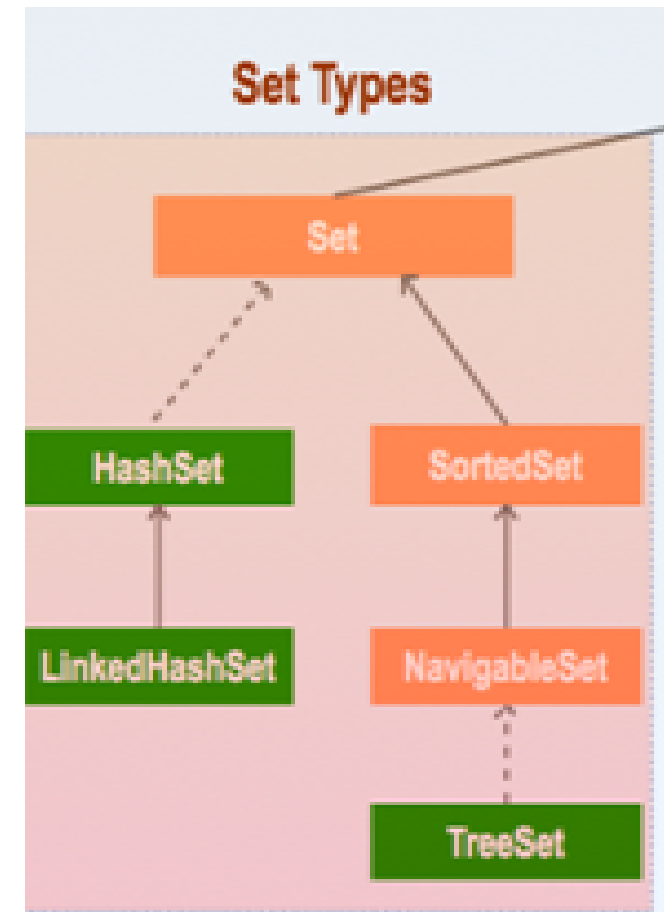
        // use push method
        stack.push(12L); // push long value 12L
        System.out.println("Pushed 12L");
        printStack(stack);
        stack.push(34567); // push int value 34567
        System.out.println("Pushed 34567");
        printStack(stack);
        stack.push(1.0F); // push float value 1.0F
        System.out.println("Pushed 1.0F");
        printStack(stack);
        stack.push(1234.5678); // push double value 1234.5678
        System.out.println("Pushed 1234.5678 ");
        printStack(stack);
    }
}
```

```
// remove items from stack
try
{
    Number removedObject = null;

    // pop elements from stack
    while (true)
    {
        removedObject = stack.pop(); // use pop method
        System.out.printf("Popped %s\n", removedObject);
        printStack(stack);
    }
}
catch (EmptyStackException emptyStackException)
{
}
```

```
private static void printStack(Stack<Number> stack)
{
    if (stack.isEmpty())
        System.out.printf("stack is empty\n\n"); // the stack is empty
    else // stack is not empty
        System.out.printf("stack contains: %s (top)\n", stack);
}
```

Set



SET

- Set is an interface which extends Collection. It is an **unordered collection of objects in which duplicate values cannot be stored.**
- Basically, Set is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).
- Set
 - Collection that contains unique elements (no duplicates)
 - HashSet (implements Set)
 - Stores elements in hash table (order determined by hashing algorithm)
 - TreeSet (implements SortedSet)
 - Stores elements in tree (sorted order)
 - `headSet(x)` returns subset with every element before x
 - `tailSet(x)` returns subset with every element including and after x

Set implementation

- in Java, sets are represented by Set interface in `java.util`
- Set is implemented by `HashSet` and `TreeSet` classes
 - `HashSet`: implemented using a "hash table";
extremely fast for all operations
elements are stored in unpredictable order
 - `TreeSet`: implemented using a "binary search tree";
very fast for all operations
elements are stored in sorted order

```
Set<Integer> numbers = new TreeSet<Integer>();  
Set<String> words = new HashSet<String>();
```

Operations

- `set1.equals(set2)` - $\text{set1} \subseteq \text{set2}$
- `set1 ∪ set2` - `set1.addAll(set2)`
- `set1 ∩ set2` - `set1.retainAll(set2)`
- `set1 - set2` - `set1.removeAll(set2)`

public interface **Set**<E> extends Collection<E>

```
public interface Set<E> extends Collection<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);           //optional  
    boolean remove(Object element);  //optional  
    Iterator<E> iterator();  
  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c); //optional  
    boolean removeAll(Collection<?> c);       //optional  
    boolean retainAll(Collection<?> c);       //optional  
    void clear();                             //optional  
  
    // Array Operations  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

Note: nothing added to Collection interface – except no duplicates allowed

Types of Sets

- HashSet
- TreeSet
- LinkedHashSet

HashSet

- Implements the Set interface, backed by a hash table (actually a HashMap instance).
- Makes no guarantees as to the **iteration order** of the set.
- HashSet doesn't maintain any kind of order of its elements.
- It does not guarantee **that the order will remain constant over time.**
- **No duplicates**
- **Permits the null element**

.

LinkedHashSet

- Extends HashSet.
- Implements doubly-linked list.
- Can retrieve elements based on insertion-order.
- Less efficient than HashSet.
- Elements gets sorted in the same sequence in which they have been added to the Set.

TreeSet

- Extends AbstractSet, implements SortedSet
- Elements can be kept in ascending order
- Elements are re-ordered upon insertion- natural ordering of a tree..
 - It does not allow to store the null elements.
 - It is a non-synchronized class.
- The data structure for the TreeSet is TreeMap; it contains a **SortedSet** & **NavigableSet** interface to keep the elements sorted in ascending order and navigated through the tree.

The "for each" loop

```
for (type name : collection) {  
    statements;  
}
```

- Provides a clean syntax for looping over the elements of a Set, List, array, or other collection

```
Set<Double> grades = new HashSet<Double>();  
...
```

```
for (double grade : grades) {  
    System.out.println("Student's grade: " + grade);  
}
```

- needed because sets have no indexes; can't get element i

```

3  import java.util.*;
5  public class SetTest {
6      private static final String colors[] = { "red", "white", "blue",
7          "green", "gray", "orange", "tan", "white", "cyan",
8          "peach", "gray", "orange" };

10     // create and output ArrayList
11     public SetTest()
12     {
13         List list = new ArrayList( Arrays.asList( colors ) );
14         System.out.println( "ArrayList: " + list );
15         printNonDuplicates( list );
16     }

18     // create set from array to eliminate duplicates
19     private void printNonDuplicates( Collection collection )
20     {
21         // create a HashSet and obtain its iterator
22         Set set = new HashSet( collection );
23         Iterator iterator = set.iterator();
24
25         System.out.println( "\nNonduplicates are: " );
26         while ( iterator.hasNext() )
27             System.out.print( iterator.next() + " " );
28
29
31     }
32     public static void main( String args[] )
33     {
34         new SetTest();
35     }

```

ArrayList: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]

Nonduplicates are:
red cyan white tan gray green orange blue peach

- `import java.util.*;`
- `4`
- `5 public class SortedSetTest {`
- `6 private static final String names[] = { "yellow", "green",`
- `7 "black", "tan", "grey", "white", "orange", "red", "green" };`
- `8`
- `9 // create a sorted set with TreeSet, then manipulate it`
- `10 public SortedSetTest()`
- `11 {`
- `12 SortedSet tree = new TreeSet(Arrays.asList(names));`
- `13`
- `14 System.out.println("set: ");`
- `15 printSet(tree);`
- `16`
- `17 // get headSet based upon "orange"`
- `18 System.out.print("\nheadSet (\"orange\"): ");`
- `19 printSet(tree.headSet("orange"));`
- `20`
- `21 // get tailSet based upon "orange"`
- `22 System.out.print("tailSet (\"orange\"): ");`
- `23 printSet(tree.tailSet("orange"));`
- `24`
- `25 // get first and last elements`
- `26 System.out.println("first: " + tree.first());`
- `27 System.out.println("last : " + tree.last());`
- `28 }`
- `29`

ArrayList: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]

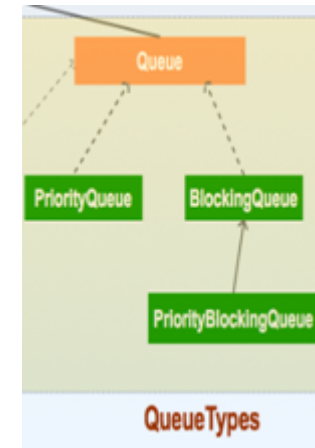
Nonduplicates are:
red cyan white tan gray green orange blue peach

- `private void printSet(SortedSet set)`
- `32 {`
- `33 Iterator iterator = set.iterator();`
- `34`
- `35 while (iterator.hasNext())`
- `36 System.out.print(iterator.next() + " ");`
- `37`
- `38 System.out.println();`
- `39 }`
- `40`
- `41 public static void main(String args[])`
- `42 {`
- `43 new SortedSetTest();`
- `44 }`
- `45 46 } // end class SortedSetTest`

```
set:
black green grey orange red tan white yellow

headSet ( "orange" ):  black green grey
tailSet ( "orange" ):  orange red tan white yellow
first: black
last : yellow
```

Queue



Queue

public interface Queue<E>

extends Collection<E>

- Queue — a collection used to hold multiple elements prior to processing.
- The Queue interface basically orders the element in FIFO(First In First Out)manner.
- It can be defined as an ordered list that is used to hold the elements which are about to be processed.

Priority Queue

- Priority Queue- PriorityQueue stores its elements internally according to their natural order(not fifo)
- The PriorityQueue class implements the Queue interface. It holds the **elements or objects which are to be processed by their priorities.**
- PriorityQueue **doesn't allow null values** to be stored in the queue.

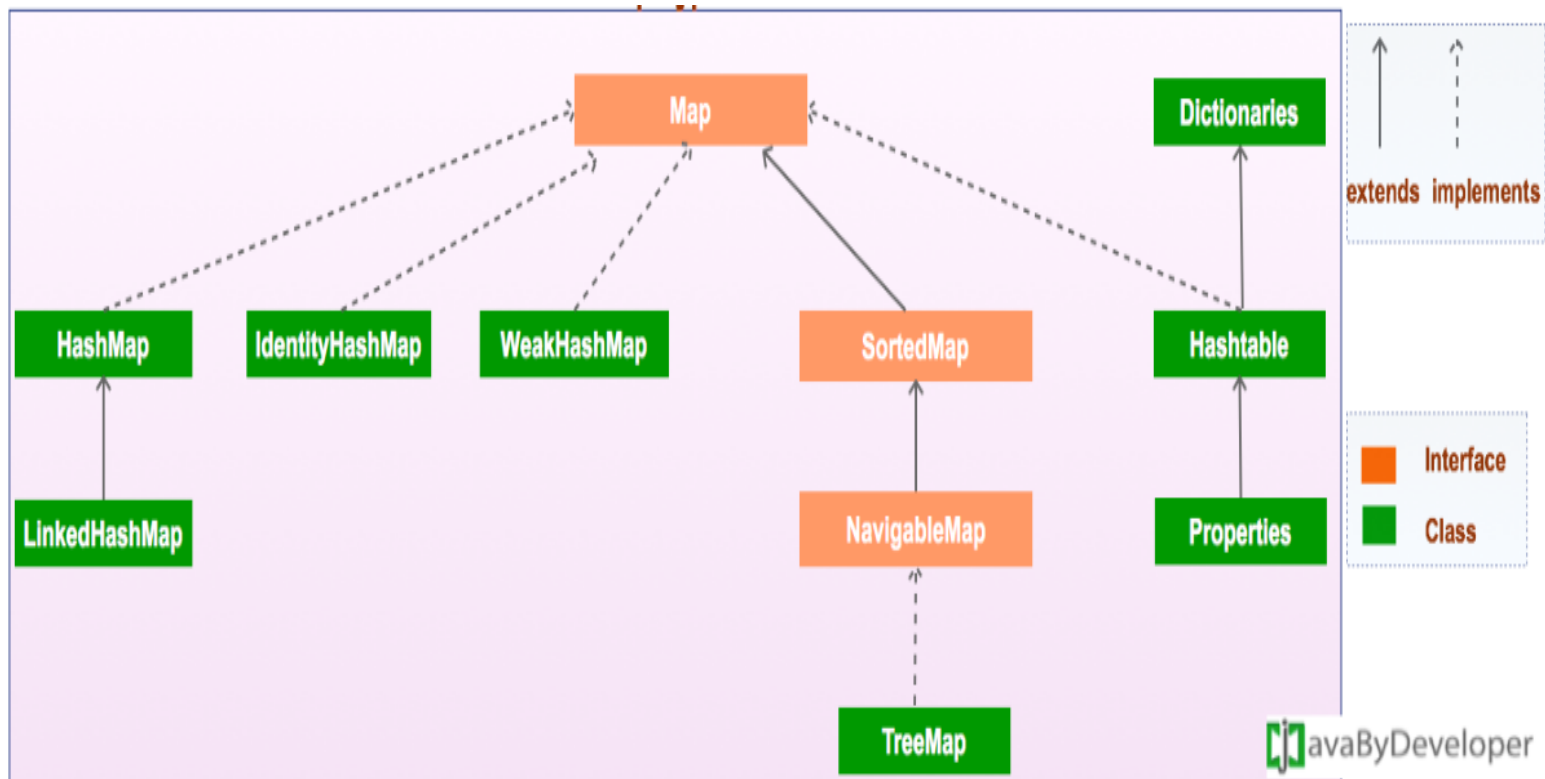
```
// ***** program *****  
import java.util.PriorityQueue;  
  
public class PriorityQueueTest  
{  
    public static void main(String[] args)  
    {  
        // queue of capacity 11  
        PriorityQueue<Double> queue = new PriorityQueue<>();  
  
        // insert elements to queue  
        queue.offer(3.2);  
        queue.offer(9.8);  
        queue.offer(5.4);  
  
        System.out.print("Polling from queue: ");  
  
        // display elements in queue  
        while (queue.size() > 0)  
        {  
            System.out.printf("%.1f ", queue.peek()); // view top element  
            queue.poll(); // remove top element  
        }  
    }  
} // end class PriorityQueueTest
```

Polling from queue: 3.2 5.4 9.8

Exercises

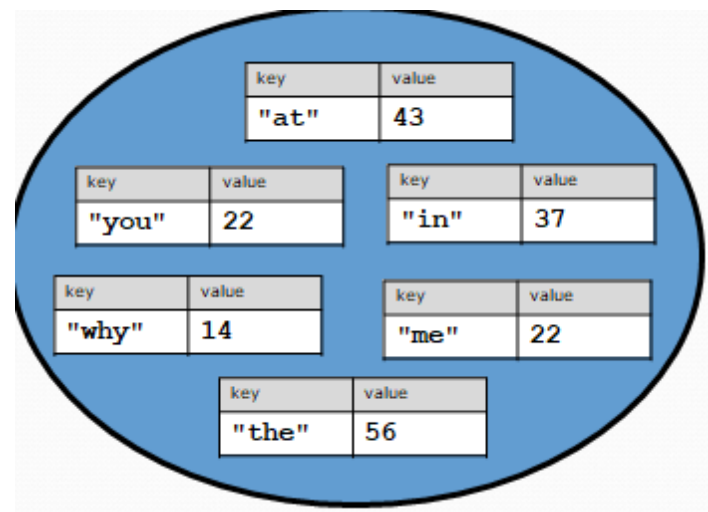
- Write a method `stutter` that accepts a queue of integers as a parameter and replaces every element of the queue with two copies of that element.
 - `front [1, 2, 3] back`
becomes
`front [1, 1, 2, 2, 3, 3] back`
- Write a method `mirror` that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order.
 - `front [a, b, c] back`
becomes
`front [a, b, c, c, b, a] back`

MAP



MAP

- A map contains values on the basis of key i.e. **key and value pair**.
- Each key and value pair is known as an entry.
- Map contains **only unique keys**.
- It contains only unique elements.
- It may have one null key and multiple null values.



MAP types

- HashMap
 - It maintains no order.
- LinkedHashMap
 - It is same as HashMap instead maintains insertion order.
- TreeMap
 - It is same as HashMap instead maintains ascending order.

Map implementation

- Java provides the Map interface in `java.util`
- Map is implemented by the `HashMap` and `TreeMap` classes
 - `HashMap`: implemented using a "hash table";
extremely fast: keys are stored in unpredictable order
 - `TreeMap`: implemented as a linked "binary tree" structure;
very fast: keys are stored in sorted order
- Maps require 2 type params: one for keys, one for values.

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
// maps from Integer keys to String values
Map<Integer, String> words = new TreeMap<Integer, String>();
```

basic map operations:

- **put**(*key*, *value*): Adds a mapping from a key to a value.
- **get**(*key*): Retrieves the value mapped to the key.
- **remove**(*key*): Removes the given key and its mapped value.

KEYS		VALUES
	Jan	327.2
	Feb	368.2
	Mar	197.6
	Apr	178.4
	May	100.0
	Jun	69.9
	Jul	32.3
Aug →	Aug	37.3 →
	Sep	19.0
	Oct	37.0
	Nov	73.2
	Dec	110.9
	Annual	1551.0

Map methods

<code>put(key, value)</code>	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
<code>get(key)</code>	returns the value mapped to the given key (<code>null</code> if not found)
<code>containsKey(key)</code>	returns <code>true</code> if the map contains a mapping for the given key
<code>remove(key)</code>	removes any existing mapping for the given key
<code>clear()</code>	removes all key/value pairs from the map
<code>size()</code>	returns the number of key/value pairs in the map
<code>isEmpty()</code>	returns <code>true</code> if the map's size is 0
<code>toString()</code>	returns a string such as " <code>{a=90, d=60, c=70}</code> "
<code>keySet()</code>	returns a set of all keys in the map
<code>values()</code>	returns a collection of all values in the map
<code>putAll(map)</code>	adds all key/value pairs from the given map to this map
<code>equals(map)</code>	returns <code>true</code> if given map has the same mappings as this one

keySet and values

- keySet method returns a Set of all keys in the map
 - can loop over the keys in a for-each loop
 - can get each key's associated value by calling get on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);
ages.put("Vicki", 57); // ages.keySet() returns Set<String>
for (String name : ages.keySet()) { // Geneva -> 2
    int age = ages.get(name); // Marty -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

- values method returns a collection of all values in the map
 - can loop over the values in a foreach loop
 - no easy way to get from a value to its associated key(s)

- `values()`
- The **`values()`** method returns a Collection of the values contained in our map:
- ```
Collection<String> values = map.values();
// [Petyr Baelish, Sansa Stark, Daenerys
Targaryen]
```

# Using foreach and Map.Entry

- This is the most common method and is preferable in most cases. We get access to both keys and values in the loop.
- ```
for (Map.Entry entry : map.entrySet())  
{  
    Sop( entry.getKey() + " -> " + entry.getValue());  
}
```

Counting the Occurrences of Words in a Text

This program counts the occurrences of words in a text and displays the words and their occurrences in ascending order of the words. The program uses a hash map to store a pair consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add the key and value 1 to the map. Otherwise, increase the value for the word (key) by 1 in the map. To sort the map, convert it to a tree map.

```
import java.util.Scanner;

public class WordTypeCount
{
    public static void main(String[] args)
    {
        // create HashMap to store String keys and Integer values
        Map<String, Integer> myMap = new HashMap<>();

        createMap(myMap); // create map based on user input
        displayMap(myMap); // display map content
    }

    // create map from user input
    private static void createMap(Map<String, Integer> map)
    {
        Scanner scanner = new Scanner(System.in); // create scanner
        System.out.println("Enter a string:"); // prompt for user input
        String input = scanner.nextLine();

        // tokenize the input
        String[] tokens = input.split(" ");

        // processing input text
        for (String token : tokens)
        {
            String word = token.toLowerCase(); // get lowercase word
```

```

// if the map contains the word
if (map.containsKey(word)) // is word in map
{
    int count = map.get(word); // get current count
    map.put(word, count + 1); // increment count
}
else
    map.put(word, 1); // add new word with a count of 1 to map
}
}

// display map content
private static void displayMap(Map<String, Integer> map)
{
    Set<String> keys = map.keySet(); // get keys

    // sort keys
    TreeSet<String> sortedKeys = new TreeSet<>(keys);

    System.out.printf("%nMap contains:%nKey\t\tValue%n");

    // generate output for each key in map
    for (String key : sortedKeys)
        System.out.printf("%-10s%10s%n", key, map.get(key));
}

```

Output

Enter a string:

this is a sample sentence with several words this is another sample sentence with several different words

Map contains:

Key	Value
a	1
another	1
different	1
is	2
sample	2
sentence	2
several	2
this	2
with	2
words	2

size: 10

Ordering Collections Interfaces

Ordering Collections

The `Comparable` and `Comparator` interfaces are useful for ordering collections:

- The `Comparable` interface imparts natural ordering to classes that implement it.
- The `Comparator` interface specifies order relation. It can also be used to override natural ordering.
- Both interfaces are useful for sorting collections.

The Comparable Interface

Imparts natural ordering to classes that implement it:

- Used for sorting
- The `compareTo` method should be implemented to make any class comparable:
 - `int compareTo(Object o)` method
- The `String`, `Date`, and `Integer` classes implement the `Comparable` interface
- You can sort the `List` elements containing objects that implement the `Comparable` interface


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

- While sorting, the `List` elements follow the natural ordering of the element types
 - `String` elements – Alphabetical order
 - `Date` elements – Chronological order
 - `Integer` elements – Numerical order

Java Comparable interface

- Java Comparable interface is **used to order the objects of user-defined class**.
- Objects of classes that implement Comparable can be ordered.
- In other words, classes that implement Comparable contain objects that can be compared in some meaningful manner
- it's possible to compare two objects of the same class if that class implements the generic interface Comparable (from package java.lang).
- All the type-wrapper classes for primitive types implement this interface.

```
public int compareTo(Object obj):
```

```
integer1.compareTo(integer2)
```

Example of the Comparable Interface

```
1  import java.util.*;
2  class Student implements Comparable {
3      String firstName, lastName;
4      int studentID=0;
5      double GPA=0.0;
6      public Student(String firstName, String lastName, int studentID,
7          double GPA) {
8          if (firstName == null || lastName == null || studentID == 0
9              || GPA == 0.0) {throw new IllegalArgumentException();}
10         this.firstName = firstName;
11         this.lastName = lastName;
12         this.studentID = studentID;
13         this.GPA = GPA;
14     }
15     public String firstName() { return firstName; }
16     public String lastName() { return lastName; }
17     public int studentID() { return studentID; }
18     public double GPA() { return GPA; }
```

Example of the Comparable Interface

```
// Implement compareTo method.
public int compareTo(Object o) {
    double f = GPA-((Student)o).GPA;
    if (f == 0.0)
        return 0;    // 0 signifies equals
    else if (f<0.0)
        return -1;   // negative value signifies less than or before
    else
        return 1;    // positive value signifies more than or after
    }
}
```

Example of the Comparable Interface

```
import java.util.*;
public class ComparableTest {
    public static void main(String[] args) {
        TreeSet studentSet = new TreeSet();
        studentSet.add(new Student("Mike", "Hauffmann", 101, 4.0));
        studentSet.add(new Student("John", "Lynn", 102, 2.8));
        studentSet.add(new Student("Jim", "Max", 103, 3.6));
        studentSet.add(new Student("Kelly", "Grant", 104, 2.3));
        Object[] studentArray = studentSet.toArray();
        Student s;
        for(Object obj : studentArray){
            s = (Student) obj;
            System.out.printf("Name = %s %s ID = %d GPA = %.1f\n",
                s.firstName(), s.lastName(), s.studentID(), s.GPA());
        }
    }
}
```

Generated Output:

```
Name = Kelly Grant ID = 104 GPA = 2.3
Name = John Lynn ID = 102 GPA = 2.8
Name = Jim Max ID = 103 GPA = 3.6
Name = Mike Hauffmann ID = 101 GPA = 4.0
```

The Comparator Interface

- Represents an order relation
- Used for sorting
- Enables sorting in an order different from the natural order
- Used for objects that do not implement the Comparable interface
- Can be passed to a sort method

You need the `compare` method to implement the Comparator interface:

- `int compare(Object o1, Object o2)` method

Comparable vs Comparator

java.lang.Comparable

int objOne.**compareTo**(objTwo)

Returns

Negative, if objOne < objTwo

Zero, if objOne == objTwo

Positive, if objOne > objTwo

You must **modify** the class whose instances you want to sort

Only **one** sort sequence can be created

Implemented frequently in the **API** by:
String, Wrapper classes, Date,
Calendar

java.util.Comparator

int **compare**(objOne, objTwo)

Same as Comparable

You **build** a class separate from the class whose instances you want to sort

Many sort sequences can be created

Meant to be implemented to sort instances of **third-party classes**

Example of the Comparator Interface

```
class Student {
    String firstName, lastName;
    int studentID=0;
    double GPA=0.0;
    public Student(String firstName, String lastName,
        int studentID, double GPA) {
        if (firstName == null || lastName == null || studentID == 0 ||
            GPA == 0.0) throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
        this.studentID = studentID;
        this.GPA = GPA;
    }
    public String firstName() { return firstName; }
    public String lastName() { return lastName; }
    public int studentID() { return studentID; }
    public double GPA() { return GPA; }
}
```



```
import java.util.*;
public class NameComp implements Comparator {
    public int compare(Object o1, Object o2) {
        return
            (((Student)o1).firstName.compareTo(((Student)o2).firstName));
    }
}
```

```
import java.util.*;
public class GradeComp implements Comparator {
    public int compare(Object o1, Object o2) {
        if (((Student)o1).GPA == ((Student)o2).GPA)
            return 0;
        else if (((Student)o1).GPA < ((Student)o2).GPA)
            return -1;
        else
            return 1;
    }
}
```

Example of the Comparator Interface

```
import java.util.*;
public class ComparatorTest {
    public static void main(String[] args) {
        Comparator c = new NameComp();
        TreeSet studentSet = new TreeSet(c);
        studentSet.add(new Student("Mike", "Hauffmann", 101, 4.0));
        studentSet.add(new Student("John", "Lynn", 102, 2.8));
        studentSet.add(new Student("Jim", "Max", 103, 3.6));
        studentSet.add(new Student("Kelly", "Grant", 104, 2.3));
        Object[] studentArray = studentSet.toArray();
        Student s;
        for(Object obj : studentArray) {
            s = (Student) obj;
            System.out.println("Name = %s %s ID = %d GPA = %.1f\n",
                               s.firstName(), s.lastName(), s.studentID(), s.GPA());
        }
    }
}
```