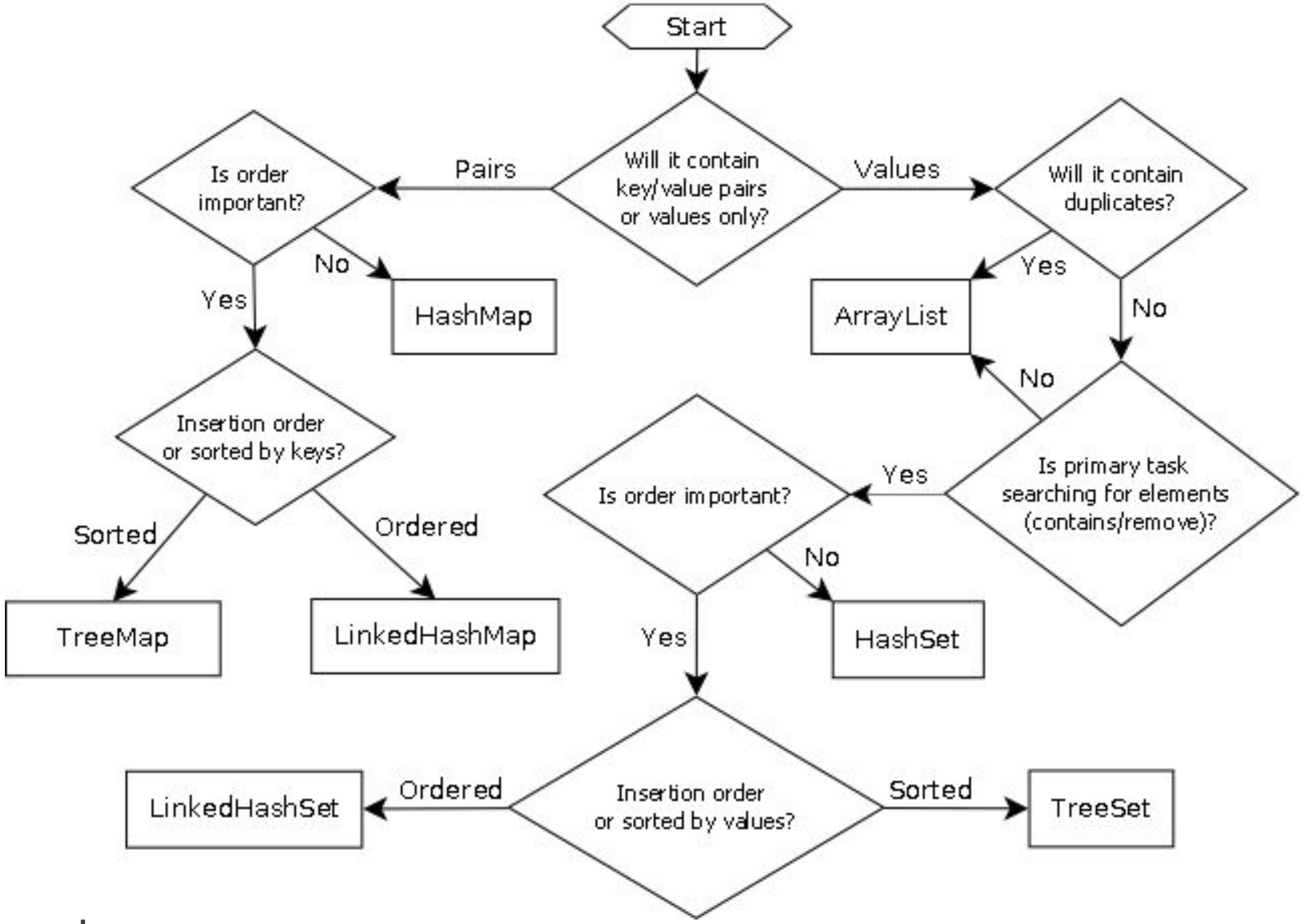
Choosing a collection


Start
Will it contain key/value pairs or values only?
Pairs
Values
Is order important?
No
HashMap
Yes
Insertion order or sorted by keys?
Sorted
TreeMap
Ordered
LinkedHashMap
Will it contain duplicates?
Yes
ArrayList
No
Is primary task searching for elements (contains/remove)?
No
Yes
Is order important?
No
HashSet
Yes
Insertion order or sorted by values?
Ordered
LinkedHashSet
Sorted
TreeSet

- see also: http://initbinder.com/bunker/wp-content/uploads/2011/03/collections.png

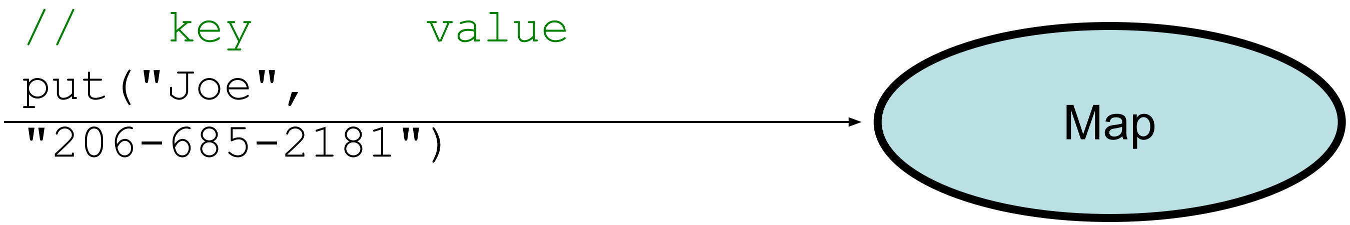
Collections summary

collection	ordering	benefits	weaknesses
array	by index	fast; simple	little functionality; cannot resize
`ArrayList`	by insertion, by index	random access; fast to modify at end	slow to modify in middle/front
`LinkedList`	by insertion, by index	fast to modify at both ends	poor random access
`TreeSet`	sorted order	sorted; O(log N)	must be comparable
`HashSet`	unpredictable	very fast; O(1)	unordered
`LinkedHashSet`	order of insertion	very fast; O(1)	uses extra memory
`TreeMap`	sorted order	sorted; O(log N)	must be comparable
`HashMap`	unpredictable	very fast; O(1)	unordered
`LinkedHashMap`	order of insertion	very fast; O(1)	uses extra memory
`PriorityQueue`	natural/comparable	fast ordered access	must be comparable

- It is important to be able to choose a collection properly based on the capabilities needed and constraints of the problem to solve.

Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

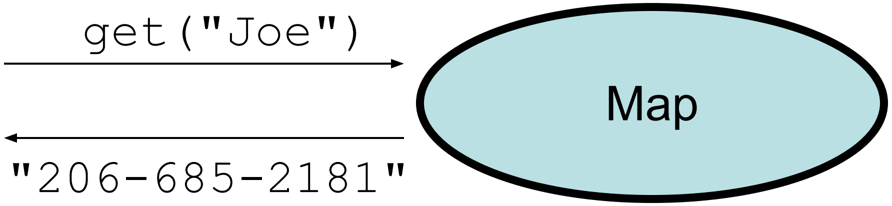


```
//    key        value
put("Joe", "206-685-2181")
```

Map

  - Later, we can supply only the key and get back the related value:

    Allows us to ask: *What is Joe's phone number?*



```
get("Joe")
"206-685-2181"
```

Map

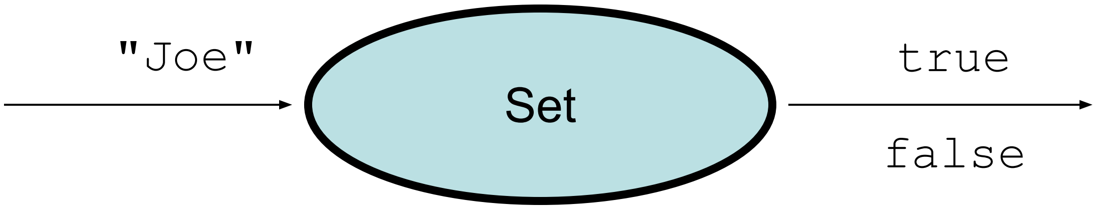
`Map` methods

`put(`**key**`, `**value**`)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**key**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**key**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**key**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`

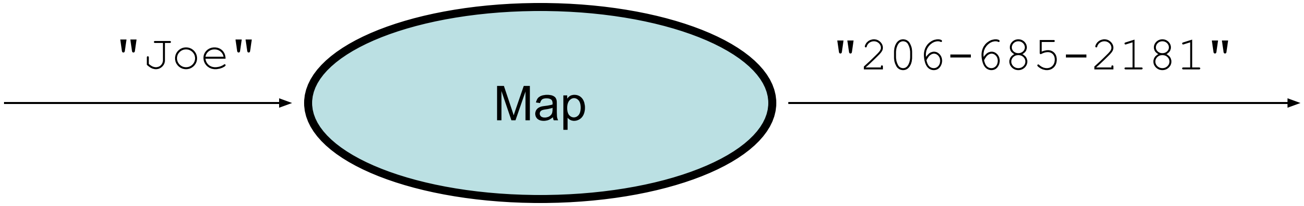
`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

Maps vs. sets

- A set is like a map from elements to `boolean` values.
  - *Set: Is Joe found in the set? (true/false)*



  - *Map: What is Joe's phone number?*



keySet and values

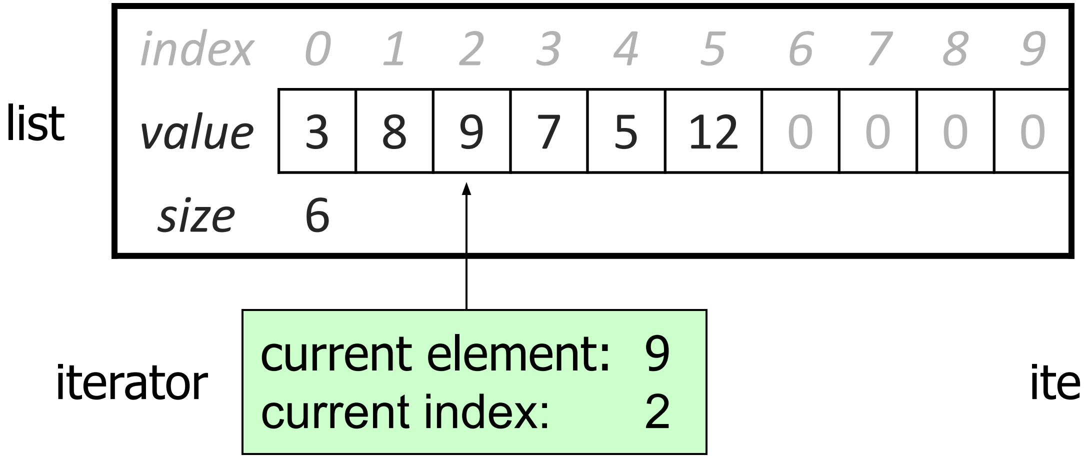
- keySet method returns Set of all keys in map
  - can loop over the keys in a foreach loop
  - can get each key's associated value by calling get on the map

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

- values method returns Collection of all values in map
  - ages.values() above returns [2, 57, 19]
  - can loop over the values with a for-each loop
  - no easy way to get from a value back to its associated key(s)

Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of any collection.
  - Remembers a position, and lets you:
    - get the element at that position
    - advance to the next position
    - remove the element at that position

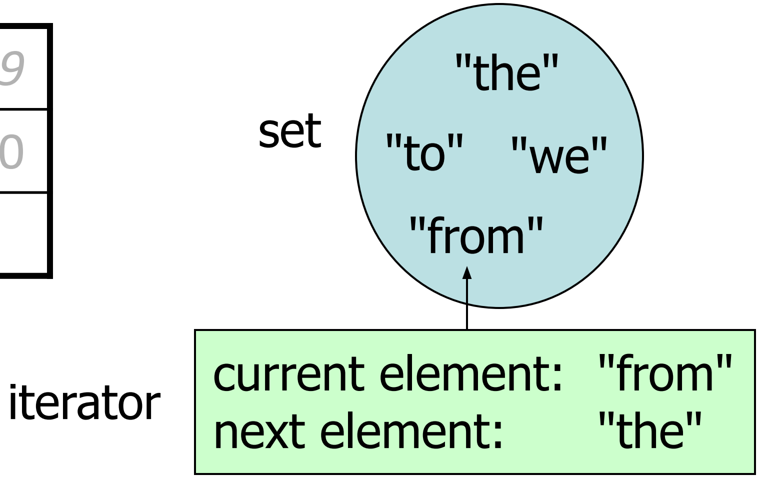


list

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	9	7	5	12	0	0	0	0
size	6									

iterator

current element: 9
current index: 2



set

"the"
"to" "we"
"from"

iterator

current element: "from"
next element: "the"

Iterator methods

hasNext()	returns true if there are more elements to examine
next()	returns the next element from the collection (throws a NoSuchElementException if there are none left to examine)
remove()	removes the last value returned by next() (throws an IllegalStateException if you haven't called next() yet)

- Iterator interface in java.util
  - every collection has an iterator() method that returns an iterator over its elements

```
Set<String> set = new HashSet<String>();
...
Iterator<String> itr = set.iterator();
...
```

Iterator example

```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38);    // Jenny
scores.add(87);
scores.add(43);    // Marty
scores.add(72);
…

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();

    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores);  // [72, 87, 94]
```

Chapter 22 – Collections

Outline

22.1 Introduction
22.2 Collections Overview
22.3 Class `Arrays`
22.4 Interface `Collection` and Class `Collections`
22.5 Lists
22.6 Algorithms
    22.6.1 Algorithm `sort`
    22.6.2 Algorithm `shuffle`
    22.6.3 Algorithms `reverse`, `fill`, `copy`, `max` and `min`
    22.6.4 Algorithm `binarySearch`
22.7 Sets

22.1 Introduction

- Java collections framework
  - Contains prepackaged data structures, interfaces, algorithms for manipulating those data structures
  - Examples of collections – hand of cards, software engineers working on same project, etc.
  - Collections – Use existing data structures without concern for how they are implemented
    - Example of code reuse

22.2 Collections Overview

- Collection
  - Data structure (object) that can hold references to other objects
- Collections framework
  - Interfaces declare operations for various collection types
  - Belong to package `java.util`
    - `Collection`
    - `Set`
    - `List`
    - `Map`

22.3 Class `Arrays`

- Class `Arrays`
  - Provides `static` methods for manipulating arrays
  - Provides “high-level” methods
    - Method `binarySearch` for searching sorted arrays
    - Method `equals` for comparing arrays
    - Method `fill` for placing values into arrays (overloaded to fill all or part of the array)
    - Method `sort` for sorting arrays (overloaded to sort all or part of the array)
    - Method `arraycopy` to copy portion of one array into another (`java.lang.System`)
  - Methods overloaded to work with primitive-type arrays and object arrays

Outline

UsingArrays.java

Line 16

Line 18

```
1   // Fig. 22.1: UsingArrays.java
2   // Using Java arrays.
3   import java.util.*;
4
5   public class UsingArrays {
6      private int intValues[] = { 1, 2, 3, 4, 5, 6 };
7      private double doubleValues[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
8      private int filledInt[], intValuesCopy[];
9
10     // initialize arrays
11     public UsingArrays()
12     {
13        filledInt = new int[ 10 ];
14        intValuesCopy = new int[ intValues.length ];
15
16        Arrays.fill( filledInt, 7 );   // fill with 7s
17
18        Arrays.sort( doubleValues );   // sort doubleValues ascending
19
20        // copy array intValues into array intValuesCopy
21        System.arraycopy( intValues, 0, intValuesCopy,
22           0, intValues.length );
23     }
24
```

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

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 `intValues` into array `intValuesCopy`

```java
   // output values in each array
   public void printArrays()
   {
      System.out.print( "doubleValues: " );

      for ( int count = 0; count < doubleValues.length; count++ )
         System.out.print( doubleValues[ count ] + " " );

      System.out.print( "\nintValues: " );

      for ( int count = 0; count < intValues.length; count++ )
         System.out.print( intValues[ count ] + " " );

      System.out.print( "\nfilledInt: " );

      for ( int count = 0; count < filledInt.length; count++ )
         System.out.print( filledInt[ count ] + " " );

      System.out.print( "\nintValuesCopy: " );

      for ( int count = 0; count < intValuesCopy.length; count++ )
         System.out.print( intValuesCopy[ count ] + " " );

      System.out.println();

   } // end method printArrays

```

Outline

UsingArrays.java

Outline

```
52    // find value in array intValues
53    public int searchForInt( int value )
54    {
55       return Arrays.binarySearch( intValues, value );
56    }
57
58    // compare array contents
59    public void printEquality()
60    {
61       boolean b = Arrays.equals( intValues, intValuesCopy );
62
63       System.out.println( "intValues " + ( b ? "==" : "!=" ) +
64          " intValuesCopy" );
65
66       b = Arrays.equals( intValues, filledInt );
67
68       System.out.println( "intValues " + ( b ? "==" : "!=" ) +
69          " filledInt" );
70    }
71
```

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

Line 55

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

UsingArrays.java

```java
   public static void main( String args[] )
   {
      UsingArrays usingArrays = new UsingArrays();

      usingArrays.printArrays();
      usingArrays.printEquality();

      int location = usingArrays.searchForInt( 5 );
      System.out.println( ( location >= 0 ? "Found 5 at element " +
         location : "5 not found" ) + " in intValues" );

      location = usingArrays.searchForInt( 8763 );
      System.out.println( ( location >= 0 ? "Found 8763 at element " +
         location : "8763 not found" ) + " in intValues" );
   }

} // end class UsingArrays
```

```
doubleValues: 0.2 3.4 7.9 8.4 9.3
intValues: 1 2 3 4 5 6
filledInt: 7 7 7 7 7 7 7 7 7 7
intValuesCopy: 1 2 3 4 5 6
intValues == intValuesCopy
intValues != filledInt
Found 5 at element 4 in intValues
8763 not found in intValues
```

22.3 Class Arrays

- Viewing an Array as a List
  - `private static final String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };`
  - `List list = new ArrayList( Arrays.asList( suits ) );`
    - `list` is independent of `suits`, changes to either does not affect the other
  - `List list = Arrays.asList( suits );`
    - `list` is a "view" of `suits`, changes made to `list` changes `suits` and vice versa

Outline

UsingAsList.java

Line 12

Line 21

```
1   // Fig. 22.2: UsingAsList.java
2   // Using method asList.
3   import java.util.*;
4
5   public class UsingAsList {
6      private static final String values[] = { "red", "white", "blue" };
7      private List list;
8
9      // initialize List and set value at location 1
10     public UsingAsList()
11     {
12        list = Arrays.asList( values );   // get List
13        list.set( 1, "green" );           // change a value
14     }
15
16     // output List and array
17     public void printElements()
18     {
19        System.out.print( "List elements : " );
20
21        for ( int count = 0; count < list.size(); count++ )
22           System.out.print( list.get( count ) + " " );
23
24        System.out.print( "\nArray elements: " );
25
```

Use `static` method `asList` of class `Arrays` to return `List` *view* of array `values`

Use method `set` of `List` object to change the contents of element `1` to `"green"`

`List` method `size` returns number of elements in `List`

`List` method `get` returns individual element in `List`

```java
      for ( int count = 0; count < values.length; count++ )
         System.out.print( values[ count ] + " " );

      System.out.println();
   }

   public static void main( String args[] )
   {
      new UsingAsList().printElements();
   }

} // end class UsingAsList
```

```
List elements : red green blue
Array elements: red green blue
```

UsingAsList.java

22.4 Interface Collection and Class Collections

- Interface Collection
  - Contains *bulk operations*
    - Adding, clearing, comparing and retaining objects
  - Interfaces Set and List extend interface Collection
  - Provides an Iterator, similar to an Enumeration (but Iterator can remove elements, while Enumeration cannot)
- Class Collections
  - Provides static methods that manipulate collections
  - Collections can be manipulated polymorphically

22.5 Lists

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

Outline

CollectionTest.java

Lines 15-20

Line 26

```
1   // Fig. 22.3: CollectionTest.java
2   // Using the Collection interface.
3   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
```

Use List method add to add objects to ArrayList

List method get returns individual element in List

```
28          // remove all String objects
29          removeStrings( list );
30
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
38
39       // remove String objects from Collection
40       private void removeStrings( Collection collection )
41       {
42          Iterator iterator = collection.iterator(); // get iterat
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       }
50
```

Method removeStrings takes a Collection as an argument; Line 29 passes List, which extends Collection, to this method

java

Line 29

Line 42

Line 45

Obtain Collection iterator

Iterator method hasNext determines whether the Iterator contains more elements

Iterator method next returns next Object in Iterator

Use Iterator method remove to remove String from Iterator

```
   public static void main( String args[] )
   {
      new CollectionTest();
   }

} // end class CollectionTest
```

```
ArrayList:
java.awt.Color[r=255,g=0,b=255] red white blue java.awt.Color
[r=0,g=255,b=255]

ArrayList after calling removeStrings:
java.awt.Color[r=255,g=0,b=255] java.awt.Color[r=0,g=255,b=255]
```

CollectionTest.java

Outline

ListTest.java

Lines 14-15

Line 23

Line 24

```
1   // Fig. 22.4: ListTest.java
2   // Using LinkLists.
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
25
```

Create two `LinkedList` objects

Use `LinkedList` method `addAll` to append `link2` elements to `link`

Nullify `link2`, so it can be garbage collected

Outline

ListTest.java

Lines 42-50

```
26          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
49          System.out.println();
50       }
```

Use `List` method get to obtain object in `LinkedList`, then print its value

```
51
52     // locate String objects and convert to uppercas
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() );
75
```

Use `ListIterator` to traverse `LinkedList` elements and convert them to upper case (if elements are `Strings`)

Use `List` methods `subList` and `clear` to remove `LinkedList` elements

ListTest.java

Lines 53-63

Line 68

java

```
      System.out.println( "\nReversed List:" );

      // print list in reverse order
      while( iterator.hasPrevious() )
         System.out.print( iterator.previous() + " " );
   }

   public static void main( String args[] )
   {
      new ListTest();
   }

} // end class ListTest
```

`ListIterator` method `hasPrevious` determines whether the List`Iterator` contains more elements

`ListIterator` method `previous` returns previous `Object` in `ListIterator`

```
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
```

```
1     // Fig. 22.5: UsingToArray.java
2     // Using method toArray.
3     import java.util.*;
4
5     public class UsingToArray {
6
7        // create LinkedList, add elements and convert to array
8        public UsingToArray()
9        {
10          String colors[] = { "black", "blue", "yellow" };
11
12          LinkedList links = new LinkedList( Arrays.asList( colors ) );
13
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
18
19          // get LinkedList elements as an array
20          colors = ( String [] ) links.toArray( new String[ links.size() ] );
21
22          System.out.println( "colors: " );
23
```

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

UsingToArray.java

Line 20

```java
      for ( int count = 0; count < colors.length; count++ )
         System.out.println( colors[ count ] );
   }

   public static void main( String args[] )
   {
      new UsingToArray();
   }

} // end class UsingToArray
```

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

UsingToArray.java

22.6 Algorithms

- `Collections` class provides set of (high-performance, efficient) algorithms
  - Implemented as `static` methods
    - `List` algorithms
      - `sort`
      - `binarySearch`
      - `reverse`
      - `shuffle`
      - `fill`
      - `copy`
    - `Collections` algorithms
      - `min`
      - `max`

22.6.1 Algorithm `sort`

- `sort`
  - Sorts `List` elements
    - `Collections.sort(list);`
    - Order is determined by natural order of element type
    - Uses that class's `compareTo` method (returns 0, negative, positive)
    - `Collections.sort(list, Comparator object)` can be used for alternate ordering

Outline

```
1   // Fig. 22.6: Sort1.java
2   // Using algorithm sort.
3   import java.util.*;
4
5   public class Sort1 {
6      private static final String suits[] =
7         { "Hearts", "Diamonds", "Clubs", "Spades" };
8
9      // display array elements
10     public void printElements()
11     {
12        // create ArrayList
13        List list = new ArrayList( Arrays.asList( suits ) );
14
15        // output list
16        System.out.println( "Unsorted array elements:\n" + list );
17
18        Collections.sort( list ); // sort ArrayList
19
20        // output list
21        System.out.println( "Sorted array elements:\n" + list );
22     }
23
```

Sort1.java

Line 13

Line 18

Create ArrayList

Use Collections method sort to sort ArrayList

```
   public static void main( String args[] )
   {
      new Sort1().printElements();
   }

} // end class Sort1
```

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

Sort1.java

Sort2.java

Line 18

Line 18

```
1   // Fig. 22.7: Sort2.java
2   // Using a Comparator object with algorithm sort.
3   import java.util.*;
4
5   public class Sort2 {
6      private static final String suits[] =
7         { "Hearts", "Diamonds", "Clubs", "Spades" };
8
9      // output List elements
10     public void printElements()
11     {
12        List list = Arrays.asList( suits ); // create List
13
14        // output List elements
15        System.out.println( "Unsorted array elements:\n" + list );
16
17        // sort in descending order using a comparator
18        Collections.sort( list, Collections.reverseOrder() );
19
20        // output List elements
21        System.out.println( "Sorted list elements:\n" + list );
22     }
23
```

Method `reverseOrder` of class `Collections` returns a `Comparator` object that represents the collection's reverse order

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

```java
   public static void main( String args[] )
   {
      new Sort2().printElements();
   }

} // end class Sort2
```

Sort2.java

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

```
1   // Fig. 22.8: Sort3.java
2   // Creating a custom Comparator class.
3   import java.util.*;
4   
5   public class Sort3 {
6   
7      public void printElements()
8      {
9         List list = new ArrayList(); // create List
10  
11        list.add( new Time2(  6, 24, 34 ) );
12        list.add( new Time2( 18, 14, 05 ) );
13        list.add( new Time2(  8, 05, 00 ) );
14        list.add( new Time2( 12, 07, 58 ) );
15        list.add( new Time2(  6, 14, 22 )
16  
17        // output List elements
18        System.out.println( "Unsorted arr
19  
20        // sort in order using a comparator
21        Collections.sort( list, new TimeComparator() );
22  
23        // output List elements
24        System.out.println( "Sorted list elements:\n" + list );
25     }
26  
```

Sort in order using a custom comparator `TimeComparator`.

Sort3.java

Line 21

Outline

```
27    public static void main( String args[] )
28    {
29       new Sort2().printElements();
30    }
31
32    private class TimeComparato
33       int hourCompare, minuteC
34       Time2 time1, time2;
35
36       public int compare(Object object1, Object object2)
37       {
38          // cast the objects
39          time1 = (Time2)object1;
40          time2 = (Time2)object2;
41
42          hourCompare = new Integer( time1.getHour() ).compareTo(
43                        new Integer( time2.getHour() ) );
44
45          // test the hour first
46          if ( hourCompare != 0 )
47             return hourCompare;
48
49          minuteCompare = new Integer( time1.getMinute() ).compareTo(
50                          new Integer( time2.getMinute() ) );
51
```

Custom comparator `TimeComparator` implements `Comparator` interface.

Implement method `compare` to determine the order of two objects.

Line 32

Line 36

Sort3.java

```
         // then test the minute
         if ( minuteCompare != 0 )
            return minuteCompare;

         secondCompare = new Integer( time1.getSecond() ).compareTo(
                         new Integer( time2.getSecond() ) );

         return secondCompare; // return result of comparing seconds
      }

   } // end class TimeComparator

} // end class Sort3
```

```
Unsorted array elements:
[06:24:34, 18:14:05, 08:05:00, 12:07:58, 06:14:22]
Sorted list elements:
[06:14:22, 06:24:34, 08:05:00, 12:07:58, 18:14:05]
```

22.6.2 Algorithm shuffle

- shuffle
  - Randomly orders List elements

```java
// Fig. 22.9: Cards.java
// Using algorithm shuffle.
import java.util.*;

// class to represent a Card in a deck of cards
class Card {
   private String face;
   private String suit;

   // initialize a Card
   public Card( String initialface, String initialSuit )
   {
      face = initialface;
      suit = initialSuit;
   }

   // return face of Card
   public String getFace()
   {
      return face;
   }

   // return suit of Card
   public String getSuit()
   {
      return suit;
   }
```

Outline

Cards.java

Cards.java

```java

   // return String representation of Card
   public String toString()
   {
      StringBuffer buffer = new StringBuffer( face + " of " + suit );
      buffer.setLength( 20 );

      return buffer.toString();
   }

} // end class Card

// class Cards declaration
public class Cards {
   private static final String suits[] =
      { "Hearts", "Clubs", "Diamonds", "Spades" };
   private static final String faces[] = { "Ace", "Deuce", "Three",
      "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten",
      "Jack", "Queen", "King" };
   private List list;

   // set up deck of Cards and shuffle
   public Cards()
   {
      Card deck[] = new Card[ 52 ];

```

Outline

```
54       for ( int count = 0; count < deck.length; count++ )
55          deck[ count ] = new Card( faces[ count % 13 ],
56             suits[ count / 13 ] );
57
58       list = Arrays.asList( deck );   // get List
59       Collections.shuffle( list );    // shuffle deck
60    }
61
62    // output deck
63    public void printCards()
64    {
65       int half = list.size() / 2 - 1;
66
67       for ( int i = 0, j = half + 1; i <= half; i++, j++ )
68          System.out.println( list.get( i ).toString() + list.get( j ) );
69    }
70
71    public static void main( String args[] )
72    {
73       new Cards().printCards();
74    }
75
76 } // end class Cards
```

Use method `shuffle` of class `Collections` to shuffle `List`

`Cards.java`

Line 59

```
King of Diamonds    Jack of Spades
Four of Diamonds    Six of Clubs
King of Hearts      Nine of Diamonds
Three of Spades     Four of Spades
Four of Hearts      Seven of Spades
Five of Diamonds    Eight of Hearts
Queen of Diamonds   Five of Hearts
Seven of Diamonds   Seven of Hearts
Nine of Hearts      Three of Clubs
Ten of Spades       Deuce of Hearts
Three of Hearts     Ace of Spades
Six of Hearts       Eight of Diamonds
Six of Diamonds     Deuce of Clubs
Ace of Clubs        Ten of Diamonds
Eight of Clubs      Queen of Hearts
Jack of Clubs       Ten of Clubs
Seven of Clubs      Queen of Spades
Five of Clubs       Six of Spades
Nine of Spades      Nine of Clubs
King of Spades      Ace of Diamonds
Ten of Hearts       Ace of Hearts
Queen of Clubs      Deuce of Spades
Three of Diamonds   King of Clubs
Four of Clubs       Jack of Diamonds
Eight of Spades     Five of Spades
Jack of Hearts      Deuce of Diamonds
```

Cards.java

22.6.3 Algorithms `reverse`, `fill`, `copy`, `max` and `min`

- `reverse`
  - Reverses the order of `List` elements
- `fill`
  - Populates (overwrites) `List` elements with values
- `copy (dest, source)`
  - Creates copy of a `List`
- `max`
  - Returns largest element in `Collection`
- `min`
  - Returns smallest element in `Collection`
- `max` and `min` can be called with comparator object as second argument

Outline

Algorithms1.java

Line 19

Line 23

```
1   // Fig. 22.10: Algorithms1.java
2   // Using algorithms reverse, fill, copy, min and max.
3   import java.util.*;
4
5   public class Algorithms1 {
6      private String letters[] = { "P", "C", "M" }, lettersCopy[];
7      private List list, copyList;
8
9      // create a List and manipulate it with methods from Collections
10     public Algorithms1()
11     {
12        list = Arrays.asList( letters );     // get List
13        lettersCopy = new String[ 3 ];
14        copyList = Arrays.asList( lettersCopy );
15
16        System.out.println( "Initial list: " );
17        output( list );
18
19        Collections.reverse( list );         // reverse order
20        System.out.println( "\nAfter calling reverse: " );
21        output( list );
22
23        Collections.copy( copyList, list ); // copy List
24        System.out.println( "\nAfter copying: " );
25        output( copyList );
26
```

Use method reverse of class Collections to obtain List in reverse order

Use method copy of class Collections to obtain copy of List

```
27        Collections.fill( list, "R" );
28        System.out.println( "\nAfter ca
29        output( list );
30
31     } // end constructor
32
33     // output List information
34     private void output( List listRef )
35     {
36        System.out.print( "The list is: " );
37
38        for ( int k = 0; k < listRef.size(); k++ )
39           System.out.print( listRef.get( k ) + " " );
40
41        System.out.print( "\nMax: " + Collections.max( listRef ) );
42        System.out.println( "  Min: " + Collections.min( listRef ) );
43     }
44
45     public static void main( String args[] )
46     {
47        new Algorithms1();
48     }
49
50  } // end class Algorithms1
```

Use method `fill` of class `Collections` to populate `List` with the letter `"R"`

Obtain maximum value in `List`

Obtain minimum value in `List`

Outline

Algorithms1.java

Line 27

Line 41

Line 42

```
Initial list:
The list is: P C M
Max: P  Min: C

After calling reverse:
The list is: M C P
Max: P  Min: C

After copying:
The list is: M C P
Max: P  Min: C

After calling fill:
The list is: R R R
Max: R  Min: R
```

Algorithms1.java

22.6.4 Algorithm binarySearch

- binarySearch
  - Collections method
  - Locates Object in List
    - Returns index of Object in List if Object exists
    - Returns negative value if Object does not exist
  - Collections.binarySearch(list,key)
  - Collections.binarySearch(list,key, comparator object)

Outline

BinarySearchTest.java

Line 14

```
1   // Fig. 22.11: BinarySearchTest.java
2   // Using algorithm binarySearch.
3   import java.util.*;
4
5   public class BinarySearchTest {
6      private static final String colors[] = { "red", "white",
7         "blue", "black", "yellow", "purple", "tan", "pink" };
8      private List list;        // List reference
9
10     // create, sort and output list
11     public BinarySearchTest()
12     {
13        list = new ArrayList( Arrays.asList( colors ) );
14        Collections.sort( list );   // sort the ArrayList
15        System.out.println( "Sorted ArrayList: " + list );
16     }
17
18     // search list for various values
19     private void printSearchResults()
20     {
21        printSearchResultsHelper( colors[ 3 ] ); // first item
22        printSearchResultsHelper( colors[ 0 ] ); // middle item
23        printSearchResultsHelper( colors[ 7 ] ); // last item
24        printSearchResultsHelper( "aardvark" );  // below lowest
25        printSearchResultsHelper( "goat" );      // does not exist
26        printSearchResultsHelper( "zebra" );     // does not exist
27     }
28
```

Sort `List` in ascending order

Outline

BinarySearchTest.java

```
   // helper method to perform searches
   private void printSearchResultsHelper( String key )
   {
      int result = 0;

      System.out.println( "\nSearching for: " + key );
      result = Collections.binarySearch( list, key );
      System.out.println( ( result >= 0 ? "Found at index " + result :
         "Not Found (" + result + ")" ) );
   }

   public static void main( String args[] )
   {
      new BinarySearchTest().printSearchResults();
   }

} // end class BinarySearchTest
```

Use method binarySearch of class Collections to search List for specified key

```
Sorted ArrayList: black blue pink purple red tan white yellow
Searching for: black
Found at index 0

Searching for: red
Found at index 4

Searching for: pink
Found at index 2

Searching for: aardvark
Not Found (-1)

Searching for: goat
Not Found (-3)

Searching for: zebra
Not Found (-9)
```

22.7 Sets

- 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

Outline

SetTest.java

Line 22

```
1   // Fig. 22.12: SetTest.java
2   // Using a HashSet to remove duplicates.
3   import java.util.*;
4
5   public class SetTest {
6      private static final String colors[] = { "red", "white", "blue",
7         "green", "gray", "orange", "tan", "white", "cyan",
8         "peach", "gray", "orange" };
9
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     }
17
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
```

Create `HashSet` from `Collection` object

Outline

SetTest.java

Lines 27-28

```
27        while ( iterator.hasNext() )
28           System.out.print( iterator.next() + " " );
29
30        System.out.println();
31     }
32
33     public static void main( String args[] )
34     {
35        new SetTest();
36     }
37
38  } // end class SetTest
```

Use Iterator to traverse Set and print nonduplicates

```
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. 22.13: SortedSetTest.java
2  // Using TreeSet and SortedSet.
3  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
```

SortedSetTest.java

Line 12

Line 19

Lines 26-27

Create TreeSet from names array

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

Use TreeSet method tailSet to get TreeSet subset including and after "orange"

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

Outline

SortedSetTest.j

35-36

```
30     // output set
31     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
```

Use Iterator to traverse Set and print values

```
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
```