9.6.3 The Comparable Interface

Implement the Comparable interface so that objects of your class can be compared, for example, in a sort method.

In the preceding sections, we defined the Measurable interface and provided an average method that works with any classes implementing that interface. In this section, you will learn about the Comparable interface of the standard Java library.

The Measurable interface is used for measuring a single object. The Comparable interface is more complex because comparisons involve two objects. The interface declares a compareTo method. The call

```
a.compareTo(b)
```

must return a negative number if a should come before b, zero if a and b are the same, and a positive number otherwise.

The Comparable interface has a single method:

```
public interface Comparable
{
   int compareTo(Object otherObject);
}
```

For example, the BankAccount class can implement Comparable like this:

```
public class BankAccount implements Comparable
{
   . . .
   public int compareTo(Object otherObject)
   {
      BankAccount other = (BankAccount) otherObject;
      if (balance < other.balance) { return -1; }
      if (balance > other.balance) { return 1; }
      return 0;
   }
   . . .
}
```

This compareTo method compares bank accounts by their balance. Note that the compareTo method has a parameter variable of type Object. To turn it into a BankAccount reference, we use a cast:

```
BankAccount other = (BankAccount) otherObject;
```

Once the BankAccount class implements the Comparable interface, you can sort an array of bank accounts with the Arrays.sort method:

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(0);
accounts[2] = new BankAccount(2000);
Arrays.sort(accounts);
```

The accounts array is now sorted by increasing balance.

The compareTo method checks whether another object is larger or smaller.

26. Suppose you want to use the `average` method to find the average salary of `Employee` objects. What condition must the `Employee` class fulfill?

27. Why can't the `average` method have a parameter variable of type `Object[]`?

28. Why can't you use the `average` method to find the average length of `String` objects?

29. What is wrong with this code?

```
Measurable meas = new Measurable();
System.out.println(meas.getMeasure());
```

30. How can you sort an array of `Country` objects by increasing area?

31. Can you use the `Arrays.sort` method to sort an array of `String` objects? Check the API documentation for the `String` class.

Practice It Now you can try these exercises at the end of the chapter: R9.14, E9.18, E9.19.

Common Error 9.6

Forgetting to Declare Implementing Methods as Public

The methods in an interface are not declared as `public`, because they are public by default. However, the methods in a class are *not* public by default. It is a common error to forget the `public` reserved word when declaring a method from an interface:

```
public class BankAccount implements Measurable
{
   double getMeasure() // Oops—should be public
   {
      return balance;
   }
   . . .
}
```

Then the compiler complains that the method has a weaker access level, namely package access instead of public access (see Special Topic 8.4). The remedy is to declare the method as `public`.

Programming Tip 9.2

Comparing Integers and Floating-Point Numbers

When you implement a comparison method, you need to return a negative integer to indicate that the first object should come before the other, zero if they are equal, or a positive integer otherwise. You have seen how to implement this decision with three branches. When you compare *nonnegative* integers, there is a simpler way: subtract the integers:

```
public class Person implements Comparable
{
   private int id; // Must be ≥ 0
   . . .
   public int compareTo(Object otherObject)
   {
      Person other = (Person) otherObject;
      return id - other.id;
   }
}
```

The difference is negative if `id < other.id`, zero if the values are the same, and positive otherwise.

This trick doesn't work if the integers can be negative because the difference can overflow (see Exercise R9.15). However, the `Integer.compare` method always works:

```
return Integer.compare(id, other.id); // Safe for negative integers
```

You cannot compare floating-point values by subtraction (see Exercise R9.16). Instead, use the `Double.compare` method:

```
public class BankAccount implements Comparable
{
   . . .
   public int compareTo(Object otherObject)
   {
      BankAccount other = (BankAccount) otherObject;
      return Double.compare(balance, other.balance);
   }
}
```

Special Topic 9.8

Constants in Interfaces

Interfaces cannot have instance variables, but it is legal to specify **constants**.

When declaring a constant in an interface, you can (and should) omit the reserved words `public static final`, because all variables in an interface are automatically `public static final`. For example,

```
public interface Measurable
{
   double OUNCES_PER_LITER = 33.814;
   . . .
}
```

To use this constant in your programs, add the interface name:

```
Measurable.OUNCES_PER_LITER
```

Special Topic 9.9

Generic Interface Types

In Section 9.6.3, you saw how to use the "raw" version of the `Comparable` interface type. In fact, the `Comparable` interface is a parameterized type, similar to the `ArrayList` type:

```
public interface Comparable<T>
{
   int compareTo(T other)
}
```

The type parameter specifies the type of the objects that this class is willing to accept for comparison. Usually, this type is the same as the class type itself. For example, the `BankAccount` class would implement `Comparable<BankAccount>`, like this:

```
public class BankAccount implements Comparable<BankAccount>
{
   . . .
   public int compareTo(BankAccount other)
   {
      return Double.compare(balance, other.balance);
   }
}
```

The type parameter has a significant advantage: You need not use a cast to convert an `Object` parameter variable into the desired type.

Similarly, the `Measurer` interface can be improved by making it into a generic type:

```
public interface Measurer<T>
{
   double measure(T anObject);
}
```

The type parameter specifies the type of the parameter of the `measure` method. Again, you avoid the cast from `Object` when implementing the interface:

```
public class AreaMeasurer implements Measurer<Rectangle>
{
   public double measure(Rectangle anObject)
   {
      double area = anObject.getWidth() * anObject.getHeight();
      return area;
   }
}
```

(See Chapter 18 for an in-depth discussion of implementing and using generic classes.)

Java 8 Note 9.1

Static Methods in Interfaces

Before Java 8, all methods in an interface had to be abstract. Java 8 allows static methods in interfaces that work exactly like static methods in classes. A static method of an interface does not operate on objects, and its purpose should relate to the interface that contains it.

For example, it would be perfectly sensible to place the `average` method from Section 9.6.1 into the `Measurable` interface:

FULL CODE EXAMPLE
Go to `wiley.com/go/bjlo2code` to download an example of a static method in an interface.

```
public interface Measurable
{
   double getMeasure();   // An abstract method
   static double average(Measurable[] objects) // A static method
   {
      . . . // Same implementation as in Section 9.6.1
   }
}
```

To call this method, provide the name of of the interface and the method name:

```
double meanArea = Measurable.average(countries);
```

Java 8 Note 9.2

Default Methods

A **default method** is a non-static method in an interface that has an implementation. A class that implements the method either inherits the default behavior or overrides it. By providing default methods in an interface, it is less work to define a class that implements an interface.

For example, the `Measurable` interface can declare `getMeasure` as a default method:

```
public interface Measurable
{
   default double getMeasure() { return 0; }
}
```

If a class implements the interface and doesn't provide a getMeasure method, then it inherits this default method.

This particular example isn't all that useful. One doesn't normally want each object to have measure zero. Here is a more interesting example, in which a default method calls another interface method:

FULL CODE EXAMPLE
Go to wiley.com/go/bjlo2code to download an example of a default method in an interface.

```
public interface Measurable
{
   double getMeasure(); // An abstract method
   default boolean smallerThan(Measurable other)
   {
      return getMeasure() < other.getMeasure();
   }
}
```

The smallerThan method tests whether an object has a smaller measure than another, which is useful for arranging objects by increasing measure.

A class that implements the Measurable interface only needs to implement getMeasure, and it automatically inherits the smallerThan method. This can be a very useful mechanism. For example, the Comparator interface that is described in Special Topic 14.5 has one abstract method but more than a dozen default methods.

Special Topic 9.10

Function Objects

In the preceding section, you saw how the Measurable interface type makes it possible to provide services that work for many classes—provided they are willing to implement the interface type. But what can you do if a class does not do so? For example, we might want to compute the average length of a collection of strings, but String does not implement Measurable.

Let's rethink our approach. The average method needs to measure each object. When the objects are required to be of type Measurable, the responsibility for measuring lies with the objects themselves, which is the cause of the limitation that we noted. It would be better if another object could carry out the measurement. Let's move the measurement method into a different interface:

```
public interface Measurer
{
   double measure(Object anObject);
}
```

The measure method measures an object and returns its measurement. We use a parameter variable of type Object, the "lowest common denominator" of all classes in Java, because we do not want to restrict which classes can be measured.

We add a parameter variable of type Measurer to the average method:

```
public static double average(Object[] objects, Measurer meas)
{
   if (objects.length == 0) { return 0; }
   double sum = 0;
   for (Object obj : objects)
   {
      sum = sum + meas.measure(obj);
   }
   return sum / objects.length;
}
```

When calling the method, you need to supply a Measurer object. That is, you need to implement a class with a measure method, and then create an object of that class. Let's do that for measuring strings:

```
public class StringMeasurer implements Measurer
{
   public double measure(Object obj)
   {
      String str = (String) obj; // Cast obj to String type
      return str.length();
   }
}
```

Note that the `measure` method must accept an argument of type `Object`, even though this particular measurer just wants to measure strings. The parameter variable must have the same type as in the `Measurer` interface. Therefore, the `Object` parameter variable is cast to the `String` type.

Finally, we are ready to compute the average length of an array of strings:

```
String[] words = { "Mary", "had", "a", "little", "lamb" };
Measurer lengthMeasurer = new StringMeasurer();
double result = average(words, lengthMeasurer); // result is set to 3.6
```

FULL CODE EXAMPLE

Go to `wiley.com/go/bjlo2code` to download a complete program that demonstrates the string measurer.

An object such as `lengthMeasurer` is called a *function object*. The sole purpose of the object is to execute a single method, in our case `measure`. (In mathematics, as well as many other programming languages, the term "function" is used where Java uses "method".)

The `Comparator` interface, discussed in Special Topic 14.4, is another example of an interface for function objects.

Java 8 Note 9.3

Lambda Expressions

In Special Topic 9.10, you saw how to use function objects for specifying variations in behavior. The `average` method needs to measure each object, and it does so by calling the `measure` method of the supplied `Measurer` object.

Unfortunately, the caller of the `average` method has to do a fair amount of work; namely, to define a class that implements the `Measurer` interface and to construct an object of that class. Java 8 has a convenient shortcut for these steps, provided that the interface has a *single abstract method*. Such an interface is called a **functional interface** because its purpose is to define a single function. The `Measurer` interface is an example of a functional interface.

To specify that single function, you can use a **lambda expression**, an expression that defines the parameters and return value of a method in a compact notation. Here is an example:

```
(Object obj) -> ((BankAccount) obj).getBalance()
```

This expression defines a function that, given an object, casts it to a `BankAccount` and returns the balance.

(The term "lambda expression" comes from a mathematical notation that uses the Greek letter lambda (λ) instead of the -> symbol. In other programming languages, such an expression is called a *function expression.*)

A lambda expression cannot stand alone. It needs to be assigned to a variable whose type is a functional interface:

```
Measurer accountMeas = (Object obj) -> ((BankAccount) obj).getBalance();
```

Now the following actions occur:

1. A class is defined that implements the functional interface. The single abstract method is defined by the lambda expression.
2. An object of that class is constructed.
3. The variable is assigned a reference to that object.

You can also pass a lambda expression to a method. Then the parameter variable of the method is initialized with the constructed object. For example, consider the call

```
double averageBalance = average(accounts,
   (Object obj) -> ((BankAccount) obj).getBalance());
```

In the same way as before, an object is constructed that belongs to a class implementing `Measurer`. The object is used to initialize the parameter variable `meas` of the `average` method. Recall that the parameter variable has type `Measurer`:

```
public static double average(Object[] objects, Measurer meas)
{
   . . .
   sum = sum + meas.measure(obj);
   . . .
}
```

The `average` method calls the `measure` method on `meas`, which in turn executes the body of the lambda expression.

In its simplest form, a lambda expression contains a list of parameters and the expression that is being computed from the parameters. If more work needs to be done, you can write a method body in the usual way, enclosed in braces and with a `return` statement:

```
Measurer areaMeas = (Object obj) ->
   {
      Rectangle r = (Rectangle) obj;
      return r.getWidth() * r.getHeight();
   };
```

Lambda expressions enable the caller of a method to provide code that is called inside the method, and they enable the implementor of the method to invoke that code as needed. This can be achieved as follows:

1. The implementor of the method defines an interface that describes the purpose of the code to be executed. That interface has a single method.
2. The method receives a parameter of that interface, and calls the single method of the interface whenever the code that can vary needs to be called.
3. The caller of the method provides a lambda expression whose body is the code that should be called in this invocation.

You will see additional examples of using lambda expressions for event handlers (Java 8 Note 10.1) and comparators (Section 14.8). Lambda expressions are extensively used in the "streams" API for processing large data sets.

WORKED EXAMPLE 9.2 Investigating Number Sequences

Learn how to use a `Sequence` interface to investigate properties of arbitrary number sequences. Go to `wiley.com/go/bjlo2examples` and download Worked Example 9.2.

© Norebbo/iStockphoto.

VIDEO EXAMPLE 9.2 Drawing Geometric Shapes

In this Video Example, you will see how to use inheritance to describe and draw different geometric shapes. Go to `wiley.com/go/bjlo2videos` to view Video Example 9.2.