16

Generic Collection Algorithms

Test 3

- Monday, June 8
- 5% of the total grade
- all course material (in book Chapters 1 – 11 and 16)
- review session next class (Wed)
- practice problems on the website
- preparation for final exam

Last time

- generic classes and methods
- collections
- Lists, Sets, and Maps in Java

Objectives

- generic classes and methods
- Collections algorithms
- Arrays algorithms

Generics

- **Generic classes**
  - **A single class declaration defines set of related classes**
  - **Also called parameterized classes (types)**
  - **For example,** `ArrayList< String >`
    and `ArrayList< Double >`

- **Generic methods (single method declaration)**

Java Collections Framework

- Contain prepackaged data structures, interfaces, and algorithms (cover some today)
- Use generics
- Provides reusable components and containers (hold references to other objects)

Interface	Description
`Collection`	The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived.
`Set`	A collection that does not contain duplicates.
`List`	An ordered collection that can contain duplicate elements.
`Map`	Associates keys to values and cannot contain duplicate keys.
`Queue`	Typically a first-in, first-out collection that models a waiting line; other orders can be specified.

Some collection framework interfaces.

Lists

- `List`
  - Ordered `Collection` that can contain duplicate elements
  - Sometimes called a *sequence*
  - Implemented via interface `List`
    - `ArrayList`
    - `LinkedList`
    - `Vector`

Sets

- `Set`
  - `Collection` that contains unique elements
  - `HashSet`
    - Stores elements in hash table
  - `TreeSet`
    - Stores elements in tree

Maps

- `Map`
  - Associates keys to values
  - Cannot contain duplicate keys
  - Called *one-to-one mapping*
- Implementation classes
  - `Hashtable`, `HashMap`
    - Store elements in hash tables
  - `TreeMap`
    - Store elements in trees

Class `Collections`

- Class `Collections`
  - Provides `static` methods that manipulate collections
  - Implement algorithms for searching, sorting and so on
  - Collections can be manipulated polymorphically

Algorithm	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`.
`reverse`	Reverses the elements of a `List`.
`shuffle`	Randomly orders a `List`’s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` into another.
`min`	Returns the smallest element in a `Collection`.
`max`	Returns the largest element in a `Collection`.
`addAll`	Appends all elements in an array to a collection.
`frequency`	Calculates how many elements in the collection are equal to the specified element.
`disjoint`	Determines whether two collections have no elements in common.

Collections algorithms.

Software Engineering Observation

The collections framework algorithms are polymorphic. That is, each algorithm can operate on objects that implement specific interfaces, regardless of the underlying implementations.

Algorithm sort

- Sorts `List` elements
  - Order is determined by natural order of elements' type
  - `List` elements must implement the `Comparable` interface
  - Or, pass a `Comparator` to method `sort`
- Sorting in ascending order
  - Collections method `sort`
- Sorting in descending order
  - Collections static method `reverseOrder`
- Sorting with a `Comparator`
  - Create a custom `Comparator` class

Algorithm `reverse`, `fill`, `copy`, `max`/`min`

- `reverse`
  - Reverses the order of `List` elements
- `fill`
  - Populates `List` elements with values
- `copy`
  - Creates copy of a `List`
- `max`
  - Returns largest element in `List`
- `min`
  - Returns smallest element in `List`

Algorithm binarySearch

- binarySearch
  - Locates object in List
    - Returns index of object in List if object exists
    - Returns negative value if Object does not exist
      - Calculate insertion point
      - Make the insertion point sign negative
      - Subtract 1 from insertion point

Algorithms `addAll`, `frequency`, `disjoint`

- `addAll`
  - Insert all elements of an array into a collection
- `frequency`
  - Calculate the number of times a specific element appear in the collection
- `disjoint`
  - Determine whether two collections have elements in common

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
    - Method `sort` for sorting arrays

```java
// Using Arrays.java
// Using Java arrays.
import java.util.Arrays;

public class UsingArrays
{
   private int intArray[] = { 1, 2, 3, 4, 5, 6 };
   private double doubleArray[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
   private int filledIntArray[], intArrayCopy[];

   // constructor initializes arrays
   public UsingArrays()
   {
      filledIntArray = new int[ 10 ]; // create int array with 10 
elements
      intArrayCopy = new int[ intArray.length ];

      Arrays.fill( filledIntArray, 7 ); // fill with 7s
      Arrays.sort( doubleArray ); // sort do

      // copy array intArray into array intArrayCopy
      System.arraycopy( intArray, 0, intArrayCopy,
         0, intArray.length );
   } // end UsingArrays constructor

```

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 `intArray` into array `intArrayCopy`

```
   // output values in each array
   public void printArrays()
   {
      System.out.print( "doubleArray: " );
      for ( double doubleValue : doubleArray )
         System.out.printf( "%.1f ", doubleValue );

      System.out.print( "\nintArray: " );
      for ( int intValue : intArray )
         System.out.printf( "%d ", intValue );

      System.out.print( "\nfilledIntArray: " );
      for ( int intValue : filledIntArray )
         System.out.printf( "%d ", intValue );

      System.out.print( "\nintArrayCopy: " );
      for ( int intValue : intArrayCopy )
         System.out.printf( "%d ", intValue );

      System.out.println( "\n" );
   } // end method printArrays

   // find value in array intArray
   public int searchForInt( int value )
   {
      return Arrays.binarySearch( intArray, value );
   } // end method searchForInt

```

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

```
    // compare array contents
    public void printEquality()
    {
       boolean b = Arrays.equals( intArray, intArrayCopy );
       System.out.printf( "intArray %s intArrayCopy\n",
          ( b ? "==" : "!=" ) );

       b = Arrays.equals( intArray, filledIntArray );
       System.out.printf( "intArray %s filledIntArray\n",
          ( b ? "==" : "!=" ) );
    } // end method printEquality

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

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

```

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

```java
      int location = usingArrays.searchForInt( 5 );
      if ( location >= 0 )
         System.out.printf(
            "Found 5 at element %d in intArray\n", location );
      else
         System.out.println( "5 not found in intArray" );

      location = usingArrays.searchForInt( 8763 );
      if ( location >= 0 )
         System.out.printf(
            "Found 8763 at element %d in intArray\n", location );
      else
         System.out.println( "8763 not found in intArray" );
   } // end main
} // 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
```