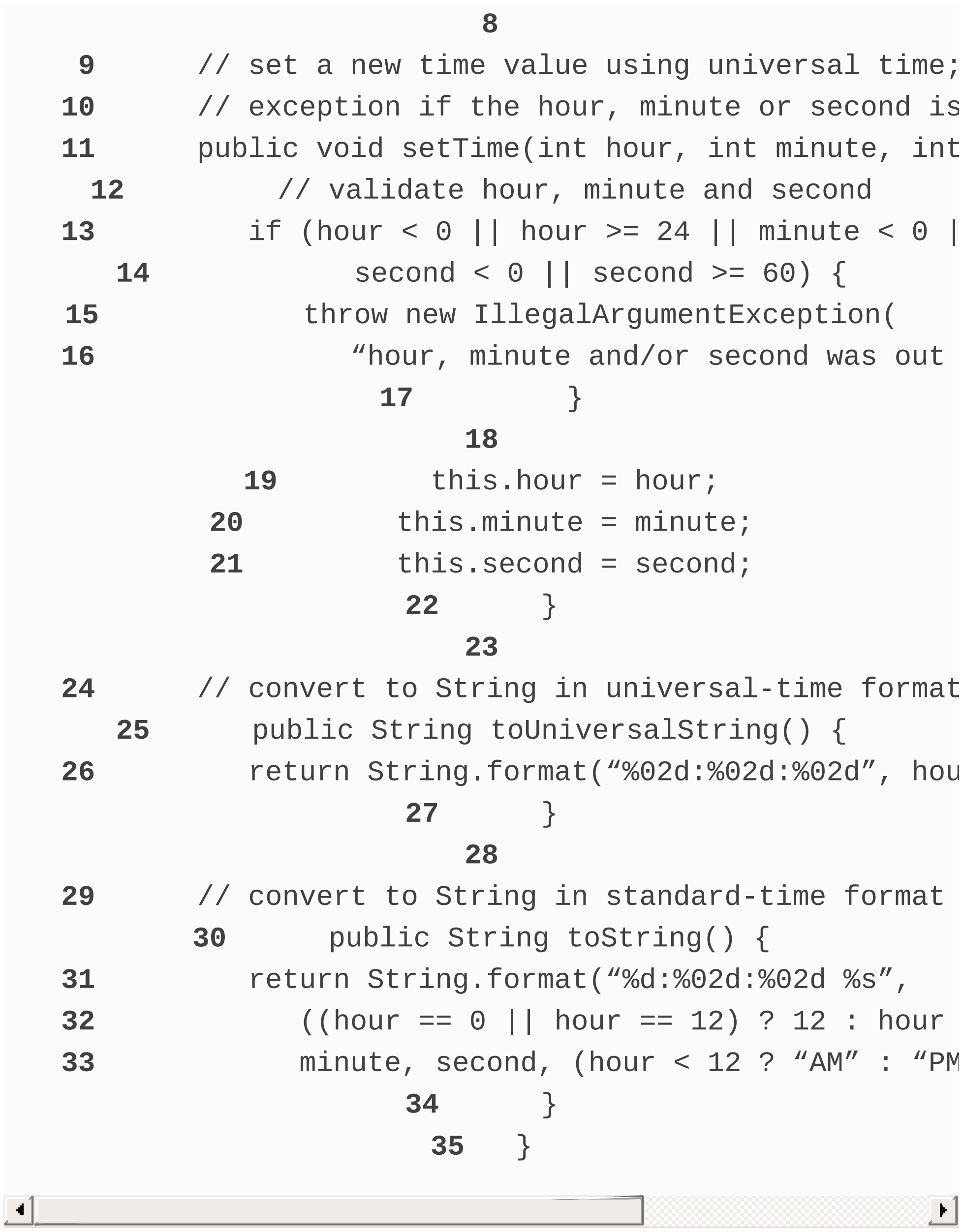
8.2 `Time` Class Case Study

Our first example consists of two classes—`Time1` (Fig. 8.1) and `Time1Test` (Fig. 8.2). Class `Time1` represents the time of day. Class `Time1Test`'s `main` method creates one object of class `Time1` and invokes its methods. The output of this program appears in Fig. 8.2.

`Time1` Class Declaration

Class `Time1`'s `private int` instance variables `hour`, `minute` and `second` (Fig. 8.1, lines 5–7) represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23, and minutes and seconds are each in the range 0–59). `Time1` contains `public` methods `setTime` (lines 11–22), `toUniversalString` (lines 25–27) and `toString` (lines 30–34). These methods are also called the `public` **services** or the `public` **interface** that the class provides to its clients.

```
1  // Fig. 8.1: Time1.java
2  // Time1 class declaration maintains the time in
3
4  public class Time1 {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
```



```java

   // set a new time value using universal time;
   // exception if the hour, minute or second is
   public void setTime(int hour, int minute, int
      // validate hour, minute and second
      if (hour < 0 || hour >= 24 || minute < 0 |
         second < 0 || second >= 60) {
         throw new IllegalArgumentException(
            “hour, minute and/or second was out
      }

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

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

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

Fig. 8.1

`Time1` class declaration maintains the time in 24-hour format.

Default Constructor

In this example, class `Time1` does *not* declare a constructor, so the compiler supplies a default constructor (as we discussed in Section 3.3.2). Each instance variable implicitly receives the default `int` value. Instance variables also can be initialized when they're declared in the class body, using the same initialization syntax as with a local variable.

Method `setTime` and Throwing Exceptions

Method `setTime` (lines 11–22) is a `public` method that declares three `int` parameters and uses them to set the time. Lines 13–14 test each argument to determine whether the value is outside the proper range. The `hour` value must be greater than or equal to `0` and less than `24`, because universal-time format represents hours as integers from 0 to 23 (e.g., 1 PM is hour 13 and 11 PM is hour 23; midnight is hour 0 and noon is hour 12). Similarly, both `minute` and `second` values must be greater than or equal to `0` and less than `60`. For values outside these ranges, `setTime` **throws an exception** of type `IllegalArgumentException` (lines 15–16), which notifies the client code that an invalid argument was passed to the method. As you learned in Section 7.5, you can use `try`...`catch` to catch exceptions and attempt to recover from them, which we'll do in Fig. 8.2. The class instance creation expression in the `throw` **statement** (Fig. 8.1; line 15) creates a new object of type `IllegalArgumentException`. The parentheses indicate a call to the `IllegalArgumentException` constructor. In this case, we call the constructor that allows us to specify a custom error message. After the exception object is created, the `throw` statement immediately terminates method `setTime` and the exception is returned to the calling method that attempted to set the time. If the argument values are all valid, lines 19–21 assign them to the `hour`, `minute` and `second` instance variables.

Software Engineering Observation 8.1

For a method like `setTime` *in Fig. 8.1, validate all of the method's arguments before using them to set instance variable values to ensure that the object's data is modified only if all the arguments are valid.*

Method `toUniversalString`

Method `toUniversalString` (lines 25–27) takes no arguments and returns a `String` in *universal-time format*, consisting of two digits each for the hour, minute and second —recall that you can use the `0` flag in a `printf` format specification (e.g., `“%02d”`) to display leading zeros for a value that doesn’t use all the character positions in the specified field width. For example, if the time were 1:30:07 PM, the method would return `13:30:07`. Line 26 uses `static` method `format` of class `String` to return a `String` containing the formatted `hour`, `minute` and `second` values, each with two digits and possibly a leading `0` (specified with the `0` flag). Method `format` is similar to method `System.out.printf` except that `format` *returns* a formatted `String` rather than displaying it in a command window. The formatted `String` is returned by method `toUniversalString`.

Method `toString`

Method `toString` (lines 30–34) takes no arguments and returns a `String` in *standard-time format*, consisting of the `hour`, `minute` and `second` values separated by colons and followed by AM or PM (e.g., `11:30:17 AM` or `1:27:06 PM`). Like method `toUniversalString`, method `toString` uses `static String` method `format` to format the `minute` and `second` as two-digit values, with leading zeros if necessary. Line 32 uses a conditional operator (`?:`) to determine the value for `hour` in the `String`—if the `hour` is `0` or `12` (AM or PM), it appears as 12; otherwise, it appears as a value from 1 to 11. The conditional operator in line 33 determines whether AM or PM will be returned as part of the `String`.

Recall that all objects in Java have a `toString` method that returns a `String` representation of the object. We chose to return a `String` containing the time in standard-time format. Method `toString` is called *implicitly* whenever a `Time1` object appears in the code where a `String` is needed, such as the value to output with a `%s` format specifier in a call to `System.out.printf`. You may also call `toString` *explicitly* to obtain a `String` representation of a `Time` object.

Using Class `Time1`

Class `Time1Test` (Fig. 8.2) uses class `Time1`. Line 7 declares the `Time1` variable `time` and initializes it with a new `Time1` object. Operator `new` implicitly invokes class `Time1`'s default constructor, because `Time1` does not declare any constructors. To confirm that the `Time1` object was initialized properly, line 10 calls the `private` method `displayTime` (lines 31–34), which, in turn, calls the `Time1` object's `toUniversalString` and `toString` methods to output the time in universal-time format and standard-time format, respectively. Note that `toString` could have been called implicitly here rather than explicitly. Next, line 14 invokes method `setTime` of the `time` object to change the time. Then line 15 calls `displayTime` again to output the time in both formats to confirm that it was set correctly.

Software Engineering Observation 8.2

Recall from Chapter 3 *that methods declared with access modifier* `private` *can be called* only *by other methods of the class in which the* `private` *methods are declared. Such methods are commonly referred to as* ***utility methods*** *or* ***helper methods*** *because they're typically used to support the operation of the class's other methods.*

```
// Fig. 8.2: Time1Test.java
// Time1 object used in an app.

public class Time1Test {
   public static void main(String[] args) {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // invokes Time1

      // output string representations of the ti
      displayTime("After time object is created"
      System.out.println();

      // change time and output updated time
      time.setTime(13, 27, 6);
      displayTime("After calling setTime", time)
      System.out.println();

      // attempt to set time with invalid values
      try {
         time.setTime(99, 99, 99); // all values
      }
      catch (IllegalArgumentException e) {
         System.out.printf("Exception: %s%n%n",
      }

      // display time after attempt to set inval
      displayTime(“After calling setTime with in
   }

   // displays a Time1 object in 24-hour and 12-
   private static void displayTime(String header
      System.out.printf(“%s%nUniversal time: %s%
         header, t.toUniversalString() t.toStrin
   }
}
```

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of rang

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 8.2

`Time1` object used in an app.

Calling `Time1` Method `setTime` with Invalid Values

To illustrate that method `setTime` *validates* its arguments, line 20 calls method `setTime` with *invalid* arguments of `99` for the `hour`, `minute` and `second`. This statement is placed in a `try` block (lines 19–21) in case `setTime` throws an `IllegalArgumentException`, which it will do since the arguments are all invalid. When this occurs, the exception is caught at lines 22–24, and line 23 displays the exception’s error message by calling its `getMessage` method. Line 27 outputs the time again in both formats to confirm that `setTime` did *not* change the time when invalid arguments were supplied.

Software Engineering of the `Time1` Class Declaration

Consider several issues of class design with respect to class `Time1`. The instance variables `hour`, `minute` and `second` are each declared `private`. The actual data representation used within the class is of no concern to the class’s clients. For example, it would be perfectly reasonable for `Time1` to represent the time internally as the number of seconds since midnight or the number of minutes and seconds since midnight. Clients could use the same `public` methods and get the same results without being aware of this. (Exercise 8.5 asks you to represent the time in class `Time2` of Fig. 8.5 as the number of seconds since midnight and show that indeed no change is visible to the clients of the class.)

Software Engineering Observation 8.3

Classes simplify programming, because the client can use only a class's `public` *methods. Such methods are usually client oriented rather than implementation oriented. Clients are neither aware of, nor involved in, a class's implementation. Clients generally care about* what *the class does but not* how *the class does it.*

Software Engineering Observation 8.4

Interfaces change less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. Hiding the implementation reduces the possibility that other program parts will become dependent on a class's implementation details.

Java SE 8—Date/Time API

8

This section’s example and several of this chapter’s later examples demonstrate various class-implementation concepts in classes that represent dates and times. In professional Java development, rather than building your own date and time classes, you’ll typically reuse the ones provided by the Java API. Though Java has always had classes for manipulating dates and times, Java SE 8 introduced a new **Date/Time API** —defined by the classes in the package `java.time`. Applications built with Java SE 8 should use the Date/Time API’s capabilities, rather than those in earlier Java versions. The new API fixes various issues with the older classes and provides more robust, easier-to-use capabilities for manipulating dates, times, time zones, calendars and more. We use some Date/Time API features in Chapter 23. You can learn more about the Date/Time API’s classes at:

```
http://docs.oracle.com/javase/8/docs/api/java/time/pa
```