18 Generic Class Parameters and the Java Collection Framework

18.1 `ArrayList`

18.1.1 Maintaining a Collection of Merchandise Items

Consider writing an application that interacts with the user to build a list of merchandise items. Each item is represented by its name and price (in dollars). Like the pizza applications from the previous chapter, the application must allow the user to read data from a file, write the data to a file, add an item, remove an item, and view the data. In addition, the program must allow the user to revise the name as well as the price of an item. Furthermore, the program must allow the user to rearrange the items on the list in the increasing order of the names or in the increasing order of the prices.

In building the application, we write the following classes:

- an object class named `Merchandise` whose objects represent a merchandise item,
- an object class named `PriceComparison` whose objects perform price comparison between `Merchandise` objects,
- an object class named `MerchandiseCollection` whose objects are collections of `Merchandise` objects, and
- an application class named `MerchandiseMain` that provides the method `main`.

18.1.2 The Class for Merchandise Item

`Merchandise` objects are compared with respect to their names using the `compareTo` method of `Merchandise`. Here is the initial part of the code for `Merchandise`. The class header (Line 1) has the `implements Comparable<Merchandise>` declaration. We plan to use a private `String` variable, `name`, and a private `int` variable, `price`, as the instance variables. The two variables represent the name and the price of a merchandise item. The constructor receives the values for these two variables, and then stores them in the instance variables. Since the parameters have the same names as the instance variables, `this.` is used for distinguishing between the instance variables and the parameters (Lines 8 and 9).

```java
public class Merchandise implements Comparable< Merchandise >
{
    // instance variables
  private String name;
  private int price;
    // constructor
  public Merchandise( String name, int price)
  {
    this.name = name;
    this.price = price;
  }
```

Listing 18.1 The class `Merchandise` (part 1). The class header, the instance variables, and the constructor

Next we show the getters, the setters, and the method `compareTo`. The method `compareTo` relies on the method `compareTo` of the class `String`.

```java
    // getters
  public String getName()
  {
    return name;
  }
  public int getPrice()
  {
    return price;
  }
    // setters
  public void setName( String name )
  {
    this.name = name;
  }
  public void setPrice( int price )
  {
    this.price = price;
  }
    // for implementing Comaparable
  public int compareTo( Merchandise o )
  {
    return name.compareTo( o.name );
  }
}
```

Listing 18.2 The class `Merchandise` (part 2). The getters, the setters, and the comparator

18.1.3 The `Comparator` Class

`PriceComparator` is an object class that implements an interface `Comparator`. `Comparator` is similar to `Comparable`, and is defined with a generic type parameter, say `<T>`. However, unlike `Comparable`, `Comparator` must be imported, and it is in the `java.util` package.

`Comparator` has only one abstract method, `compare`. The method `compare` receives two data of type `T` and returns an `int` value. This is not from the actual source code for `Comparator`, but after removing the comments, we can imagine that the source code is like this one:

```java
public interface Comparator<T>
{
  public int compare( T o1, T o2 );
}
```

The generic type parameter is highly complex topic. We only see a small fraction of coding with generic type parameters in this book. We mention the following general rules about generic type parameters:

- Any single capital letter can be used as a generic type parameter, so there are only 26 possibilities for a type parameter.
- Classes and interfaces that require multiple type parameters can be defined.
- To declare a class or an interface with multiple generic type parameters, assign distinct letters to the generic types, use a single pair of `<>` where the parameters appear together with commas in between (for instance, `<T,K,E>`).

Here is a source code for `PriceComparator`. The first line of the code is the required `import` statement. In Line 2, the generic type parameter appearing in `Comparator` is substituted with the actual data type, `Merchandise`. The method `compare` receives two `Merchandise` objects, and compares their prices by simply subtracting one price value from the other. Like the method `compareTo` of `Comparable`, the method `compare` is expected to report the result of comparison by the sign of the return value. In other words, the return value is strictly positive if `o1` is greater than `o2`, strictly negative if `o1` is smaller than `o2`, and 0 if `o1` is equal to `o2`.

```java
1 import java.util.Comparator;
2 public class PriceComparator implements Comparator< Merchandise >
3 {
4   public int compare( Merchandise o1, Merchandise o2 )
5   {
6     return o1.getPrice() - o2.getPrice();
7   }
8 }
```

Listing 18.3 The class `PriceComparator`

The code for `PriceComparator` does not have an instance variable. There is no need to define a constructor, but still a constructor can be called with `new PriceComparator()`.

18.1.4 The Collection of Merchandise Items That Uses `ArrayList`

18.1.4.1 `ArrayList`

To write the class `MerchandiseCollection`, we use the class `ArrayList`, from the `java.util` package. `ArrayList` is a class for maintaining a list of objects. `ArrayList` improves upon the arrays with capacity and size from Sect. 13.6. Following are some features of `ArrayList`:

- The elements in the list are given unique indexes starting from 0, and can be retrieved or changed with their indexes.
- It is possible to insert an element at a specific position, while preserving the order of appearance of other elements.
- For removing an element at a specific index, `ArrayList` shifts all the elements appearing after the index. The order of appearance of the remaining elements is thus preserved.
- There is no explicit limit on the number of elements to store the list.
- The class implements an interface `List`. `List` is in the package `java.util`. Because `ArrayList` implements `List`, for-each loops can be used to iterate over the elements in an `ArrayList`.
- An `ArrayList` object takes a generic type parameter with it. An instantiation of an `ArrayList` requires the specification of the type of the elements stored in the list.

If `T` is an object type, the declaration of an `ArrayList` data type, `list`, for storing elements of type `T`, and its instantiation will be as follows:

```java
  ArrayList<T> list;
  list = new ArrayList<T>();
```

`ArrayList` offers many instance methods, here is a short list:

- `int size()`: The method returns the number of elements currently stored in the list.
- `boolean add( T e )`: The method attempts to add the element `e` at the end of the list, and then returns whether or not the operation was successful.
- `boolean add( int index, T e )`: The method attempts to inserts the element `e` at the position `index` in the list, and then returns whether or not the operation was successful.
- `void clear()`: The method removes all the elements from the list.
- `T get( int index )`: The method returns the element at position `index`.
- `T remove( int index )`: The method attempts to remove the element at position `index`, and then returns the element removed.
- `T set( int index, T e )`: The method replaces the element at position `index` with `e`, and then returns the element that was stored at `index` prior to the replacement.

18.1.4.2 The Instance Variables, the "Getters", and the "Setters"

Here is a source code for `MerchandiseCollection`. The class has two imports, `java.util.*` and `java.io.*`. There is only one instance variable, `theList`. The type of the instance variable is `ArrayList<Merchandise>`. The variable is instantiated with `new ArrayList <Merchandise>()`. The instantiation produces an object of `ArrayList<Merchandise>` with no elements being stored in the list.

```
 1 import java.io.*;
 2 import java.util.*;
 3 public class MerchandiseCollection
 4 {
 5      //---- instance variable
 6   private ArrayList<Merchandise> theList;
 7      //---- constructor
 8   public MerchandiseCollection()
 9   {
10     theList = new ArrayList<Merchandise>();
11   }
```

Listing 18.4 The class `MerchandiseCollection` (part 1). The class header, the instance variable, and the constructor

The “getters” and “setters” appear next. The method `size()` (Lines 13) returns `theList.size()` (Line 15). The returned value is the number of merchandise items stored in the list. The method `getName( int i )` (Lines 17) obtains the element at index `i` and returns its names. This is accomplished by `theList.get( i ).getName()` (Line 19). The method `getPrice( int i )` (Lines 21) obtains the element at index `i` and returns its price. This is accomplished by `theList.get( i ).getPrice()` (Line 23).

```
12       //---- getters
13   public int size()
14   {
15     return theList.size();
16   }
17   public String getName( int i )
18   {
19     return theList.get( i ).getName();
20   }
21   public int getPrice( int i )
22   {
23     return theList.get( i ).getPrice();
24   }
```

Listing 18.5 The class `MerchandiseCollection` (part 2). The “getters”

The method `add( String name, int price )` (Line 26) adds a merchandise item composed of `name` and `price` at the end of the list (Line 28). The method `remove( int i )` (Line 30) removes the element at `i` (Line 32). The method `set( int i, String name, int price )` (Line 34) replaces the name and price of the element at `i` with the values given as parameters (Lines 36 and 37).

```
25      //---- setters
26    public void add( String name, int price )
27    {
28      theList.add( new Merchandise( name, price ) );
29    }
30    public void remove( int i )
31    {
32      theList.remove( i );
33    }
34    public void set( int i, String name, int price )
35    {
36      theList.get( i ).setName( name );
37      theList.get( i ).setPrice( price );
38    }
```

Listing 18.6 The class `MerchandiseCollection` (part 3). The “setters”

18.1.4.3 The File Read/Write

The data format for a merchandise collection requires two lines per item, with the first line representing the name and the second line representing the price. The name may contain the white space character. Since the length of an `ArrayList` is not fixed, the data file does not need to provide information about how many elements are in the list.

Reading the data from a file is accomplished using the method `read`. The method `read` receives a file path as its formal parameter (Line 40). The method opens a `Scanner` to read from the `File` object instantiated with the file path (Line 42). So long as there is a token remaining in the file (Line 43), the method reads one line as the name, and then another line as the price. The second line is converted to an `int` value using `Integer.parseInt`. These two values are passed to the method `add`. This sequence of actions is compressed into a single statement appearing in Line 45. The method has a `throws FileNotFoundException` declaration because the constructor of a `Scanner` may throw an exception of the type. After reading is completed, the `Scanner` is closed using the method `close` (Line 47).

Writing the data to a file is accomplished using the method `write`. The method receives a file path as its parameter (Line 49). The method opens a `PrintStream` with the file specified in the formal parameter (Line 51). Then, the method uses a for-each loop to iterate over the elements in the list. The loop header `for ( Merchandise m : theList )` (Line 52) implies that the elements are retrieved in sequence from the list `theList`, the type of the elements is `Merchandise`, and the element can be referred to by the variable `m`. For each element retrieved, the method writes two lines into the file. The first line is the name and the second is the price. The `printf` in Line 54 accomplishes both using the format `%s\n%d\n`, where the first placeholder is for the name and the second is for the price. After finishing to write the information of the elements, the method closes the stream using the method `close`. This method also has a `throws FileNotFoundException` declaration.

18.1.4.4 The Sorting Methods

The last part of `MerchandiseCollection` has two methods for rearranging the elements in the list. In the case of arrays, sorting is accomplished using the method `sort` in the class `Arrays`. In the case of `ArrayList`, the class that provides the functions for sorting is `Collections`. The method `sortByName` (Lines 59–62) sorts the elements using the method `compareTo` that is natively available in the class `Merchandise`.

```
39       //---- read & write
40    public void read( String fileName ) throws FileNotFoundException
41    {
42      Scanner sc = new Scanner( new File( fileName ) );
43      while ( sc.hasNext() )
44      {
45        add( sc.nextLine(), Integer.parseInt( sc.nextLine() ) );
46      }
47      sc.close();
48    }
49    public void write( String fileName ) throws FileNotFoundException
50    {
51      PrintStream st = new PrintStream( new File( fileName ) );
52      for ( Merchandise m : theList )
53      {
54        st.printf( "%s\n%d\n", m.getName(), m.getPrice() );
55      }
56      st.close();
57    }
```

Listing 18.7 The class `MerchandiseCollection` (part 4). The methods `read` and `write`

The method `sortByPrice` (Lines 63–66) sorts the elements using the method `compare` defined in the class `PriceComparator`. The method `sort` of `Collections` has a version that receives the comparison method as an additional parameter. The accepted type for the parameter is a `Comparator<T>` where `T` is the type of the elements in list. Since we have written a class `PriceComparator` that implements `Comparator<Merchandise>`, an object of `PriceComparator` can be used as the method for comparison. Line 65 has `new PriceComparator()` as the second parameter, and the object returned by the constructor is passed to the method `sort`.

```
58       //--- sorting
59    public void sortByName()
60    {
61      Collections.sort( theList );
62    }
63    public void sortByPrice()
64    {
65      Collections.sort( theList, new PriceComparator() );
66    }
67  }
```

Listing 18.8 The class `MerchandiseCollection` (part 5). The methods `sortByName` and `sortByPrice`

18.1.5 The Main Class

18.1.5.1 The Method for Printing Prompt

In `MerchandiseMain`, the user is presented with as many as 11 choices for an action to be performed, including one to terminate the program. The class has a method, `printPrompt`, to print these choices neatly. The choices are presented in a table-like manner with three choices appearing in one line, as shown next:

```
----Choose action to be performed----
  1. add an item      2. remove an item   3. view list
  4. sort by name     5. sort by price    6. change item
  7. change name      8. change price     9. read from a file
  10. write to a file 0. quit
```

The source code of `printPrompt` appears next.

```java
import java.util.*;
import java.io.*;
public class MerchandiseMain
{
  public static void printPrompt()
  {
    System.out.println( "----Choose action to be performed----" );
    System.out.printf(
          "  %-20s%-20s%-20s\n  %-20s%-20s%-20s\n"
        + "  %-20s%-20s%-20s\n  %-20s%-20s\n",
         "1. add an item", "2. remove an item", "3. view list       ",
         "4. sort by name", "5. sort by price", "6. change item",
         "7. change name", "8. change price", "9. read from a file",
         "10. write to a file", "0. quit" );
  }

```

Listing 18.9 The class `MerchandiseMain` (part 1). The method `printPrompt`

18.1.5.2 The Method `main`

The method `main` of `MerchandiseMain` uses a `MerchandiseCollection` variable, `data`, to record the collection (Line 19), a `Scanner` variable, `keyboard`, to receive input from the keyboard (Line 20), a `String` variable, `name`, to store the name information (Line 21), and three `int` variables, `choice`, `price`, and `pos`, for storing the index to the action to be performed, the price value, and the position of a merchandise item in the list (Line 22). The method is a large while-loop whose continuation condition is `choice != 0` (Line 23). The initial value of `choice` is 1, so the loop-body is executed at least once.

The loop-body is placed in a `try-catch`. The try-part conducts all the necessary action. The catch-part is for recovering from some anticipated run-time errors. The choice of action is received after presenting the prompt (Lines 26–28).

18.1.5.3 Adding and Removing

After receiving the number indicating the action to be performed, the program uses a switch-statement to direct the flow to the required method. The choice 1 is for adding an item (Line 31). The program receives the name (Lines 32 and 33) and the price (Lines 34 and 35), and then calls the method `add` of the `MerchandiseCollection` variable `data` with the name and price received (Line 36). The program uses `nextLine` exclusively to receive input from the user. To interpret the input from the user as an integer, the program uses `Integer.parseInt`. If the user enters a `String` data that

```
17    public static void main( String[] args )
18    {
19      MerchandiseCollection data = new MerchandiseCollection();
20      Scanner keyboard = new Scanner( System.in );
21      String name;
22      int choice = 1, price, pos;
23      while ( choice != 0 )
24      {
25        try {
26          printPrompt();
27          System.out.print( "Enter your choice: " );
28          choice = Integer.parseInt( keyboard.nextLine() );
```

Listing 18.10 The class `MerchandiseMain` (part 2). The start of the method `main`, including the variable declarations, the start of the loop, and the try-clause

cannot be converted to an integer, a run-time error of `NumberFormatException` occurs. This error is handled by a catch-clause appearing in Lines 91 and 92. `NumberFormatException` may occur in other places of the code, and all of them are handled by this catch-clause. For `removing` an item (Line 38), the program receives the position of the item to be removed (Lines 39 and 40), and then calls the method `remove` (Line 41).

```
29          switch ( choice )
30          {
31          case 1: // add
32            System.out.print( "Enter the name: " );
33            name = keyboard.nextLine();
34            System.out.print( "Enter the price: " );
35            price = Integer.parseInt( keyboard.nextLine() );
36            data.add( name, price );
37            break;
38          case 2: // remove
39            System.out.print( "Enter the position: " );
40            pos = Integer.parseInt( keyboard.nextLine() );
41            data.remove( pos );
42            break;
```

Listing 18.11 The class `MerchandiseMain` (part 3). The part that handles adding and removing an item

18.1.5.4 Viewing and Sorting

Choices 3, 4, and 5 are for viewing and sorting the elements. The program presents the data on the screen after sorting, so the choice 3 can be thought of as a special case where sorting is not performed at all. The three cases have the same entry points (Lines 43–45) and the following series of actions is performed:

- If the action is sorting by the name (i.e., the choice is 4), call `sortByName` (Lines 46–49).
- If the action is sorting by the price (i.e., the choice is 5), call `sortByPrice` (Lines 50–53).
- If the action is viewing the data (i.e., the choice is 3), call neither.

After sorting the elements, the program prints the items on the list using a for-loop, where the number of items in the list is obtained using the method `size` (Line 54). For each item, the program prints the index, the name, and the price (Lines 56–58).

```
43          case 3: // view
44          case 4: // sort by name
45          case 5: // sort by price
46            if ( choice == 4 )
47            {
48              data.sortByName();
49            }
50            else if ( choice == 5 )
51            {
52              data.sortByPrice();
53            }
54            for ( pos = 0; pos < data.size(); pos ++ )
55            {
56              System.out.printf( "%3d: %s : $%,d\n", pos,
57                  data.getName( pos ), data.getPrice( pos ) );
58            }
59            break;
```

Listing 18.12 The class `MerchandiseMain` (part 4). The part that handles sorting and viewing the data

18.1.5.5 Making Changes

The choices 6, 7, and 8 are for making changes on an element in the list. The three cases have the same entry points (Lines 60–62). If the choice is 6, the change is for both the name and the price. If the choice is 7, the change is for the name only. If the choice is 8, the change is for the price only. The program first obtains the index to the item to be updated, and stores it in the variable `pos` (Lines 63 and 64). To handle the three possibilities in one place, the program obtains the present values of the name and price of the item at `pos`, and stores them in `name` and `price` (Lines 65 and 66). The idea is that the item at `pos` will be updated with the values of `name` and `price`, but prior to the update, the user will be asked to provide new values for the two variables. Which of the two variables the user can update depends on the choice of the action.

- If the choice is not 8 (i.e., the choice is either 6 or 7), the new name is obtained and stored in `name` (Lines 68–72).
- If the choice is not 7 (i.e., the choice is either 6 or 8), the new price is obtained and stored in `price` (Lines 73–77).

Finally, the program stores the values of `name` and `price` by calling the method `set` (Line 78).

```
        case 6: // change
        case 7: // change name
        case 8: // change price
          System.out.print( "Enter the position: " );
          pos = Integer.parseInt( keyboard.nextLine() );
          name = data.getName( pos );
          price = data.getPrice( pos );
          System.out.printf( "The item is: %s : $%,d\n", name, price );
          if ( choice != 8 )
          {
            System.out.print( "Enter the new name: " );
            name = keyboard.nextLine();
          }
          if ( choice != 7 )
          {
            System.out.print( "Enter the new price: " );
            price = Integer.parseInt( keyboard.nextLine() );
          }
          data.set( pos, name, price );
          break;
```

Listing 18.13 The class `MerchandiseMain` (part 5). The part that is responsible for changing the name and the price

18.1.5.6 File Read/Write

If the choice is 9, the program receives a file path from the user, and then calls the method `read` (Lines 80–83). If the choice is 10, the program receives a file path from the user and calls the method `write` (Lines 84–87). In the case where the choice is 0, the program prints a message to inform that the program will terminate (Lines 88 and 89).

```
80          case 9: // read from a file
81            System.out.print( "Enter an input file path: " );
82            data.read( keyboard.nextLine() );
83            break;
84          case 10: // write to a file
85            System.out.print( "Enter an output file path: " );
86            data.write( keyboard.nextLine() );
87            break;
88          case 0: // quit
89            System.out.println( "...Terminating" );
90          }
```

Listing 18.14 The class `MerchandiseMain` (part 6). The part responsible for reading from a data file and writing to a data file

There are two catch-clauses. One is for catching a run-time error that occurs when the user enters a line that cannot be interpreted as an integer, and the other is for catching `FileNotFoundException` (Lines 93 and 94). The program prints an error message stating the error, and then returns to the loop.

Here is an example of using the program. Consider maintaining a list of real estate properties, with the address and the price. Suppose a text file, `propertyData.txt`, has seven properties in the format readable by the program, as follows:

```
      } catch ( NumberFormatException e ) {
        System.out.println( "Incorrect input!!!" );
      } catch ( FileNotFoundException e ) {
        System.out.println( "No such file found|!!" );
      }
    }
  }
}
```

Listing 18.15 The class `MerchandiseMain` (part 7). The `catch` clauses

```
10 Wilkinson Road
343450
11 Wilkinson Drive
766000
34 Coral Way
807500
7900 Plainview Drive
195500
7000 Flamingo Drive
790000
1 Presidential Place
1535000
2 Marigold Terrace
615000
```

Starting with this file, we can manipulate the data as follows:

```
----Choose action to be performed----
  1. add an item      2. remove an item    3. view list
  4. sort by name     5. sort by price     6. change item
  7. change name      8. change price      9. read from a file
  10. write to a file 0. quit
Enter your choice: 9
Enter an input file path: propertyData.txt
----Choose action to be performed----
  1. add an item      2. remove an item    3. view list
  4. sort by name     5. sort by price     6. change item
  7. change name      8. change price      9. read from a file
  10. write to a file 0. quit
Enter your choice: 5
  0: 7900 Plainview Drive : $195,500
  1: 10 Wilkinson Road : $343,450
  2: 2 Marigold Terrace : $615,000
  3: 11 Wilkinson Drive : $766,000
  4: 7000 Flamingo Drive : $790,000
  5: 34 Coral Way : $807,500
  6: 1 Presidential Place : $1,535,000
----Choose action to be performed----
  1. add an item      2. remove an item    3. view list
  4. sort by name     5. sort by price     6. change item
  7. change name      8. change price      9. read from a file
  10. write to a file 0. quit
Enter your choice: 1
Enter the name: 71 Canary Drive
Enter the price: 199700
```

```
----Choose action to be performed----
  1. add an item      2. remove an item   3. view list
  4. sort by name     5. sort by price    6. change item
  7. change name      8. change price     9. read from a file
  10. write to a file 0. quit
Enter your choice: 1
Enter the name: 2 Atkins Road
Enter the price: 280000
----Choose action to be performed----
  1. add an item      2. remove an item   3. view list
  4. sort by name     5. sort by price    6. change item
  7. change name      8. change price     9. read from a file
  10. write to a file 0. quit
Enter your choice: 4
  0: 1 Presidential Place : $1,535,000
  1: 10 Wilkinson Road : $343,450
  2: 11 Wilkinson Drive : $766,000
  3: 2 Atkins Road : $280,000
  4: 2 Marigold Terrace : $615,000
  5: 34 Coral Way : $807,500
  6: 7000 Flamingo Drive : $790,000
  7: 71 Canary Drive : $199,700
  8: 7900 Plainview Drive : $195,500
----Choose action to be performed----
  1. add an item      2. remove an item   3. view list
  4. sort by name     5. sort by price    6. change item
  7. change name      8. change price     9. read from a file
  10. write to a file 0. quit
Enter your choice: 2
Enter the position: 2
----Choose action to be performed----
  1. add an item      2. remove an item   3. view list
  4. sort by name     5. sort by price    6. change item
  7. change name      8. change price     9. read from a file
  10. write to a file 0. quit
Enter your choice: 3
  0: 1 Presidential Place : $1,535,000
  1: 10 Wilkinson Road : $343,450
  2: 2 Atkins Road : $280,000
  3: 2 Marigold Terrace : $615,000
  4: 34 Coral Way : $807,500
  5: 7000 Flamingo Drive : $790,000
  6: 71 Canary Drive : $199,700
  7: 7900 Plainview Drive : $195,500
----Choose action to be performed----
  1. add an item      2. remove an item   3. view list
  4. sort by name     5. sort by price    6. change item
  7. change name      8. change price     9. read from a file
  10. write to a file 0. quit
```

```
Enter your choice: 8
Enter the position: 7
The item is: 7900 Plainview Drive : $195,500
Enter the new price: 205000
----Choose action to be performed----
  1. add an item      2. remove an item    3. view list
  4. sort by name     5. sort by price     6. change item
  7. change name      8. change price      9. read from a file
  10. write to a file 0. quit
Enter your choice: 5
  0: 71 Canary Drive : $199,700
  1: 7900 Plainview Drive : $205,000
  2: 2 Atkins Road : $280,000
  3: 10 Wilkinson Road : $343,450
  4: 2 Marigold Terrace : $615,000
  5: 7000 Flamingo Drive : $790,000
  6: 34 Coral Way : $807,500
  7: 1 Presidential Place : $1,535,000
----Choose action to be performed----
  1. add an item      2. remove an item    3. view list
  4. sort by name     5. sort by price     6. change item
  7. change name      8. change price      9. read from a file
  10. write to a file 0. quit
Enter your choice: 10
Enter an output file path: propertyData2.txt
----Choose action to be performed----
  1. add an item      2. remove an item    3. view list
  4. sort by name     5. sort by price     6. change item
  7. change name      8. change price      9. read from a file
  10. write to a file 0. quit
Enter your choice: 0
...Terminating
```

The execution example has resulted in the following revised data file, `propertyData2.txt`:

```
71 Canary Drive
199700
7900 Plainview Drive
205000
2 Atkins Road
280000
10 Wilkinson Road
343450
2 Marigold Terrace
615000
7000 Flamingo Drive
790000
34 Coral Way
807500
1 Presidential Place
1535000
```

18.2 The Dynamic Maintenance of the Largest K Values

We often encounter a problem of writing a code for selecting, from an indefinite number of elements, a fixed number of, say k, best ones. A straightforward solution to this problem is to store all the elements in a list, sort the list, and then select the k best ones. However, if the total number of elements considered is much larger than k, this approach may be slightly wasteful of memory. An alternative to this approach is to use a shorter list, whose length will be no more than k + 1. At the start the list has 0 elements. Each element is added to the list, and the elements in the list are sorted. After that, if the list has k + 1 elements, the "worst" of the k + 1 elements is removed, thereby bringing the length of the list back to k. When all the elements have been processed, the list consists of the k best ones.

Using `ArrayList`, this idea can be easily implemented. Let `E` be a generic data type that represents the data type of the elements to be processed, such that `E` is comparable. We use an `ArrayList<E>` variable, `theList`, to maintain the best elements that have seen so far. We initialize `theList` with `new ArrayList<E>()`. To process a new element, say `o`, we add `o` to `theList`, use `Collections.sort` to sort the list, and if `theList.size()` is equal to k + 1, remove the candidate at position 0, because `sort` sorts the element in increasing order. After all the candidates have been processed, we may need to access the best ones. The access is provided by the method `get( int i )` that returns the element at rank `i` in the list.

Here is the class `TopK` written based upon the above discussion. The syntax for stating that `E` has `compareTo` is `<E extends Comparable<E»`. Since `E` is the parameter associated with `TopK`, the overall class declaration becomes the one in Line 2:

```
public class TopK<E extends Comparable<E»
```

The class has two instance variables (Lines 4 and 5). One is an `ArrayList<E>` named `theList` and the other is an `int` variable, `k`. The variable `k` specifies the number of elements to keep. The constructor receives the value for `k`, stores it in `this.k` (Line 9), and instantiates the list (Line 10).

```
 1  import java.util.*;
 2  public class TopK< E extends Comparable< E > >
 3  {
 4    private ArrayList<E> theList;
 5    private int k;
 6
 7    public TopK( int k )
 8    {
 9      this.k = k;
10      theList = new ArrayList<E>();
11    }
```

Listing 18.16 The class `TopK` (part 1)

The method `get` receives an `int` value named `i` as the parameter (Line 12), and returns the element at rank `i`. This is accomplished by returning the element obtained through `get( k - i )` of the list because the elements appear in the decreasing order of their ranks (Line 14). The method `add` receives an element `o` of type `E` (Line 16), adds it to `theList` using the method `add` of `ArrayList`, sorts the list using `Collections.sort`, and then, if the length of `theList` is equal to `k + 1` (Line 20), removes the element at index 0 using the method `remove` of `ArrayList` (Line 22).

```java
  public E get( int i )
  {
    return theList.get( k - i );
  }
  public void add( E o )
  {
    theList.add( o );
    Collections.sort( theList );
    if ( theList.size() == k + 1 )
    {
      theList.remove( 0 );
    }
  }
}
```

Listing 18.17 The class TopK (part 2)

Here is a source code for testing the class TopK. The program instantiates a TopK<Integer> object top with the value of k set to 10 (Line 6). The program then adds integers 1, ..., 10,000 one after another in top (Lines 7–10), and then retrieves the recorded top ten elements (Line 11–14). Since the elements appear in increasing order. The program generates ranks from 1 to 10 with a variable named i (Line 11), and prints the element located at rank i (Line 13).

```java
 1  public class TopKTest
 2  {
 3    public static void main( String[] arg )
 4    {
 5      System.out.println( "Test using integers 1..10000" );
 6      TopK<Integer> top = new TopK<Integer>( 10 );
 7      for ( int i = 1; i <= 10000; i ++ )
 8      {
 9        top.add( i );
10      }
11      for ( int i = 1; i <= 10; i ++ )
12      {
13        System.out.printf( "%2d: %d\n", i, top.get( i ) );
14      }
```

Listing 18.18 The class TopKTest (part 1). Top 10 of 1, ..., 10,000

The program then repeats the same action, this time with a 10,000 random real number with values 0 and 1,000,000 (Lines 17–20). The instantiation (Line 16) uses new TopK<Double>(10).

```
    System.out.println( "Test using random double" );
    TopK<Double> newtop = new TopK<Double>( 10 );
    for ( int i = 1; i <= 10000; i ++ )
    {
      newtop.add( Math.random() * 1000000 );
    }
    for ( int i = 1; i <= 10; i ++ )
    {
      System.out.printf( "%2d: %f\n", i, newtop.get( i ) );
    }
  }
}
```

Listing 18.19 The class `TopKTest` (part 2). Testing with random real numbers

Here is an execution example of the code. In the first test, the top ten numbers are 10,000, ..., 9991 and the program correctly identifies them. In the second test, the top ten numbers are expected to be close to the upper bound, 1,000,000. In the example, the top ten numbers are indeed greater than 999,000.

```
Test using integers 1..10000
 1: 10000
 2: 9999
 3: 9998
 4: 9997
 5: 9996
 6: 9995
 7: 9994
 8: 9993
 9: 9992
10: 9991
Test using random double
 1: 999990.863047
 2: 999867.865257
 3: 999831.136887
 4: 999685.778237
 5: 999663.635342
 6: 999629.397141
 7: 999591.778661
 8: 999538.840688
 9: 999424.382109
10: 999248.272207
```

18.3 The Java Collection Framework

18.3.1 The Framework

The `List` and its implementation `ArrayList` belong to a large group of interfaces, abstract classes, and concrete classes, called the **Java Collection Framework**. The Java Collection Framework provides various tools for dynamically maintaining a collection of data through such actions as adding, removing, and searching. Some interfaces and classes in the framework are shown in Fig. 18.1. Like `ArrayList`, all interfaces and classes in the framework take a generic type parameter. In the following description, we use `E` to refer to the generic type parameter.

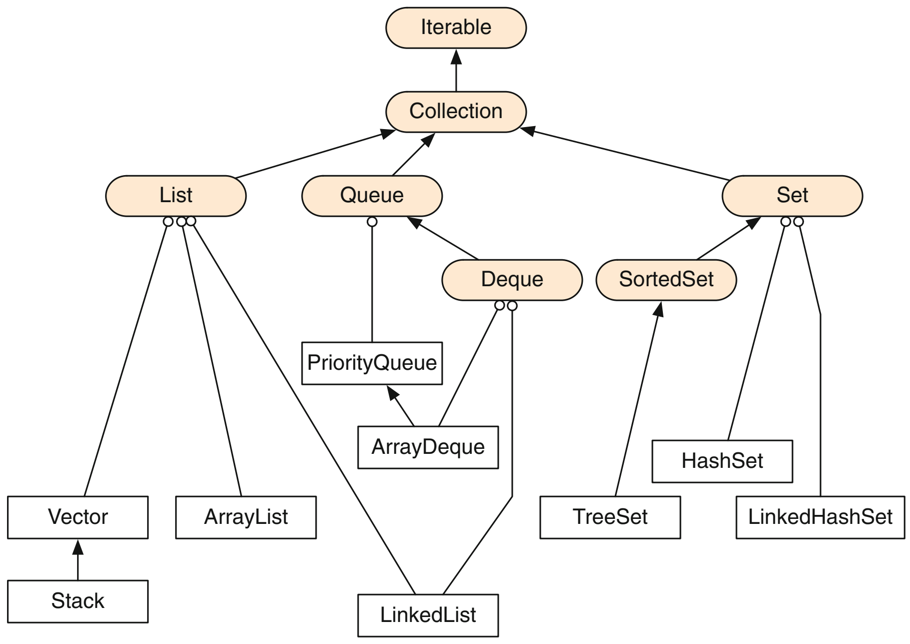

Fig. 18.1 The Java Collection Framework. The squashed rectangles represent interfaces and the rectangles represent classes. The arrows represent extensions and the round-head arrows represent implementations

An interface `Iterable<E>` is at the top of the hierarchy in the diagram. `Iterable<E>` defines only one method, `iterator`. The method `iterator` returns an object of type `Iterator<E>`. `Iterator<E>` is an interface that defines two abstract methods that enable iterative access to the elements in some collection of data of type `E`. The two methods are `hasNext` and `next`. The method `hasNext` returns a `boolean` value. The return value of `hasNext` is interpreted as the indication of whether or not there is any element remaining in the iteration. The method `next` returns an object of type `<E>` and is interpreted as the next element in the iteration. If a class implements `Iterable<E>`, for-each loops are available. The two methods are reminiscent of the methods having the same names in `Scanner`. Although the methods have the same names, `Scanner` does not implement `Iterable<String>`.

In the diagram, an interface `Collection<E>` appears immediately under `Iterable<E>`. `Collection<E>` is a collection of data of type `E`. The elements in the collection may not appear sequentially. Here are some methods of `Collection<E>`.

- `add( E o )`: This method attempts to add the data `o` of type `E` to the collection, and then returns a `boolean` indicating whether or not the attempt was successful.
- `clear()`: This method empties the collection. The return type is `void`.
- `contains( Object o )`: This method returns a `boolean` value representing whether or not the collection contains an element matching `o`.
- `isEmpty()`: This method returns a `boolean` representing whether or not the collection is empty.

- `remove( Object o )`: This method removes one occurrence of an object matching `o`. The method returns a `boolean` representing whether or not such an element has been removed.
- `size()`: This method returns an `int` representing the number of elements in the collection.

The interfaces `List<E>`, `Queue<E>` (pronounced "cue" as in the "billiard cue"), `Deque<E>` (pronounced "deck" as in the "deck of cards"), `Set<E>`, and `SortedSet<E>` extend `Collection<E>`, and represent certain organizations of the elements in the collections.

- `List<E>` is a sequential list of data. Like arrays, `List<E>` assigns sequential indexes starting from 0 are assigned to its elements. The elements are accessible with their indexes.
- `Queue<E>` is a sequential list of data, but only the element at one end can be examined or removed, elements can be added only at the other end.
- `Deque<E>` is a sub-interface of `Queue<E>` and is a "double-ended" version of `Queue<E>`. Addition, examination, and removal can be made at each end, but nowhere else in the collection.
- `Set<E>` is a set of data without duplication.
- `SortedSet<E>` is a sub-interface of `Set<E>`, where the elements are in order.

18.3.2 Some Classes from the Framework

`ArrayList<E>` we studied earlier in the chapter is a class that implements `List<E>`.

Here we explore two additional classes, `LinkedList<E>` and `HashSet<E>`, from the diagram. The former implements `List<E>`, `Queue<E>`, and `Deque<E>`. The latter implements `Set<E>` and `SortedSet<E>`. `LinkedList<E>` uses a chain of objects. Each object in the chain holds a value of type `E`, a reference to its next object, and a reference to its previous object. The chain has two ends, the "head" and the "tail". The "previous" object of the "head" is `null` and the "next" object of the "tail" is `null`. As in the case of `ArrayList<E>`, `LinkedList<E>` provides the methods `add( E o )`, `get( int i )`, `set( int i, E o )`, and `remove( int i )`. In addition, the class offers access to the "head" and the "tail" (see Fig. 18.2). Here are some methods available in `LinkedList<E>` beyond those defined in `List<E>`.

- `addFirst( E o )` inserts `o` to the chain as its "head" element (the present "head", if it exists, will become the "next" element of the new "head"). The return type of this method is `void`.
- `addLast( E o )` inserts `o` to the chain as its "tail" element (the present "tail", if it exists, will become the 'previous" element of the new "tail"). The return type of this method is `void`.
- `getFirst()` returns the "head" element of the chain, if it exists.
- `getLast( E o )` returns the "tail" element of the chain, if it exists.
- `removeFirst()` removes the "head" element of the chain, if it exists, and returns it.
- `removeLast( E o )` removes the "tail" element of the chain, it if exists, and returns it.

Because `LinkedList<E>` offers access to each end of the list, the function to be performed in `Deque<E>` are already available in `LinkedList<E>`. The abstract methods of `Queue<E>` and of `Deque<E>` are overridden in `LinkedList<E>` through adaptations of the above methods. Here are some of such methods. For `Queue<E>`, we have the following:

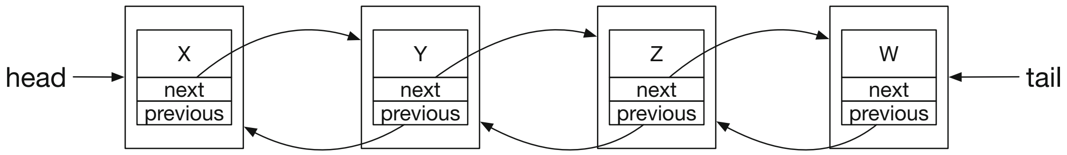

Fig. 18.2 A `LinkedList`. `X`, `Y`, `Z`,and `W` are the data elements in the collection. `next` and `previous` present the references to the "next" and "previous" elements

- `offer( E o )` adds the element `o` at the "head" of the queue, and returns a `boolean` representing whether or not the attempt was successful.
- `peek()` returns, without removing, the element at the "head" of the queue. If the queue is empty, the method returns `null`.
- `poll()` removes the element at the "head" of the queue and returns it. If the queue is empty, the method returns `null`.

The methods for `Deque<E>` that are overridden in `LinkedList<E>` include the following:

- `offerFirst( E o )` adds the element `o` at the "head" of the queue, and returns a `boolean` representing whether or not the element has been added.
- `offerLast( E o )` adds the element `o` at the "tail" of the queue and returns a `boolean` representing whether or not the element has been added.
- `peekFirst( E o )` returns, without removing it, the "head" element of the queue.
- `peekLast( E o )` returns, without removing it, the "tail" element the queue.
- `pollFirst( E o )` removes and returns the "head" element of the queue.
- `pollLast( E o )` removes and returns the "tail" element of the queue.

`HashSet<E>` uses a structure called a **hash table** to record the elements in the collection. A hash table is an array. Each element has an assigned index in the array. The assigned index is calculated from a value given by a so-called **hash function**. The index of an element is the remainder of the value of the hash function of the element when divided by the length of the array. Adding an element, removing an element, and search for an element are carried out by examining the array at the assigned index of the element. Since the assigned indexes are based upon the hash function, it may happen that two elements to be stored in the hash-table have the same indexes. We call such a phenomenon a **collision**. Collisions are usually resolved by allocating a list of `E` to each slot of the array, rather than making each slot available for one element exclusively (see Fig. 18.3). Adding an element, removing an element, and searching for an element are executed on the list at the index assigned to the element. The order in which the elements appear in the `HashSet<E>` depends on the length of the hash table, the hash function, and the mechanisms used for adding an element and removing an element on the lists representing the slots of the hash table. `HashSet<E>` implements `add( E o )`, `clear()`, `contains( Object o )`, `remove( Object o )`, `isEmpty()`, and `size()`.

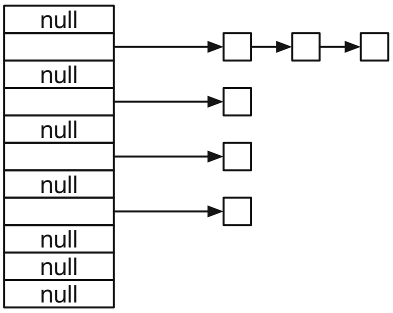

Fig. 18.3 A hash table

18.3.3 A Demonstration

We present a program that demonstrates how the interfaces `List`, `Queue`, and `Set` operate. `List` and `Queue` are implemented with `LinkedList` and `Set` is implemented with `HashSet`. For the type parameter, `String` is chosen.

The method `main` of the class calls three methods, `listExperiment`, `queueExperiment`, and `setExperiment` (Lines 6–8). They demonstrate the use of `List<String>`, `Queue <String>`, and `Set<String>`. The three methods have common tasks: (i) add the elements from a `String` array constant `SEQUENCE` (appearing in Lines 11–13), (ii) remove some elements, and then (iii) print the contents of the collection.

The task (iii) is accomplished by the method `printInfo`. The method `printInfo` receives a `Collection<String>` data named `c` as its parameter (Line 15). The methods executes the following:

- The method reports the size of `c` obtained by the method `size` of `Collection<String>` (Line 18).
- The method reports the contents of `c` using the for-each loop (Lines 19–24).
- The method prints, for each element from `SEQUENCE`, whether or not the element appears in the collection `c`. The method `contains` of `Collection<String>` is used for this purpose (Lines 25–29).

The method `listExperiment` (Line 32) instantiates a `LinkedList<String>` object, and then assigns the object to a `List<String>` variable named `myList` (Line 35). The method then adds each element of `SEQUENCE` successively twice using the method `add` of `List<String>` (Lines 36–40). Next, the method removes the element at position 3 (the position indexes start from 0) three times successively (Lines 41–47). To remove an element, the method uses the method `remove` of `List<String>` (Line 43). Since the method `remove` of `List<String>` returns the element that has been removed, `listExperiment` reports the element returned after each removal (Line 44). Furthermore, immediately after each removal, using the method `get` of `List<String>`, the method prints the element at index 3 (Lines 45 and 46). Next, `listExperiment` replaces the element at position 3 with a new `String` literal, `"jackie"`, using the method `set` of `List<String>`, and reports the element at index 3 after the modification (Lines 48–51). Finally, the method calls `printInfo` to print the information about `myList` (Line 52).

```
 1 import java.util.*;
 2 public class CollectionExperiment
 3 {
 4   public static void main( String[] args )
 5   {
 6     listExperiment();
 7     queueExperiment();
 8     setExperiment();
 9   }
10
11   public static final String[] SEQUENCE = new String[]{
12       "suzie", "grace", "carol", "deborah", "janis"
13   };
14
15   public static void printInfo( Collection<String > c )
16   {
17     System.out.println();
18     System.out.printf( "The collection has %d elements\n", c.size() );
19     System.out.print( "The elements are:" );
20     for ( String w : c )
21     {
22       System.out.print( " " + w );
23     }
24     System.out.println();
25     System.out.println( "The results of membership tests are:" );
26     for ( String w : SEQUENCE )
27     {
28       System.out.printf( "%20s:%s\n", w, c.contains( w ) );
29     }
30   }
31
```

Listing 18.20 The class `CollectionExperiment` (part 1). The method `main`, the array constant `SEQUENCE`, and the method `printfInfo`

After the initial insertion of the elements, the contents of `myList` are expected to be:

```
"suzie", "suzie", "grace", "grace", "carol", "carol", "deborah",
                  "deborah","janis", "janis"
```

with the second occurrence of `"grace"` at position 3. Removing the element at position 3 three times successively must turns this to:

```
"suzie", "suzie", "grace", "deborah", "deborah", "janis", "janis"
```

and changing the element at position 3 to `"jackie"` must change the contents to:

```
"suzie", "suzie", "grace", "jackie", "deborah", "janis", "janis"
```

The method `queueExperiment` (Line 54) instantiates a `LinkedList<String>` object, and then assigns the object to a `Queue<String>` variable named `myQueue` (Line 58). The method then adds each element of `SEQUENCE` successively twice using the method `offer` of `Queue<String>` (Lines 59–63). Next, the method examines the head element using the method `peek` of `Queue<String>` (Lines 66 and 67) and then removes the head element of `myQueue` using the method `poll` of `Queue<String>` (Lines 68 and 69). The method repeats this action three times. Finally, the method calls `printInfo` to print the information about `myQueue` (Line 71).

```java
  public static void listExperiment()
  {
    System.out.println( "-----Linked List Demo-----" );
    List<String> myList = new LinkedList<String>();
    for ( String w : SEQUENCE )
    {
      myList.add( w );
      myList.add( w );
    }
    for ( int i = 1; i <= 3; i ++ )
    {
      String y = myList.remove( 3 );
      System.out.printf( "Removed the element at 3 %s\n", y );
      y = myList.get( 3 );
      System.out.printf( "The new element at 3 is %s\n", y );
    }
    myList.set( 3, "jackie" );
    String z = myList.get( 3 );
    System.out.println( "Changed the element at 3 to \"jackie\"" );
    System.out.printf( "The new element at 3 is %s\n", z );
    printInfo( myList );
  }

```

Listing 18.21 The class `CollectionExperiment` (part 2). The experiment with a `List<String>` data

```java
  public static void queueExperiment()
  {
    System.out.println( "-----Queue Demo-----" );
    Queue<String> myQueue = new LinkedList<String>();
    for ( String w : SEQUENCE )
    {
      myQueue.offer( w );
      myQueue.offer( w );
    }
    for ( int i = 1; i <= 3; i ++ )
    {
      String y = myQueue.peek();
      System.out.printf( "The head is %s\n", y );
      y = myQueue.poll();
      System.out.printf( "The head %s has been removed\n", y );
    }
    printInfo( myQueue );
  }

```

Listing 18.22 The class `CollectionExperiment` (part 3). The experiment with a `Queue<String>` data

After the initial insertion of the elements, the contents of `myQueue` are expected to be: `"suzie", "suzie", "grace", "grace", "carol", "carol", "deborah",`

Removing the head element three times in a row must turn this set to:

```
"grace", "carol", "carol", "deborah", "deborah", "janis", "janis"
```

The method `setExperiment` (Line 74) instantiates a `HashSet<String>` object, and then assigns the object to a `Set<String>` variable named `myQueue` (Line 77). The method then adds each element of `SEQUENCE` successively twice using the method `add` of `Set<String>` (Lines 78–82). Next, the method removes `"suzie"` using the method `remove` and `Set<String>` (Lines 83 and 84). Finally, the method calls `printInfo` to generate a report about `mySet` (Line 85).

```
74    public static void setExperiment()
75    {
76      System.out.println( "-----Set Demo-----" );
77      Set<String> mySet = new HashSet<String>();
78      for ( String w : SEQUENCE )
79      {
80        mySet.add( w );
81        mySet.add( w );
82      }
83      mySet.remove( "suzie" );
84      System.out.printf( "Removed %s\n", "suzie" );
85      printInfo( mySet );
86    }
87  }
```

Listing 18.23 The class `CollectionExperiment` (part 4). The experiment with a `Set<String>` data

After the initial insertion of the elements, the contents of the list are expected to be the set consisting of:

`"suzie", "grace", "carol", "deborah", "janis"`

Removing `"suzie"` must turn this set to:

`"grace", "carol", "deborah", "janis"`

Here is the result of executing the program.

```
-----Linked List Demo-----
Removed the element at 3 grace
The new element at 3 is carol
Removed the element at 3 carol
The new element at 3 is carol
Removed the element at 3 carol
The new element at 3 is deborah
Changed the element at 3 to "jackie"
The new element at 3 is jackie

The collection has 7 elements
The elements are: suzie suzie grace jackie deborah janis janis
The results of membership tests are:
               suzie:true
               grace:true
               carol:false
             deborah:true
               janis:true
```

```
-----Queue Demo-----
The head is suzie
The head suzie has been removed
The head is suzie
The head suzie has been removed
The head is grace
The head grace has been removed

The collection has 7 elements
The elements are: grace carol carol deborah deborah janis janis
The results of membership tests are:
               suzie:false
               grace:true
               carol:true
             deborah:true
               janis:true
               -----Set Demo-----
Removed suzie

The collection has 4 elements
The elements are: carol janis grace deborah
The results of membership tests are:
               suzie:false
               grace:true
               carol:true
             deborah:true
               janis:true
```

Note that the order in which the elements are retrieved from the set is different from the order in which the elements are added to the set.

Summary

- A generic type parameter is a parameter for an object type.
- A generic type parameter is represented with an upper case letter inside a pair of “greater than” and “less than” symbols, < >.
- `Comparable<E>` is an interface with one abstract method `compareTo`.
- `Comparator<E>` is an interface that defines an abstract method `compare` that compares two items of type `E`.
- `Iterable<E>` is an interface with one abstract method `iterator` that returns a data of type `Iterator<E>`.
- `Iterator<E>` is an interface with two abstract methods `hasNext` and `next`.
- For-each loops can be applied to an object of a class implementing `Iterable<E>`.
- `ArrayList<E>` is a class that builds a list of an indefinite length.
- The Java Collection Framework is a large body of interfaces and classes. The framework provides ways to maintain dynamic collections of data. Major units in the framework include interfaces `Iterable<E>`, `Iterator<E>`, `Collection<E>`, `List<E>`, `Queue<E>`, `Dequeue<E>`, `Set<E>`, and `SortedSet<E>` and classes `ArrayList<E>`, `LinkedList<E>`, and `HashSet<E>`.

Exercises

1. **A class for a bottom K list** The class `TopK` was designed to maintain the largest `k` elements in a sequence of an indefinite number of elements of some type `E`, where `E` is a class parameter, where `E` extends `Comparable<E>`. Write a class, `BottomK`, for maintaining the smallest `k` elements instead.
2. **A class for a pair of values** Write a class, `Pair`, that takes two class parameters, `K` and `E`, where `K` extends `Comparable<K>`. An object of the class has a pair of instance variables, one being of type `K` and the other being of type `E`. A constructor for `Pair<K,E>` receives values for the two variables and stores them in the instance variables. The class has a "getter" for each of the two variables, `getOne` and `getTwo`. The class is also expected to be comparable with respect to the first instance variable. The header for the class is:

```
public class Pair< E extends Comparable<E>, K> implements
    Comparable< Pair< E,K > >
```

3. **A Collatz Conjecture competition** Recall that Collatz Conjecture states that every positive integer can be converted to 1 with successive applications of the following transformation rule: if the number is even, divide it by 2; otherwise, multiply it by 3 and then add 1. Write a program, `CollatzCompetition`, that receives three input integers, say `num`, `max`, and `k`, from the user, generates `num` random integers between 1 and `max`, and among the random numbers generated, finds which `k` numbers require the most transformation steps before becoming 1. The minimum possible value for `k` is 3. If the user enters a number less than 3, the number must be adjusted to 3.
   Here is a possible output of the program:

```
Testing the Collatz Conjecture
Enter  #Trials, Max, K: 10000 1000000 10
rank=10  length=436   number=775035
rank=9   length=369   number=900093
rank=8   length=361   number=772009
rank=7   length=361   number=781862
rank=6   length=357   number=293199
rank=5   length=356   number=815273
rank=4   length=348   number=747070
rank=3   length=344   number=270847
rank=2   length=344   number=270222
rank=1   length=343   number=789302
```

   In keeping the top 10, use the class `Pair<Integer,Integer>` written in the previous question, where the first element of the pair is the number of steps that the first number had to undergo before becoming 1.
4. **The set of unique tokens** Write a program, `UniqueTokens`, that produces an ordered list of tokens that appear in a file, where the path to the file is given by `args[ 0 ]`. The program must use a `Set<String>` variable instantiated with `HashSet<String>` to record the tokens. After opening the file, the program scans all the tokens in the file, and adds the tokens to the set using the method `set` of the class `Set<String>`. After finishing collecting the tokens, the program generates an `ArrayList<String>` consisting of all the tokens appearing in the set. To accomplish this, the program retrieves all the tokens in the set using a for-each loop, and adds it to the list. After building the list, the program sorts it using `Collections.sort`. Finally, the program prints the tokens, say three tokens per line, to produce the output.

Here is the output generated by supplying the source code of `TopK.java` that was presented in this textbook:

```
                     (                     )                    );
                     +                     0                     1
                     =                    ==                     >
          ArrayList<E>       ArrayList<E>();     Collections.sort(
           Comparable<                     E                 TopK(
                 TopK<                  add(                 class
               extends                  get(                     i
                    if                import                   int
          java.util.*;                     k                    k;
                   new                     o               private
                public                return               theList
          theList.add(          theList.get(       theList.remove(
        theList.size()              theList;                this.k
                  void                     {                     }
```

To produce this output, the program uses `printf` with 22 character spaces allocated for each token.

5. **The set of unique tokens, keeping only the alphabet, the apostrophe, and the dash** As a variation of the previous question, write a program, `UniqueTokensAlt`, that generates a sorted list of the tokens appearing in a given file, where all the characters other than the letters of the alphabet, the apostrophe, and the dash are converted to the whitespace during the process of reading tokens, and each uppercase letter is converted to lowercase.

   To accomplish the task, the program reads the input file line by line, transforms the line to a new `String` that contains only the letters, the apostrophe, the dash, and the whitespace, and then reads the tokens from the newline using a `Scanner`. The transformation should be done as follows: convert the line to all lowercase, instantiate a `StringBuilder` object, append the characters of the lowercase version after replacing each character that is not a lowercase letter, an apostrophe, or a dash to a whitespace. For example,

   ```
   System.out.printf( "\%s\n", "abc" );
   ```

   must be converted to:

   ```
   system out printf s n abc
   ```

   The tokens obtained from this `String` are `"system"`, `"out"`, `"printf"`, `"s"`, `"n"`, and `"abc"`. Here is the output of the program with the source code of `TopK.java` as the input.

```
                   add             arraylist                 class
           collections            comparable                     e
               extends                   get                     i
                    if                import                   int
                  java                     k                   new
                     o               private                public
                remove                return                  size
                  sort               thelist                  this
                  topk                  util                  void
```

6. **A comparator for `File`** Write a class, `CompareFile`, that implements `Comparator <File>`. The comparison should be based upon the names of the `File` objects accessed with `getName`.

Programming Projects

7. **Name and count** Write a class named NameAndCount for recording a String data named name and an int data named count. A constructor for the class takes two values, one for the name and the other for the count, and stores the two values in the instance variables. The class must be comparable, with the declaration of implements Comparable<NameAndCount>. The class has a "getter" for the name, named getName, and a "getter" for the count, named count. For "setters", there is a method named increment that increases the value of count by 1. There is another instance method equals, which receives a String data as its parameter, and returns a boolean value indicating whether or not the contents of the String data is equal to the contents of name. Additionally, the class must implement the compareTo method, which returns the result of comparing the values of count.
8. **A comparator for the NameAndCount** Write a class named CompareNameAndCount that implements Comparator<NameAndCount>. The comparison should be based upon the names.
9. **Taking an inventory** Using the NameAndCount class from the previous question, write a program NameInventory that takes an inventory of the tokens appearing in a file, where the file path is given by args[0]. The program instantiates a LinkedList of NameAndString objects and retrieves all the tokens from the file. For each token it retrieves, the program scans the list for a match with the token using the equals method. If there is a match, the program increases the count by calling the method increment of the NameAndCount object that matches the token. If there is no match found, the program adds a new NameAndCount object with the token and the initial count of 1. After incorporating all the tokens, the program sorts the collection, and prints the names with their counts.

   The program can be used to count votes in a write-in election. Here is the result of processing the file containing one million random tokens that are chosen from a set of five names:

```
           Ron:19799           Dudley:19883         Hermione:20058
         Draco:20125            Harry:20135
```