16

Generic Collections

Test 3

- Monday, June 8
- 5% of the total grade
- all course material (in book Chapters 1 – 11 and 16)
- 5 problems
- practice test will be available today
- preparation for final exam

Last time

- data streams
- files and URLs
- various I/O examples

Objectives

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

Generics

- New feature since Java 5 (current version Java 8)
- Provide compile-time type safety
  - Catch invalid types at compile time
  - Only reference types can be used with generics
- Generic classes
  - A single class declaration
  - A set of related classes
- Generic methods (single method declaration)

Generic Classes

- **Generic classes**
  - **Use a simple, concise notation to indicate the actual type(s)**
  - **At compilation time, Java compiler ensures the type safety**

- **Parameterized classes**
  - **Also called parameterized types**
  - **E.g.,** `ArrayList< String >`

Java Collections

- **Java collections framework**
  - **Contain prepackaged data structures, interfaces, algorithms**
  - **Use generics**
  - **Provides reusable componentry**

Collections Overview

- **Collection**
  - **Data structure that can hold references to other objects**
- **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 since Java 5**

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.

Interface `Collection`

- 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, comparing and retaining objects
  - Provide method to return an `Iterator` object
    - Walk through collection and remove elements from collection

Software Engineering Observation

`Collection` is used commonly as a method parameter type to allow polymorphic processing of all objects that implement interface `Collection`.

Lists

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

Performance Tip

`ArrayLists` behave like `Vectors` without synchronization and therefore execute faster than `Vectors` because `ArrayLists` do not have the overhead of thread synchronization.

Vector

- Class `Vector`
  - Array-like data structures that can resize themselves dynamically
  - Contains a *capacity*
  - Grows by *capacity increment* if it requires additional space
  - Synchronized for multiple threads access

Sets

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

```
1  // 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 >( col
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  // 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 =
17          new TreeSet< String >( Arrays.asList( names ) );
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
```

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

```
1  // 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 Hash
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
28
29     // create St
30     StringTokenizer toke
31
32     // processi
33     while ( toke
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 ); // ge
41           map.put( word, count + 1 ); // inc
42        } // end if
43        else
44           map.put( word, 1 ); // add ne
45     } // end while
46  } // end method createMap
47
```

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

Use `StringTokenizer` method `hasMoreTokens` to determine whether there are more tokens in the string

Use `StringTokenizer` method `nextToken` to obtain the next token

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

Use method `get` to obtain the

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
52
53       // sort keys
54       TreeSet< String > sortedKeys = new TreeSet< String >( keys );
55
56       System.out.println( "Map contains:\nK
57
58       // generate output fo
59       for ( String key : so
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
```