Chapter 10. Developing Robust Programs

Leo Tolstoy began his classic *Anna Karenina* with the statement “Happy families are all alike; every unhappy family is unhappy in its own way.” So far in this text, we have made it a general practice of assuming “happy” scenarios – our users always behave as expected, programmers always use our APIs appropriately, and the files we read always contain properly-formatted data. In Chapter 7 for example we assumed that no user would ever try to enter the string “forty” (rather than the digits “40”) as a frame rate, in Chapter 9 we assumed that no programmer would ever try to construct a temperature below absolute zero, and in Chapter 8 we assumed that the country names stored in the input files never have spaces in their names, e.g., “United States”.

These assumptions are hopelessly naïve. As Tolstoy observed, for every happy scenario there are myriad unhappy scenarios. We know that users can and probably will break every assumption our code makes, either purposely or accidentally. We know that programmers who don’t understand our class APIs may use them improperly; truth be told, we acknowledge that over time even we forget how to use our own APIs properly. We know that the content of data files is notoriously inconsistent and incomplete.

In addition to ignoring problems related to the use of our code, we have also largely ignored the problem of testing our code. We have, for the most part, developed our applications, run them a few times to see if they seem to be working and then moved on to the next programming task. As our applications grow increasingly complex, the viability of this ad hoc approach to testing decreases alarmingly. There are simply too many things to test when we write code and too many subtle problems that can be caused each time we modify it.

Our goal is to develop correct programs, which means that our programs must produce appropriate behaviors in both the happy scenarios and the unhappy scenarios. Fortunately, the Java development environment provides tools that help us with this task. It provides unit testing tools to help ensure that our programs produce correct results in the happy scenarios, and exception handling tools to help ensure that our programs respond appropriately in the unhappy scenarios. This chapter discusses exception handling in Java and then unit testing using JUnit. It also introduces the notion of user testing for the purpose of ensuring the usability of our interactive applications. Finally, it introduces enumerated types as a way to make our programs easier to understand and therefore easier to modify and maintain correctly.

10.1. Example: Temperature Conversion

Some computing applications must work with temperature measurements, each of which may be represented using a different measurement scale, e.g., Celsius, Fahrenheit or Kelvin. To support such applications, Chapter 9 created a Temperature class that encapsulates the key features of temperature objects. Each temperature object knows its own magnitude in degrees (e.g., 98.6) and the scale in which it is represented (e.g., Fahrenheit) and it knows how to convert its magnitude from one scale to another.

This chapter focuses on maintaining the integrity of the temperature objects constructed by our class so that they behave in a robust, correct manner in all situations. In Chapter 9, we included some preliminary code that rejects attempts to create invalid temperatures, say temperatures below absolute zero or temperatures specified in unknown scales, but that code was heavy-handed, using `System.exit()` to terminate the program entirely if any problem, large or small, was found. Java provides better mechanisms for handling these issues more even-handedly.

In Chapter 9, we also attempted to ensure that the temperature class performed scale conversions properly. For example, when we created a 0.0° Celsius temperature object and asked it to convert itself to Fahrenheit, we expected the result to be a 32.0° Fahrenheit temperature object (because we know that 0.0° C = 32.0° F). We tried to ensure the correctness of these conversions by manually running a set of test cases, e.g., 0° C = 32° F , 100° C = 212° F , etc, but this proved to be both tedious and error-prone. There are too many test cases to remember and they need to be run every time we change the implemention of the temperature class. Instead, we need to be more systematic in our testing. Given the range of temperatures that we could represent, it would be impractical to test every possible case and every possible conversion. However, we need to test a sufficient range of cases to assure ourselves that the system can handle anything that it will see in a production environment and we would like to automate this testing so that it is easy to run and re-run whenever we modify our temperature class definition. Java provides mechanisms for doing this as well.

To illustrate these things, this chapter will reuse the temperature class code from Chapter 9 and construct a simple console application for temperature conversion that will look something like the example shown in Figure 10-1. This application should allow the user to enter a temperature value in any of the scales and then should automatically convert the value into the desired scale. Our class should resist any attempt to construct objects representing temperatures below absolute zero, and it should perform scale transformations correctly in all legal cases. The figure shows one happy scenario in which the user enters 98.6° Fahrenheit, which we know to be a legal temperature, and the system should correctly convert that value into the appropriate Kelvin equivalent. At this point, we can't remember what that Kelvin value should be, but we'll figure that all out when we specify the formal tests.

Enter a temperature: 98.6 F
Convert to: K
Result: *??*

Figure 10-1. A simple temperature converter utility

This chapter focuses less on the development of the class itself, which was discussed at length in Chapter 9, but focuses instead on the development of a robust temperature class that gives correct results in happy scenarios and responds appropriately in unhappy scenarios.

10.2. Exception Handling

This section discusses Java tools for anticipating and handling unhappy scenarios. In particular, it discusses Java's exception handling mechanism.

10.2.1. Working with Exceptions

When the Java environment encounters a fault of some sort during the execution of a program, it creates an object representing that fault. These objects are called *exceptions*. Java creates exceptions in a number of different situations. For example, the following code segment attempts to print the result of 1/0:

```
System.out.println(1/0);
```

In this unhappy scenario, we've attempted to divide by zero, which we know to be undefined. The Java environment prints the following message on the text output console:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at c10quality.TemperatureConsole.main(TemperatureConsole.java:13)
```

This output indicates that a division-by-zero exception has occurred at line 13 of the `main()` method defined in `TemperatureConsole.java`. In this case, we say that Java has *thrown* an `ArithmeticException`. When Java throws an exception like this, it does not automatically terminate the program. We can write code to deal with this exception in some appropriate manner, say by printing an alternate message on the screen. This is called *handling* an exception, and we discuss the flexibility of this technique below.

For our temperature class, consider what happens if the programmer attempts to construct a temperature object whose value is below absolute zero (e.g., -1.0° Kelvin)? In Chapter 9, we added code to the explicit-value constructor to deal (heavy-handedly) with this case:

```
public Temperature(double degrees, char scale) {
      if (isValidTemperature(degrees, scale)) {
            myDegrees = degrees;
            myScale = Character.toUpperCase(scale);
      } else {
            System.err.println("Invalid Temperature: (" + degrees + ","
                                                        + scale + ")");
            System.exit(-1);
      }
}
```

If a programmer attempts to create valid temperature object, everything works as normal. If, on the other hand, the programmer attempts to create a sub-absolute-zero temperature object, this code prints an error message and exits the application. This implementation had the benefit of enforcing our established invariant, which was good enough for our purposes in Chapter 9, but it puts the `Temperature` class in charge of what happens to the application; the `TemperatureConsole` application therefore loses control over the application. This is not a good division of responsibilities. The `Temperature` class should control the integrity of each temperature object and the `TemperatureConsole` should control what happens with the application. In this scenario, it would probably be preferable to keep the application running and to print a message instead, warning the user gently about the evils of sub-absolute-zero temperatures.

An alternative approach might be to program `TemperatureConsole` to handle sub-absolute-zero temperatures before even trying to construct one using the temperature explicit-value constructor. This

could work, but would require that `TemperatureConsole` know about what makes a temperature valid or invalid, which puts the `TemperatureConsole` in charge of `Temperature` objects. This is not a good division of responsibilities either. It's difficult and unnecessary for the `TemperatureConsole` to know about the temperature class's established invariants; these are things best hidden inside the `Temperature` class.

A better option would be to: (1) program the `Temperature` to determine if its invariants have been violated and to alert its calling program, here the `TemperatureConsole`, whenever that happens; and (2) program the `TemperatureConsole` to listen for the alerts and manage the application appropriately when they happen. This puts the temperature class in charge of the integrity of temperature objects and the temperature console in charge of the application, which is the correct division of responsibilities.

Though we cannot easily implement these behaviors with the control structures we have discussed so far in this text, Java provides an exception handling mechanism that supports both of them. The next two sections discuss each of these behaviors in turn.

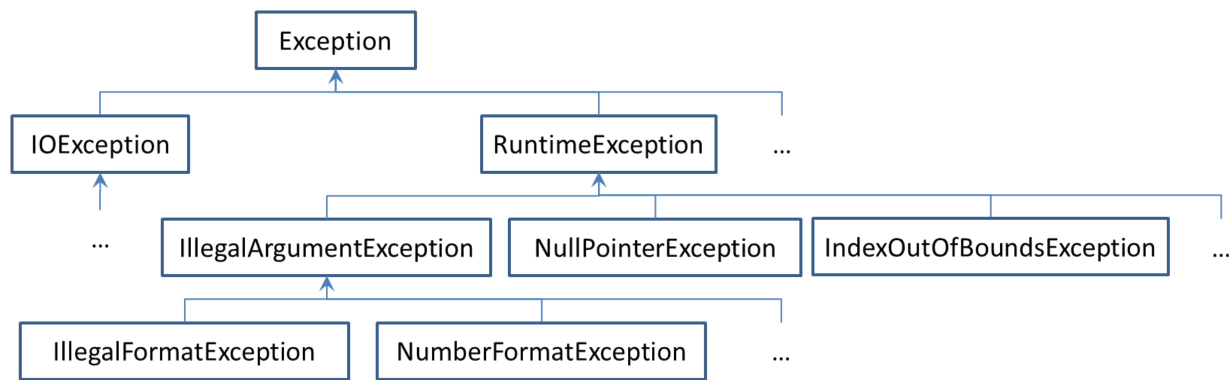
10.2.2. Throwing Exceptions

The temperature class's explicit-value constructor can "alert" its calling program to problems by constructing and *throwing* an exception object as follows:

```
public Temperature(double degrees, char scale) throws Exception {
      if (isValidTemperature(degrees, scale)) {
            myDegrees = degrees;
            myScale = Character.toUpperCase(scale);
      } else {
            throw new Exception("Invalid Temperature: " + degrees + " "
                                                      + scale);
      }
}
```

If the received values are valid, then this code constructs the specified temperature object. If, on the other hand, the received values are invalid, this code constructs a new object of type `Exception` and "throws" it back to the calling program. The constructor for the Exception class receives a string that should specify a useful message indicating the nature of the exception. The `throw` command terminates the execution of the method and returns control back to the calling program. Methods that throw `Exception` objects must add the `throws Exception` clause to their declaration. In this case, the exception indicates that the constructor has received an illegal temperature specification ("Invalid Temperature:…"). Given this new definition of the explicit-value constructor, programmers that construct new temperature objects must handle the thrown exception as discussed in the next section.

The example uses a generic `Exception` object. Java supports a variety of specialized exception types, modeled as a hierarchy shown here:



This hierarchy includes as its root the `Exception` type used above. It also includes some exception types that you may have encountered in other circumstances, including the `NumberFormatException`, thrown when the Integer class tries to read non-digits as an integer (e.g., when scanning "forty" rather than "40"), the `IndexOutOfBoundsException`, thrown when indexing beyond the end of any array, and the `NullPointerException`, thrown when Java tries to use an object handle that has not been initialized. Though it is generally a good idea to use the most specific exception type appropriate for the given situation, this text adopts the simplifying practice of using `Exception`, the most general exception type.

As shown above, throwing an exception of type `Exception` requires that the method declare that it might throw an exception. This is called a *checked* exception because the throwing method must specify a `throws` declaration and the calling program must handle the exception. It is generally a good idea to be explicit about exceptions in this way, but it does require extra work of the programmer. To help ease the burden somewhat, Java also provides *unchecked* exceptions, which do not require the `throws` declaration or exception handling. The `RuntimeException` type and its children are unchecked exceptions. If a calling program does not *handle* an unchecked exception, then that calling program also terminates and returns control back to its calling program, and so forth. This propagation continues until some code handles the exception. Ultimately, the Java runtime environment will handle the exception by printing the error message to the output console, as shown in the previous section.[1]

The pattern for using the `throw` statement is given here.

Throw Pattern

```
throw ExceptionObject;
```

- *`ExceptionObject`* is an object of some type in the `Exception` class hierarchy; checked exception types must add a `throws` clause to the method declaration line.

[1] A discussion of the checked-unchecked exception issue can be found here: http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html.

10.2.3. Handling Exceptions

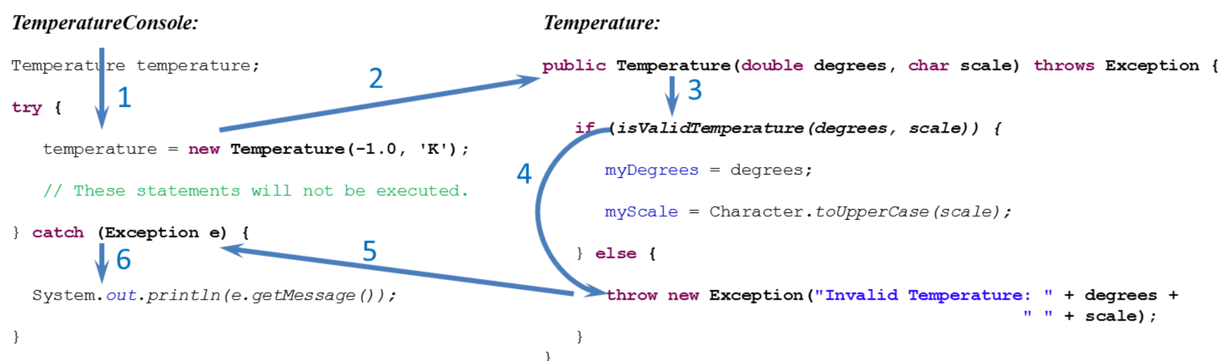
`throw` statements are an alternate form of the `return` statement in that they return control back to the calling program, but a `throw` must be handled differently than a simple `return`. The calling program must handle a thrown error using a `try-catch` block.

```
try {
      Temperature temperature1 = new Temperature(-1.0, 'k');
} catch (Exception e) {
      System.out.println(e.getMessage());
}
```

The `try{}` block contains calls to methods that might throw exceptions. In this example, the call to the temperature explicit-value constructor may or may not throw an exception depending upon whether it likes the arguments it receives. The `catch{}` block contains the statements that respond to any exception that is raised and "receives" the exception that it is to handle using a construct similar to a parameter list, i.e., `(Exception e)`. In this example, the `TemperatureConsole` attempts to create a -1° Kelvin temperature, which causes the Temperature class to throw an exception. The try-catch block of the console class catches the exception, giving it the identifier `e`, and accesses the error message specified by the constructor using the `getMessage()` method on the exception object `e`. The output is shown here:

```
Invalid Temperature: -1.0 K
```

The flow of control looks something like this:



Control starts in the temperature console application code (on the left), proceeds as normal to the call to the explicit-value constructor, which passes control to the definition of the constructor in the temperature class (on the right). Because the temperature is invalid, the constructor executes its `else` clause, which constructs and throws an exception object back to the calling program. The `catch` clause receives the thrown exception object and prints out the message contained therein.

If no statement in the try block throws an exception, then control proceeds to the end of the `try` block, skips the `catch{}` block and goes on to the statements following the try-catch blocks. If any statement in the `try` block does throw an exception, then Java looks for a corresponding `catch` block that specifies the type of exception that was thrown (or one of its parents in the exception hierarchy) and

transfers control to that block. In the example given above, the method throws an exception of type `Exception` and the `catch` block specifies the same type `Exception`. Thus, control passes from the method call that threw the error directly to the catch block, skipping any as-of-yet unexecuted statements in the `try` block.

Each `try-catch` block has one `try` block that may contain multiple statements that could throw exceptions. Each `try-catch` block must have at least one `catch` block, but it can have multiple `catch` blocks each handling a different type of exception. Consider the following code, which may raise and handle two different types of exceptions:

```
Temperature temperature;
Scanner scanner = new Scanner(System.in);
System.out.print("Please enter a temperature: ");
try {
      temperature = new Temperature(scanner.nextDouble(),
                                    scanner.next().charAt(0));
      System.out.println("You entered: " + temperature);
} catch (InputMismatchException e) {
      System.out.println("Please enter a valid degree value.");
} catch (Exception e) {
      System.out.println(e.getMessage());
}
```

In this code, the `try` block contains several statements that could throw exceptions:

- the temperature class's explicit-value constructor, as we programmed it above, can throw an exception of type `Exception` if the temperature values are not valid;
- `nextDouble()` can throw an exception of type `InputMismatchException` if the value it is reading is not a valid double number.

This code also includes two `catch` blocks, each handling its particular exception in its own way. Note that the ordering of these two `catch` blocks is important. We should handle more specific exception types before handling more general types. For example, because `InputMismatchException` is a special type, that is, it is below `Exception` in the exception hierarchy, we must catch it first, before catching the more general `Exception`; otherwise, Java treats the `InputMismatchException` as an `Exception` and generates an error indicating that control can never reach the trailing `InputMismatchException catch` block.

The try-catch pattern is as follows:

try-catch Pattern

```
try {
   Statements
} catch (ExceptionClass1 exception1) {
   ExceptionHandlerStatements1
...
} catch (ExceptionClassn exceptionn) {
   ExceptionHandlerStatementsn
} [finally {
   CleanupStatements
}]
```

- `Statements` is a block of statements each of which may throw exceptions;
- `ExceptionClass`$_i$ `exception`$_i$ is a parameter list specifying a class from the Exception hierarchy and an associated identifier;
- `ExceptionHandlerStatements`$_i$ are statements that handle exceptions of type `ExceptionClass`$_i$; there can be as many catch blocks as required;
- `finally { CleanupStatements }` is a set of statements designed to be executed after the chosen catch block, regardless of which block that is. The finally clause is optional (as indicated by the "`[ ]`" given in the pattern) and is commonly used to cleanup file processing commands.

10.2.4. Revisiting the Example

Given the exception handling techniques discussed in this section, we can extend the temperature class defined in Chapter 9 with a more effective means of establishing and maintaining its invariant. The upgraded version of the explicit-value constructor is discussed in the previous section. The additional code required to upgrade the `Temperature` class is shown in this section.

The upgraded version of the change() method is as follows:

```
public void change(double amount) throws Exception {
      double newDegrees = myDegrees + amount;
      if (isValidTemperature(newDegrees, myScale)) {
            myDegrees = newDegrees;
      } else {
            throw new Exception("Invalid Temperature change: "
                                          + newDegrees + " "
                                          + myScale);
      }
}
```

```
public void setScale(char scale) throws Exception {
```

```java
        if (Character.toUpperCase(scale) == 'C') {
            convertToCelsius();
        } else if (Character.toUpperCase(scale) == 'F') {
            convertToFahrenheit();
        } else if (Character.toUpperCase(scale) == 'K') {
            convertToKelvin();
        } else {
            throw new Exception("Invalid Temperature scale: " + scale);
        }
    }
```

These two mutators must be as careful with the invariant as the explicit-value constructor. The `change()` mutator throws an exception for invalid temperatures just as the constructor does; the `setScale()` mutator throws an exception if the scale is invalid.

The `copy()` method must specify the `throws Exception` clause, as shown here:

```java
    public Temperature copy() throws Exception {
        return new Temperature(myDegrees, myScale);
    }
```

Because `copy()` doesn't do anything that could invalidate the invariant, it does not, itself, create new exceptions to be thrown, but it does pass along errors thrown by the explicit-value constructor it calls. In this case, Java requires that `copy()` also specify the `throws Exception` clause, which indicates that it or something it calls could throw an exception. This same thing is true for all the comparison methods, all of which call this new version of the `copy()` method.

This upgraded version of the temperature class guards all the points at which the data items are either initialized or modified. This is one of the reasons that we've maintained the practice of declaring local data items to be private because this forces other code to use our carefully designed constructors and mutators rather than allowing them to directly set data items in an unrestrained manner.

10.3. Functional Testing

If part of our goal as programmers is to produce code that is *correct*, i.e., that it does what it is supposed to do, then we should consider the question of how to test that we have achieved this goal. In some design contexts, it may be difficult to specify what the correct behavior of a program should be. For example, in Chapter 2 it would have been difficult to specify a "correct" poster layout; so many layouts could work and it would be difficult to distinguish them in terms of correctness. In other contexts, however, our notion of correctness can and should be carefully specified and tested. For example, a basic tool like our `Temperature` class could potentially be used in safety-critical applications where correctness is absolutely critical such as life-support monitors or flight control systems. To help ensure that our `Temperature` class is up to this level, our testing should follow these principles:

- Testing should be systematic, addressing the full range of scenarios in which our code might find itself;
- Testing should be done early and often – the sooner we find an error, the more quickly and efficiently it can be corrected;

To achieve these goals, we should make testing part of our development process, and, as far as possible, write our code in a way that facilitates systematic and frequent testing. This section discusses Java tools that support these goals.

10.3.1. Specifying Test Cases

Good testing is based on good *test cases*. The analysis phase of any software development project should specify a set of test cases that is sufficient to demonstrate the correctness of any implementation of our analysis and design. These test cases should be built systematically and should specify concrete instances of a system input along with the correct system output. For example, one good test case for our temperature class is that it should convert 0 Celsius to 32 Fahrenheit. It's concrete and we know the input and the correct result.

It is clear that we need to run more than one test case before we are confident that our program is correct. Our program should handle a variety of conversions, in different directions, with positive and negative values, and it should also handle the exception conditions discussed in the previous section. It's also clear that we can't generally test all possible cases. Even in the temperature conversion case, we wouldn't have time to test that all possible real values for Celsius are converted properly to corresponding values in Fahrenheit and Kelvin and vice-versa.

Fortunately, it is sufficient to specify a subset of all the possible test cases that are representative of the behavior required of our program. This set should include a variety of test cases that cover all the desired functionality of the system; in our example we'd want to test temperature object construction and scale conversions in all possible directions. This set should also include cases that pay particular attention to extreme conditions, often known as the *boundary conditions*; in our example, we'd want to test cases at or around absolute zero, cases in which the scale settings are close but not correct, and potentially very large temperature values. For the temperature class, we might want to specify the following list of test cases:

- Freezing and boiling points:
    - 0.0° C = 32.0° F = 273.15° K (Let this be short-hand for the two individual test cases: 0C should convert to 32F and 32F should convert to 273.15K.)
    - 373.15° K = 212.0° F =100.0° C
- Extreme temperatures:
    - -273.15° C = -459.67° F = 0.0° K
    - 1799540.33° F = 999726.85° C =1000000.0° K
- Exception conditions
    - -1° K results in thrown error
    - -460° F results in thrown error
    - -274.0° C results in thrown error

  While this is not an exhaustive list, we would be justified in being reasonably confident in the correctness of our temperature class if it handled all these cases correctly. [2]

[2] We assume here that the range of the `double` type is sufficient for all possible applications; further treatment of the rather extreme cases that raise this issue is beyond the scope of this text.

As previously mentioned, it is a good idea to specify the test cases during the analysis phase. You don't need to know the design of the system to specify these test cases, and it might even be dangerous to know the design or implementation given that you might be tempted to create test cases that you know will work. It is also a good idea to have people other than the programmers specify the test cases, just to make sure that the test cases don't manifest the same misconceptions that the programmers might have about the requirements.

Specifying test cases such as these is the first step in testing the functionality of the application, commonly called *functional testing*. Note that we've focused here on testing the function of the temperature class constructors and mutators rather than on the converter's user interface. As we'll see in a later section, testing the usability of a user interface requires a different sort of testing, commonly called *usability testing*.

10.3.2. Automating Unit Testing

Once specified, there are a number of ways that we might run the set of tests, sometimes called a *test suite*. If we had an interactive user interface of some sort, we could run them by hand. For example, we could fire up the temperature console program and go through each test case manually as we did in the previous section for our first test case (i.e., 0.0°C = 32.0° F = 273.15° K), showing that if the user enters 0.0° C, the Fahrenheit and Kelvin conversions are correct. In the case of exception conditions, we'd have to verify that the application generated appropriate exceptions in the unhappy scenarios and did not crash or print any un-handled exceptions in the happy scenarios. Then we could be reasonably certain that the application is operating correctly.

While this would be possible, it certainly would be a tedious task, particularly when we realize that each time we modify the Temperature class in any way, we'd need to not only specify and run tests of the new or modified features, but also re-run all the previous tests just to make sure that nothing was inadvertently broken in the process of making the modifications. This process of re-running previous tests is called *regression testing*, that is, we are testing that the system did not "regress" when we are trying to make progress. When we develop code iteratively, as we've been doing in this text, we'd need to do regression testing early and often.

Another approach to running tests is to automate them. One particular framework that has been influential in computing is JUnit, a framework for *unit testing*.[3] Unit testing focuses on individual "units" in a program rather than on the whole program itself. In our temperature example, unit tests would focus on the temperature class itself, ignoring the user interface and any other unrelated "units" entirely.

The basic idea of JUnit testing is that we can write simple test cases that "exercise" aspects of our code and alert us to any problems. JUnit uses predefined assertion methods for this purpose. For example, if we wanted to satisfy ourselves that Java is implementing addition properly, we could write a set of JUnit assertions such as:

```
assertEquals(1+1, 2);
```

---

[3] Complete information on JUnit can be found at http://www.junit.org/.

When executed as part of a full JUnit test, this individual assertion command evaluates its two arguments and reports an error if they are not equal. This assertion runs without reporting an error because 1+1 is, indeed, equal to 2. We could then add more assertions to test the range of addition functionality.

We can also run the following overloaded versions of the assertion command:

```
assertTrue(true || false);
assertEquals(3.14159, Math.PI, 1e-5);
```

The first call evaluates its argument and reports an error if the result is not true; this example does not report an error because the Boolean expression (**`true`** OR **`false)`** is, indeed, **`true`**. The second call is similar to the `assertEquals()` shown above except that it expects double values for its arguments; because double values may be tested with differing levels of precision, JUnit requires that this test supply a third argument specifying the required precision; this example does not report an error because the Java Math library's pre-defined value for PI is, indeed, equal to 3.14159 to the precision of 5 decimal places (`1e-5` $= 10^{-5}$).

For our temperature class example, we might want to write testing code that looks like this:

```
Temperature temperature = new Temperature(0.0, 'C');
temperature.setScale('F');
assertEquals(32.0, temperature.getDegrees(), 1e-5);
temperature.setScale('K');
assertEquals(273.15, temperature.getDegrees(), 1e-5);
```

This code segment constructs a temperature object, `temp`, initializing it to 0° C. It then resets that temperature object's scale to Fahrenheit and asserts that its degrees value is 32.0 as required. The code then resets the scale to Kelvin and asserts that its degrees value is 273.15 as required. If this code runs through without any errors, we know that our Temperature class handles the first test case properly.

JUnit provides a framework for specifying suites of test cases such as this, running those suites automatically and reporting any errors that it finds. For example, we could write a JUnit test class that implements this given test case as follows:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class TemperatureTest {

      @Test
      public void equalityTests() {
            Temperature t;
            try {
                  t = new Temperature(0.0, 'C');
                  t.setScale('F');
                  assertEquals(32.0, t.getDegrees(), 1e-5);
                  t.setScale('K');
                  assertEquals(273.15, t.getDegrees(), 1e-5);
            } catch (Exception e) {
                  fail("inappropriate exception raised...");
            }
      }
}
```

This code implements our first test case (0.0°C = 32.0° F = 273.15° K) as a class, `TemperatureTest`, containing a method, `equalityTests()`. However, this is no ordinary class. We can't run `TemperatureTest` as a Java application because it has no `main()` method. Instead, we run this class as a JUnit test class. Your Java IDE will likely provide an execution option for this purpose.

Furthermore, `equalityTests()` is no ordinary method. It is marked with the `@Test` annotation, which tells Java that this is a unit test method that should be run as a JUnit test case.[4] JUnit runs all the test methods and reports any failed assertions that it finds.

This particular test method repeats the sample test code described above, but adds the try-catch block that Java requires we add in order to handle the exception that the `setScale()` method might raise for invalid temperatures. If our Temperature class is coded properly, it should pass all the assertions and not throw any exceptions.

However, if our Temperature class is coded improperly, we would like JUnit to report this to us so that we can fix our code. JUnit does this as follows. Java runs the `try{}` block through one statement at a time. As discussed above, JUnit will report an error if any of the assertion commands fail. As discussed in the previous section on exception handling, Java transfers control from the `try{}` block to the `catch{}` block if any of the `try{}` block code raises an exception. In the case of this example, we don't want any exceptions to be raised because 0.0C and its conversions are perfectly valid temperature objects. Thus, our Temperature code would be wrong to raise any exception. For this reason, the `catch{}` block calls JUnit's `fail()` method, which tells JUnit to report that the test has failed to execute properly, in this case because the Temperature class raised an exception inappropriately.

The patterns for using some of JUnit's more useful methods are as follows:[5]

Method	Description
`assertTrue(`*booleanExpression*`)`	Tests to see if *booleanExpression* returns true.
`assertEquals(`*expectedValue*`, `*actualValue*`)`	Tests to see if *expectedValue* is equal to *actualValue*..
`assertEquals(`*expectedDoubleValue*`, `*actualDoubleValue*`, `*tolerance*`)`	A special version of assertEquals designed for testing double values – it tests to see if *expectedDoubleValue* is equal to *actualDoubleValue* plus-or-minus the given tolerance.
`fail(`*errorMessage*`)`	Fail the unit test and report the given *errorMessage*.

[4] Java annotations such as this are new feature in Java 5 that we will not discuss in detail, see http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html for more details.

[5] JUnit supports a variety of other test facilities, including set-up and tear-down methods; we don't discuss them here. Tutorials and more details can be found at http://junit.sourceforge.net/.

The test cases that we previously identified include some scenarios in which we want our code to throw an exception. For example, we want the `Temperature` class to throw an exception if the user tries to construct a new Temperature object with the value -1.0° Kelvin. Testing this behavior, though a bit trickier than testing simple return values, can be done as shown in the following code.

```
try {
      Temperature t = new Temperature(-1.0, 'K');
      fail("should have thrown an error...");
} catch (IllegalArgumentException e) {
      // Do nothing here; this code should throw an exception!
}
```

This test method uses a try-catch block, as discussed in the previous section, but it uses it a bit differently.[6] It tries to construct a temperature with the value -1.0° Kelvin, which we know should throw an exception because of the way we programmed the explicit-value constructor. If the Temperature class throws this exception as expected then control will jump directly to the catch block, and in this case the catch block has no statements, so JUnit concludes that everything worked properly. This will pass the unit test. If, on the other hand, the Temperature class does not throw an exception as we hope, then control will stay in the `try{}` block and pass directly to the `fail()` statement, which tells JUnit that this test has failed. Thus, this JUnit code passes the test if the constructor throws an exception and fails the test if the constructor does not throw an exception.

To run a set of unit tests, we must ask our development environment to run the test class as a JUnit test case. The result is a summary of the success and/or failure of the test cases specified in the class. A fully successful test run is generally indicated by displaying the well-known "green bar"; a "red bar" indicates that at least one of the tests failed.

By default, JUnit tests pass unless there is a failed assertion or an explicit call to `fail()`. As a consequence of this, and empty test, such as the following test, passes.

```
@Test
public void emptyTest() {
      // Empty tests pass by default.
}
```

A set of unit tests should be run whenever changes are made to the implementation. This helps to verify that the changes did not affect the correctness of the code. While it may require additional work to specify our test suite in this manner, it will almost certainly pay off in the long run given how much easier regression testing will be. One particularly vexing problem in software development is the situation in which changes to one part of a system cause unexpected errors in another part of that system. Unit testing the entire system whenever changes are made to any part of the system helps to smoke out these sorts of problems.

Another benefit of automated testing accrues when we would like to modify our code without changing its functional behavior. This process is known as *refactoring*. For example, we might want to make the

---

[6] JUnit 4 provides an annotation-based alternative to this version 3 practice; this approach uses this annotation: `@Test(expected=Exception.class)`, and no try-catch in the test method itself. This text adopts the practice of using try-catch. For more information, see http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

implementation more efficient or more understandable, without changing its functionality. In this case, refactoring should not affect the code's ability to pass the automated unit tests, so passing the tests after refactoring gives us confidence that our modifications did not affect the functionality. We'll see an example of this later in this chapter.

10.3.3. Revisiting the Example

We can use the techniques shown in this section to automate a set of unit tests for the `Temperature` class as follows.

```
@Test
public void defaultConstructorTests() {
	// Verify that the default temperature is 0.0 Celsius.
	Temperature t1 = new Temperature();
	assertEquals(0.0, t1.getDegrees(), 1e-5);
	assertEquals('C', t1.getScale());
}

@Test
public void explicitValueConstructorTests() {
	// Verify that other temperatures can be constructed and
accessed.
	try {
		Temperature t1 = new Temperature(0, 'K');
		assertEquals(0.0, t1.getDegrees(), 1e-5);
		assertEquals('K', t1.getScale());

		t1 = new Temperature(98.6, 'f');
		assertEquals(98.6, t1.getDegrees(), 1e-5);
		assertEquals('F', t1.getScale());
	} catch (Exception e) {
		fail("inappropriate exception raised...");
	}
}

@Test
public void badExplicitValueConstructorTests() {
	// Try some invalid temperatures.
	badExplicitValueUtility(10.0, 'X');
	badExplicitValueUtility(-0.01, 'K');
}

// This utility tries to construct invalid temperatures using try-catch
private void badExplicitValueUtility(double degrees, char scale) {
	Temperature t;
	try {
		t = new Temperature(degrees, scale);
		fail("Exception should have been thrown on: " + degrees +
								" " + scale);
	} catch (Exception e) {
		// Do nothing; this code should throw an exception.
	}
}
```

These unit tests test the default and explicit-value constructors. Note that they test both happy and unhappy scenarios. For example, the simplest happy scenario is the default constructor, which should

construct a new temperature representing 0.0° Celsius. Other happy scenarios tested include 0.0° Kelvin and 98.6° Fahrenheit. Unhappy scenarios may occur as well, so they must also be tested. The `badExplicitValueConstructorTest()` unit test tries to create two invalid temperatures: 10.0° X (an invalid scale) and -1.0° Kelvin (below absolute zero). Both of these tests are handled by `badExplicitValueUtility()`, and utility method that handles the details of running code that is supposed to throw errors.

```
@Test
public void badSetScaleTest() {
        Temperature t = new Temperature();
        try {
                t.setScale(' ');
                fail("Exception should have been thrown...");
        } catch (Exception e) {
                // Do nothing; it should throw this exception.
        }
}
```

This test uses code that is similar to that used in `badExplicitValueUtility()`, but it is designed to test the `setScale()` mutator instead. Clearly the space character is not a valid scale designator and this call should, therefore, throw an error.

```
@Test
public void equalityTests() {
        Temperature t;
        try {
                // Test freezing temperature equivalents.
                t = new Temperature(0.0, 'C');
                t.setScale('F');
                assertEquals(32.0, t.getDegrees(), 1e-5);
                t.setScale('K');
                assertEquals(273.15, t.getDegrees(), 1e-5);

                // Test boiling temperature equivalents.
                t = new Temperature(373.15, 'K');
                t.setScale('F');
                assertEquals(212.0, t.getDegrees(), 1e-5);
                t.setScale('C');
                assertEquals(100.0, t.getDegrees(), 1e-5);

                // Test absolute zero temperature equivalents.
                t = new Temperature(-459.67, 'F');
                t.setScale('K');
                assertEquals(0.0, t.getDegrees(), 1e-5);
                t.setScale('C');
                assertEquals(-273.15, t.getDegrees(), 1e-5);

                // Test large temperature equivalents.
                t = new Temperature(1799540.33, 'F');
                t.setScale('C');
                assertEquals(999726.85, t.getDegrees(), 1e-5);
                t.setScale('K');
                assertEquals(1000000.0, t.getDegrees(), 1e-5);
        } catch (Exception e) {
                fail("inappropriate exception raised...");
        }
}
```

```
@Test
public void comparatorsTest() {
      Temperature t1;
      try {
            t1 = new Temperature(5.0, 'K');
            assertTrue(t1.equals(t1));
            assertFalse(t1.lessThan(t1));
            assertTrue(t1.lessThanOrEqualTo(t1));
            assertFalse(t1.greaterThan(t1));
            assertTrue(t1.greaterThanOrEqualTo(t1));

            Temperature t2 = new Temperature(25.0, 'C');
            assertFalse(t1.equals(t2));
            assertTrue(t1.lessThan(t2));
            assertFalse(t1.greaterThan(t2));
            assertTrue(t1.lessThanOrEqualTo(t2));
            assertFalse(t1.greaterThanOrEqualTo(t2));

            Temperature t3 = new Temperature(5.0, 'F');
            assertFalse(t1.equals(t3));
            assertTrue(t1.lessThan(t3));
            assertTrue(t1.lessThanOrEqualTo(t3));
            assertFalse(t1.greaterThan(t2));
            assertFalse(t1.greaterThanOrEqualTo(t3));

      } catch (Exception e) {
            fail("inappropriate exception raised...");
      }
}

@Test
public void changeTests() {
      try {
            Temperature t1 = new Temperature(50.0, 'F');
            t1.change(10.0);
            assertEquals(60.0, t1.getDegrees(), 0e-5);
            t1.change(-20.0);
            assertEquals(40.0, t1.getDegrees(), 0e-5);
      } catch (Exception e) {
            fail("inappropriate exception raised...");
      }
}

public void testCopy() {
      Temperature t1;
      try {
            t1 = new Temperature(10000.5, 'C');
            Temperature t2 = t1.copy();
            assertTrue(t1.equals(t2));
            assertNotSame(t1, t2);
      } catch (Exception e) {
            fail("inappropriate exception raised...");
      }
}
```

These tests all exercise happy scenarios, but they must, nevertheless, wrap their code in try-catch blocks because the methods they call could potentially throw errors. The compiler requires that the try-catch blocks be there even if we happen to believe that the execution should not throw errors. The unit tests help to verify that this is, in fact, the case.

10.4. User Testing

The previous section discussed approaches for testing program correctness. Correctness is a necessary but not sufficient criterion for robust, high-quality software. The *usability* of the system, that is, the measure of how easy it is to learn for an end user, is also important. A correct program that is not usable by its intended users has little more value than an incorrect program. It is, therefore, important to test both correctness and usability.

10.4.1. Testing Usability

Usability cannot easily be tested using JUnit because JUnit tests don't have easy access to the interaction between the user and the system. This is true both of the console-based applications we have seen so far as well as the graphical user interfaces we'll see in later chapters. More importantly, JUnit tests cannot anticipate the sorts of questions that users are likely to have when encountering the GUI for the first time. This kind of testing can only be done effectively by asking representative users to actually use the interface.

Note that while developers should certainly try out the GUI prototypes to make sure that they run as expected, we cannot expect a developer to find the sort of usability problems common to today's interactive systems. Developers know too much about GUIs and far too much about the system being developed to ever have a good idea of what it would be like to encounter the interface for the first time. Thus, we must perform our usability testing with the help of more representative users who have not seen the program before.

The design and execution of systematic usability tests is based on the principles and concepts of experimental psychology and is, thus, beyond the scope of this text. As a simple rule of thumb, however, it would be a good idea to ask four to six users to try out your system. Let these test users be representative of your real users. If your system is a math quiz for high school students, then find some high school math students; if it's a layout design tool for art students, then find some art students. You can expect these test users to see your system as anyone else would see it, and that is potentially valuable to you as a system developer.

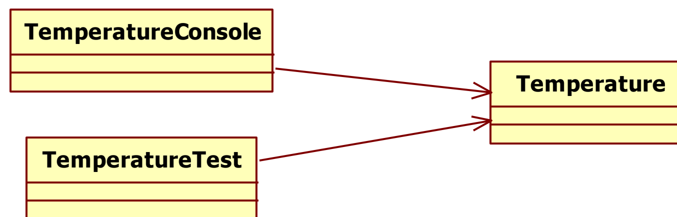
Our working definition of software quality throughout this text has been that our applications should be correct, efficient, understandable to other programmers and usable to its prospective users. It's not good enough to produce applications that satisfy only the first three criteria. We want more; our potential users deserve more; they deserve an application that doesn't just work, but is a joy to use.

10.4.2. Separating the Model from the View

The distinction between functional unit testing and usability testing has implications for the class design of our systems. To support these two types of tests, we would like, as much as possible, to design our class structure to separate what are commonly called the *model* and the *view*:

- **Model** – The model comprises the core functionality of the class itself. In the temperature example, the model is the implementation of the temperature class itself, with its implementations of the attributes and behaviors of the temperature objects.
- **View** – The view comprises the use, or uses, of the class in context. In the temperature example, the views include the use of the temperature class in the `TemperatureConsole` application and in the `TemperatureTest` test cases.

The class structure that we desire can be depicted as follows, where the views are on the left and the model on the right.



Separating these concerns is generally a good object-oriented design practice. The temperature class should only concern itself with temperature-function-related elements. It shouldn't care how it is used. That's the job of the view classes, which may use temperature objects in a console-based application, as `TemperatureConsole` does, or in unit tests, as `TemperatureTest` does. This practice generally makes it easier to design and implement the classes.

One additional benefit of this design is that it allows us to use automated unit testing on all the functionality provided by the model. As discussed above, we can't easily unit test user interfaces, so encapsulating all non-interface-related functions in the model allows us to unit test those functions easily. We then only have to do manual user testing on the user interface.

10.4.3. Revisiting the example

In this section we consider the usability of a temperature conversion console application. The user experience envisaged in Section 10.1 was rather simple, allowing the user to enter a temperature and a target scale, as follows:

```
Enter a temperature: 40 f
Convert to: c
Result: 4.444444444444445 C
```

The system output is shown here in black and the user input is shown in green. This is not the most friendly of interfaces given that users are unlikely to know how to enter a temperature value. It also turns out that the current implementation doesn't handle unhappy scenarios properly, as seen here:

```
Enter a temperature: forty degrees fahrenheit
Exception in thread "main" java.util.InputMismatchException
	at java.util.Scanner.throwFor(Scanner.java:909)
	at java.util.Scanner.next(Scanner.java:1530)
	at java.util.Scanner.nextDouble(Scanner.java:2456)
	at
c10quality.text.temperature.TemperatureConverterConsole.main(TemperatureConver
terConsole.java:21)
```

Here, the user had entered their temperature using an unsupported format (i.e., "forty" rather than "40") and has received an unhandled exception in response. This is not a usable interface in its current form. Asking real users to try the system out in a usability test would almost certainly identify these problems.

While there is only so much we can do to make a console application like this one more usable, we can certainly give the users some idea of what the input should look like and handle the exceptions more gracefully. Here is a console application that does that:

```
/**
 * TemperatureConverterConsole implements a simple text-based interface for the
 * temperature converter application.
 *
 * @author kvlinden
 * @version Fall, 2009
 */
public class TemperatureConverterConsole {

	public static void main(String[] args) {
		System.out.print("Enter a temperature (e.g., 98.6 F): ");
		// System.out.print("Enter a temperature: ");

		Scanner keyboard = new Scanner(System.in);
		try {
			double degrees = keyboard.nextDouble();
			char scale = keyboard.next().charAt(0);
			Temperature temperature = new Temperature(degrees,
								scale);
			System.out.print("Convert to: ");
			temperature.setScale(keyboard.next().charAt(0));
			System.out.println("Result: " + temperature);
		} catch (InputMismatchException e) {
			System.out.println("Please use a valid format.");
		} catch (Exception e) {
			System.out.println(e.getMessage());
		}
	}

}
```

This code gives the user an example of the temperature input and helps them through the following happy scenario:

```
Enter a temperature (e.g., 98.6 F): 40 f
Convert to: c
Result: 4.444444444444445 C
```

This code also handles some of the unhappy scenarios a bit more gracefully:

```
Enter a temperature (e.g., 98.6 F): forty degrees fahrenheit
Please use a valid format.
```

and

```
Enter a temperature (e.g., 98.6 F): -1.0 K
Invalid Temperature: -1.0 K
```

We'll use graphical techniques to develop more useable interfaces when we get to Chapter 12.

10.5. Enumerated Types

As a final enhancement to our temperature program, we consider the understandability of the temperature class. Where usability measures how easy an application is to learn for its end users, *understandability* measures how easy the units of the system are to learn for other programmers. As an example, consider the representation of the scale of a temperature object. Currently, we represent the scale using a character value (e.g., 'F', 'C' or 'K') and we must admit that this is somewhat clunky. Programmers using our temperature class in their applications are required to remember and abide by these codes.

The Java compiler helps a bit. It only accepts characters for the explicit-value constructor and the `setScale()` mutator, and thus an invocation like `new Temperature(98.6, "Fahrenheit")` would be flagged as a compiler error. `"Fahrenheit"` is not a character.

However, there are other errors that the compiler doesn't catch. For example, the invocation `new Temperature(98.6, 'X')` is syntactically correct. Currently, the temperature class throws exceptions in these cases, but it would be nice if there were some way to specify that there are exactly three scales and provide some convenient way of enumerating them. This would not improve the correctness of the system, but it would help to improve its understandability to other programmers and simplify the implementation.

Fortunately, Java provides a mechanism that allows us to enumerate a finite list of field values and treat that list as a new type. This mechanism is called *enumerated types*.

10.5.1. Working with Enumerated Types

We can define an enumerated type as shown here. This example implements a finite list of currency types that we could profitably use in a currency conversion application.

```
enum Currency {US$, CANADIAN$, AUSTRALIAN$};
```

This statement declares a new type called `Currency` whose legal values are restricted to only those fields listed in the declaration: `US$`, `CANADIAN$`, `AUSTRALIAN$`. The fields can be any legal Java identifier (recall that identifiers can contain the `$` character). This declaration allows us to work with variables of this new type.

```
Currency aCurrency = Currency.US$;
```

Here, Java expects us to use the qualified version of the `Currency` value (`Currency.US$`) rather than just the unqualified version (`US$`). Because `aCurrency` is declared to be of type `Currency`, we cannot use any other values than those listed in the `Currency` definition.

The following code segment prints out a message based on the value of `aCurrency`.[7]

```
if (aCurrency == Currency.US$) {
      System.out.println("it's US dollars...");
} else if (aCurrency == Currency.CANADIAN$) {
      System.out.println("it's Canadian dollars...");
} else {
      System.out.println("it's Australian dollars...");
}
```

Note that in this code, we don't need to provide fourth option because we know that the compiler will prevent the programmer from ever using any `Currency` value not listed in the `Currency` declaration.

The key advantage of using enumerated types is that it allows us to tell Java about this new type and engage the Java compiler as a partner in ensuring the robustness of our application. We no longer have to explicitly program exception handling code to handle the situations in which programmers use the wrong terms; the Java compiler now does this for us. For example, if we forget how to spell "Australian", we can rely on the compiler to enforce the use of the correct term (`Currency.AUSTRALIAN$`). In addition, using an incorrect name leads to a compiler error that we can fix quickly and easily rather than to an exception thrown at runtime that takes longer to discover and to fix.

10.5.2. Revisiting the Example

In this iteration, we refactor the Temperature class to use an enumerated type to represent the legal list of temperature scales.

```
public class Temperature {

      public static enum Scale {
            FAHRENHEIT, CELSIUS, KELVIN
      }

      private double myDegrees;
      private Scale myScale;

      // The other declarations are unchanged.

      public Temperature() {
            myDegrees = 0;
            myScale = Scale.CELSIUS;
      }
```

[7] Because the new enumerated type is integer-compatible, we can more effectively use a `switch` statement rather than a multi-branch if statement to distinguish its values.

```
    public Temperature(double degrees, Scale scale) throws Exception {
        if (isValidTemperature(degrees, scale)) {
            myDegrees = degrees;
            myScale = scale;
        } else {
            throw new Exception("Invalid Temperature: " + degrees + " "
                                                + scale);
        }
    }

    // The getDegrees() accessor is unchanged.

    public Scale getScale() {
        return myScale;
    }

    // The comparison and copy methods unchanged.


    public void setScale(Scale scale) throws Exception {
        if (scale == Scale.CELSIUS) {
            convertToCelsius();
        } else if (scale == Scale.FAHRENHEIT) {
            convertToFahrenheit();
        } else {
            convertToKelvin();
        }
    }

    private void convertToCelsius() {
        if (myScale == Scale.FAHRENHEIT) {
            myDegrees = 5.0 / 9.0 * (myDegrees - 32.0);
        } else if (myScale == Scale.KELVIN) {
            myDegrees = myDegrees - 273.15;
        }
        myScale = Scale.CELSIUS;
    }
    // The other conversion methods are unchanged.
    public static boolean isValidTemperature(double degrees, Scale scale) {
        if (scale == Scale.CELSIUS) {
            return degrees >= ABSOLUTE_ZERO_CELSIUS;
        } else if (scale == Scale.FAHRENHEIT) {
            return degrees >= ABSOLUTE_ZERO_FAHRENHEIT;
        } else {
            return degrees >= ABSOLUTE_ZERO_KELVIN;
        }
    }

    public String toString() {
        return myDegrees + " " + myScale;
    }
}
```

This new version of the Temperature class declares a new type, `Scale`, that enumerates the three supported temperature scales. We choose to declare this type as public and static so that it is accessible to

other classes and is shared by all temperature objects respectively. We've also replaced the char type of all those parameters and variables that represented scales with `Scale`.

This new Temperature class requires changes to the controller class as well.

```
import static c10quality.text.temperatureEnumerations.Temperature.Scale;
import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * TemperatureConverterConsole implements a simple text-based interface
 * for the temperature converter application.
 *
 * @author kvlinden
 * @version Fall, 2012
 */
public class TemperatureConverterConsole {

    public static void main(String[] args) {
        System.out.print("Enter a temperature (e.g., 98.6 F): ");

        Scanner scanner = new Scanner(System.in);
        try {
            double degrees = scanner.nextDouble();
            Temperature.Scale scale = readScale(scanner);
            Temperature temperature = new Temperature(degrees,
                                                      scale);
            System.out.print("Convert to: ");
            temperature.setScale(readScale(scanner));
            System.out.println("Result: " + temperature);
        } catch (InputMismatchException e) {
            System.out.println("Please use a valid format.");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    // Read char scale from the keyboard and convert to
    // Temperature.Scale.
    public static Scale readScale(Scanner scanner) throws Exception {
        char scaleInput = scanner.next().charAt(0);
        if (Character.toUpperCase(scaleInput) == 'C') {
            return Scale.CELSIUS;
        } else if (Character.toUpperCase(scaleInput) == 'F') {
            return Scale.FAHRENHEIT;
        } else if (Character.toUpperCase(scaleInput) == 'K') {
            return Scale.KELVIN;
        } else {
            throw new Exception("Invalid scale: " + scaleInput);
        }
    }
}
```

This controller must configure its temperature object using, in part, the enumerate scale names, so it imports those names (see the very first `import static` line) and uses them when it must read a new scale from the user. The process of converting from character input to `Temperature.Scale` is performed by the `readScale()` method. This method is a bit messy, but moving the input and

conversion to `Temperature.Scale` to the console interface greatly simplifies the Temperature code and makes it less dependent on the implementation of the user interface.

This modified version of the Temperature class should behave in the same manner as the original version. That is, the upgrades only should affect its efficiency and understandability, but not its correctness. Thus, the code should pass the automated unit tests without any modification to the test cases themselves. If it does, we have confidence that we've done the refactoring correctly (see the discussion of refactoring earlier in the chapter).

10.6. Example Revisited

Now that we have a Temperature class in which we have some confidence of correctness and understandability, we consider the application's usability. Having designed the system through several iterations, we have a good idea of how it works and would likely find it hard to imagine that anyone would have problems figuring out how to use it. Nevertheless, we should ask potential users to try the application out and closely observe how they fare with the features and user interface.

We will not run such a study for this text, but there are a number of things that we'd like to determine if we did, including:

- Will the users understand that they must enter numeric values (e.g., "40") rather than textual ones (e.g., "   40", with leading spaces, or "forty") for the degree magnitudes? We've provided a simple input example and have programmed the application to catch this error and respond with a simple text message, but it's unclear whether this will be enough to make the system's use seem easy to users.
- Will the users know that the three legal scales are coded by F, C and K? Given that the interface provides no information on this, it's likely that they'll see the Fahrenheit scale in the example and guess the Celsius scale, but what about the Kelvin scale?
- Will the users immediately understand and appreciate the decidedly textual mode of the system output, or would they respond more favorably to a graphical, animated form of output?

Because we haven't done such a study, we can't provide definitive answers to these questions. As programmers, we know this interface very well, from both the internal and external perspectives, and we are, therefore, unlikely to guess how someone else would experience our program. This fact about the nature of developers makes user testing critical to the development of usable applications.