Data Structures

04 - Java Collections Framework

David Drohan

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709

Objectives

- Describe and use class `Arrays` for array manipulations.
- Define and use an instance of **`ArrayList`**
- Introduction to the Java Collections Framework
- Describe, create, use **`Iterators`**
- Define, use classes with generic types

Class Arrays

- Class **`Arrays`**
  - Provides `static` methods for manipulating arrays
  - Provides the following "high-level" methods
    - Method `binarySearch` for searching sorted arrays
    - Method `equals` for comparing arrays
    - Method `fill` for placing values into arrays
    - Method `sort` for sorting arrays

```
1  // Fig. 19.2: UsingArrays.java
2  // Using Java arrays.
3  import java.util.Arrays;
4
5  public class UsingArrays
6  {
7     private int intArray[] = { 1, 2, 3, 4, 5, 6 };
8     private double doubleArray[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
9     private int filledIntArray[], intArrayCopy[];
10
11    // constructor initializes arrays
12    public UsingArrays()
13    {
14       filledIntArray = new int[ 10 ]; // create int array with 10 elements
15       intArrayCopy = new int[ intArray.length ];
16
17       Arrays.fill( filledIntArray, 7 ); // fill with 7s
18       Arrays.sort( doubleArray ); // sort doubleArray ascending
19
20       // copy array intArray into array intArrayCopy
21       System.arraycopy( intArray, 0, intArrayCopy, 0, intArray.length );
22
23    } // end UsingArrays constructor
24
```

Use `static` method `fill` of class `Arrays` to populate array with 7s

Use `static` method `sort` of class `Arrays` to sort array's elements in ascending order

Use `static` method `arraycopy` of class `System` to copy array `intArray` into array `intArrayCopy`

```
25    // output values in each array
26    public void printArrays()
27    {
28       System.out.print( "doubleArray: " );
29       for ( double doubleValue : doubleArray )
30          System.out.printf( "%.1f ", doubleValue );
31
32       System.out.print( "\nintArray: " );
33       for ( int intValue : intArray )
34          System.out.printf( "%d ", intValue );
35
36       System.out.print( "\nfilledIntArray: " );
37       for ( int intValue : filledIntArray )
38          System.out.printf( "%d ", intValue );
39
40       System.out.print( "\nintArrayCopy: " );
41       for ( int intValue : intArrayCopy )
42          System.out.printf( "%d ", intValue );
43
44       System.out.println( "\n" );
45    } // end method printArrays
46
47    // find value in array intArray
48    public int searchForInt( int value )
49    {
50       return Arrays.binarySearch( intArray, value );
51    } // end method searchForInt
52
```

Use `static` method `binarySearch` of class `Arrays` to perform binary search on array

```
53    // compare array contents
54    public void printEquality()
55    {
56       boolean b = Arrays.equals( intArray, intArrayCopy );
57       System.out.printf( "intArray %s intArrayCopy\n",
58          ( b ? "==" : "!=" ) );
59
60       b = Arrays.equals( intArray, filledIntArray );
61       System.out.printf( "intArray %s filledIntArray\n",
62          ( b ? "==" : "!=" ) );
63    } // end method printEquality
64
65    public static void main( String args[] )
66    {
67       UsingArrays usingArrays = new UsingArrays();
68
69       usingArrays.printArrays();
70       usingArrays.printEquality();
71
```

Use `static` method `equals` of class `Arrays` to determine whether values of the two arrays are equivalent

```
72       int location = usingArrays.searchForInt( 5 );
73       if ( location >= 0 )
74          System.out.printf(
75             "Found 5 at element %d in intArray\n", location );
76       else
77          System.out.println( "5 not found in intArray" );
78
79       location = usingArrays.searchForInt( 8763 );
80       if ( location >= 0 )
81          System.out.printf(
82             "Found 8763 at element %d in intArray\n", location );
83       else
84          System.out.println( "8763 not found in intArray" );
85    } // end main
86 } // end class UsingArrays
```

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
intArray: 1 2 3 4 5 6
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

ArrayLists

Array-Based Data Structures: Outline

- The Class **`ArrayList`**
- Creating an Instance of **`ArrayList`**
- Using Methods of **`ArrayList`**
- Programming Example: A To-Do List
- Parameterized Classes and Generic Data Types

Class `ArrayList`

- Consider limitations of Java arrays
  - Array length is not dynamically changeable
  - Possible to create a new, larger array and copy elements – but this is awkward, contrived
- More elegant solution is use instance of `ArrayList`
  - Length is changeable at run time

Class `ArrayList`

- Drawbacks of using `ArrayList`
  - Less efficient than using an array
  - Can only store objects
  - Cannot store primitive types
- Implementation
  - Actually does use arrays
  - Expands capacity in manner previously suggested

Class `ArrayList`

- Class ArrayList is an implementation of an **A**bstract **D**ata **T**ype (ADT) called a *list*
- Elements can be added
  - At end
  - At beginning
  - In between items
- Possible to edit, delete, access, and count entries in the list

Creating Instance of `ArrayList`

- Necessary to

```
import java.util.ArrayList;
```

- Create and name instance

```
ArrayList<String> list =
                    new ArrayList<String>(20);
```

- This list will
  - Hold `String` objects
  - Initially hold up to 20 elements

Using Methods of `ArrayList`

- Object of an ArrayList used like an array
  - But methods must be used, not square bracket [] notation
- Given

```
ArrayList<String> aList =
                    new ArrayList<String>(20);
```

  - Assign a value with

```
aList.add("Hello Everybody");
aList.add(index, "Hi Mam");
aList.set(index, "Well Dad");
```

Programming Example

- A To-Do List
  - Maintains a list of everyday tasks
  - User enters as many as desired
  - Program displays the list
- View source code **`class ArrayListDemo`**

class ArrayListDemo

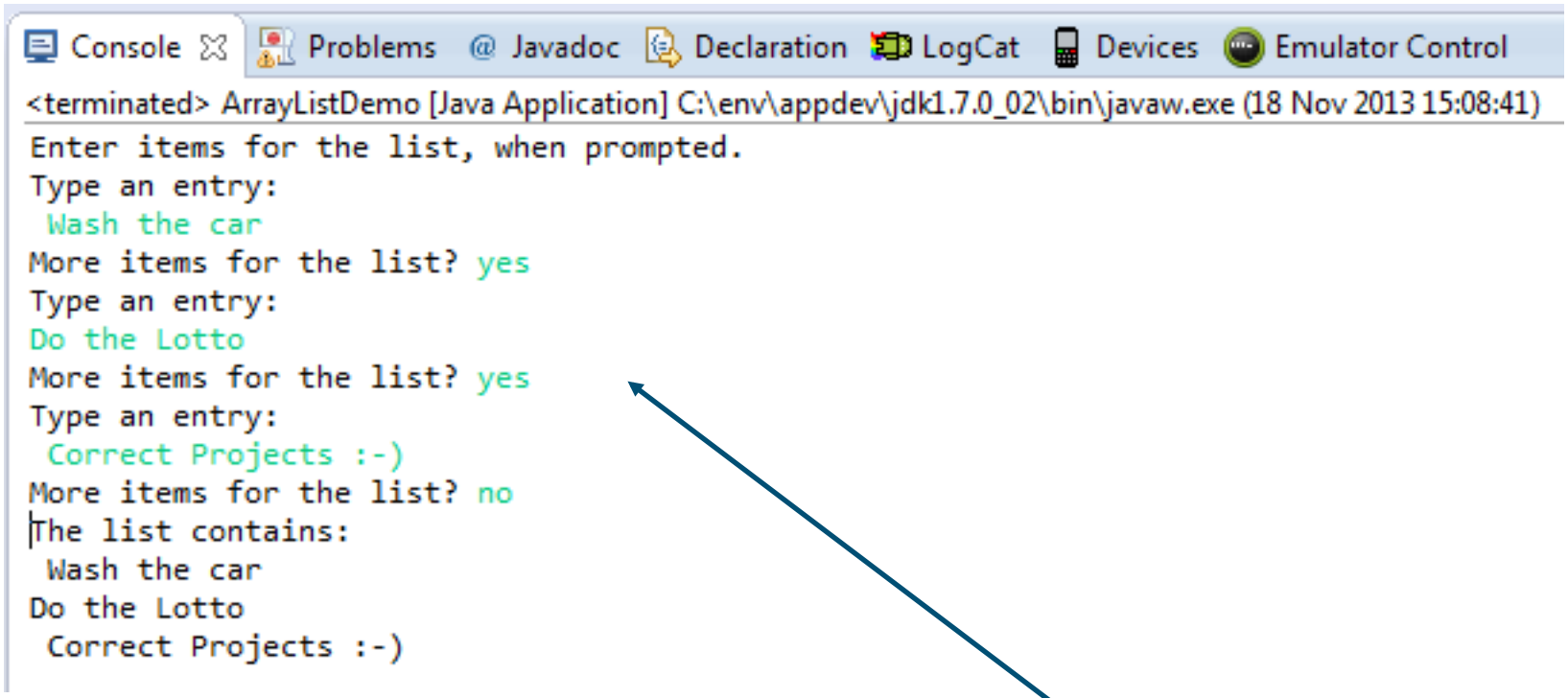
```
public class ArrayListDemo
{
   public static void main(String[] args)
   {
      ArrayList<String> toDoList = new ArrayList<String>();
      System.out.println("Enter items for the list, when prompted.");
      boolean done = false;
      Scanner keyboard = new Scanner(System.in);

      while (!done)
      {
          System.out.println("Type an entry:");
          String entry = keyboard.nextLine( );
          toDoList.add(entry);
          System.out.print("More items for the list? ");
          String ans = keyboard.nextLine( );
          if (!ans.equalsIgnoreCase("yes"))
              done = true;
      }

      System.out.println("The list contains:");
      for (int pos = 0; pos < toDoList.size( ); pos++)
          System.out.println(toDoList.get(pos));

/* Alternate code for displaying the list
      System.out.println("The list contains:");
      for (String element : toDoList)
          System.out.println(element);
*/
   }
}
```

Programming Example : Output



Sample screen output

Notes on Using `ArrayList`

- When accessing all elements of an `ArrayList` object
  - Use a For-Each loop
- Use the `trimToSize` method to save memory
- To copy an `ArrayList`
  - **Do not** use just an assignment statement (why not??)
  - Use the `clone` method, e.g.

```
ArrayList<Integer> a = new ArrayList<Integer>();
a.add(5);
ArrayList<Integer> b =(ArrayList<Integer>)a.clone();
a.add(6);
```

Parameterized Classes, Generic Data Types

- Class **`ArrayList`** is a *parameterized class*
  - It has a parameter which is a type
- Possible to declare our own classes which use types as parameters

```
ArrayList<Device> d = new ArrayList<Device>();
```

- Note : earlier versions of Java had a type of **`ArrayList`** that was not parameterized

Collections & Primitive Data Types

- Note that Collections can only hold Objects
  - One cannot put a fundamental/primitive data type into a Collection
- Java has defined "wrapper" classes which hold fundamental data type values within an Object
  - These classes are defined in java.lang
  - Each fundamental data type is represented by a wrapper class
- The wrapper classes are:

```
Boolean
Byte
Character
Double
Float
Short
Integer
Long
```

Collections & Primitive Data Types

- The wrapper classes are usually used so that fundamental data values can be placed within a Collection
- The wrapper classes have useful class constant variables
  - Integer.MAX_VALUE, Integer.MIN_VALUE
  - Double.MAX_VALUE, Double.MIN_VALUE, Double.NaN, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY
- They also have useful class methods
  - Double.parseDouble(String) - converts a String to a double
  - Integer.parseInt(String) - converts a String to an integer

The Java Collections Framework

The Java Collections Framework

- A collection of interfaces and classes that implement useful data structures and algorithms
- The `Collection` interface specifies how objects can be added, removed, or accessed from a `Collection`
- Brief introduction to a number of implementations
  - See next slide

The Java Collections Framework

- The java classes that implement the collection interfaces, generally have names in combination of the type of implementation prefixed to the type of interface, for example,
  - `ArrayList`, `LinkedList`, also (`Vector` and `Stack`) classes implement the `List` interface.
  - `PriorityQueue` implement the `Queue` interface.
  - `HashSet`, `TreeSet`, `LinkedHashSet` typical general purpose classes that implement the `Set` interface.
  - `HashMap`,`TreeMap` and `LinkedHashMap` implement the `Map` interface
- The public interface `Collection` is the root interface in the collection hierarchy and is part of `java.util.Collection` API.
- All java developers should know about the **Collections Framework!**

Collections Overview Distinction

- A `Collection` class
  - Data structure (object) that can hold references to other objects
- The Collections Framework
  - Interfaces declare operations for various collection types
  - Provide high-performance, high-quality implementations of common data structures
  - Enable software reuse
  - Enhanced with generics capabilities in J2SE 5.0
    - Compile-time type checking

Interface `Collection` & Class `Collections`

Interface `Collection`

- Root interface in the collection hierarchy
- Interfaces `Set`, `Queue`, `List` extend interface `Collection`
  - `Set` – collection does not contain duplicates
  - `Queue` – collection represents a waiting line
  - `List` – ordered collection can contain duplicate elements
- Contains *bulk operations*
  - Adding, clearing and comparing objects
- Provides a method to return an `Iterator` object
  - Walk through collection and remove elements from collection

The `Collection` Interface

- 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[])
```

Interface `Collection` & Class `Collections`

- Class `Collections`
  - Provides `static` methods that manipulate Collection objects
    - Implement algorithms for searching, sorting and so on (later section in notes)
  - Collections can be manipulated polymorphically (at a generic level)
- Synchronized collection
- Unmodifiable collection

The `List` Interface

- Lists allow duplicate entries within the collection
- Lists are an ordered collection much like an array
  - Lists grow automatically when needed
  - The list interface provides accessor methods based on index
- The List interface extends the Collections interface and add the following method definitions:

```
void add(int index, Object)
boolean addAll(int index, Collection)
Object get(int index)
int indexOf(Object)
int lastIndexOf(Object)
```

```
ListIterator listIterator()
ListIterator listIterator(int index)
Object remove(int index)
Object set(int index, Object)
List subList(int fromIndex, int toIndex)
```

`List` Implementations

- Java provides 3 concrete classes which implement the list interface
  - `ArrayList`
  - `LinkedList`
  - `Vector`
- Vectors try to optimize storage requirements by growing and shrinking as required
  - Contains a *capacity (defaults to size 10)*
  - Methods are synchronized (used for Multi-threading)
- ArrayList is roughly equivalent to Vector
  - Methods are not synchronized
- LinkedList implements a doubly linked list of elements
  - Methods are not synchronized

The `List` Interface

- NOTE : **`LinkedLists`** can be used to create Stacks, Queues, Trees and Deques (double-ended queues, pronounced “decks”).
- The collections framework provides implementations of some of these data structures.

Iterators

- A variable that allows you to step through a collection of nodes in a linked list
  - For arrays, we use an integer
- Common to place elements of a linked list into an array
  - For display purposes, array is easily traversed

```
String[] array = myList.toArray();
 for (String element : array)
     System.out.println(element);
```

The `Iterator` Interface

- Java formally considers an iterator to be an object
- Note interface named Iterator with methods
  - `hasNext` – returns boolean value
  - `next` – returns next element in iteration
  - `remove` – removes element most recently returned by `next` method

`ArrayList` and `Iterator` Example

- Demonstrate Collection interface capabilities
- Place two *`String`* arrays in `ArrayLists`
- Use `Iterator` to remove elements from `ArrayList`

```
1  // Fig. 19.3: CollectionTest.java
2  // Using the Collection interface.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class CollectionTest
9  {
10    private static final String[] colors =
11       { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
12    private static final String[] removeColors =
13       { "RED", "WHITE", "BLUE" };
14
15    // create ArrayList, add Colors to it and manipulate it
16    public CollectionTest()
17    {
18       List< String > list = new ArrayList< String >();
19       List< String > removeList = new ArrayList< String >();
20
```

Create `ArrayList` objects and assign their references to variable `list` and `removeList`, respectively

```
21       // add elements in colors array to list
22       for ( String color : colors )
23          list.add( color );
24
25       // add elements in removeColors to removeList
26       for ( String color : removeColors )
27          removeList.add( color );
28
29       System.out.println( "ArrayList: " );
30
31       // output list contents
32       for ( int count = 0; count < list.size(); count++ )
33          System.out.printf( "%s ", list.get( count ) );
34
35       // remove colors contained in removeList
36       removeColors( list, removeList );
37
38       System.out.println( "\n\nArrayList after calling removeColors: " );
39
40       // output list contents
41       for ( String color : list )
42          System.out.printf( "%s ", color );
43    } // end CollectionTest constructor
44
```

Use `List` method `add` to add objects to `list` and `removeList`, respectively

Use `List` method `size` to get the number of `ArrayList` elements

Use `List` method `get` to retrieve individual element values

Method `removeColors` takes two `Collection`s as arguments; Line 36 passes two `List`s, which extends `Collection`, to this method

```
45    // remove colors specified in collection2 from collection1
46    private void removeColors(
47       Collection< String > collection1, Collection< String > collection2 )
48    {
49       // get iterator
50       Iterator< String > iterator = collection1.iterator();
51
52       // loop while collection has items
53       while ( iterator.hasNext() )
54
55          if ( collection2.contains( iterator.next() ) )
56             iterator.remove(); // remove current Color
57    } // end method removeColors
58
59    public static void main( String args[] )
60    {
61       new CollectionTest();
62    } // end main
63 } // end class CollectionTest
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

Method `removeColors` allows any `Collection`s containing strings to be passed as arguments to this method

Obtain `Collection` iterator

`Iterator` method `hasNext` determines whether the `Iterator` contains more elements

`Iterator` method `next` returns a reference to the next element

Collection method `contains` determines whether `collection2` contains the element returned by `next`

Use `Iterator` method `remove` to remove `String` from `Iterator`

`LinkedList` and `ListIterator` Example

- Add elements of one `List` to the other
- Convert `Strings` to uppercase
- Delete a range of elements

```
1  // Fig. 19.4: ListTest.java
2  // Using LinkLists.
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.ListIterator;
6
7  public class ListTest
8  {
9     private static final String colors[] = { "black", "yellow",
10       "green", "blue", "violet", "silver" };
11    private static final String colors2[] = { "gold", "white",
12       "brown", "blue", "gray", "silver" };
13
14    // set up and manipulate LinkedList objects
15    public ListTest()
16    {
17       List< String > list1 = new LinkedList< String >();
18       List< String > list2 = new LinkedList< String >();
19
20       // add elements to list link
21       for ( String color : colors )
22          list1.add( color );
23
```

Create two `LinkedList` objects

Use `List` method `add` to append elements from array `colors` to the end of `list1`

```
24       // add elements to list link2
25       for ( String color : colors2 )
26          list2.add( color );
27
28       list1.addAll( list2 ); // concatenate lists
29       list2 = null; // release resources
30       printList( list1 ); // print list1 elements
31
32       convertToUppercaseStrings( list1 ); // convert to upper case string
33       printList( list1 ); // print list1 elements
34
35       System.out.print( "\nDeleting elements 4 to 6..." );
36       removeItems( list1, 4, 7 ); // remove items 4-7 from list
37       printList( list1 ); // print list1 elements
38       printReversedList( list1 ); // print list in reverse order
39    } // end ListTest constructor
40
41    // output List contents
42    public void printList( List< String > list )
43    {
44       System.out.println( "\nlist: " );
45
46       for ( String color : list )
47          System.out.printf( "%s ", color );
48
49       System.out.println();
50    } // end method printList
51
```

Use `List` method `add` to append elements from array `colors2` to the end of `list2`

Use `List` method `addAll` to append all elements of `list2` to the end of `list1`

Method `printList` allows any `Lists` containing strings to be passed as arguments to this method

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

Method `convertToUppercaseStrings` allows any `List`s containing strings to be passed as arguments to this method

Invoke `List` method `listIterator` to get a bidirectional iterator for the List

Invoke `ListIterator` method `hasNext` to determine whether the `List` contains another element

Invoke `ListIterator` method `next` to obtain the next `String` in the `List`

Invoke `ListIterator` method `set` to replace the current `String` to which `iterator` refers with the `String` returned by method `toUpperCase`

Method `removeItems` allows any `List`s containing strings to be passed as arguments to this method

Invoke `List` method `subList` to obtain a portion of the `List`

Method `printReversedList` allows any `List`s containing strings to be passed as arguments to this method

Invoke `List` method `listIterator` with one argument that specifies the starting position to get a bidirectional iterator for the list

```
75       System.out.println( "\nReversed List:" );
76
77       // print list in reverse order
78       while ( iterator.hasPreviously() )
79          System.out.printf( "%s ", iterator.previous() );
80    } // end method printReversedList
81
82    public static void main( String args[] )
83    {
84       new ListTest();
85    } // end main
86 } // end class ListTest
```

The `while` condition calls method `hasPrevious` to determine whether there are more elements while traversing the list backward

Invoke `ListIterator` method `previous` to get the previous element from the list

```
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
```

`LinkedList` (Cont.)

- `static` method **`asList`** of class **`Arrays`**
  - View an array as a **`List`** collection
  - Allow programmer to manipulate the array as if it were a list
  - Any modification made through the **`List`** view change the array
  - Any modification made to the array change the **`List`** view
  - Only operation permitted on the view returned by **`asList`** is ***`set`***

```
1  // Fig. 19.5: UsingToArray.java
2  // Using method toArray.
3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsingToArray
7  {
8     // constructor creates LinkedList, adds elements and converts to array
9     public UsingToArray()
10    {
11       String colors[] = { "black", "blue", "yellow" };
12
13       LinkedList< String > links =
14          new LinkedList< String >( Arrays.asList( colors ) );
15
16       links.addLast( "red" ); // add as last item
17       links.add( "pink" );    // add to the end
18       links.add( 3, "green" );  // add at 3rd index
19       links.addFirst( "cyan" ); // add as first item
20
```

Call method `asList` to create a `List` view of array `colors`, which is then used for creating a `LinkedList`

Call `LinkedList` method `addLast` to add "`red`" to the end of `links`

Call `LinkedList` method `add` to add "`pink`" as the last element and "`green`" as the element at index 3

Call `LinkedList` method `addFirst` to add "`cyan`" as the new first item in the `LinkedList`

```
21       // get LinkedList elements as an array
22       colors = links.toArray( new String[ links.size() ] );
23
24       System.out.println( "colors: " );
25
26       for ( String color : colors )
27          System.out.println( color );
28    } // end UsingToArray constructor
29
30    public static void main( String args[] )
31    {
32       new UsingToArray();
33    } // end main
34 } // end class UsingToArray
```

```
colors:
cyan
black
blue
yellow
green
red
pink
```

Use `List` method `toArray` to obtain array representation of `LinkedList`

Vector Example

```
1  // Fig. 19.6: VectorTest.java
2  // Using the Vector class.
3  import java.util.Vector;
4  import java.util.NoSuchElementException;
5
6  public class VectorTest
7  {
8     private static final String colors[] = { "red", "white", "blue" };
9
10    public VectorTest()
11    {
12       Vector< String > vector = new Vector< String >();
13       printVector( vector ); // print vector
14
15       // add elements to the vector
16       for ( String color : colors )
17          vector.add( color );
18
19       printVector( vector ); // print vector
20
```

Create `Vector` of type `String` with initial capacity of 10 element and capacity increment of zero

Call `Vector` method `add` to add objects (`String`s in this example) to the end of the `Vector`

```
21      // output the first and last elements
22      try
23      {
24         System.out.printf( "First element: %s\n", vector.firstElement());
25         System.out.printf( "Last element: %s\n", vector.lastElement() );
26      } // end try
27      // catch exception if vector is empty
28      catch ( NoSuchElementException exception )
29      {
30         exception.printStackTrace();
31      } // end catch
32
33      // does vector contain "red"?
34      if ( vector.contains( "red" ) )
35         System.out.printf( "\n\"red\" found at index %d\n\n",
36            vector.indexOf( "red" ) );
37      else
38         System.out.println( "\n\"red\" not found\n" );
39
40      vector.remove( "red" ); // remove the string "red"
41      System.out.println( "\"red\" has been removed" );
42      printVector( vector ); // print vector
43
```

Call `Vector` method `firstElement` to return a reference to the first element in the `Vector`

Call `Vector` method `lastElement` to return a reference to the last element in the `Vector`

`Vector` method `contains` returns `boolean` that indicates whether `Vector` contains a specific `Object`

`Vector` method `remove` removes the first occurrence of its argument `Object` from `Vector`

`Vector` method `indexOf` returns index of first location in `Vector` containing the argument

```
44       // does vector contain "red" after remove operation?
45       if ( vector.contains( "red" ) )
46          System.out.printf(
47             "\"red\" found at index %d\n", vector.indexOf( "red" ) );
48       else
49          System.out.println( "\"red\" not found" );
50
51       // print the size and capacity of vector
52       System.out.printf( "\nSize: %d\nCapacity: %d\n", vector.size(),
53          vector.capacity() );
54    } // end Vector constructor
55
56    private void printVector( Vector< String > vectorToOutput )
57    {
58       if ( vectorToOutput.isEmpty() )
59          System.out.print( "vector is empty" ); // vectorToOutput is empty
60       else  // iterate through the elements
61       {
62          System.out.print( "vector contains: " );
63
64          // output elements
65          for ( String element : vectorToOutput )
66             System.out.printf( "%s ", element );
67       } // end else
68
```

`Vector` methods `size` and `capacity` return number of `elements` in `Vector` and `Vector` capacity, respectively

Method `printVector` allows any `Vector`s containing strings to be passed as arguments to this method

`Vector` method `isEmpty` returns true if there are no elements in the Vector

```
69          System.out.println( "\n" );
70       } // end method printVector
71
72       public static void main( String args[] )
73       {
74          new VectorTest(); // create object and call its constructor
75       } // end main
76    } // end class VectorTest
```

```
vector is empty

vector contains: red white blue

First element: red
Last element: blue

"red" found at index 0

"red" has been removed
vector contains: white blue

"red" not found

Size: 2
Capacity: 10
```

Class `Stack`

- Implements a stack data structure
- Extends class `Vector`
- Stores references to objects
- Elements removed from ADT in reverse order of initial insertion
  - LIFO Implementation

```
1  // Fig. 19.16: StackTest.java
2  // Program to test java.util.Stack.
3  import java.util.Stack;
4  import java.util.EmptyStackException;
5
6  public class StackTest
7  {
8     public StackTest()
9     {
10       Stack< Number > stack = new Stack< Number >();
11
12       // create numbers to store in the stack
13       Long longNumber = 12L;
14       Integer intNumber = 34567;
15       Float floatNumber = 1.0F;
16       Double doubleNumber = 1234.5678;
17
18       // use push method
19       stack.push( longNumber ); // push a long
20       printStack( stack );
21       stack.push( intNumber ); // push an int
22       printStack( stack );
23       stack.push( floatNumber ); // push a float
24       printStack( stack );
25       stack.push( doubleNumber ); // push a double
26       printStack( stack );
27
```

Create an empty `Stack` of type `Number`

`Stack` method `push` adds object to top of `Stack`

```
28       // remove items from stack
29       try
30       {
31          Number removedObject = null;
32
33          // pop elements from stack
34          while ( true )
35          {
36             removedObject = stack.pop(); // use pop method
37             System.out.printf( "%s popped\n", removedObject );
38             printStack( stack );
39          } // end while
40       } // end try
41       catch ( EmptyStackException emptyStackException )
42       {
43          emptyStackException.printStackTrace();
44       } // end catch
45    } // end StackTest constructor
46
47    private void printStack( Stack< Number > stack )
48    {
49       if ( stack.isEmpty() )
50          System.out.print( "stack is empty\n\n" ); // the stack is empty
51       else  // stack is not empty
52       {
53          System.out.print( "stack contains: " );
54
```

Stack method pop removes element from top of Stack

Stack method isEmpty returns true if Stack is empty

```
55          // iterate through the elements
56          for ( Number number : stack )
57             System.out.printf( "%s ", number );
58
59          System.out.print( "(top) \n\n" ); // indicates top of the stack
60       } // end else
61    } // end method printStack
62
63    public static void main( String args[] )
64    {
65       new StackTest();
66    } // end main
67 } // end class StackTest
```

```
stack contains: 12 (top)

stack contains: 12 34567 (top)

stack contains: 12 34567 1.0 (top)

stack contains: 12 34567 1.0 1234.5678 (top)

1234.5678 popped
stack contains: 12 34567 1.0 (top)

1.0 popped
stack contains: 12 34567 (top)

34567 popped
stack contains: 12 (top)

12 popped
stack is empty

java.util.EmptyStackException
        at java.util.Stack.peek(Unknown Source)
        at java.util.Stack.pop(Unknown Source)
        at StackTest.<init>(StackTest.java:36)
        at StackTest.main(StackTest.java:65)
```

Interface `Queue` & Class `PriorityQueue`

- Interface `Queue`
  - New collection interface introduced in J2SE 5.0
  - Extends interface `Collection`
  - Provides additional operations for inserting, removing and inspecting elements in a queue fashion
- Class `PriorityQueue`
  - Implements the `Queue` interface
  - Orders elements by their natural ordering
    - Specified by `Comparable` elements' `compareTo` method **OR**
    - `Comparator` object supplied through constructor (discussed later)
- For an Unordered `Queue`, implement as `LinkedList`

```java
// Fig. 19.17: PriorityQueueTest.java
// Standard library class PriorityQueue test program.
import java.util.PriorityQueue;

public class PriorityQueueTest
{
   public static void main( String args[] )
   {
      // queue of capacity 11
      PriorityQueue< Double > queue = new PriorityQueue< Double >();

      // 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 while
   } // end main
} // end class PriorityQueueTest
```

Create a `PriorityQueue` that stores `Doubles` with an initial capacity of 11 elements and orders the elements according to the object's natural ordering

Use method `offer` to add elements to the priority queue

Use method `size` to determine whether the priority queue is empty

Use method `peek` to retrieve the highest-priority element in the queue

Use method `pool` to remove the highest-priority element from the queue

```
Polling from queue: 3.2 5.4 9.8
```

The `Set` Interface

- The Set interface also extends the **`Collection`** interface but does not add any methods to it.
- Collection classes which implement the Set interface have the added stipulation that Sets ***CANNOT*** contain duplicate elements
- Elements are compared using the equals method

NOTE: exercise caution when placing mutable objects within a set. Objects are tested for equality upon addition to the set. If the object is changed after being added to the set, the rules of duplication may be violated.

The `SortedSet` Interface

- SortedSet provides the same mechanisms as the `Set` interface, except that SortedSets maintain the elements in *ascending order*.
- Ordering is based on natural ordering (Comparable) or by using a Comparator.
- We will discuss `Comparable` and `Comparators` in later sections.

`Set` Implementations

- Java provides 2 concrete classes which implement the Set interface
  - `HashSet`
  - `TreeSet`
- HashSet behaves like a HashMap except that the elements **cannot** be duplicated.
- TreeSet behaves like TreeMap except that the elements **cannot** be duplicated.

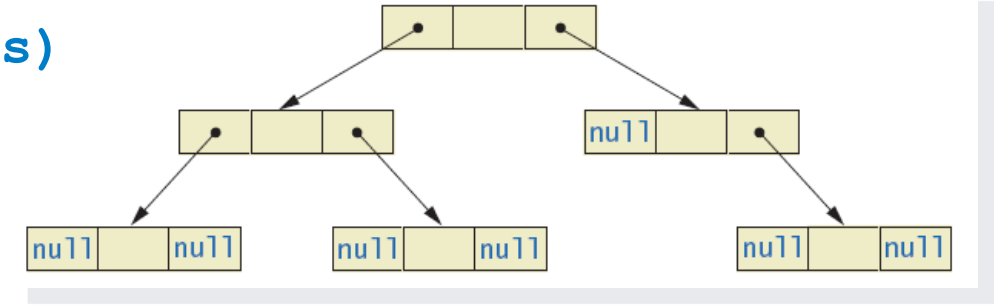
- Note: Sets are not as commonly used as Lists

Class HashSet

- HashSet implements the Set interface. It creates a collection that uses a hash table for storage.
  - A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code.
  - The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.
- The HashSet class supports a number of constructors.
  - `HashSet( )`
  - `HashSet(Collection c)`
  - `HashSet(int capacity)`

Class `TreeSet`

- TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.
  - Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.
- The TreeSet class supports a number of constructors.
  - `TreeSet( )`
  - `TreeSet(Collection c)`
  - `TreeSet(Comparator comp)`
  - `TreeSet(SortedSet s)`



```
1  // Fig. 19.18: SetTest.java
2  // Using a HashSet to remove duplicates.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.HashSet;
6  import java.util.Set;
7  import java.util.Collection;
8
9  public class SetTest
10 {
11    private static final String colors[] = { "red", "white", "blue",
12       "green", "gray", "orange", "tan", "white", "cyan",
13       "peach", "gray", "orange" };
14
15    // create and output ArrayList
16    public SetTest()
17    {
18       List< String > list = Arrays.asList( colors );
19       System.out.printf( "ArrayList: %s\n", list );
20       printNonDuplicates( list );
21    } // end SetTest constructor
22
```

Create a `List` that contains `String` objects

```
23    // create set from array to eliminate duplicates
24    private void printNonDuplicates( Collection< String > collection )
25    {
26       // create a HashSet
27       Set< String > set = new HashSet< String >( collection );
28
29       System.out.println( "\nNonduplicates are: " );
30
31       for ( String s : set )
32          System.out.printf( "%s ", s );
33
34       System.out.println();
35    } // end method printNonDuplicates
36
37    public static void main( String args[] )
38    {
39       new SetTest();
40    } // end main
41 } // end class SetTest
```

Method `printNonDuplicates` accepts a `Collection` of type `String`

Construct a `HashSet` from the `Collection` argument

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

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

```
1  // Fig. 19.19: SortedSetTest.java
2  // Using TreeSet and SortedSet.
3  import java.util.Arrays;
4  import java.util.SortedSet;
5  import java.util.TreeSet;
6
7  public class SortedSetTest
8  {
9     private static final String names[] = { "yellow", "green",
10       "black", "tan", "grey", "white", "orange", "red", "green" };
11
12    // create a sorted set with TreeSet, then manipulate it
13    public SortedSetTest()
14    {
15       // create TreeSet
16     SortedSet< String > tree = new TreeSet< String >( Arrays.asList( names ) );
17
18
19       System.out.println( "sorted set: " );
20       printSet( tree ); // output contents of tree
21
```

Create `TreeSet` from `names` array

```
22       // get headSet based on "orange"
23       System.out.print( "\nheadSet (\"orange\"):  " );
24       printSet( tree.headSet( "orange" ) );
25
26       // get tailSet based upon "orange"
27       System.out.print( "tailSet (\"orange\"):  " );
28       printSet( tree.tailSet( "orange" ) );
29
30       // get first and last elements
31       System.out.printf( "first: %s\n", tree.first() );
32       System.out.printf( "last : %s\n", tree.last() );
33    } // end SortedSetTest constructor
34
35    // output set
36    private void printSet( SortedSet< String > set )
37    {
38       for ( String s : set )
39          System.out.printf( "%s ",  s );
40
```

Use `TreeSet` method `headSet` to get `TreeSet` subset less than `"orange"`

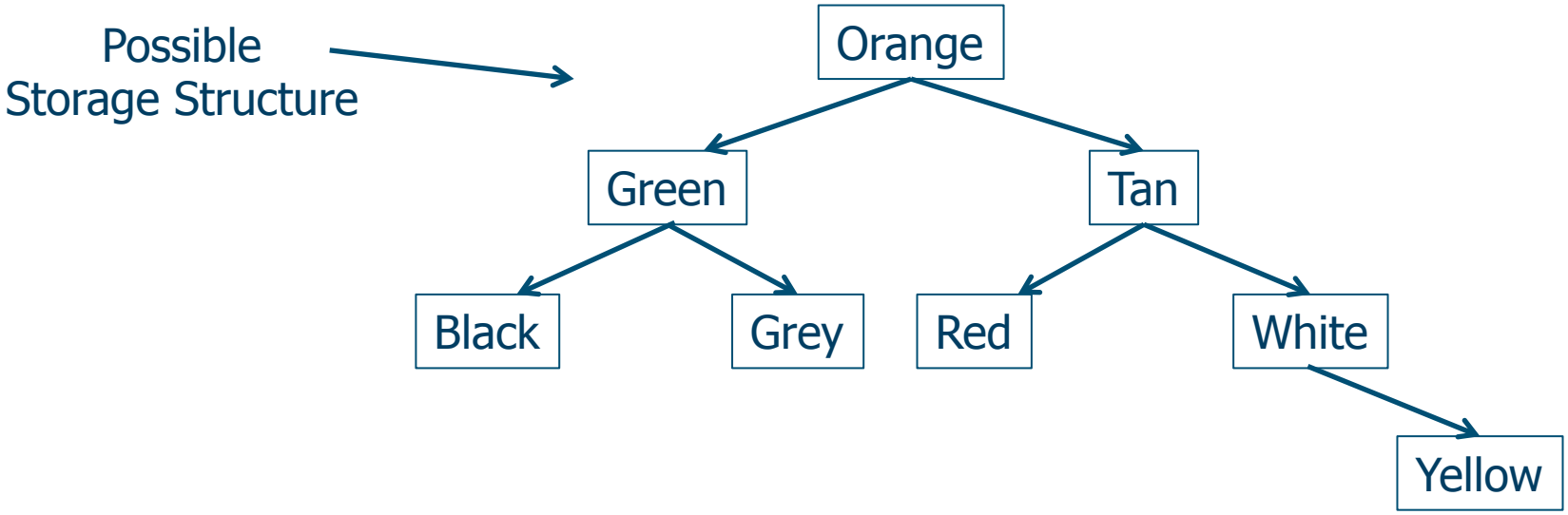
Use `TreeSet` method `tailSet` to get `TreeSet` subset greater than `"orange"`

Methods `first` and `last` obtain smallest and largest `TreeSet` elements, respectively

```
41       System.out.println();
42    } // end method printSet
43
44    public static void main( String args[] )
45    {
46       new SortedSetTest();
47    } // end main
48 } // end class SortedSetTest
```

```
sorted 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
```



The `Map` Interface

- The Map interface provides the basis for dictionary or key-based collections in Java.
  - Associates keys to values
  - Cannot contain duplicate keys – (*one-to-one mapping)*
- Interface `SortedMap` maintains its keys in sorted order
- Implementation classes `HashMap`, `TreeMap`
- The interface includes:

```
void clear()
boolean containsKey(Object)
boolean containsValue(Object)
Set entrySet()
boolean equals(Object)
Object get(Object)
boolean isEmpty()
```

```
Set keySet()
Object put(Object key, Object value)
void putAll(Map)
boolean remove(Object key)
int size()
Collection values()
```

Performance Tip

- The load factor in a hash table is a classic example of a memory/speed trade-off:
- By increasing the load factor, we get better memory utilization, but the program runs slower, due to increased hashing collisions.
- By decreasing the load factor, we get better program speed, because of reduced hashing collisions, but we get poorer memory utilization, because a larger portion of the hash table remains empty.

```
1  // Fig. 19.20: WordTypeCount.java
2  // Program counts the number of occurrences of each word in a string
3  import java.util.StringTokenizer;
4  import java.util.Map;
5  import java.util.HashMap;
6  import java.util.Set;
7  import java.util.TreeSet;
8  import java.util.Scanner;
9
10 public class WordTypeCount
11 {
12    private Map< String, Integer > map;
13    private Scanner scanner;
14
15    public WordTypeCount()
16    {
17       map = new HashMap< String, Integer >(); // create HashMap
18       scanner = new Scanner( System.in ); // create scanner
19       createMap(); // create map based on user input
20       displayMap(); // display map content
21    } // end WordTypeCount constructor
22
```

Create an empty `HashMap` with a default capacity 16 and a default load factor 0.75. The keys are of type `String` and the values are of type `Integer`

```
23  // create map from user input
24  private void createMap()
25  {
26     System.out.println( "Enter a string:" ); // prompt for user input
27     String input = scanner.nextLine();
28
29     // create StringTokenizer for input
30     StringTokenizer tokenizer = new StringTokenizer( input );
31
32     // processing input text
33     while ( tokenizer.hasMoreTokens() ) // while more input
34     {
35        String word = tokenizer.nextToken().toLowerCase(); // get word
36
37        // if the map contains the word
38        if ( map.containsKey( word ) ) // is word in map
39        {
40           int count = map.get( word ); // get current count
41           map.put( word, count + 1 );  // increment count
42        } // end if
43        else
44           map.put( word, 1 ); // add new word with a count of 1 to map
45     } // end while
46  } // end method createMap
47
```

Create a `StringTokenizer` to break the input string argument into its component individual words

`Map` method `containsKey` determines whether the key specified as an argument is in the hash table

Use method `get` to obtain the key's associated value in the map

Increment the value and use method `put` to replace the key's associated value

Create a new entry in the map, with the word as the key and an `Integer` object containing 1 as the value

```
48    // display map content
49    private void displayMap()
50    {
51       Set< String > keys = map.keySet(); // get keys
52
53       // sort keys
54       TreeSet< String > sortedKeys = new TreeSet< String >( keys );
55
56       System.out.println( "Map contains:\nKey\t\tValue" );
57
58       // generate output for each key in map
59       for ( String key : sortedKeys )
60          System.out.printf( "%-10s%10s\n", key, map.get( key ) );
61
62       System.out.printf(
63          "\nsize:%d\nisEmpty:%b\n", map.size(), map.isEmpty() );
64    } // end method displayMap
65
```

Use `HashMap` method `keySet` to obtain a set of the keys

Access each key and its value in the map

Call `Map` method `size` to get the number of key-value pairs in the `Map`

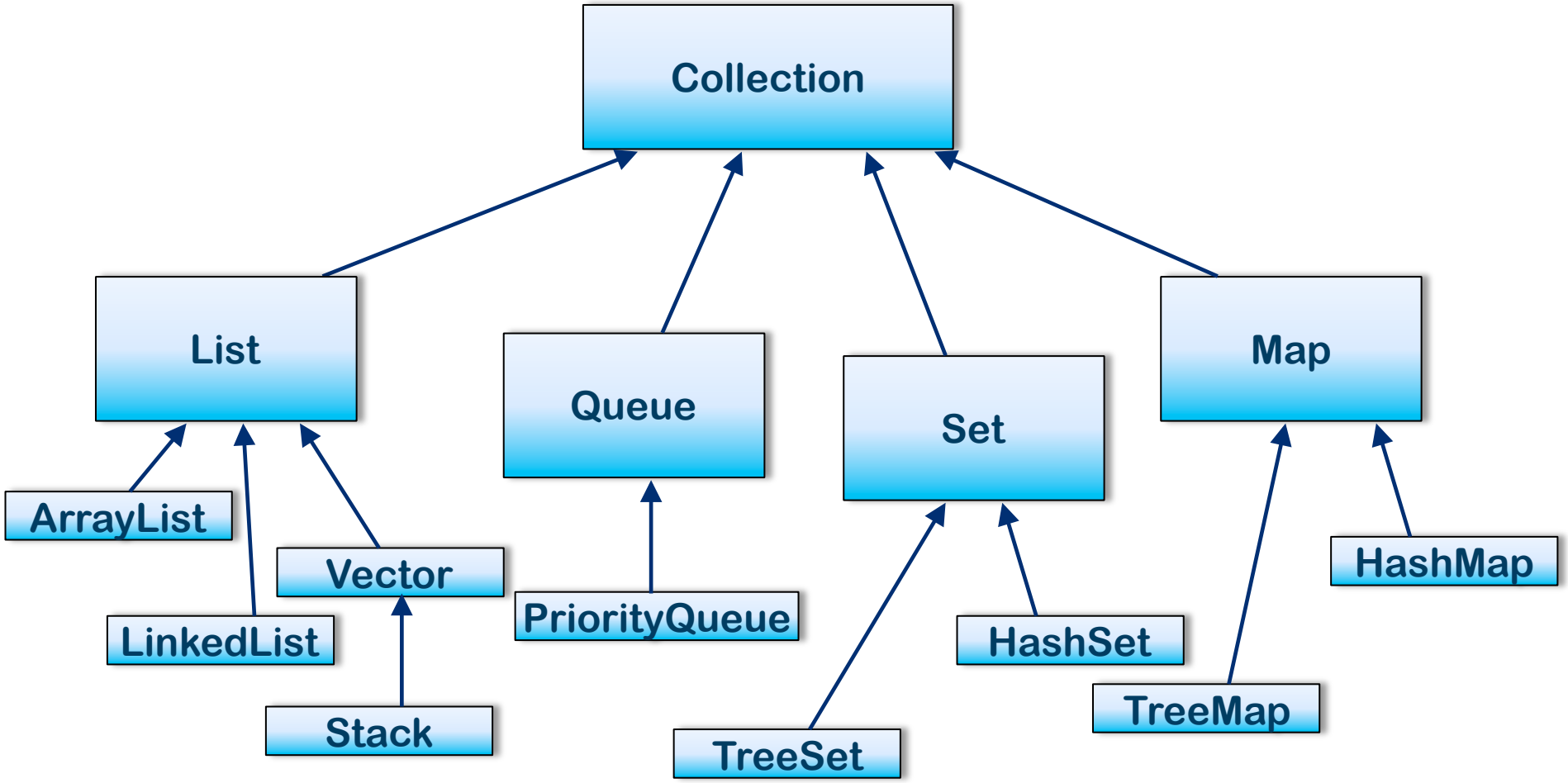
Call `Map` method `isEmpty` to determine whether the `Map` is empty

```
66    public static void main( String args[] )
67    {
68       new WordTypeCount();
69    } // end main
70 } // end class WordTypeCount
```

```
Enter a string:
To be or not to be: that is the question Whether 'tis nobler to suffer
Map contains:
Key                  Value
'tis                    1
be                      1
be:                     1
is                      1
nobler                  1
not                     1
or                      1
question                1
suffer                  1
that                    1
the                     1
to                      3
whether                 1

size:13
isEmpty:false
```

The Collections Hierarchy



Which class should I use?

- You'll notice that collection classes all provide the same or similar functionality. The difference between the different classes is how the structure is implemented - This generally has an impact on performance.
- Use Vector
  - Fast access to elements using index
  - Optimized for storage space
  - Not optimized for inserts and deletes
- Use ArrayList
  - Same as Vector except the methods are not synchronized. But better performance as a result..

Which class should I use?

- Use LinkedList
  - Fast inserts and deletes
  - Stacks and Queues (accessing elements near the beginning or end)
- Use Set
  - When you need a collection which does not allow duplicate entries
- Use Maps
  - Very Fast access to elements using keys
  - Fast addition and removal of elements
  - No duplicate keys allowed
- When choosing a class, it is worthwhile to read the class's documentation in the Java API specification. There you will find notes about the implementation of the Collection class and within which contexts it is best to use.

Generics

Basics of Generics

- Beginning with Java 5.0, class definitions may include parameters for types
  - Called *generics*
- Programmer now can specify ***any*** class type for the type parameter
- View definition

```
class Sample<T>
```

- Note use of `<T>` for the type parameter

class Sample<T>

```
public class Sample<T>
{
    private T data;

    public void setData(T newValue)
    {
        data = newValue;
    }

    public T getData( )
    {
        return data;
    }
}
```

Basics of Generics

- Legal to use parameter T almost anywhere you can use class type
  - Cannot use type parameter when allocating memory such as `anArray = new T[20];`
- Example declaration

```
Sample <String> sample1 =
        new Sample<String>();
```

- Cannot specify a primitive type for the type parameter

Summary

- Java Class Library includes **`ArrayList`**
  - Like an array that can grow in length
  - Includes methods to manipulate the list
- Java Collections Framework contains many classes to store and manipulate objects
- **`Iterators`** used to navigate lists
- Class can be declared with type parameter
- Object of a parameterized class replaces type parameter with an actual class type
- Classes **`ArrayList`**, **`HashSet`**, **`HashMap`** and **`LinkedList`** are parameterized classes

Questions?