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.
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.

```
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.println("Sorted numbers: ");
        for (i = 0; i < NUM_ELEMENTS; ++i) {
            System.out.print(userInts.get(i) + " ");
        }
        System.out.println("\n");
    }
}
```

The Collections’ sort() method operates on lists of Integer objects.

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:

```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:

```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;
        String userCommand = "";
        String emplFirstName = "";
        String emplLastName = "";
        Integer emplID = 0;
        Integer deptNum = 0;
        int i = 0; // Loop counter
```
do {
    // Prompt user for input
    System.out.println("Enter command ('a' to add new employee, 'p' to print all employees, 'q' to quit): ");
    userCommand = scnr.next();

    // Add new employee entry
    if (userCommand.equals("a")) {
        System.out.println("First Name: ");
        emplFirstName = scnr.next();
        System.out.println("Last Name: ");
        emplLastName = scnr.next();
        System.out.println("ID: ");
        emplID = scnr.nextInt();
        System.out.println("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("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;
}
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.

<table>
<thead>
<tr>
<th>Employees:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faraday Michael</td>
</tr>
<tr>
<td>Maxwell James</td>
</tr>
<tr>
<td>Lovelace Ada</td>
</tr>
<tr>
<td>Turing Alan</td>
</tr>
</tbody>
</table>

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

```java
EmployeeData.java
EmployeeRecords.java
```

```
 Reset

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
```
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

```java
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.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The following statement sorts an ArrayList called <code>prevEmployees</code>. Assume <code>prevEmployees</code> is an appropriately initialized <code>ArrayList</code> of <code>EmployeeData</code> elements. <code>sort(prevEmployees);</code></td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>An interface contains method declarations, as opposed to method definitions.</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>An interface cannot be instantiated.</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>The <code>EmployeeData</code> class, as defined above, is not required to override the <code>compareTo()</code> method declared by the <code>Comparable</code> interface.</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>A class may not simultaneously &quot;extend&quot; a class and &quot;implement&quot; an interface.</td>
<td>True</td>
</tr>
</tbody>
</table>
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.]

```java
// 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.

---

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**
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 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

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fill in the blank. &lt;MyType extends Comparable&lt;MyType&gt;&gt;</td>
<td>TheType</td>
</tr>
<tr>
<td></td>
<td>public static &lt;MyType extends Comparable&lt;MyType&gt;&gt; GetMax3 (MyType i, MyType j, MyType k) { }</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Fill in the blank. &lt;_____ extends Comparable&lt;_____&gt;&gt;</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>public static &lt;_____ extends Comparable&lt;_____&gt;&gt; T TripleMedian(T item1, T item2, T item3) { }</td>
<td>TheType</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>For the earlier TripleMin generic method, what happens if a call is TripleMin(i, j, k) but those arguments are of type Character?</td>
<td>The compiler generates an error message because only Integer and Double are supported.</td>
</tr>
</tbody>
</table>

---

The new code snippet above demonstrates how bounded type parameters can be used in generic methods.

The `GetMax3` method takes three arguments of type `MyType` and returns the maximum value among them. The `TripleMedian` method calculates the median of three values of type `T`.

The `compareTo()` method is automatically available due to the type bound, allowing for comparisons between `Integer` and `Character` objects.

When using primitive types, the corresponding wrapper classes (`Integer`, `Character`, etc.) should be used to ensure type safety and proper method invocation.

---

For the earlier TripleMin generic method, attempting to use `Character` as the argument type would result in a compiler error because the method definition expects `Integer` or `Double` parameters, not `Character`. Therefore, the compiler generates an error message indicating that the method only supports `Integer` and `Double` arguments.

---

The code snippet illustrates how type bounds enable method overriding and access to class members, ensuring that only compatible types are used, thereby preventing runtime errors.

---

To summarize, type bounds in generic methods allow for precise type checking and method invocation, ensuring that only compatible types are used. This feature is crucial for maintaining type safety and preventing errors at runtime.
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`. 

During runtime, the Character values are forced to be Integer values.

The compiler creates a method with Character types and calls that method.

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.

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.
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.

```java
public class ItemMinimum {
    public static <TheType extends Number> Double tripleAvg(TheType item1, TheType item2, TheType item3) {
        Double tripleSum = 0.0;
        tripleSum = item1 + item2 + item3;
        return tripleSum / 3.0;
    }
    public static void main(String[] args) {
        Integer intVal1 = 55;
        Integer intVal2 = 99;
        Integer intVal3 = 66;
        Double doubleVal1 = 14.5;
        Double doubleVal2 = 12.3;
    }
}
```

Exploring further:
- Introduction to generics from Oracle’s Java tutorials
- Introduction to bounded type parameters from Oracle’s Java tutorials
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**:

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

**TripleItemManager.java**:

```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() + 

        triShorts.printAll();
        System.out.println("Min: " + triShorts.minItem());
        return;
    }
```
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)
```
A generic class may have multiple type parameters, separated by commas. Additionally, each type parameter may have type bounds.

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.

```java
import java.util.Scanner;

public class DuplicateCounter {
    public static void main(String[] args) {
        Scanner scnr = new Scanner(System.in);
        ItemCount<String> wordCounter = new ItemCount<String>();
        String inputWord = "";
        wordCounter.setItem("that");
        System.out.println("Enter words (END at end):");
        // Read first word
        inputWord = scnr.next();
        // Keep reading until word read equals <end>
        while( !inputWord.equals("END") ) {
            if (wordCounter.getItem().compareTo(inputWord) == 0) {
                // Run
            }
        }
    }
}
```

DuplicateCounter.java  ItemCount.java

Reset

END
Exploring further:
- Introduction to generics from Oracle’s Java tutorials

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

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 public class GenericMappingArrays {
3   public static <MapType extends Comparable<MapType>>
4     MapType getMapping(MapType mapMe, MapType [] mappings) {
5       MapType result = mapMe;
6       int i = 0;
7   
8   }
```java
int i = 0;
int len = mappings.length;
boolean keepLooking = true;

System.out.println();
System.out.println("Mapping range: ");
for (i = 0; i < len; ++i) {
    System.out.print(mappings[i] + " ");
}
System.out.println();
i = 0; // Restart counting
while ((i < len) && keepLooking) {
```

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

A solution to the above problem follows.

```java
import java.util.Scanner;

public class GenericMappingArraysSolution {
    public static <MapType extends Comparable<MapType>> MapType getMapping(MapType mapMe, MapType[] mappings) {
        MapType result = mapMe;
        int i = 0;
        int len = mappings.length;
        boolean keepLooking = true;

        System.out.println();
        System.out.println("Mapping range: ");
        for (i = 0; i < len; ++i) {
            System.out.print(mappings[i] + " ");
        }
        System.out.println();

        i = 0;  // Restart counting
        while ((i < len) && keepLooking) {
```

4,44444

Hi