Chapter 14 - Generics

Section 14.1 - Comparable Interface: Sorting an ArrayList

Sorting the elements of an ArrayList into ascending or descending order is a common programming task. Java's ***Collections*** class provides static methods that operate on various types of lists such as an ArrayList. The sort() method sorts collections into ascending order provided that the elements within the collection implement the Comparable interface (i.e., the elements are also of the type Comparable). For example, each of the primitive wrapper classes (e.g., Integer, Double, etc.) implements the ***Comparable*** interface, which declares the compareTo() method. Classes implementing the Comparable interface must define a custom implementation of the compareTo() method. A programmer may use sort() to sort an ArrayList in which the elements implement the Comparable interface (e.g., Integer). The programmer must import java.util.Collections to use the sort() method. The following example demonstrates the use of sort() to sort an ArrayList of Integer objects.

Figure 14.1.1: Collections' sort() method operates on lists of Integer objects.

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Collections;

public class ArraySorter {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      final int NUM_ELEMENTS = 5;                              // Number of items in array
      ArrayList<Integer> userInts = new ArrayList<Integer>(); // Array of user defined values
      int i = 0;                                               // Loop index

      // Prompt user for input, add values to array
      System.out.println("Enter " + NUM_ELEMENTS + " numbers...");
      for (i = 1; i <= NUM_ELEMENTS; ++i) {
         System.out.print(i + ": ");
         userInts.add(new Integer(scnr.nextInt()));
      }

      // Sort ArrayList of Comparable elements
      Collections.sort(userInts);

      // Print sorted array
      System.out.print("\nSorted numbers: ");
      for (i = 0; i < NUM_ELEMENTS; ++i) {
         System.out.print(userInts.get(i) + " ");
      }
      System.out.println("");

      return;
   }
}
```

```
Enter 5 numbers...
1: -10
2: 99
3: 31
4: 5
5: 31

Sorted numbers: -10 5 31 31 99
```

The Collections' sort() method calls the compareTo() method on each object within the ArrayList to determine the order and produce a sorted list.

The sort() method can also be used to sort an ArrayList containing elements of a user-defined class type. The only requirement, however, is that the user-defined class must also implement the Comparable interface and override the compareTo() method, which should return a number that determines the ordering of the two objects being compared as shown below.

compareTo(otherComparable) compares a Comparable object to otherComparable, returning a number indicating if the Comparable object is less than, equal to, or greater than otherComparable. The method compareTo() will return 0 if the two Comparable objects are equal. Otherwise, compareTo() returns a negative number if the Comparable object is less than otherComparable, or a positive number if the Comparable object is greater than otherComparable.

The following program allows a user to add new employees to an ArrayList and print employee information in sorted order. The Employee class implements `Comparable<EmployeeData>` and overrides the compareTo()

method in order to enable the use of the Collections class's sort() method.

Figure 14.1.2: Sorting an ArrayList of employee records.

EmployeeData.java:

```
public class EmployeeData implements Comparable<EmployeeData> {
   private String firstName; // First Name
   private String lastName;  // Last Name
   private Integer emplID;   // Employee ID
   private Integer deptNum;  // Department Number

   EmployeeData(String firstName, String lastName, Integer emplID, Integer deptNum) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.emplID = emplID;
      this.deptNum = deptNum;
   }

   @Override
   public int compareTo(EmployeeData otherEmpl) {
      String fullName = "";      // Full name, this employee
      String otherFullName = ""; // Full name, comparison employee
      int comparisonVal = 0;     // Outcome of comparison

      // Compare based on department number first
      comparisonVal = deptNum.compareTo(otherEmpl.deptNum);

      // If in same organization, use name
      if (comparisonVal == 0) {
         fullName = lastName + firstName;
         otherFullName = otherEmpl.lastName + otherEmpl.firstName;
         comparisonVal = fullName.compareTo(otherFullName);
      }

      return comparisonVal;
   }

   @Override
   public String toString() {
      return lastName + " " + firstName +
             "       \tID: " + emplID +
             "\t\tDept. #: " + deptNum;
   }
}
```

EmployeeRecords.java:

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Collections;

public class EmployeeRecords {

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<EmployeeData> emplList = new ArrayList<EmployeeData>(); // Stores all employee
      EmployeeData emplData;                                            // Stores info for one
      String userCommand = "";                                          // User defined add/pr
      String emplFirstName = "";                                        // User defined employ
      String emplLastName = "";                                         // User defined employ
      Integer emplID = 0;                                               // User defined employ
      Integer deptNum = 0;                                              // User defined employ
      int i = 0;                                                        // Loop counter
```

```
      do {
         // Prompt user for input
         System.out.println("Enter command ('a' to add new employee, 'p' to print all employee
         userCommand = scnr.next();

         // Add new employee entry
         if (userCommand.equals("a")) {
            System.out.print("First Name: ");
            emplFirstName = scnr.next();
            System.out.print("Last Name: ");
            emplLastName = scnr.next();
            System.out.print("ID: ");
            emplID = scnr.nextInt();
            System.out.print("Department Number: ");
            deptNum = scnr.nextInt();
            emplData = new EmployeeData(emplFirstName, emplLastName, emplID, deptNum);
            emplList.add(emplData);
         }
         // Print all entries
         else if (userCommand.equals("p")) {

            // Sort employees by department number first
            // and name second
            Collections.sort(emplList);

            System.out.println("");
            System.out.println("Employees: ");
            // Access employee records
            for (i = 0; i < emplList.size(); ++i) {
               System.out.println(emplList.get(i).toString());
            }
            System.out.println("");
         }
      } while (!userCommand.equals("q"));

      return;
   }
}
```

```
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
a
First Name: Michael
Last Name: Faraday
ID: 124
Department Number: 1
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
a
First Name: Ada
Last Name: Lovelace
ID: 203
Department Number: 2
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
a
First Name: James
Last Name: Maxwell
ID: 123
Department Number: 1
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
a
First Name: Alan
Last Name: Turing
ID: 201
Department Number: 2
Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
p
```

```
Employees:
Faraday Michael          ID: 124          Dept. #: 1
Maxwell James            ID: 123          Dept. #: 1
Lovelace Ada             ID: 203          Dept. #: 2
Turing Alan              ID: 201          Dept. #: 2

Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit):
q
```

Interface implementation is a concept similar to class inheritance. The ***implements*** keyword tells the compiler that a class implements, instead of extends, a particular interface (e.g., Comparable<EmployeeData>). Like with inheritance, an Employee object is of type Comparable<EmployeeData> as well as EmployeeData. However, an interface differs from a typical super class in that interfaces cannot be instantiated and the methods declared by an interface must be overridden and defined by the implementing class. In this example, the built-in Comparable interface declares the compareTo() method, which EmployeeData must override. Failing to override compareTo() results in the following compiler error: "EmployeeData is not abstract and does not override abstract method compareTo(EmployeeData) in java.lang.Comparable".

The ArrayList of EmployeeData elements is sorted via the sort() method, as in `Collections.sort(emplList);`. The sort() method invokes each element's compareTo() method in order to determine the ordering and sort the ArrayList. EmployeeData's compareTo() method performs comparison between two EmployeeData objects, prioritizing department number over an employee's name. Thus, an employee hired within a numerically smaller department number will precede another employee with a numerically larger department number, and vice versa. If two employees are located in the same department, they are compared lexicographically based on their names. The end result is that employees are sorted according to department number, and employees in the same department are sorted in alphabetical order according to their names.

Participation Activity

14.1.1: Sort Employee elements using employee IDs.

Modify EmployeeData's compareTo() method so that elements are sorted based on the employees' department number (deptNum) and ID (emplID). Specifically, employee's should first be sorted in ascending order according to department number first, and those employees within the same department should be sorted in ascending order according to the employee ID.

EmployeeData.java | EmployeeRecords.java

Reset

```
1
2  public class EmployeeData implements Comparable<EmployeeData> {
3     private String firstName; // First Name
4     private String lastName;  // Last Name
5     private Integer emplID;   // Employee ID
6     private Integer deptNum;  // Department Number
7
8     EmployeeData(String firstName, String lastName, Integer emplID, Integer deptNum) {
9        this.firstName = firstName;
10       this.lastName = lastName;
11       this.emplID = emplID;
12       this.deptNum = deptNum;
13    }
14
15    @Override
16    public int compareTo(EmployeeData otherEmpl) {
17       String fullName = "";      // Full name, this employee
18       String otherFullName = ""; // Full name, comparison employee
19       int comparisonVal = 0;     // Outcome of comparison
```

```
a Michael Faraday 124 1
a Ada Lovelace 203 2
a James Maxwell 123 1
a [illegible]ring 201 2
```

Run

Classes that already inherit from a base class can also be defined to implement an interface. For example, the above EmployeeData class could have been defined so that it extends a Person class and implements the Comparable interface, as in
`public class EmployeeData extends Person implements Comparable<EmployeeData> { ... }`.

Finally, note that Comparable's compareTo() method is meant to work with any class. Thus, a programmer must append the class name in angle brackets to "Comparable", as in `Comparable<EmployeeData>`, in order to tell the compiler that the compareTo() method requires an argument of the indicated class type. Generic methods, classes, and interfaces are discussed in more detail elsewhere.

Participation Activity

14.1.2: Sorting elements in an ArrayList.

#	Question	Your answer
1	The following statement sorts an ArrayList called prevEmployees. Assume prevEmployees is an appropriately initialized ArrayList of EmployeeData elements. `sort(prevEmployees);`	True False
2	An interface contains method declarations, as opposed to method definitions.	True False
3	An interface cannot be instantiated.	True False
4	The EmployeeData class, as defined above, is not required to override the compareTo() method declared by the Comparable interface.	True False
5	A class may not simultaneously "extend" a class and "implement" an interface.	True False

Exploring further:

- Introduction to interfaces from Oracle's Java tutorials
- Introduction to object ordering from Oracle's Java tutorials
- Oracle's Java Comparable class specification

Section 14.2 - Generic methods

Multiple methods may be nearly identical, differing only in their data types, as below.

Figure 14.2.1: Methods may have identical behavior, differing only in data types.

```
// Find the minimum of three **ints**
public static Integer tripleMinInt(Integer item1, Integer item2, Integer item3) {
   Integer minVal = item1;

   if (item2.compareTo(minVal) < 0) {
      minVal = item2;
   }
   if (item3.compareTo(minVal) < 0) {
      minVal = item3;
   }
   return minVal;
}


// Find the minimum of three **chars**
public static Character tripleMinChar(Character item1, Character item2, Character item3) {
   Character minVal = item1;

   minVal = item1;
   if (item2.compareTo(minVal) < 0) {
      minVal = item2;
   }
   if (item3.compareTo(minVal) < 0) {
      minVal = item3;
   }
   return minVal;
}
```

Writing and maintaining redundant methods that only differ by data type can be time-consuming and error-prone. The language supports a better approach.

A ***generic method*** is a method definition having a special type parameter that may be used in place of types in the method.

Figure 14.2.2: A generic method enables a method to handle various class types.

```
public class ItemMinimum {
   public static <TheType extends Comparable<TheType>>
   TheType tripleMin(TheType item1, TheType item2, TheType item3) {
      TheType minVal = item1; // Holds min item value, init to first item

      if (item2.compareTo(minVal) < 0) {
         minVal = item2;
      }
      if (item3.compareTo(minVal) < 0) {
         minVal = item3;
      }
      return minVal;
   }

   public static void main(String[] args) {
      Integer num1 = 55;     // Test case 1, item1
      Integer num2 = 99;     // Test case 1, item2
      Integer num3 = 66;     // Test case 1, item3

      Character let1 = 'a'; // Test case 2, item1
      Character let2 = 'z'; // Test case 2, item2
      Character let3 = 'm'; // Test case 2, item3

      String str1 = "zzz";  // Test case 3, item1
      String str2 = "aaa";  // Test case 3, item2
      String str3 = "mmm";  // Test case 3, item3

      // Try tripleMin method with Integers
      System.out.println("Items: " + num1 + " " + num2 + " " + num3);
      System.out.println("Min: " + tripleMin(num1, num2, num3) + "\n");

      // Try tripleMin method with Characters
      System.out.println("Items: " + let1 + " " + let2 + " " + let3);
      System.out.println("Min: " + tripleMin(let1, let2, let3) + "\n");

      // Try tripleMin method with Strings
      System.out.println("Items: " + str1 + " " + str2 + " " + str3);
      System.out.println("Min: " + tripleMin(str1, str2, str3) + "\n");

      return;
   }
}
```

```
run:
Items: 55 99 66
Min: 55

Items: a z m
Min: a

Items: zzz aaa mmm
Min: aaa
```

The method return type is preceded by `<TheType extends Comparable<TheType>>` (highlighted yellow), where TheType can be any identifier. That type is known as a ***type parameter*** and can be used throughout the method for any parameter types, return types, or local variable types (highlighted orange).

A type parameter may be associated with a ***type bound*** to specify the class types for which a type parameter is valid. Type bounds are specified using the extends keyword and appear after the corresponding type parameter.

For example, the code `<TheType extends Comparable<TheType>>` specifies that TheType is bounded by the type bound `Comparable<TheType>`. Thus, TheType may only represent types that implement the Comparable interface. If the type bound is a class type (e.g., the Number class), the type parameter may only represent types that are of the type specified by the type bound or any derived classes.

Type bounds are also necessary to enable access to the class members of the class specified by the type bound (e.g., compareTo()) via a variable of a generic type (e.g., item1, item2, item3, and min). By bounding TheType to the Comparable interface, the programmer is able to invoke the Comparable interface's compareTo() method with the generic types, as in `item2.compareTo(min);`. Attempting to invoke a class member via a generic type without specifying the appropriate type bound results in a compiler error.

The compiler automatically generates a unique method definition for each type appearing in generic method calls. Thus, the above example's calls would create three tripleMin() method method definitions using Integer and Character as in this section's introductory example. The programmer never sees those method definitions.

Importantly, type arguments cannot be primitive types such as int, char, and double. Instead, the type arguments must be reference types. If primitive types are desired, a programmer should use the corresponding primitive wrapper classes (e.g., Integer, Character, Double, etc.), discussed elsewhere.

Participation Activity

14.2.1: Generic methods.

#	Question	Your answer

1	Fill in the blank.
	`public static <MyType extends Comparable<MyType>>`
	`_____ GetMax3 (MyType i, MyType j, MyType k) {`
	`  ...`
	`};`
		TheType
		Integer
		MyType

2	Fill in the blank.
	`public static <_____ extends Comparable<_____>>`
	`T TripleMedian(T item1, T item2, T item3) {`
	`  ...`
	`}`
		Integer
		TheType
		T
		Not possible; T is not a valid type.

	For the earlier TripleMin generic method, what happens if a call is TripleMin(i, j, k) but those arguments are of type Character?	The compiler generates an error message because only Integer and Double are supported.

3		During runtime, the Character values are forced to be Integer values.
		The compiler creates a method with Character types and calls that method.
4	For the earlier TripleMin generic method, what happens if a call is TripleMin(i, j, k) but those arguments are String objects?	The method will compare the Strings.
		The compiler generates an error, because only numerical types can be passed.
5	For the earlier TripleMin generic method, what happens if a call is TripleMin(i, j, z), where i and j are Integers, but z is a String?	The method will compare the Integer and String objects.
		The compiler will generate an error, because TheType must be the same for all three arguments.

Programmers optionally may explicitly specify the generic type as a special argument, as in `ItemMinimum.<Integer>tripleMin(num1, num2, num3);`.

A generic method may have multiple parameters:

Construct 14.2.1: Method definition with multiple generics.

```
modifiers <Type1 extends BoundType1, Type2 extends BoundType2>
ReturnType methodName(parameters) {
   ...
}
```

Note that the modifiers represent a space delimited list of valid modifiers like `public` and `static`.

Participation Activity

14.2.2: Generic methods.

This program currently fails to compile because the parameters cannot be automatically converted to Double in the statement `tripleSum = item1 + item2 + item3;`. Because TheType is bound to the class Number, the Number class' doubleValue() method can be called to get the value of the parameters as a double value. Modify tripleAvg() method to use the doubleValue() method to convert each of the parameters to a double value before adding them.

```
1
2  public class ItemMinimum {
3
4     public static <TheType extends Number>
5     Double tripleAvg(TheType item1, TheType item2, TheTyp
6        Double tripleSum = 0.0;
7
8        tripleSum = item1 + item2 + item3;
9
10       return tripleSum / 3.0;
11    }
12
13    public static void main(String[] args) {
14       Integer intVal1 = 55;
15       Integer intVal2 = 99;
16       Integer intVal3 = 66;
17
18       Double doubleVal1 = 14.5;
19       Double doubleVal2 = 12.3;
```

Run

Exploring further:

- Introduction to generics from Oracle's Java tutorials
- Introduction to bounded type parameters from Oracle's Java tutorials

Section 14.3 - Class generics

Multiple classes may be nearly identical, differing only in their data types. The following shows a class managing three Integer numbers, and a nearly identical class managing three Short numbers.

Figure 14.3.1: Classes may be nearly identical, differing only in data type.

```
public class TripleInt {
   private Integer item1; // Data value 1
   private Integer item2; // Data value 2
   private Integer item3; // Data value 3

   public TripleInt(Integer i1, Integer i2, Integer i3) {
      item1 = i1;
      item2 = i2;
      item3 = i3;
   }

   // Print all data member values
   public void printAll() {
      System.out.println("(" + item1 + "," + item2 + "," + item3 + ")");
      return;
   }

   // Return min data member value
   public Integer minItem() {
      Integer minVal = item1; // Holds min item value, init to first item

      if (item2.compareTo(minVal) < 0) {
         minVal = item2;
      }
      if (item3.compareTo(minVal) < 0) {
         minVal = item3;
      }
      return minVal;
   }
}

public class TripleShort {
   private Short item1; // Data value 1
   private Short item2; // Data value 2
   private Short item3; // Data value 3

   public TripleShort(Short i1, Short i2, Short i3) {
      item1 = i1;
      item2 = i2;
      item3 = i3;
   }

   // Print all data member values
   public void printAll() {
      System.out.println("(" + item1 + "," + item2 + "," + item3 + ")");
      return;
   }

   // Return min data member value
   public Short minItem() {
      Short minVal = item1; // Holds min item value, init to first item

      if (item2.compareTo(minVal) < 0) {
         minVal = item2;
      }
      if (item3.compareTo(minVal) < 0) {
         minVal = item3;
      }
      return minVal;
   }
}
```

Writing and maintaining redundant classes that only differ by data type can be time-consuming and error-prone. The language supports a better approach.

A ***generic class*** is a class definition having a special type parameter that may be used in place of types in the class. A variable defined of that ***generic*** class type must indicate a specific type.

Figure 14.3.2: A generic class enables one class to handle various data types.

TripleItem.java:

```
public class TripleItem <TheType extends Comparable<TheType>> {
   private TheType item1; // Data value 1
   private TheType item2; // Data value 2
   private TheType item3; // Data value 3

   public TripleItem(TheType i1, TheType i2, TheType i3) {
      item1 = i1;
      item2 = i2;
      item3 = i3;
   }

   // Print all data member values
   public void printAll() {
      System.out.println("(" + item1 + "," + item2 + "," + item3 + ")");
      return;
   }

   // Return min data member value
   public TheType minItem() {
      TheType minVal = item1; // Holds min item value, init to first item

      minVal = item1;
      if (item2.compareTo(minVal) < 0) {
         minVal = item2;
      }
      if (item3.compareTo(minVal) < 0) {
         minVal = item3;
      }
      return minVal;
   }
}
```

TripleItemManager.java:

```
public class TripleItemManager {
   public static void main(String[] args) {

      // TripleItem class with Integers
      TripleItem<Integer> triInts = new TripleItem<Integer>(9999, 5555, 6666);

      // TripleItem class with Shorts
      TripleItem<Short> triShorts = new TripleItem<Short>((short)99, (short)55, (short)66);

      // Try functions from TripleItem
      triInts.printAll();
      System.out.println("Min: " + triInts.minItem() + "\n");

      triShorts.printAll();
      System.out.println("Min: " + triShorts.minItem());

      return;
   }
}
```

```
(9999,5555,6666)
Min: 5555

(99,55,66)
Min: 55
```

The class name is succeeded by `<TheType ... >` (highlighted yellow), where TheType can be any identifier. That type is known as a ***type parameter*** and can be used throughout the class, such as for parameter types, method return types, or field types. An object of this class can be instantiated by appending after the class name a specific type in angle brackets (highlighted orange), such as `TripleItem<Short> triShorts = new TripleItem<Short>((short)99, (short)55, (short)66);`.

Each type parameter can be associated with type bounds to define the data types a programmer is allowed to use for the type arguments. As with generic methods, type bounds (discussed elsewhere) also allow a programmer to utilize the class members defined by the bounding type with variables of a generic type (e.g., item1, item2, item3, and min). Thus, above, Triple is a generic class whose instances expect type arguments that implement the Comparable<TheType> interface. By bounding the generic class's type parameter to the Comparable interface, a programmer can invoke the Comparable interface's compareTo() method with the generic types, as in `item2.compareTo(min)`.

Participation Activity

14.3.1: Generic classes.

#	Question	Your answer
1	A class has been defined using the type GenType throughout, where GenType is intended to be chosen by the programmer when defining and initializing a variable of this class. The code that should immediately follow the class's name in the class definition is `<GenType>`	True
		False
2	A key advantage of generic classes is relieving the programmer from having to write redundant code that differs only by type.	True
		False
3	For a generic class with type parameters defined as `public class Vehicle <T> { ... }`, an appropriate instantiation of that class would be `Vehicle<T> v1 = new Vehicle<T>();`.	True
		False

A generic class may have multiple type parameters, separated by commas. Additionally, each type parameter may have type bounds.

Construct 14.3.1: Generic class template with multiple parameters.

```
public class ClassName <Type1 extends BoundType1, Type2 extends BoundType2> {
   ...
}
```

Importantly, type arguments cannot be primitive types such as int, char, and double. Instead, the type arguments must be reference types. If primitive types are desired, a programmer should use the corresponding primitive wrapper classes (e.g., Integer, Char, Double, etc.), discussed elsewhere.

Note that Java's ArrayList class is a generic class, which is why a variable defined as an ArrayList indicates the type in angle brackets, as in `ArrayList<Integer> nums = new ArrayList<Integer>();`.

Participation Activity

14.3.2: Class generics.

The following program using a generic class ItemCount to count the number of times the same word is read from the user input. Modify the program to:

- Complete the incrementIfDuplicate() method and update the main() method within the DuplicateCounter class to use the incrementIfDuplicate() method.
- Modify the program to count the number of times to a specific integer value is read from the user input. Be sure to use the Integer class.

DuplicateCounter.java | ItemCount.java

Reset

```
1
2  import java.util.Scanner;
3
4  public class DuplicateCounter {
5     public static void main(String[] args) {
6        Scanner scnr = new Scanner(System.in);
7        ItemCount<String> wordCounter = new ItemCount<String>();
8        String inputWord = "";
9
10       wordCounter.setItem("that");
11
12       System.out.println("Enter words (END at end):");
13
14       // Read first word
15       inputWord = scnr.next();
16
17       // Keep reading until word read equals <end>
18       while( !inputWord.equals("END") ) {
19          if (wordCounter.getItem().compareTo(inputWord) == 0) {
```

```
that that is is not that that is not
END
```

Run

Exploring further:

- Introduction to generics from Oracle's Java tutorials

Section 14.4 - Java example: Map values using a generic method

Participation Activity | 14.4.1: Map a value using a generic method.

The program below uses a generic method to map numeric, string, or character values to a shorter list of values. The program demonstrates a mapping for integers using a table of:

100
200
300
400
500
600

The program gets an integer value from a user and returns the first value in the table that is greater than or equal to the user value, or the user value itself if that value is greater than the largest value in the table. Ex:

165 returns 200
444 returns 500
888 returns 888

1. Run the program and notice the input value 137 is mapped to 200. Try changing the input value and running again.
2. Modify the program to call the getMapping method for a double and a string, similar to the integer. .
3. Run the program again and enter an integer, a double, and a string

Reset

```
1 import java.util.Scanner;
2
3 public class GenericMappingArrays {
4    public static <MapType extends Comparable<MapType>>
5                  MapType getMapping(MapType mapMe, MapType [] mappings) {
6       MapType result = mapMe;
7       int i = 0;
```

```
7        int i = 0;
8        int len = mappings.length;
9        boolean keepLooking = true;
10
11       System.out.println();
12       System.out.print("Mapping range: ");
13       for (i = 0; i < len; ++i) {
14          System.out.print(mappings[i] + "  ");
15       }
16       System.out.println();
17
18       i = 0;      // Restart counting
19       while ((i < len) && keepLooking) {
```

```
137
```

Run

Participation Activity

14.4.2: Map a value using a generic method (solution).

A solution to the above problem follows.

Reset

```
1  import java.util.Scanner;
2
3  public class GenericMappingArraysSolution {
4     public static <MapType extends Comparable<MapType>>
5                    MapType getMapping(MapType mapMe, MapType[] mappings) {
6        MapType result = mapMe;
7        int i = 0;
8        int len = mappings.length;
9        boolean keepLooking = true;
10
11       System.out.println();
12       System.out.print("Mapping range: ");
13       for (i = 0; i < len; ++i) {
14          System.out.print(mappings[i] + "  ");
15       }
16       System.out.println();
17
18       i = 0;      // Restart counting
19       while ((i < len) && keepLooking) {
```

```
137
4.44444
Hi
```

Run