8.4 Referring to the Current Object's Members with the `this` Reference

Every object can access a *reference to itself* with keyword `this` (sometimes called the `this` **reference**). When an instance method is called for a particular object, the method's body *implicitly* uses keyword `this` to refer to the object's instance variables and other methods. This enables the class's code to know which object should be manipulated. As you'll see in Fig. 8.4, you can also use keyword `this` *explicitly* in an instance method's body. Section 8.5 shows another interesting use of keyword `this`. Section 8.11 explains why keyword `this` cannot be used in a `static` method.

Figure 8.4 demonstrates implicit and explicit use of the `this` reference. This example is the first in which we declare *two* classes in one file—class `ThisTest` is declared in lines 4–9, and class `SimpleTime` in lines 12–41. When you compile a `.java` file containing more than one class, the compiler produces a separate `.class` file for every class. In this case, two separate files are produced—`SimpleTime.class` and `ThisTest.class`. When one source-code (`.java`) file contains multiple class declarations, the compiler places the `.class` files in the *same* directory. Note also in Fig. 8.4 that only class `ThisTest` is declared `public`. A source-code file can contain only *one* `public` class—otherwise, a compilation error occurs. *Non-*`public` *classes* can be used only by other classes in the *same package*—recall from Section 3.2.5 that classes compiled into the same directory are in the same package. So, in this example, class `SimpleTime` can be used only by class `ThisTest`.

```
// Fig. 8.4: ThisTest.java
// this used implicitly and explicitly to refer

public class ThisTest {
   public static void main(String[] args) {
      SimpleTime time = new SimpleTime(15, 30, 1
      System.out.println(time.buildString());
   }
}

// class SimpleTime demonstrates the “this” refe
class SimpleTime {
   private int hour; // 0-23
   private int minute; // 0-59
   private int second; // 0-59

   // if the constructor uses parameter names id
   // instance variable names the “this” referen
   // required to distinguish between the names
   public SimpleTime(int hour, int minute, int s
      this.hour = hour; // set “this” object’s h
      this.minute = minute; // set “this” object
      this.second = second; // set “this” object
   }

   // use explicit and implicit “this” to call t
   public String buildString() {
      return String.format(“%24s: %s%n%24s: %s”,
         “this.toUniversalString()”, this.toUniv
         “toUniversalString()”, toUniversalStrin
31      }
32
33      // convert to String in universal-time format
34      public String toUniversalString() {
35         // “this” is not required here to access i
36         // because method does not have local vari
37         // names as instance variables
38         return String.format(“%02d:%02d:%02d”,
39            this.hour, this.minute, this.second);
40      }
41   }
```

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

Fig. 8.4

`this` used implicitly and explicitly to refer to members of an object.

Class `SimpleTime` (lines 12–41) declares three `private` instance variables—`hour`, `minute` and `second` (lines 13–15). The class’s constructor (lines 20–24) receives three `int` arguments to initialize a `SimpleTime` object. Once again, we used parameter names for the constructor that are *identical* to the class’s instance-variable names (lines 13–15), so we use the `this` reference to refer to the instance variables in lines 21–23.

Error-Prevention Tip 8.1

Most IDEs will issue a warning if you say `x = x`*; instead of* `this.x = x`*;. The statement* `x = x`*; is often called a no-op (no operation).*

Method `buildString` (lines 27–31) returns a `String` created by a statement that uses the `this` reference explicitly and implicitly. Line 29 uses it *explicitly* to call method `toUniversalString`. Line 30 uses it *implicitly* to call the same method. Both lines perform the same task. You typically will not use `this` explicitly to reference other methods within the current object. Also, line 39 in method `toUniversalString` explicitly uses the `this` reference to access each instance variable. This is *not* necessary here, because the method does *not* have any local variables that shadow the instance variables of the class.

Performance Tip 8.1

There's only one copy of each method per class— every *object of the class shares the method's code. Each object, on the other hand, has its own copy of the class's instance variables. The class's non-static methods implicitly use this to determine the specific object of the class to manipulate.*

Class `ThisTest`'s `main` method (lines 5–8) demonstrates class `SimpleTime`. Line 6 creates an instance of class

`SimpleTime` and invokes its constructor. Line 7 invokes the object’s `buildString` method, then displays the results.