8.5 `Time` Class Case Study: Overloaded Constructors

As you know, you can declare your own constructor to specify how objects of a class should be initialized. Next, we demonstrate a class with several **overloaded constructors** that enable objects of that class to be initialized in different ways. To overload constructors, simply provide multiple constructor declarations with different signatures.

Class `Time2` with Overloaded Constructors

The `Time1` class's default constructor in Fig. 8.1 initialized `hour`, `minute` and `second` to their default `0` values (i.e., midnight in universal time). The default constructor does not enable the class's clients to initialize the time with nonzero values. Class `Time2` (Fig. 8.5) contains five overloaded constructors that provide convenient ways to initialize objects. In this program, four of the constructors invoke a fifth, which in turn ensures that the value supplied for `hour` is in the range 0 to 23, and the values for `minute` and `second` are each in the range 0 to 59. The compiler invokes the appropriate constructor by matching the number, types and order of the types of the arguments specified in the constructor call with the number, types and order of the types of the parameters specified in each constructor declaration. Class `Time2` also provides *set* and *get* methods for each instance variable.

```
// Fig. 8.5: Time2.java
// Time2 class declaration with overloaded const

public class Time2 {
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // Time2 no-argument constructor:
   // initializes each instance variable to zero
   public Time2() {
      this(0, 0, 0); // invoke constructor with
   }

   // Time2 constructor: hour supplied, minute a
   public Time2(int hour) {
      this(hour, 0, 0); // invoke constructor wi
   }

   // Time2 constructor: hour and minute supplie
   public Time2(int hour, int minute) {
      this(hour, minute, 0); // invoke construct
   }

   // Time2 constructor: hour, minute and second
   public Time2(int hour, int minute, int second
      if (hour < 0 || hour >= 24) {
         throw new IllegalArgumentException(“hou
      }

      if (minute < 0 || minute >= 60) {
         throw new IllegalArgumentException(“min
      }
```

```

        if (second < 0 || second >= 60) {
            throw new IllegalArgumentException(“sec
        }

        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // Time2 constructor: another Time2 object su
    public Time2(Time2 time) {
        // invoke constructor with three arguments
        this(time.hour, time.minute, time.second);
    }

    // Set Methods
    // set a new time value using universal time;
    // validate the data
    public void setTime(int hour, int minute, int
        if (hour < 0 || hour >= 24) {
            throw new IllegalArgumentException(“hou
        }

        if (minute < 0 || minute >= 60) {
            throw new IllegalArgumentException(“min
        }

        if (second < 0 || second >= 60) {
            throw new IllegalArgumentException(“sec
        }

        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // validate and set hour
    public void setHour(int hour) {
        if (hour < 0 || hour >= 24) {
            throw new IllegalArgumentException(“hou
         }

         this.hour = hour;
      }

      // validate and set minute
      public void setMinute(int minute) {
         if (minute < 0 || minute >= 60) {
            throw new IllegalArgumentException(“min
         }

         this.minute = minute;
      }

      // validate and set second
      public void setSecond(int second) {
         if (second < 0 || second >= 60) {
            throw new IllegalArgumentException(“sec
         }

         this.second = second;
      }

      // Get Methods
      // get hour value
      public int getHour() {return hour;}

      // get minute value
      public int getMinute() {return minute;}

      // get second value
      public int getSecond() {return second;}

      // convert to String in universal-time forma
      public String toUniversalString() {
         return String.format(
            “%02d:%02d:%02d”, getHour(), getMinute
      }

```

```
    // convert to String in standard-time format
    public String toString() {
        return String.format(“%d:%02d:%02d %s”,
            ((getHour() == 0 || getHour() == 12) ?
            getMinute(), getSecond(), (getHour() <
    }
}
```

Fig. 8.5

`Time2` class declaration with overloaded constructors.

Class `Time2`’s Constructors—Calling One Constructor from Another via `this`

Lines 11–13 declare a so-called **no-argument constructor** that’s invoked without arguments. Once you declare any constructors in a class, the compiler will *not* provide a *default constructor*. This no-argument constructor ensures that class `Time2`’s clients can create `Time2` objects with default values. Such a constructor simply initializes the object as specified in the constructor’s body. In the body, we introduce a use of `this` that’s allowed only as the *first* statement in a constructor’s body. Line 12 uses `this` in method-call syntax to invoke the `Time2` constructor that takes three parameters (lines 26–42) with values of 0 for the `hour`, `minute` and `second`. Using `this` as shown here is a popular way to *reuse* initialization code provided by another of the class's constructors rather than defining similar code in the no-argument constructor's body. A constructor that calls another constructor in this manner is known as a **delegating constructor**. We use this syntax in four of the five `Time2` constructors to make the class easier to maintain and modify. If we need to change how objects of class `Time2` are initialized, only the constructor that the class's other constructors call will need to be modified.

Common Programming Error 8.2

It's a compilation error when `this` *is used in a constructor's body to call another of the class's constructors if that call is not the first statement in the constructor. It's also a compilation error when a method attempts to invoke a constructor directly via* `this`.

Lines 16–18 declare a `Time2` constructor with a single `int` parameter representing the `hour`, which is passed with 0 for the `minute` and `second` to the constructor at lines 26–42. Lines 21–23 declare a `Time2` constructor that receives two `int` parameters representing the `hour` and `minute`, which are passed with 0 for the `second` to the constructor at lines 26–42. Like the no-argument constructor, each of these constructors invokes the three-argument constructor to minimize code duplication. Lines 26–42 declare the `Time2` constructor that receives three `int` parameters representing the `hour`, `minute` and `second`. This constructor validates and initializes the instance variables.

Lines 45–48 declare a `Time2` constructor that receives a reference to another `Time2` object. The argument object’s values are passed to the three-argument constructor to initialize the `hour`, `minute` and `second`. Line 47 directly accesses the `hour`, `minute` and `second` values of the argument `time` with the expressions `time.hour`, `time.minute` and `time.second`—even though `hour`, `minute` and `second` are declared as `private` variables of class `Time2`. This is due to a special relationship between objects of the same class.

Software Engineering Observation 8.5

When one object of a class has a reference to another object of the same class, the first object can access all *the second object’s data and methods (including those that are* `private`*).*

Class `Time2`’s `setTime`

Method

Method `setTime` (lines 53–69) throws an `IllegalArgumentException` (lines 55, 59 and 63) if any of the method’s arguments is out of range. Otherwise, it sets `Time2`’s instance variables to the argument values (lines 66–68).

Notes Regarding Class `Time2`’s Set and Get Methods and Constructors

`Time2`’s *get* methods are called from other methods of the class. In particular, methods `toUniversalString` and `toString` call `getHour`, `getMinute` and `getSecond` in line 111 and lines 117–118, respectively. In each case, these methods could have accessed the class’s private data directly without calling the *get* methods. However, consider changing the representation of the time from three `int` values (requiring 12 bytes of memory) to a single `int` value representing the total number of seconds that have elapsed since midnight (requiring only four bytes of memory). If we made such a change, only the bodies of the methods that access the `private` data directly would need to change—in particular, the three-argument constructor, the `setTime` method and the individual *set* and *get* methods for the `hour`, `minute` and `second`. There would be no need to modify the bodies of methods `toUniversalString` or `toString` because they do *not* access the data directly. Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.

Similarly, each `Time2` constructor could include a copy of the appropriate statements from the three-argument constructor. Doing so may be slightly more efficient, because the extra constructor calls are eliminated. But, *duplicating* statements makes changing the class's internal data representation more difficult. Having the `Time2` constructors call the constructor with three arguments requires that any changes to the implementation of the three-argument constructor be made only once. Also, the compiler can optimize programs by removing calls to simple methods and replacing them with the expanded code of their declarations—a technique known as **inlining the code**, which improves program performance.

Using Class `Time2`'s Overloaded Constructors

Class `Time2Test` (Fig. 8.6) invokes the overloaded `Time2` constructors (lines 6–10 and 21). Line 6 invokes the `Time2` no-argument constructor. Lines 7–10 demonstrate passing arguments to the other `Time2` constructors. Line 7 invokes the single-argument constructor that receives an `int` at lines 16–18 of Fig. 8.5. Line 8 of Fig. 8.6 invokes the two-argument constructor at lines 21–23 of Fig. 8.5. Line 9 of Fig. 8.6

invokes the three-argument constructor at lines 26–42 of Fig. 8.5. Line 10 of Fig. 8.6 invokes the single-argument constructor that takes a `Time2` at lines 45–48 of Fig. 8.5. Next, the app displays the `String` representations of each `Time2` object to confirm that it was initialized properly (lines 13–17 of Fig. 8.6). Line 21 attempts to initialize `t6` by creating a new `Time2` object and passing three *invalid* values to the constructor. When the constructor attempts to use the invalid hour value to initialize the object's `hour`, an `IllegalArgumentException` occurs. We catch this exception at line 23 and display its error message, which results in the last line of the output.

```
1   // Fig. 8.6: Time2Test.java
2   // Overloaded constructors used to initialize Ti
3
4   public class Time2Test {
5      public static void main(String[] args) {
6         Time2 t1 = new Time2(); // 00:00:00
7         Time2 t2 = new Time2(2); // 02:00:00
8         Time2 t3 = new Time2(21, 34); // 21:34:00
9         Time2 t4 = new Time2(12, 25, 42); // 12:25
10        Time2 t5 = new Time2(t4); // 12:25:42
11
12        System.out.println("Constructed with:");
13        displayTime("t1: all default arguments", t
14        displayTime("t2: hour specified; default m
15        displayTime("t3: hour and minute specified
16        displayTime("t4: hour, minute and second s
17        displayTime("t5: Time2 object t4 specified
18
19        // attempt to initialize t6 with invalid v
20        try {
21           Time2 t6 = new Time2(27, 74, 99); // in
22        }
```

```
      catch (IllegalArgumentException e) {
         System.out.printf(“%nException while in
            e.getMessage());
      }
   }

   // displays a Time2 object in 24-hour and 12-
   private static void displayTime(String header
      System.out.printf(“%s%n %s%n %s%n”,
         header, t.toUniversalString(), t.toStri
   }
}
```

```
Constructed with:
t1: all default arguments
   00:00:00
   12:00:00 AM
t2: hour specified; default minute and second
   02:00:00
   2:00:00 AM
t3: hour and minute specified; default second
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: Time2 object t4 specified
   12:25:42
   12:25:42 PM

Exception while initializing t6: hour must be 0-23
```

Fig. 8.6

Overloaded constructors used to initialize `Time2` objects.