“Bowling Scores” Case Study
by Michael Clancy © 2005

Background

The game of bowling is a world-wide sport. More than 95 million people bowl, in 90 countries spanning six continents*. It is played by rolling a ball down a bowling lane to knock down the pins at the end of the lane. The situation is diagrammed in Figure 1.

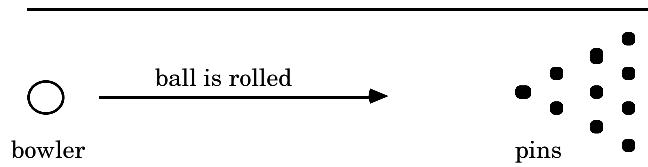


Figure 1
The bowling lane

A bowling game is divided into ten *frames*. The bowler is allowed up to two *balls* in each frame to knock down all ten pins. Knocking down all the pins with the first ball ends the frame, and is called a *strike*. Knocking down all the pins with the first two balls is called a *spare*. Knocking down fewer than ten pins with the first two balls is called a *miss*.

A bowler earns one point for each pin knocked down, plus bonuses for each strike and spare. A strike scores 10 points plus the total of the next two balls; a spare scores 10 points plus the score of the next ball. A strike in the tenth frame earns the bowler two extra balls; a spare in the tenth frame earns one extra ball. An example of how a game is scored is shown in Table 1.

Bowlers use a score sheet that’s organized to clearly display the score for each frame and each ball. By convention, they use an X to represent a strike and a / (slash) to represent a spare. Here’s how the game in Table 1 would be scored:

9 /	0 /	X	X	6 2	7 /	8 /	X	9 0	X X 8
10	30	56	74	82	100	120	139	148	176

In olden days, bowlers kept their own scores. Computers, however, have brought automatic scorekeeping to bowling. After each ball, sensors count and tally the number of pins knocked down.

*. Source: The International Bowling Museum web site, www.bowlingmuseum.com.

frame	balls	score for the frame	cumulative score
1	9+1 (spare)	10+0=10	10
2	0+10 (spare)	10+10=20	30
3	10 (strike)	10+10+6=26	56
4	10 (strike)	10+6+2=18	74
5	6+2	8	82
6	7+3 (spare)	10+8=18	100
7	8+2 (spare)	10+10=20	120
8	10 (strike)	10+9+0=19	139
9	9+0	9	148
10	10 (strike) +10+8 (bonuses)	10+10+8=28	176

Table 1: A sample bowling game

Problem

Design and implement a class that represents an *automatic bowling scorer* object. It will provide several methods:

- a constructor;
- an `int frameNumber` method that returns the number of the frame containing the ball about to be rolled;
- an `int scoreSoFar` method that returns the score in the game so far;
- a `boolean gameIsOver` method that returns true when the tenth frame has been scored and false otherwise, and which causes the next roll to start a new game;
- an `int [ ] roll` method that, given the number of pins knocked down by a roll of the ball, returns an array whose length is the number of frames completely scored and whose contents are the cumulative scores for those frames.

The arrays returned by `roll` in the game just described appear in Table 2.

You may assume that the argument to `roll` correctly represents the number of pins knocked down by the roll. That is, no error checking is necessary.

Preparation

The reader should have experience with defining objects, using methods, loops, and arrays, and testing programs with JUnit.

argument to roll	*return value*	*argument to* roll	*return value*
9	{ }	3	{10, 30, 56, 74, 82}
1	{ }	8	{10, 30, 56, 74, 82, 100}
0	{10}	2	{10, 30, 56, 74, 82, 100}
10	{10}	10	{10, 30, 56, 74, 82, 100, 120}
10	{10, 30}	9	{10, 30, 56, 74, 82, 100, 120}
10	{10, 30}	0	{10, 30, 56, 74, 82, 100, 120, 139, 148}
6	{10, 30, 56}	10	{10, 30, 56, 74, 82, 100, 120, 139, 148}
2	{10, 30, 56, 74, 82}	10	{10, 30, 56, 74, 82, 100, 120, 139, 148}
7	{10, 30, 56, 74, 82}	8	{10, 30, 56, 74, 82, 100, 120, 139, 148, 176}

Table 2: Results of calls to roll

Exercises

Analysis 1. What is the *minimum* number of balls a bowler can roll during a game?

Analysis 2. If a bowler rolls the minimal number of balls in a game, what is the smallest number of points he/she can score?

Analysis 3. What is the *maximum* number of balls a bowler can roll during a game?

Analysis 4. What is the maximum score a bowler can achieve during a game?

Analysis 5. Can all scores up to the maximum be bowled in a game?

Analysis 6. What score results from the following sequence of balls?

```
9 1 10 10 10 10 0 10 0 10 5 4 1 9 10 5 4
```

Analysis 7. Explain why the length of the array returned by `roll` increased by 2 for the eighth roll.

Analysis 8. List all conditions under which this can happen (i.e. the length of `roll`'s return increases by 2).

Analysis 9. Suppose `frameNumber` is called immediately after each call to `roll`. What are the values returned by `frameNumber` for the game in Table 1?

Reflection 10. What makes the scoring of a bowling game complicated?

Reflection 11. What scoring errors are inexperienced bowlers likely to make?

Design of a bowling Scorer class

How might a programmer approach this problem?

A good approach to designing a class is to *role-play* the corresponding object. That is, we put ourselves in its place, and act out what we would do for each method call. Occasionally we will find that we must remember information between method calls; this remembered information will be stored in class instance variables.

How does a human score a bowling game?

Here's how we might score the example game in the problem statement.

ball	*action*
9	We take note of the 9.
1	The roll completes a spare; we take note of that, but can't supply a score for the frame yet.
0	We note the 0 as the first ball in the frame 2. Moreover, we can now score frame 1.
10	We take note of a spare for frame 2, but can't score the frame yet.
10	We score frame 2 and note the strike in frame 3.
10	We take note of the strike for frame 4, but we still can't score frame 3.
6	We note the 6 as the first ball in frame 5 and score frame 3.
2	We score frames 4 and 5.
7	We note the 7 as the first ball in frame 6.
3	We note the spare in frame 6, but can't score it yet.
8	We note the 8 as the first ball in frame 7, and score frame 6.
2	We note the spare for frame 7.
10	We note the strike for frame 8, and score frame 7.
9	We note the 9 for frame 9, but can't score frame 8.
0	We score frames 8 and 9.
10	We indicate strike for frame 10.
10	We note 10 for the first bonus ball.
8	We score frame 10. The game is over.

Stop and predict → *What instance variables will be needed to keep track of the bowling score?*

What information must the bowling scorer keep track of?

A human bowling scorer might maintain various kinds of information between balls:

- whether or not a strike has been rolled one or two balls ago;
- whether or not a spare has been completed by the preceding ball;
- whether the ball being rolled is the first or second ball o f the frame;
- the number of pins knocked down by the first ball in the frame;
- the number of the frame containing the ball being rolled;
- the number of the earliest frame not yet scored;
- the collection of cumulative scores for frames already rolled;
- the score so far.

Stop and help → *Identify an instance in the scoring of the game from the problem statement that shows the need for each of the pieces of information just described.*

Stop and consider → *Is any of the information in the list above redundant? That is, can it be determined from the other information in the list? Explain why or why not.*

Stop and predict → *Devise an English or pseudocode algorithm for handling a ball roll.*

How should the scoring algorithm be organized?

A human bowler, asked to explain how to tally a ball value, will probably organize the explanation in terms of the "whether or not" information:

```
if the previous two balls were both strikes, then handle the ball one way;
otherwise, if the previous ball was a strike, then
    handle the ball another way;
etc.
```

Programmers refer to these "whether or not" conditions as *state*, or situational, information, and often use a *state diagram* to represent all the possible situations and paths between them. In a state diagram, the different states are represented by boxes, and the ways to get from state to state are represented by labeled arrows. (Analogously, players of computer adventure games create maps of the various "rooms" of the adventure and the ways to get from room to room.) Code designed from the state diagram will involve variables that represent the state information. It will include a section of code for each state that will modify the variables to represent a *transition* from one state to another.

What does the state diagram for bowling scoring look like?

To draw the state diagram for bowling scoring, we start by describing the situation at the start of the game:

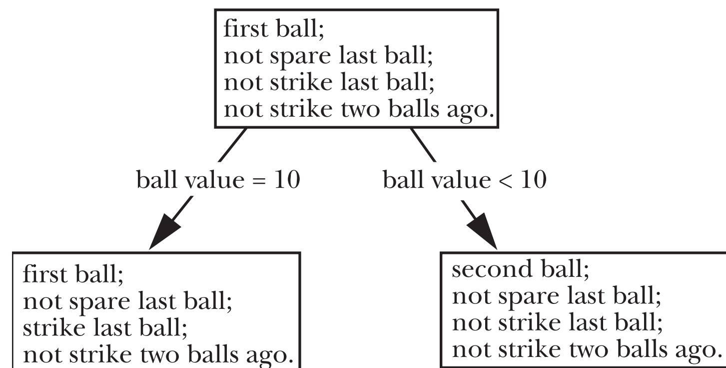
rolling the first ball of the frame;
no spare on the most recent ball;
no strike on either the most recent ball or two balls ago.

There are two possibilities for the input value. Either it's a 10, representing a strike, or it's a value from 0 to 9. That leads to two different situations:

(input value is 10)
rolling the first ball of the frame;
no spare on the most recent ball;
strike on the most recent ball;
no strike from two balls ago.

(input value < 10)
rolling the second ball of the frame;
no spare on the most recent ball;
no strike on the most recent ball;
no strike from two balls ago.

This results in the diagram below.



Each of these situations leads to two additional situations. For the strike, these are

(input = 10—two consecutive strikes)
rolling the first ball;
no spare on the most recent ball;
strike on the most recent ball;
strike two balls ago.

(input < 10)
rolling the second ball;
no spare on the most recent ball;
no strike on the most recent ball;
strike two balls ago.

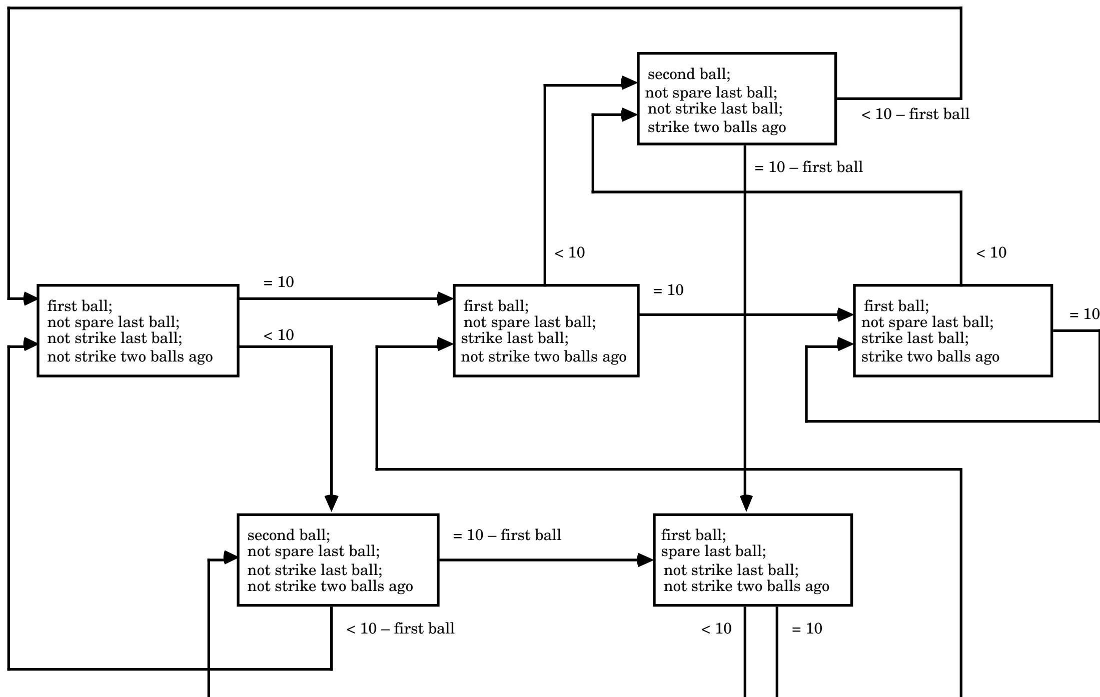
These are new situations and require two new boxes in the diagram. The other case, not a strike, leads to the following possibilities:

(first ball + second ball = 10—a spare)
rolling the first ball;
spare on the most recent ball;
no strike on the most recent ball;
no strike two balls ago.

(first ball + second ball < 10)
rolling the first ball;
no spare on the most recent ball;
no strike on the most recent ball;
no strike two balls ago.

The first possibility is a new situation. The second, however, is just the same as the start situation. In the diagram, an arrow is drawn back to the start state.

As it turns out (making a long story short), the situations just described are the only ones that will occur. The complete state diagram, slightly reoriented to be less messy, appears below.



Stop and help → *Trace through the sequence of states that results from the first few balls of the example game in the problem statement.*

A good way to check that the six states in the diagram are the only ones possible is to make a table, with a row for each possibility. In this case, the possibilities are first ball or second ball, spare or no spare on the last ball, strike or no strike on the last ball and strike or no strike two balls ago. Here's the table:

which ball in frame?	*spare last ball?*	*strike last ball?*	*strike two balls ago?*	*possible?*
1	yes	yes	yes	no
1	yes	yes	no	no
1	yes	no	yes	no
1	yes	no	no	yes
1	no	yes	yes	yes
1	no	yes	no	yes
1	no	no	yes	no
1	no	no	no	yes
2	yes	yes	yes	no
2	yes	yes	no	no
2	yes	no	yes	no
2	yes	no	no	no
2	no	yes	yes	no
2	no	yes	no	no
2	no	no	yes	yes
2	no	no	no	yes

Stop and consider →

Describe how the rows in the table are ordered. For example, explain why the row containing (1, yes, no, yes, no) *precedes the row containing* (1, yes, no, no, yes).

Stop and help →

Show, with as few arguments as possible, that the missing entries in the table are all "no". For instance, if there were a spare on the last ball, there can't have been a strike on the most recent ball, so four of the rows of the table have to be "no".

What organization does the state diagram suggest for the roll **method?**

The state diagram suggests an algorithm for the roll method:

if the game is in state "first ball, no spare, no strike last
    ball, no strike two balls ago", then
        update variables appropriately according to the ball
            value;
otherwise, if the game is in state "first ball, no spare, strike
    last ball, no strike two balls ago", then
        update variables appropriately according to the ball
            value;
otherwise ...

What instance variables are appropriate for the `Scorer` class?

It's time now to start designing in detail. Recall the list of information that the scorer object needs to maintain between calls:

- whether or not a strike has been rolled one or two balls ago;
- whether or not a spare has been completed by the preceding ball;
- whether the ball being rolled is the first or second ball o f the frame;
- the number of pins knocked down by the first ball in the frame;
- the number of the frame containing the ball being rolled;
- the number of the earliest frame not yet scored;
- the collection of cumulative scores for frames already rolled;
- the score so far.

We choose names for these quantities that suggest their purpose. The frame numbers can be `int` variables that take on values between 1 and 10; we'll name them `rollingFrame` and `firstUnscoredFrame`. Similarly, we'll store the value of the first roll in each frame in an `int` variable named `firstBallInFrame`, and the score so far in an `int` variable named `scoreSoFar`. The collection of cumulative scores should use an array; we'll call it `frameScores`, suggesting the assignment of one score per frame.

How should the state of the bowling game be represented?

The next step is to decide how to represent the state. This will allow further decomposition of the "if game is in state …" tests along with the "update variables appropriately" steps.

One representation of the state would use four variables: the ball number, plus indications of whether the last ball was a spare, the last ball was a strike, or two balls ago there was a strike. The "if game is in state …" tests would then merely be tests of these variables.

A better way is to represent the state as a *single* variable. In general, programs that let a single variable represent more than one different piece of information are confusing and difficult to debug. In this solution, however, the state variable is representing the *collection* of related pieces of information about strikes and spares, namely, the state of the bowling game.

What are the characteristics of the state variable?

The variable that indicates the current state of the bowling game must be able to hold (at least) six values. That rules out using type `boolean`, and suggests type `int` or `char`*. Using this

---

* An `enum` would be a better choice in Java 1.5.

approach, the program represents the state with the following characteristics by the value 0[*]:

rolling the first ball;
no spare on the most recent ball;
no strike on the most recent ball;
no strike from two balls ago.

The state with the following characteristics gets the value 1:

rolling the second ball;
no spare on the most recent ball;
no strike on the most recent ball;
no strike from two balls ago.

How is the state variable used in the scoring process?

And so on. The code for state 0 would then include the following statements:

```
if (ball < 10) {
    state = 1;
        ...
} else ...
```

But this is unnecessarily obscure. Java allows named constants to be defined as `static final` variables. Using this facility, we choose the following names for state values:

```
private static final int ROLLING_FIRST_BALL = 0;
private static final int ROLLING_SECOND_BALL = 1;
private static final int STRIKE_LAST_BALL = 2;
private static final int TWO_CONSEC_STRIKES = 3;
private static final int STRIKE_2_BALLS_AGO = 4;
private static final int SPARE_LAST_BALL = 5;
```

(It is conventional in Java for constants to have names that are all upper case.) This change makes the code easier to understand:

```
if (ball < 10) {
    state = ROLLING_SECOND_BALL;
        ...
} else {
    state = STRIKE_LAST_BALL;
        ...
```

This choice of a single state variable also allows the use of a `switch` statement whose cases represent the handling of each state.

The `roll` method is coming together:[†]

```
public void roll (int ball) {
    if (state == ROLLING_FIRST_BALL) {
        ...
    } else if (state == ROLLING_SECOND_BALL) {
        ...
    } else if (state == STRIKE_LAST_BALL) {
        ...
```

---

* Numbering in Java programs generally starts at 0 rather than 1.

† This could also be coded as a `switch`.

```
    } else if (state == TWO_CONSEC_STRIKES) {
        ...
    } else if (state == STRIKE_2_BALLS_AGO) {
        ...
    } else if (state == SPARE_LAST_BALL) {
        ...
    } else {
        // signal an invalid state error
    }
}
```

We now have designed the overall structure of the roll method, and have chosen instance variables. The next task is to code and test the Scorer methods; we will do this via an approach referred to as "test-driven development".

Exercises

Analysis 12. After starting the game with three calls to roll, the state is STRIKE_2_BALLS_AGO. What were the arguments to roll?

Analysis 13. The number of different *transitions* from state to state is interesting from the standpoint of testing a program based on a state diagram. How many different state-to-state transitions are there in the state diagram just designed?

Analysis 14. How many of the different state-to-state transitions are made in scoring the sample game in the problem statement?

Analysis 15. What is the maximum number of different state-to-state transitions that can be made in scoring a game that includes no strikes?

Analysis 16. Explain why it is impossible to be rolling the first ball in a frame, with a strike two balls ago and not a strike or spare on the last ball.

Application 17. The game of table tennis (Ping-Pong) is played by two players. One player *serves* the ball, then the players hit it back and forth until one of the players misses. One way to score table tennis is as follows:

- If the server misses, the serve goes to the other player.
- If the server's opponent misses, the server scores 1.
- The winner is whoever reaches a score of at least 15 and is at least two points ahead of the opponent.

Organize this method of scoring table tennis as a state diagram.

Reflection 18. Summarize the decisions made in the design so far.

Test-driven development of the Scorer class, part 1

What is test-driven development?

Many students, and even some experienced programmers, regard testing as something to be done after a program is completely coded. Delayed testing, however, can lead to problems:

- bugs involving interaction of program components can be extremely difficult to find;
- situations that are important to test are easily overlooked.

Test-driven development is a technique that addresses these problems. Coding and testing—*development* of the program—is done in small pieces, with tests being written *before* the code. The cycle of designing a test for a bit of functionality, then writing and testing the code that implements that functionality, is repeated until the program is complete. By writing only enough code to pass each test, the programmer also limits where bugs can occur and makes them easy to find.

What is `JUnit`, and how is it used?

The `JUnit` tool simplifies test-driven development. It allows the design of test methods composed of calls to *assertion methods* that check return values and internal state for correctness. The most commonly used assertion methods are the following.

- `void assertEquals (errmsg, expected, actual)`
  which compares the expected value of a variable or expression (supplied as the "expected" argument) to its actual value (the "actual" argument) and signals an error (printing the given message) if they don't match;
- `void assertTrue (errmsg, expression)`
  evaluates the given expression and signals an error (printing the given message) if it's not `true`.
- `void assertFalse (errmsg, expression)`
  evaluates the given expression and signals an error (printing the given message) if it is `true`.

The names of the test methods all must start with the word "test", for example, "testGame".

The development process then consists of adding one or more `JUnit` test methods to the collection already invented, then supplying the code with which the test will be satisfied. Since each round of the development cycle invents a test, then supplies the code to go with it, the collection of tests should be *exhaustive*; that is, it should test *all* the code written so far.

What will be the first `JUnit` tests?

We start with tests that merely exercise the `Scorer` constructor, along with the access method. This may seem like a trivial amount of code, but follows the principle of testing only a bit of functionality at each step.

The constructor should initialize the frame number to 1 and the total score to 0. A test of the constructor would thus use the access functions that return these values, and make sure that the game isn't over yet. Following good programming practice, we give the test method a name that suggests its purpose. Here's the code.

```
import junit.framework.TestCase;

public class ScorerTest extends TestCase {

    public void testConstructor ( ) {
        Scorer s = new Scorer ( );
        assertEquals ("bad frame # for new Scorer",
            1, s.frameNumber ( ));
        assertEquals ("bad score for new Scorer",
            0, s.scoreSoFar ( ));
        assertFalse ("game is over for new Scorer",
            s.gameIsOver ( );
    }

}
```

We now write the code to be tested:

```
public class Scorer {

    // The number of the frame in which the next ball
    //     will be rolled; ranges from 1 to 11
    private int rollingFrame;

    // 0 if firstUnscoredFrame == 1,
    //   frameScores[firstUnscoredFrame] thereafter
    private int totalScore;

    public Scorer ( ) {
        rollingFrame = 1;
        totalScore = 0;
    }

    public int frameNumber ( ) {
        return rollingFrame;
    }

    public int scoreSoFar ( ) {
        return totalScore;
    }

    public boolean gameIsOver ( ) {
        return false;
    }
}
```

The Scorer constructor and the gameIsOver method will clearly need to be expanded in the production version, but we are patient and provide only code that we can test. Running the test case with JUnit gives us an "all tests passed" signal. So far, so good.

Stop and help → *Change the* Scorer *code so that one of the tests is invalidated, for instance by initializing* rollingFrame *to 3. Then run the tests to see how* JUnit *signals the invalid value.*

What should be tested after the Scorer constructor?

A reasonable next step is to test a single roll that's not a strike. In addition to checking the results from frameNumber, scoreSoFar, and gameIsOver, we check the result of roll, which must be an empty array. Here is the code.

```
public void test1stRoll ( ) {
    Scorer s = new Scorer ( );
    int [ ] result = s.roll (1);
    assertEquals ("result of 1st roll is wrong",
        0, result.length);
    assertEquals ("frame # after 1st ball is wrong",
        1, s.frameNumber ( ));
    assertEquals ("score after 1st ball is wrong",
        0, s.scoreSoFar ( ));
    assertFalse ("game is over after 1st ball",
        s.gameIsOver ( ));
}
```

In the Scorer class, we add a new variable to store the scores, and initialize it in the constructor:

```
private int [ ] frameScores;

public Scorer ( ) {
        ...
    frameScores = new int [0];
}
```

Finally, we supply the roll method:

```
public int [ ] roll (int ball) {
    return frameScores;
}
```

We run the test cases; they both pass.

What should be tested after a single nonstrike roll?

The next easiest thing to test is a second roll that leaves pins standing (i.e. completes a miss). This should produce a score for the first frame.

```
public void testFrame1miss ( ) {
    Scorer s = new Scorer ( );
    int [ ] result = s.roll (1);
    result = s.roll (2);
    assertEquals ("bad result after frame 1",
        1, result.length);
    assertEquals ("bad result[0] after frame 1",
        3, result[0]);
    assertEquals ("frame # after frame 1 is wrong",
        2, s.frameNumber ( ));
    assertEquals ("score after frame 1 is wrong",
        3, s.scoreSoFar ( ));
    assertFalse ("game is over after 1st frame",
        s.gameIsOver ( ));
}
```

Stop and consider → *Why are no assertions provided immediately after the first call to* roll*?*

Stop and consider → *Rolls of 1 followed by 2 are actually quite atypical in a bowling game. Why aren't more tests needed that provide more typical rolls?*

What Scorer code must be added to handle multiple rolls?

The Scorer class must now include some way to distinguish the first ball in a frame from the second, since the score so far should be increased after the second roll but not after the first. Looking ahead to the six-state program, we code this as follows:

```
private int state;
private static final int ROLLING_FIRST_BALL = 0;
private static final int ROLLING_SECOND_BALL = 1;
```

and add a statement in the constructor that initializes state. Also required is the variable firstBallInFrame in which to save the first ball value. It is not initialized in the constructor since its value is undefined at the start of a frame.

```
private int firstBallInFrame;
```

Finally, we turn to roll, based on the pattern we designed earlier:

```
public int [ ] roll (int ball) {
    if (state == ROLLING_FIRST_BALL) {
        // handle the first ball;
    } else if (state == ROLLING_SECOND_BALL) {
        // handle the second ball;
    } else {
        // signal an error;
    }
    return frameScores;
}
```

There are several things to worry about. Tested directly are the frameScores array, which needs to include the result of the first frame; the frameNumber, which needs to be incremented; and the score so far. There also needs to be a way of switching states.

What code handles the first ball in a frame?

We start with the ROLLING_FIRST_BALL case. This just assigns the ball value to firstBallInFrame and updates the state:

```
if (state == ROLLING_FIRST_BALL) {
    firstBallInFrame = ball;
    state = ROLLING_SECOND_BALL;
}
```

What code handles the second ball in a frame?

In the case for the second ball, we instantiate a new array for frameScores, update the frame number and the score, record the score in frameScores[0], and switch states.

```
} else if (state == ROLLING_SECOND_BALL) {
    frameScores = new int [1];
    rollingFrame++;
    totalScore = totalScore + firstBallInFrame + ball;
    frameScores[0] = totalScore;
    state = ROLLING_FIRST_BALL;
}
```

Can this code handle a second frame?

Tests pass so far. Can we handle a third and fourth roll? Let's try:

```
public void test3rdBall ( ) {
    Scorer s = new Scorer ( );
    int [ ] result = s.roll(1);
    result = s.roll(2);
    result = s.roll(4);
    assertEquals ("bad result after ball 3",
        1, result.length);
    assertEquals ("bad result[0] after ball 3",
        3, result[0]);
    assertEquals ("frame # after ball 3 is wrong",
        2, s.frameNumber ( ));
    assertEquals ("score after ball 3 is wrong",
        3, s.scoreSoFar ( ));
    assertFalse ("game is over after 1st frame",
         s.gameIsOver ( ));
}
public void testFrame2miss ( ) {
    Scorer s = new Scorer ( );
    int [ ] result = s.roll(1);
    result = s.roll(2);
    result = s.roll(4);
    result = s.roll(1);
    assertEquals ("bad result after frame 2",
        2, result.length);
    assertEquals ("bad result[0] after frame 2",
        3, result[0]);
    assertEquals ("bad result[1] after frame 2",
        8, result[0]);
    assertEquals ("frame # after frame 2 is wrong",
        3, s.frameNumber ( ));
    assertEquals ("score after frame 2 is wrong",
        8, s.scoreSoFar ( ));
    assertFalse ("game is over after frame 2",
        s.gameIsOver ( ));
}
```

Stop and predict →

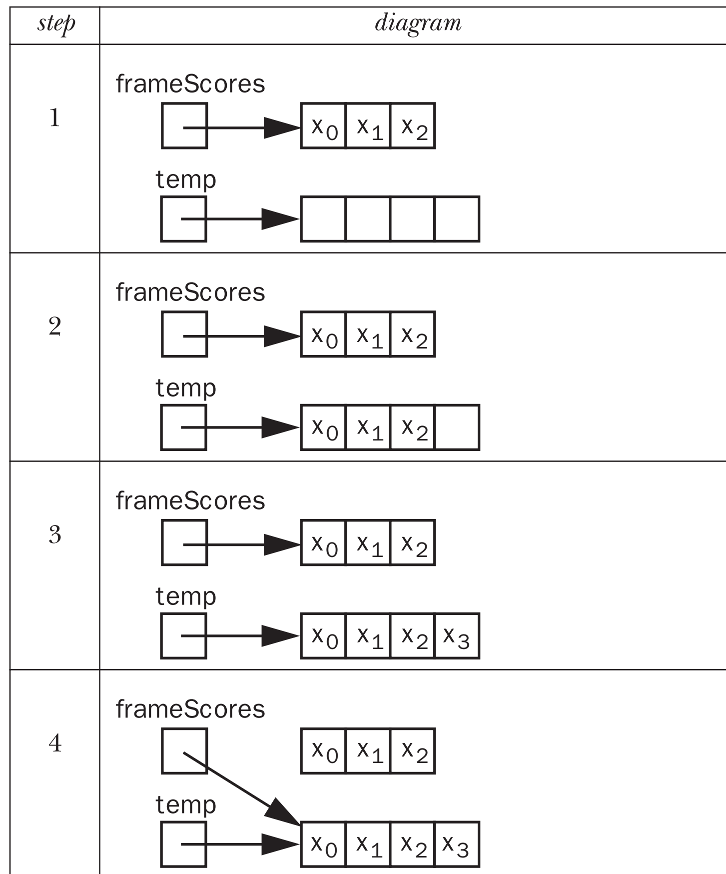
The testFrame2miss *method has a bug. Find it.*

Interestingly, only one test fails: the length of the result array after four rolls is 1 rather than 2. What changes are needed to pass this test? The problem is that frameScores must grow as the game progresses, and must also retain scores of earlier frames.

What's needed to accumulate scores across calls to roll?

An array, once constructed, stays the same length. What's needed is a *second* array—we'll call it temp—that's one element longer than frameScores. The scores from frameScores are copied into temp, the score for the just-completed frame is added, and the temp reference is assigned to frameScores. The diagram below displays this process in the situation where

the fourth frame is being scored. (The x's represent the cumulative score values stored in frameScores.)



How are arrays expanded in Java?

This operation, essentially an expansion of frameScores by one element, is a common one. Here's the code.

```
int [ ] temp = new int [frameScores.length+1];
for (int k=0; k<frameScores.length; k++) {
    temp[k] = frameScores[k];
}
temp[frameScores.length] = totalScore;
frameScores = temp;
```

We plug in that code, and fail another test in testFrame2miss: element #1 of the resulting array contains 3, not 8. Here the problem was in the test code. It was caused by cutting and pasting, then neglecting to fix the pasted line. Both the statements below test result[0]; the second statement should test result[1].

```
assertEquals ("bad result[0] after frame 2",
    3, result[0]);
assertEquals ("bad result[1] after frame 2",
    8, result[0]);
```

After fixing this somewhat embarrassing error, the test cases all run successfully. Appendix A contains the code designed so far.

How can individual test methods be combined into a single, more general test method?

Right now we have separate test cases to check 0, 1, 2, 3, and 4 calls to roll, assuming none of them produces a strike or a spare. This will shortly get out of control, and thus we turn to reorganizing the tests.

The fact that the program is executing code for 1 roll, 2 rolls, and so on suggests a loop:

```
for (int numRolls=1; the game isn't over; numRolls++) {
    test the case of numRolls calls to roll;
}
```

Each test with one or more calls to roll in addition does the following.

- It checks that the result of the last call is an array of specified size and contents.
- It checks the frame number and total score so far.
- It makes sure the game isn't over.

Thus the loop body will implement the following pseudocode.

```
result = roll (kth ball);
assertEquals ("result length check",
    expected length of kth roll call, result.length);
for (int j=0; j<result.length; j++) {
    assertEquals ("frame check",
        expected frame contents, result[j]);
}
assertEquals ("frame number check",
    expected frane number after kth roll,
    frameNumber return value);
assertEquals ("total score check",
    expected total score after kth roll,
    scoreSoFar return value);
assertFalse ("game over check",
    gameIsOver return value);
```

How are arrays used to generalize the testing?

Anything that refers to "the kth …" can be set up to use an array with k as an index. For example, in testFrame2miss, we tested the roll sequence 1, 2, 4, 1; a general test method would use an array named balls declared as

```
int [ ] balls = {1, 2, 4, 1};
```

Similarly, arrays can store the expected lengths of calls to roll and expected values of calls to frameNumber. For the testFrame2miss example, these would be declared as

```
int [ ] lengths = {0, 1, 1, 2};
int [ ] frames = {1, 2, 2, 3};
```

Finally, an array named finalResult will store the result returned by the final call to roll. For the testFrame2miss example, this would be

```
int [ ] finalResult = {3, 8};
```

The kth roll result should contain the first result.length elements of finalResult. After the values in result are verified

against those in finalResult, they can be used to test the scoreSoFar method:

```
assertEquals ("checking score after ball " + k,
    result[result.length-1], s.scoreSoFar ( ));
```

We assume for now that gameIsOver should always return false.

This technique of reorganizing code to make it shorter, or more elegant, or more easily understood is called *refactoring*. Here's a testGame method that represents the refactoring of the four previously implemented methods test1stRoll, testFrame1miss, test3rdBall, and testFrame2miss. .

```
private void testGame (Scorer s; int [ ] balls, int [ ] lengths,
  int [] frames, int [ ] finalResult) {
    int [ ] result;
    int [ ] balls = {1, 2, 4, 1};
    int [ ] lengths = {0, 1, 1, 2};
    int [ ] frames = {1, 2, 2, 3};
    int [ ] finalResult = {3, 8};
    for (int k=0; k<balls.length; k++) {
        result = s.roll (balls[k]);
        assertEquals ("checking length of result of ball " + k,
            lengths[k], result.length);
        for (int j=0; j<result.length; j++) {
            assertEquals ("checking frame " + j + " in result of ball " + k,
                finalResult[j], result[j]);
        }
        assertEquals ("checking frame number after ball " + k,
            frames[k], s.frameNumber ( ));
        if (lengths[k] == 0) {
            assertEquals ("checking score after ball " + k,
                0, s.scoreSoFar());
        } else {
            assertEquals ("checking score after ball " + k,
                result[result.length-1], s.scoreSoFar ( ));
        }
        assertFalse ("checking if game is over after ball " + k,
            s.gameIsOver ( ));
    }
}
```

All the tests of those four methods are made with the method

```
public void test4balls ( ) {
    int [ ] balls = {1, 2, 4, 1};
    int [ ] lengths = {0, 1, 1, 2};
    int [ ] frames = {1, 2, 2, 3};
    int [ ] finalResult = {3, 8};
    Scorer s = new Scorer ( );
    testGame (s, balls, lengths, frames, finalResult);
}
```

Moreover, testGame can handle tests of any number of rolls up to but not including the last roll of a game with all misses.

Stop and predict → *Can* testGame *handle games with strikes and spares?*

Stop and consider → *Why can't* frames[k] *be replaced by* lengths[k]+1*?*

How is the end of a game detected?

We focus next on detecting the end of a game. The problem statement says that the game ends when the score for the tenth frame is recorded. To simplify testing, however, we make the number of the last frame a *variable* named lastFrameNumber.

```
private int lastFrameNumber;
```

Then we invent another constructor that takes the number of frames as an argument. The 0-argument constructor will be used for the standard game of ten frames; to avoid duplication of code, we write it as a call to the new constructor.

```
private int lastFrameNumber;

public Scorer ( ) {
    this (10);
}

public Scorer (int frameCount) {
    lastFrameNumber = frameCount;
    rollingFrame = 1;
    totalScore = 0;
    frameScores = new int [0];
    state = ROLLING_FIRST_BALL;
}
```

Stop and help → *Explain how making the number of the last frame a variable simplifies testing.*

For the moment, since we're only worrying about games without strikes and spares, we can code gameIsOver as

```
public boolean gameIsOver ( ) {
    return rollingFrame > lastFrameNumber;
}
```

(Recall that rollingFrame is the number of the frame in which the next ball will be rolled.) In the tests, we may assume that the array of balls constitute a complete game; thus it ought to be the case that the game is over after the last ball is rolled:

```
assertEquals ("game-over check, ball " + k,
    k == balls.length-1, s.gameIsOver ( ));
```

Here's the new test4balls:

```
public void test4Balls ( ) {
    Scorer s = new Scorer (2);
    int [ ] balls = {1, 2, 4, 1};
    int [ ] lengths = {0, 1, 1, 2};
    int [ ] frames = {1, 2, 2, 3};
    int [ ] finalResult = {3, 8};
    testGame (s, balls, lengths, frames, finalResult);
}
```

Appendix B contains the code produced so far. It handles any game not containing a strike or a spare.

Exercises

Reflection 19. Why would a programmer believe that testing should be delayed until after a program is completely coded?

Analysis 20. The `testGame` method is called with a `balls` array that contains 8, 1, 7, 2. What are the values of the other arguments?

Application 21. Using an array, rewrite the following program segment as a loop.

```
assertEquals (5, f(1));
assertEquals (13, f(2));
assertEquals (19, f(3));
assertEquals (100, f(4));
assertEquals (38, f(5));
assertEquals (71, f(6));
```

Analysis 22. What is the effect of the following program segment?

```
int [ ] temp;
temp = new int [frameScores.length+1];
temp = frameScores;
frameScores = temp;
```

Analysis 23. The constructor initializes `frameScores` as follows:

```
frameScores = new int [0];
```

Is this the same as setting `frameScores` to `null`? Why or why not?

Application 24. Write a program `insertAtStart` that inserts an integer `k` at the beginning of an array `values`. The length of `values` should be increased by 1 as a result.

Application 25. Combining a collection of similar program segments into a single segment that uses a loop is one example of refactoring. Moving a program segment that's used several times into a method is another. Describe a third.

Analysis 26. What does the program in Appendix B do with a ball sequence that contains a spare, such as 8, 2, 7, 3?

Reflection 27. Summarize the design and development so far.

Test-driven development of the Scorer class, part 2

What tests are needed for spare handling?

We move on now to handling more complicated games, namely those that include strikes and spares. Spares seem easier to deal with than strikes, so we start with them.

Handling a spare correctly involves two cases: a spare before the end of the game and one at the end. We add the additional case of a spare in the first frame; while this should be handled the same way as a spare in frame 2, the boundary case may reveal a bug that the more general test doesn't. We decide also to test the occurrence of two spares in a row, and the roll of a 0 (a "gutter ball") immediately after a spare. A game of three frames will be sufficient to allow all these situations. Example ball sequences are the following:

6, 4, 3, 4, 5, 2 (spare first frame)
6, 3, 4, 6, 5, 2 (spare second frame)
6, 3, 3, 4, 8, 2, 4 (spare last frame)
6, 4, 3, 7, 5, 2 (spare first two frames)
6, 3, 3, 7, 8, 2, 4 (spare last two frames)
6, 4, 0, 10, 5, 2 (spare first two frames, with a 0)

Stop and help → *Describe all the "boundaries" that are exercised with the above test cases.*

How can these tests make use of `testGame`?

With luck, we can implement these tests with the `testGame` method just devised. This method requires a `Scorer` object, and several arrays:

- the ball sequence;
- the lengths of return values from `roll`;
- the frame numbers, that is, for each ball, the frame that the next ball will be rolled in; and
- the final result.

All the arrays but the final result contain one value per ball rolled.

Let's see what arrays would accompany a ball sequence of 6, 3, 3, 7, 8, 2, 4.

ball	`roll` *return*	*length of* `roll` *return*	*frame for next ball*
6	{ }	0	1
3	{9}	1	2
3	{9}	1	2
7	{9}	1	3
8	{9, 27}	2	3
2	{9, 27}	2	3
4	{9, 27, 41}	3	4

The array arguments to testGame would then be

balls	{6, 3, 3, 7, 8, 2, 4}
lengths	{0, 1, 1, 1, 2, 2, 3}
frame numbers	{1, 2, 2, 3, 3, 3, 4}
final result	{9, 27, 41}

This looks promising. Trusting that the rest of our test cases can be similarly implemented with calls to testGame, we turn to Scorer to invent the code for handling spares.

How are spares handled?

A new state SPARE_LAST_BALL is necessary, as we noted in the first part of this case study. This state is entered from ROLLING_SECOND_BALL when the argument and the first ball in the frame total 10. Here is code:

```
} else if (state == ROLLING_SECOND_BALL) {
    if (firstBallInFrame + ball == 10) {
        rollingFrame++;
        state = SPARE_LAST_BALL;
    } else {
        ... // code already implemented for this state
    }
} ...;
```

Handling the SPARE_LAST_BALL state involves incrementing the score as in ROLLING_SECOND_BALL:

```
} else if (state == SPARE_LAST_BALL) {
    totalScore = totalScore + 10 + ball;
    // extend frameScores by one frame
    int [ ] temp = ...
        ...
    frameScores = temp;
    firstBallInFrame = ball;
    state = ROLLING_SECOND_BALL;
} ...
```

Noticing that the frameScores extension code now appears in two places, we refactor it into a private method named addFrame:

```
private void addFrame ( ) {
    int [ ] temp = new int [frameScores.length+1];
    for (int k=0; k<frameScores.length; k++) {
        temp[k] = frameScores[k];
    }
    temp[frameScores.length] = totalScore;
    frameScores = temp;
}
```

How are the test cases implemented?

We return to the six test cases, implementing them as calls to `testGame` as described earlier. Here is the code.

```
public void testSpareMissMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 4, 3, 4, 5, 2};
    int [ ] lengths = {0, 0, 1, 2, 2, 3};
    int [ ] frames = {1, 2, 2, 3, 3, 4};
    int [ ] finalResult = {13, 20, 27};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testMissSpareMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 3, 4, 6, 5, 2};
    int [ ] lengths = {0, 1, 1, 1, 2, 3};
    int [ ] frames = {1, 2, 2, 3, 3, 4};
    int [ ] finalResult = {9, 24, 31};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testMissMissSpare ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 3, 3, 4, 8, 2, 4};
    int [ ] lengths = {0, 1, 1, 2, 2, 2, 3};
    int [ ] frames = {1, 2, 2, 3, 3, 3, 4};
    int [ ] finalResult = {9, 16, 30};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testSpareSpareMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 4, 3, 7, 5, 2};
    int [ ] lengths = {0, 0, 1, 1, 2, 3};
    int [ ] frames = {1, 2, 2, 3, 3, 4};
    int [ ] finalResult = {13, 28, 35};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testMissSpareSpare ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 3, 3, 7, 8, 2, 4};
    int [ ] lengths = {0, 1, 1, 1, 2, 2, 3};
    int [ ] frames = {1, 2, 2, 3, 3, 3, 4};
    int [ ] finalResult = {9, 27, 41};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testSpare0SpareMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 4, 0, 10, 5, 2};
    int [ ] lengths = {0, 0, 1, 1, 2, 3};
    int [ ] frames = {1, 2, 2, 3, 3, 4};
    int [ ] finalResult = {10, 25, 32};
    testGame (s, balls, lengths, frames, finalResult);
}
```

What aspects of spare handling remain to be implemented?

Four of the six tests succeed. The two that fail are those with a spare in the last frame; the frame check failed for the next-to-last ball, returning 4 instead of 3. Here's what's happening in testMissMissSpare:

ball	*explanation*
6	still in frame 1 for next ball
3	entering frame 2
3	one ball left in frame 2
4	entering frame 3
8	next ball will still be in frame 3
2	if a game were longer than three frames, the next ball would be in frame 4; however, in a three-frame game, there is still one ball to roll in frame 3

Some investigation reveals that handling the second ball of a spare *always* increments rollingFrame. The incrementing should only happen if the spare occurs before the last frame.

How should the last frame be handled?

There are several ways to deal with this problem. One is to wrap a test around the incrementing of rollingFrame in the ROLLING_SECOND_BALL code. A problem then would be to distinguish a spare in the last frame from a spare in the next-to-last frame. Another approach would be to allow rolling-Frame to take on the "wrong" value, but then to determine its correct value in the frameNumber method. We do the latter.

Stop and consider →

Explore the approach of delaying the incrementing of rollingFrame *in the* ROLLING_SECOND_BALL *code.*

How should frameNumber's return value be adjusted?

As just noted, the value of rollingFrame will be too large when a spare is rolled in the last frame. We add a test for this:

```
if (state == SPARE_LAST_BALL
  && rollingFrame > lastFrameNumber) {
    return rollingFrame-1;
} else {
    return rollingFrame;
}
```

This modification passes all the tests.

Does testGame work with strikes?

We move now to testing and debugging code that handles strikes. First, we check that testGame works for strikes, using a variety of ball sequences with a three-frame game:

10, 10, 10, 10, 10 (all strikes)
10, 0, 10, 10, 0, 10 (mixture of strikes and spares)
3, 6, 10, 10, 7, 3 (mixture of strikes, spares, and misses)

ball	roll *return*	*frame for next ball*	*ball*	roll *return*	*frame for next ball*	*ball*	roll *return*	*frame for next ball*
10	{ }	2	10	{ }	2	3	{ }	1
10	{ }	3	0	{ }	2	6	{9}	2
10	{30}	3	10	{20}	3	10	{9}	3
10	{30, 60}	3	10	{20, 40}	3	10	{9}	3
10	{30, 60, 90}	4	0	{20, 40}	3	7	{9, 36}	3
			10	{20, 40, 60}	4	3	{9, 36, 56}	4

Three test methods result:

```
public void testAllStrikes ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 10, 10, 10, 10};
    int [ ] lengths = {0, 0, 1, 2, 3};
    int [ ] frames = {2, 3, 3, 3, 4};
    int [ ] finalResult = {30, 60, 90};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testStrikeSpareStrikeSpare ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 0, 10, 10, 0, 10};
    int [ ] lengths = {0, 0, 1, 2, 2, 3};
    int [ ] frames = {2, 2, 3, 3, 3, 4};
    int [ ] finalResult = {20, 40, 60};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testMissStrikeStrikeSpare ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {3, 6, 10, 10, 7, 3};
    int [ ] lengths = {0, 1, 1, 1, 2, 3};
    int [ ] frames = {1, 2, 3, 3, 3, 4};
    int [ ] finalResult = {9, 36, 56};
    testGame (s, balls, lengths, frames, finalResult);
}
```

What other test cases are needed?

Rather than handling all strikes right away, we limit ourselves to dealing only with nonconsecutive strikes. (This means we will only have to implement two more states in Scorer at this stage rather than all three of the states that remain.) A variety of test cases will be needed, enumerated below:

a. a spare followed by a strike

b. a strike followed by a spare

c. a miss followed by a strike

d. a strike followed by a miss

e. a strike in the first frame

We will also need to test all possibilities involving a strike in the last frame. These cases are particularly important because of earlier last-frame bugs. The possibilities for the last frame are the following:

f. strike followed by a spare

g. strike followed by a miss

h. spare followed by a strike

We may also wish to test rolls of 0. Here is a collection of ball sequences that cover all these cases.

6, 2, 10, 5, 4	10, 7, 2, 10, 6, 3
7, 3, 10, 5, 4	10, 0, 0, 4, 3
10, 7, 3, 8, 2, 10	10, 6, 0, 4, 3
10, 7, 2, 10, 6, 4	10, 0, 6, 4, 3

We add the corresponding test… methods to our test suite.

Stop and help → *Provide the test methods for the ball sequences just described.*

Stop and consider → *Provide a smaller set of ball sequences that covers all of the cases described above.*

Stop and consider → *Is it appropriate to test two or more cases in a single ball sequence? Explain why or why not.*

What changes to the Scorer roll method are necessary?

As in earlier versions, additions to roll include the code to handle the new states, plus the code to enter those states from existing states. The two new states are STRIKE_LAST_FRAME, entered from itself and ROLLING_FIRST_BALL, and STRIKE_2_BALLS_AGO, entered from STRIKE_LAST_FRAME. ROLLING_FIRST_BALL splits into two cases:

```
if (state == ROLLING_FIRST_BALL) {
    if (ball == 10) {  // the new code
        rollingFrame++;
        state = STRIKE_LAST_BALL;
    } else {
        firstBallInFrame = ball;
        state = ROLLING_SECOND_BALL;
    }
} ...
```

Here's the code for STRIKE_LAST_FRAME and STRIKE_2_BALLS_AGO:

```
} else if (state == STRIKE_LAST_BALL) {
    if (ball == 10) {
        rollingFrame++;
        state = TWO_CONSEC_STRIKES;
    } else {
        firstBallInFrame = ball;
        state = STRIKE_2_BALLS_AGO;
    }
} ...
```

```
} else if (state == STRIKE_2_BALLS_AGO) {
    totalScore = totalScore
        + 10 + firstBallInFrame + ball;
    addFrame ( );
    if (firstBallInFrame + ball == 10) {
        rollingFrame++;
        state = SPARE_LAST_BALL;
    } else {
        totalScore = totalScore
            + firstBallInFrame + ball;
        addFrame ( );
        rollingFrame++;
        state = ROLLING_FIRST_BALL;
    }
} ...
```

What problems arise?

Two problems arise involving strikes in the tenth frame, namely that values returned by frameNumber and scoreSoFar are too high. (We encountered a similar problem handling spares in the tenth frame.) Ball sequences ending with a strike and a miss, such as 10, 7, 2, 10, 6, 3, reveal both problems; the table below provides the details.

after roll	*score so far*	*correct score*	*next frame*	*correct next frame*	*next state*
2	28	28	3	3	ROLLING_FIRST_BALL
10	28	28	4	3	STRIKE_LAST_BALL
6	28	28	4	3	STRIKE_2_BALLS_AGO
3	56	47	5	4	ROLLING_FIRST_BALL

The frame number is incremented after the 2 (the end of an actual frame) as well as after the 10 (not the end of a frame). Handling the last roll calls addFrame twice, tallying the 6 and the 3 both for the strike and for the (non-)frame following the strike.

Stop and help → *Find a different pattern of strikes, spares, and misses in the tenth frame that displays the frame number bug. Do the same for the score bug.*

Our solution for an incorrect frame when handling spares was to add a case in frameNumber that checks for the incorrect case:

```
if (state == SPARE_LAST_BALL
  && rollingFrame > lastFrameNumber) {
    return rollingFrame-1;
} ...
```

What fixes correct the problem?

We choose, with some misgivings, to add cases to the frameNumber code. (It took a few debugging runs to get these right.)

```
public int frameNumber ( ) {
    if (state == SPARE_LAST_BALL
      && rollingFrame > lastFrameNumber) {
        return rollingFrame-1;
    } else if (state == STRIKE_LAST_BALL
      && rollingFrame > lastFrameNumber) {
        return rollingFrame-1;
    } else if (state == STRIKE_2_BALLS_AGO
      && rollingFrame > lastFrameNumber) {
        return rollingFrame-1;
    } else if (state == ROLLING_FIRST_BALL
      && rollingFrame > lastFrameNumber+1) {
        return rollingFrame-1;
    } else {
        return rollingFrame;
    }
}
```

Stop and help → *Suggest a reason why a programmer might view the above code as flawed.*

Stop and predict → *Suggest a cleaner way to identify the correct frame number when processing strikes and spares in the tenth frame.*

The problem statement suggests a way to handle scoreSoFar. The description of the gameIsOver method says that the game is over "when the tenth frame has been scored". In scoreSoFar, we can check for this explicitly:

```
public int scoreSoFar ( ) {
    if (frameScores.length == lastFrameNumber) {
        return frameScores[lastFrameNumber-1];
    } else {
        return totalScore;
    }
}
```

The resulting program passes all the tests.

How are consecutive strikes tested?

There is one more thing to implement: handling consecutive strikes. First come the tests:

all strikes;
two strikes, then a miss;
a miss, two strikes, and a spare;
a miss, two strikes, and a miss.

Stop and consider → *Why do we not need to test the sequence* strike, strike, spare*? What about* spare, strike, strike, miss*?*

Stop and help → *Write the code for a method named* testMissStrikeStrikeSpare.

How are consecutive strikes handled?

After providing test… methods for the ball sequences just described, we turn to the Scorer class. Here, we addcode in roll to handle TWO_CONSEC_STRIKES, and to make the transition from STRIKE_LAST_BALL to TWO_CONSEC_STRIKES, and provide whatever new cases are necessary to produce the

correct frame number. Here is the modification to the STRIKE_LAST_BALL code:

```
} else if (state == STRIKE_LAST_BALL) {
    if (ball == 10) {  // new code
        rollingFrame++;
        state = TWO_CONSEC_STRIKES;
    } else {
        firstBallInFrame = ball;
        state = STRIKE_2_BALLS_AGO;
    }
} ...
```

Two consecutive strikes are handled as follows:

```
} else if (state == TWO_CONSEC_STRIKES) {
    totalScore = totalScore + 20 + ball;
    addFrame ( );
    if (ball == 10) {
        rollingFrame++;
    } else {
        firstBallInFrame = ball;
        state = STRIKE_2_BALLS_AGO;
    }
} ...
```

Then, after some experimentation, we come up with two more cases for frameNumber:

```
public int frameNumber ( ) {
    if ... {
    } else if (state == TWO_CONSEC_STRIKES
      && rollingFrame == lastFrameNumber+1) {
        return rollingFrame-1;
    } else if (state == TWO_CONSEC_STRIKES
      && rollingFrame > lastFrameNumber+1) {
        return rollingFrame-2;
    } else {
        return rollingFrame;
    }
}
```

The resulting program passes all the tests. It appears in Appendix C.

What remains to be done?

We now review the Scorer code, looking for ways to improve it. The extensive tests designed throughout the process of completing the first draft of Scorer should simplify these improvements.

We first notice that every call to addFrame is preceded by an assignment to totalScores. This assignment can be folded into addFrame:

```
private void addFrame (int toAdd) {
    totalScore = totalScore + toAdd;
    if (frameScores.length < lastFrameNumber) {
        ...
    }
}
```

How should frameNumber be rewritten?

Next, we examine frameNumber. The existing version was difficult to design. The cases are inconsistent, both in what they test and what they return, and the code is thus quite difficult to understand.

The scoreSoFar method, on the other hand, is much simpler. Its structure is basically the following:

```
if the game is over,
    return the score as of the last frame;
otherwise
    return totalScore;
```

This is a good model for frameNumber:

```
if the game is over,
    return lastFrameNumber+1;
otherwise ...
```

The next case to test is when the last frame is being rolled, but rollingFrame indicates a higher frame number:

```
otherwise if (rollingFrame > lastFrameNumber),
    return lastFrameNumber
otherwise
    return rollingFrame
```

Translating to Java is straightforward:

```
public int frameNumber ( ) {
    if (frameScores.length == lastFrameNumber) {
        return lastFrameNumber+1;
    } else if (rollingFrame > lastFrameNumber) {
        return lastFrameNumber;
    } else {
        return rollingFrame;
    }
}
```

This passes all the tests, and is much more clear. Appendix D contains the complete Scorer class.

Exercises

Application 28. Describe a development sequence that would result from handling strikes *before* spares.

Debugging 29. You accidentally delete a statement in the program in Appendix D. A run of JUnit tells you the following.

In testStrikeStrikeMiss, the length of the result of ball 3 is 2 instead of 3.
In testMissStrikeStrikeSpare, the score in frame 2 in the result of ball 5 is 70 instead of 63.
In testMissStrikeStrikeMiss, the score in frame 2 in the result of ball 5 is 67 instead of 60.

Which statement is missing?

Debugging 30. The program treats a spare as if it were a miss; that is, it scores the frame 8, 2 as 10 rather than as 10 plus the next ball. Describe two possible locations for the error.

Modification 31. Modify the program to allow up to *three* rolls per frame. Strikes and spares—that is, knocking down all the pins with one or two rolls—are scored as in the current program. Knocking down all the pins with three rolls earns no extra premium.

Application 32. How would the JUnit test suite in Appendix D change as a result?

Modification 33. Modify the program so that, instead of a single one-argument roll method, there are *two* methods:

```
// Handle a strike.
int [ ] roll ( );
// Handle a miss or a spare.
int [ ] roll (int first, int second);
```

Application 34. How would the JUnit test suite in Appendix D change as a result?

Analysis 35. How would the modification of exercise 33 have simplified the design of the program?

Application 36. What additional tests would be necessary?

Analysis 37. Add a checkForConsistency method to the Scorer class that would check that the values of the instance variables are not self-contradictory.

Reflection 38. List some criteria that a programmer should use to decide when to refactor a program.

Reflection 39. Summarize the last part of the case study.

Outline of design and development decisions

Design of a bowling scorer class

How might a programmer approach this problem?
How does a human score a bowling game?
- What information must the bowling scorer keep track of?
- How should the scoring algorithm be organized?

What does the state diagram for bowling scoring look like?
What organization does the state diagram suggest for the roll method?
What instance variables are appropriate for the Scorer class?
- How should the state of the bowling game be represented?
- What are the characteristics of the state variable?

How is the state variable used in the scoring process?

Test-driven development of the Scorer class, part 1

What is test-driven development?
- What is JUnit, and how is it used?

What will be the first JUnit tests?
What should be tested after the Scorer constructor?
What should be tested after a single nonstrike roll?
What Scorer code must be added to handle multiple rolls?
- What code handles the first ball in a frame?
- What code handles the second ball in a frame?
- Can this code handle a second frame?
- What's needed to accumulate scores across calls to roll?
- How are arrays expanded in Java?

How can individual test methods be combined into a single, more general test method?
- How are arrays used to generalize the testing?

How is the end of a game detected?

Test-driven development of the Scorer class, part 2

What tests are needed for spare handling?
- How can these tests make use of testGame?

How are spares handled?
- How are the test cases implemented?
- What aspects of spare handling remain to be implemented?
- How should the last frame be handled?
- How should frameNumber's return value be adjusted?

Does testGame work with strikes?
What other test cases are needed?
What changes to the Scorer roll method are necessary?
- What problems arise?
- What fixes correct the problem?

How are consecutive strikes tested?
How are consecutive strikes handled?
What remains to be done?
- How should frameNumber be rewritten?

Exercises

Analysis 40. In exercise 13, the *transitions* between states in a state diagram were discussed as a guide to testing a state-based program. Specify a ball sequence with *as few balls as possible* that exercises all transitions between states.

Analysis 41. Specify a ball sequence that spans *as few frames as possible* that exercises all transitions between states.

Analysis 42. Suppose that the problem specifications did not include the `frameNumber` method. How would that have simplified the design and development of the program?

Modification 43. A bowling game currently ends after the tenth frame. This places an upper limit on the game score, namely 300. Because of better ball technology and better designed bowling lanes, however, it's becoming easier to roll a 300 game. Modify the program in Appendix D so that a strike in the tenth frame awards an extra frame, a strike in that frame awards a frame 12, and so on. The game ends only when a nonstrike is bowled. The last frame then consists of the last strike and the two rolls that follow; it is scored the way the tenth frame would be scored in the original program.

Analysis 44. How much of the test suite in Appendix D could be used with the modification of exercise 43?

Modification 45. Modify the program in Appendix D to keep score for the game of Ninepins. Everything in this game is the same as for regular bowling, except there are only nine pins to knock down in each frame instead of ten.

Analysis 46. How much of the test suite in Appendix D could be used with the modification of exercise 45?

Analysis 47. Describe, as completely as possible, the legal arguments to the `roll` method.

Reflection 48. Consider the way that you would have designed and developed a program for keeping track of bowling scores. How would you have designed the program differently? How would you have coded and tested the program differently?

Reflection 49. There is roughly twice as much testing code in Appendix D as there is code in the `Scorer` class. What do you think of this?

Reflection 50. List three things that you learned from this case study.

Appendix A: Initial code to test and handle misses

ScorerTest.java

```
import junit.framework.TestCase;

public class ScorerTest extends TestCase {

    public void testConstructor ( ) {
        Scorer s = new Scorer ( );
        assertEquals ("frame number for new Scorer is wrong", 1, s.frameNumber ( ));
        assertEquals ("score for new Scorer is wrong", 0, s.scoreSoFar ( ));
        assertFalse ("game is over for new Scorer", s.gameIsOver ( ));
    }

    public void test1stRoll ( ) {
        Scorer s = new Scorer ( );
        int [ ] result = s.roll (1);
        assertEquals ("result of 1st roll is wrong", 0, result.length);
        assertEquals ("frame number after 1st ball is wrong", 1, s.frameNumber ( ));
        assertEquals ("score after 1st ball is wrong", 0, s.scoreSoFar( ));
        assertFalse ("game is over after 1st ball", s.gameIsOver ( ));
    }

    public void testFrame1miss ( ) {
        Scorer s = new Scorer ( );
        int [ ] result = s.roll (1);
        result = s.roll (2);
        assertEquals ("bad result after frame 1", 1, result.length);
        assertEquals ("bad result[0] after frame 1", 3, result[0]);
        assertEquals ("frame # after frame 1 is wrong", 2, s.frameNumber ( ));
        assertEquals ("score after frame 1 is wrong", 3, s.scoreSoFar ( ));
        assertFalse ("game is over after 1st frame", s.gameIsOver ( ));
    }

    public void test3rdBall ( ) {
        Scorer s = new Scorer ( );
        int [ ] result = s.roll(1);
        result = s.roll(2);
        result = s.roll(4);
        assertEquals ("bad result after ball 3", 1, result.length);
        assertEquals ("bad result[0] after ball 3", 3, result[0]);
        assertEquals ("frame # after ball 3 is wrong", 2, s.frameNumber ( ));
        assertEquals ("score after ball 3 is wrong", 3, s.scoreSoFar ( ));
        assertFalse ("game is over after 1st frame", s.gameIsOver ( ));
    }

    public void testFrame2miss ( ) {
        Scorer s = new Scorer ( );
        int [ ] result = s.roll(1);
        result = s.roll(2);
        result = s.roll(4);
        result = s.roll(1);
        assertEquals ("bad result after frame 2", 2, result.length);
        assertEquals ("bad result[0] after frame 2", 3, result[0]);
        assertEquals ("bad result[1] after frame 2", 8, result[1]);
        assertEquals ("frame # after frame 2 is wrong", 3, s.frameNumber ( ));
        assertEquals ("score after frame 2 is wrong", 8, s.scoreSoFar ( ));
        assertFalse ("game is over after frame 2", s.gameIsOver ( ));
    }
}
```

Scorer.java

```
public class Scorer {

    // the number of the frame in which the next ball
    //   will be rolled; ranges from 1 to 11
    private int rollingFrame;

    // 0 if firstUnscoredFrame == 1, frameScores[firstUnscoredFrame] thereafter
    private int totalScore;

    // frameScores[k] == cumulative score up through frame k;
    private int [ ] frameScores;

    private int state;
    private static final int ROLLING_FIRST_BALL = 0;
    private static final int ROLLING_SECOND_BALL = 1;

    // the number of pins knocked down by the first ball in frame number rollingFrame
    private int firstBallInFrame;

    public Scorer ( ) {
        rollingFrame = 1;
        totalScore = 0;
        frameScores = new int [0];
        state = ROLLING_FIRST_BALL;
    }

    public int frameNumber ( ) {
        return rollingFrame;
    }

    public int scoreSoFar ( ) {
        return totalScore;
    }

    public boolean gameIsOver ( ) {
        return false;
    }

    public int [ ] roll (int ball) {
        if (state == ROLLING_FIRST_BALL) {
            firstBallInFrame = ball;
            state = ROLLING_SECOND_BALL;
        } else if (state == ROLLING_SECOND_BALL) {
            totalScore = totalScore + firstBallInFrame + ball;
            rollingFrame++;
            int [ ] temp = new int [frameScores.length+1];
            for (int k=0; k<frameScores.length; k++) {
                temp[k] = frameScores[k];
            }
            temp[frameScores.length] = totalScore;
            frameScores = temp;
            state = ROLLING_FIRST_BALL;
        } else {
            System.out.println ("Invalid state: " + state);
            System.exit (1);
        }
        return frameScores;
    }
}
```

Appendix B: Refactored test code, plus Scorer code that handles any game with no strikes or spares (differences from Appendix A appear in boldface)

ScorerTest.java

```
import junit.framework.TestCase;

public class ScorerTest extends TestCase {

    public void testConstructor ( ) {
        Scorer s = new Scorer ( );
        assertEquals ("frame number for new Scorer is wrong", 1, s.frameNumber ( ));
        assertEquals ("score for new Scorer is wrong", 0, s.scoreSoFar ( ));
        assertFalse ("game is over for new Scorer", s.gameIsOver ( ));
    }

    public void testFourBalls ( ) {
        Scorer s = new Scorer ( );
        int [ ] balls = {1, 2, 4, 1};
        int [ ] lengths = {0, 1, 1, 2};
        int [ ] frames = {1, 2, 2, 3};
        int [ ] finalResult = {3, 8};
        testGame (s, balls, lengths, frames, finalResult);
    }

    private void testGame (Scorer s, int [ ] balls, int [ ] lengths, int [ ] frames,
      int [ ] finalResult) {
        int [ ] result;
        for (int k=0; k<balls.length; k++) {
            result = s.roll (balls[k]);
            assertEquals ("checking length of result of ball " + k,
                lengths[k], result.length);
            for (int j=0; j<result.length; j++) {
                assertEquals ("checking frame " + j + " in result of ball " + k,
                    finalResult[j], result[j]);
            }
            assertEquals ("checking frame number after ball " + k,
                frames[k], s.frameNumber ( ));
            if (lengths[k] == 0) {
                assertEquals ("checking score after ball " + k, 0, s.scoreSoFar());
            } else {
                assertEquals ("checking score after ball " + k,
                    result[result.length-1], s.scoreSoFar ( ));
            }
            assertFalse ("checking if game is over after ball " + k, s.gameIsOver ( ));
        }
    }
}
```

Scorer.java

```
public class Scorer {

    // the number of the frame in which the next ball will be rolled;
    // ranges from 1 to 11
    private int rollingFrame;

    // 0 if firstUnscoredFrame == 1,
    //   frameScores[firstUnscoredFrame] thereafter
    private int totalScore;

    // frameScores[k] == cumulative score up through frame k;
    private int [ ] frameScores;

    private int state;
    private static final int ROLLING_FIRST_BALL = 0;
    private static final int ROLLING_SECOND_BALL = 1;

    // the number of pins knocked down by the first ball
    //   in frame number rollingFrame
    private int firstBallInFrame;

    // indicates which frame is the last in the game
    private int lastFrameNumber;

    public Scorer ( ) {
        this (10);
    }

    public Scorer (int frameCount) {
        lastFrameNumber = frameCount;
        rollingFrame = 1;
        totalScore = 0;
        frameScores = new int [0];
        state = ROLLING_FIRST_BALL;
    }

    public int frameNumber ( ) {
        return rollingFrame;
    }

    public int scoreSoFar ( ) {
        return totalScore;
    }

    public boolean gameIsOver ( ) {
        return rollingFrame > lastFrameNumber;
    }

    public int [ ] roll (int ball) {
        if (state == ROLLING_FIRST_BALL) {
            firstBallInFrame = ball;
            state = ROLLING_SECOND_BALL;
        } else if (state == ROLLING_SECOND_BALL) {
            totalScore = totalScore + firstBallInFrame + ball;
            rollingFrame++;
            int [ ] temp = new int [frameScores.length+1];
            for (int k=0; k<frameScores.length; k++) {
```

```
            temp[k] = frameScores[k];
         }
         temp[frameScores.length] = totalScore;
         frameScores = temp;
         state = ROLLING_FIRST_BALL;
      } else {
         System.out.println ("Invalid state: " + state);
         System.exit (1);
      }
      return frameScores;
   }
}
```

Appendix C: Program that passes tests for handling spares and strikes correctly

Scorer.java

```
public class Scorer {

   // the number of the frame in which the next ball
   //   will be rolled; ranges from 1 to 11
   private int rollingFrame;

   // 0 if firstUnscoredFrame == 1,
   //   frameScores[firstUnscoredFrame] thereafter
   private int totalScore;

   // frameScores[k] == cumulative score up through frame k;
   private int [ ] frameScores;

   private int state;
   private static final int ROLLING_FIRST_BALL = 0;
   private static final int ROLLING_SECOND_BALL = 1;
   private static final int STRIKE_LAST_BALL = 2;
   private static final int TWO_CONSEC_STRIKES = 3;
   private static final int STRIKE_2_BALLS_AGO = 4;
   private static final int SPARE_LAST_BALL = 5;

   // the number of pins knocked down by the first ball
   //   in frame number rollingFrame
   private int firstBallInFrame;

   // indicates which frame is the last in the game
   private int lastFrameNumber;

   public Scorer ( ) {
      this (10);
   }

   public Scorer (int frameCount) {
      lastFrameNumber = frameCount;
      rollingFrame = 1;
      totalScore = 0;
      frameScores = new int [0];
      state = ROLLING_FIRST_BALL;
   }
```

```
public int frameNumber ( ) {
    if (state == SPARE_LAST_BALL && rollingFrame > lastFrameNumber) {
        return rollingFrame-1;
    } else if (state == STRIKE_LAST_BALL && rollingFrame > lastFrameNumber) {
        return rollingFrame-1;
    } else if (state == STRIKE_2_BALLS_AGO && rollingFrame > lastFrameNumber) {
        return rollingFrame-1;
    } else if (state == ROLLING_FIRST_BALL && rollingFrame > lastFrameNumber+1) {
        return rollingFrame-1;
    } else if (state == TWO_CONSEC_STRIKES && rollingFrame == lastFrameNumber+1) {
        return rollingFrame-1;
    } else if (state == TWO_CONSEC_STRIKES && rollingFrame > lastFrameNumber+1) {
        return rollingFrame-2;
    } else {
        return rollingFrame;
    }
}
public int scoreSoFar ( ) {
    if (frameScores.length == lastFrameNumber) {
        return frameScores[lastFrameNumber-1];
    } else {
        return totalScore;
    }
}
public boolean gameIsOver ( ) {
    return frameNumber ( ) > lastFrameNumber;
}
public int [ ] roll (int ball) {
    if (state == ROLLING_FIRST_BALL) {
        if (ball == 10) {
            rollingFrame++;
            state = STRIKE_LAST_BALL;
        } else {
            firstBallInFrame = ball;
            state = ROLLING_SECOND_BALL;
        }
    } else if (state == ROLLING_SECOND_BALL) {
        if (firstBallInFrame + ball == 10) {
            rollingFrame++;
            state = SPARE_LAST_BALL;
        } else {
            totalScore = totalScore + firstBallInFrame + ball;
            rollingFrame++;
            addFrame ( );
            state = ROLLING_FIRST_BALL;
        }
    } else if (state == SPARE_LAST_BALL) {
        totalScore = totalScore + 10 + ball;
        addFrame ( );
        if (ball == 10) {
            rollingFrame++;
            state = STRIKE_LAST_BALL;
        } else {
            firstBallInFrame = ball;
            state = ROLLING_SECOND_BALL;
        }
```

```
        } else if (state == STRIKE_LAST_BALL) {
            if (ball == 10) {
                rollingFrame++;
                state = TWO_CONSEC_STRIKES;
            } else {
                firstBallInFrame = ball;
                state = STRIKE_2_BALLS_AGO;
            }
        } else if (state == TWO_CONSEC_STRIKES) {
            totalScore = totalScore + 20 + ball;
            addFrame ( );
            if (ball == 10) {
                rollingFrame++;
            } else {
                firstBallInFrame = ball;
                state = STRIKE_2_BALLS_AGO;
            }
        } else if (state == STRIKE_2_BALLS_AGO) {
            totalScore = totalScore + 10 + firstBallInFrame + ball;
            addFrame ( );
            if (firstBallInFrame + ball == 10) {
                rollingFrame++;
                state = SPARE_LAST_BALL;
            } else {
                totalScore = totalScore + firstBallInFrame + ball;
                rollingFrame++;
                addFrame ( );
                state = ROLLING_FIRST_BALL;
            }
        } else {
            System.out.println ("Invalid state: " + state);
            System.exit (1);
        }
        return frameScores;
    }

    private void addFrame ( ) {
        if (frameScores.length < lastFrameNumber) {
            int [ ] temp = new int [frameScores.length+1];
            for (int k=0; k<frameScores.length; k++) {
                temp[k] = frameScores[k];
            }
            temp[frameScores.length] = totalScore;
            frameScores = temp;
        }
    }
}
```

Appendix D: Final versions of the ScorerTest and Scorer classes

ScorerTest.java

```
import junit.framework.TestCase;

public class ScorerTest extends TestCase {

    public void testConstructor ( ) {
        Scorer s = new Scorer ( );
        assertEquals ("frame number for new Scorer is wrong", 1, s.frameNumber ( ));
        assertEquals ("score for new Scorer is wrong", 0, s.scoreSoFar ( ));
        assertFalse ("game is over for new Scorer", s.gameIsOver ( ));
    }

    public void testFourBalls ( ) {
        Scorer s = new Scorer (2);
        int [ ] balls = {1, 2, 4, 1};
        int [ ] lengths = {0, 1, 1, 2};
        int [ ] frames = {1, 2, 2, 3};
        int [ ] finalResult = {3, 8};
        testGame (s, balls, lengths, frames, finalResult);
    }

    public void testSpareMissMiss ( ) {
        Scorer s = new Scorer (3);
        int [ ] balls = {6, 4, 3, 4, 5, 2};
        int [ ] lengths = {0, 0, 1, 2, 2, 3};
        int [ ] frames = {1, 2, 2, 3, 3, 4};
        int [ ] finalResult = {13, 20, 27};
        testGame (s, balls, lengths, frames, finalResult);
    }

    public void testMissSpareMiss ( ) {
        Scorer s = new Scorer (3);
        int [ ] balls = {6, 3, 4, 6, 5, 2};
        int [ ] lengths = {0, 1, 1, 1, 2, 3};
        int [ ] frames = {1, 2, 2, 3, 3, 4};
        int [ ] finalResult = {9, 24, 31};
        testGame (s, balls, lengths, frames, finalResult);
    }

    public void testMissMissSpare ( ) {
        Scorer s = new Scorer (3);
        int [ ] balls = {6, 3, 3, 4, 8, 2, 4};
        int [ ] lengths = {0, 1, 1, 2, 2, 2, 3};
        int [ ] frames = {1, 2, 2, 3, 3, 3, 4};
        int [ ] finalResult = {9, 16, 30};
        testGame (s, balls, lengths, frames, finalResult);
    }

    public void testSpareSpareMiss ( ) {
        Scorer s = new Scorer (3);
        int [ ] balls = {6, 4, 3, 7, 5, 2};
        int [ ] lengths = {0, 0, 1, 1, 2, 3};
        int [ ] frames = {1, 2, 2, 3, 3, 4};
        int [ ] finalResult = {13, 28, 35};
        testGame (s, balls, lengths, frames, finalResult);
    }
```

```
public void testMissSpareSpare ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 3, 3, 7, 8, 2, 4};
    int [ ] lengths = {0, 1, 1, 1, 2, 2, 3};
    int [ ] frames = {1, 2, 2, 3, 3, 3, 4};
    int [ ] finalResult = {9, 27, 41};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testSpare0SpareMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 4, 0, 10, 5, 2};
    int [ ] lengths = {0, 0, 1, 1, 2, 3};
    int [ ] frames = {1, 2, 2, 3, 3, 4};
    int [ ] finalResult = {10, 25, 32};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testMissStrikeMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 2, 10, 5, 4};
    int [ ] lengths = {0, 1, 1, 1, 3};
    int [ ] frames = {1, 2, 3, 3, 4};
    int [ ] finalResult = {8, 27, 36};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testSpareStrikeMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {7, 3, 10, 5, 4};
    int [ ] lengths = {0, 0, 1, 1, 3};
    int [ ] frames = {1, 2, 3, 3, 4};
    int [ ] finalResult = {20, 39, 48};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testStrikeSpareSpareStrike ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 7, 3, 8, 2, 10};
    int [ ] lengths = {0, 0, 1, 2, 2, 3};
    int [ ] frames = {2, 2, 3, 3, 3, 4};
    int [ ] finalResult = {20, 38, 58};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testStrikeMissStrikeSpare ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 7, 2, 10, 6, 4};
    int [ ] lengths = {0, 0, 2, 2, 2, 3};
    int [ ] frames = {2, 2, 3, 3, 3, 4};
    int [ ] finalResult = {19, 28, 48};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testStrikeMissStrikeMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 7, 2, 10, 6, 3};
    int [ ] lengths = {0, 0, 2, 2, 2, 3};
    int [ ] frames = {2, 2, 3, 3, 3, 4};
    int [ ] finalResult = {19, 28, 47};
    testGame (s, balls, lengths, frames, finalResult);
}
```

```
public void testStrike00Miss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 0, 0, 4, 3};
    int [ ] lengths = {0, 0, 2, 2, 3};
    int [ ] frames = {2, 2, 3, 3, 4};
    int [ ] finalResult = {10, 10, 17};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testStrike60miss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 6, 0, 4, 3};
    int [ ] lengths = {0, 0, 2, 2, 3};
    int [ ] frames = {2, 2, 3, 3, 4};
    int [ ] finalResult = {16, 22, 29};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testStrike06miss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 0, 6, 4, 3};
    int [ ] lengths = {0, 0, 2, 2, 3};
    int [ ] frames = {2, 2, 3, 3, 4};
    int [ ] finalResult = {16, 22, 29};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testAll5Strikes ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 10, 10, 10, 10};
    int [ ] lengths = {0, 0, 1, 2, 3};
    int [ ] frames = {2, 3, 3, 3, 4};
    int [ ] finalResult = {30, 60, 90};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testStrikeStrikeMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {10, 10, 3, 4};
    int [ ] lengths = {0, 0, 1, 3};
    int [ ] frames = {2, 3, 3, 4};
    int [ ] finalResult = {23, 40, 47};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testMissStrikeStrikeSpare ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 4, 10, 10, 3, 7};
    int [ ] lengths = {0, 0, 1, 1, 2, 3};
    int [ ] frames = {1, 2, 3, 3, 3, 4};
    int [ ] finalResult = {20, 43, 63};
    testGame (s, balls, lengths, frames, finalResult);
}
public void testMissStrikeStrikeMiss ( ) {
    Scorer s = new Scorer (3);
    int [ ] balls = {6, 4, 10, 10, 3, 4};
    int [ ] lengths = {0, 0, 1, 1, 2, 3};
    int [ ] frames = {1, 2, 3, 3, 3, 4};
    int [ ] finalResult = {20, 43, 60};
    testGame (s, balls, lengths, frames, finalResult);
}
```

```
    private void testGame (Scorer s, int [ ] balls, int [ ] lengths, int [ ] frames,
      int [ ] finalResult) {
        int [ ] result;
        for (int k=0; k<balls.length; k++) {
            result = s.roll (balls[k]);
            assertEquals ("checking length of result of ball " + k,
                lengths[k], result.length);
            for (int j=0; j<result.length; j++) {
                assertEquals ("checking frame " + j + " in result of ball " + k,
                    finalResult[j], result[j]);
            }
            assertEquals ("checking frame number after ball " + k,
                frames[k], s.frameNumber ( ));
            if (lengths[k] == 0) {
                assertEquals ("checking score after ball " + k, 0, s.scoreSoFar());
            } else {
                assertEquals ("checking score after ball " + k,
                    result[result.length-1], s.scoreSoFar ( ));
            }
            assertEquals ("checking if game is over after ball " + k,
                k == balls.length-1, s.gameIsOver ( ));
        }
    }
}
```

Scorer.java

```
public class Scorer {

    // the number of the frame in which the next ball
    //   will be rolled; ranges from 1 to 11
    private int rollingFrame;

    // 0 if firstUnscoredFrame == 1,
    //   frameScores[firstUnscoredFrame] thereafter
    private int totalScore;

    // frameScores[k] == cumulative score up through frame k;
    private int [ ] frameScores;

    private int state;
    private static final int ROLLING_FIRST_BALL = 0;
    private static final int ROLLING_SECOND_BALL = 1;
    private static final int STRIKE_LAST_BALL = 2;
    private static final int TWO_CONSEC_STRIKES = 3;
    private static final int STRIKE_2_BALLS_AGO = 4;
    private static final int SPARE_LAST_BALL = 5;

    // the number of pins knocked down by the first ball
    //   in frame number rollingFrame
    private int firstBallInFrame;

    // indicates which frame is the last in the game
    private int lastFrameNumber;

    public Scorer ( ) {
        this (10);
    }
```

```
public Scorer (int frameCount) {
    lastFrameNumber = frameCount;
    rollingFrame = 1;
    totalScore = 0;
    frameScores = new int [0];
    state = ROLLING_FIRST_BALL;
}

public int frameNumber ( ) {
    if (frameScores.length == lastFrameNumber) {
        return lastFrameNumber+1;// game is over
    } else if (rollingFrame > lastFrameNumber) {
        return lastFrameNumber;// we're in last frame
    } else {
        return rollingFrame;
    }
}

public int scoreSoFar ( ) {
    if (frameScores.length == lastFrameNumber) {
        return frameScores[lastFrameNumber-1];
    } else {
        return totalScore;
    }
}

public boolean gameIsOver ( ) {
    return frameNumber ( ) > lastFrameNumber;
}

public int [ ] roll (int ball) {
    if (state == ROLLING_FIRST_BALL) {
        if (ball == 10) {
            rollingFrame++;
            state = STRIKE_LAST_BALL;
        } else {
            firstBallInFrame = ball;
            state = ROLLING_SECOND_BALL;
        }
    } else if (state == ROLLING_SECOND_BALL) {
        if (firstBallInFrame + ball == 10) {
            rollingFrame++;
            state = SPARE_LAST_BALL;
        } else {
            rollingFrame++;
            addFrame (firstBallInFrame + ball);
            state = ROLLING_FIRST_BALL;
        }
    } else if (state == SPARE_LAST_BALL) {
        addFrame (10 + ball);
        if (ball == 10) {
            rollingFrame++;
            state = STRIKE_LAST_BALL;
        } else {
            firstBallInFrame = ball;
            state = ROLLING_SECOND_BALL;
        }
```

```
        } else if (state == STRIKE_LAST_BALL) {
            if (ball == 10) {
                rollingFrame++;
                state = TWO_CONSEC_STRIKES;
            } else {
                firstBallInFrame = ball;
                state = STRIKE_2_BALLS_AGO;
            }
        } else if (state == TWO_CONSEC_STRIKES) {
            addFrame (20 + ball);
            if (ball == 10) {
                rollingFrame++;
            } else {
                firstBallInFrame = ball;
                state = STRIKE_2_BALLS_AGO;
            }
        } else if (state == STRIKE_2_BALLS_AGO) {
            addFrame (10 + firstBallInFrame + ball);
            if (firstBallInFrame + ball == 10) {
                rollingFrame++;
                state = SPARE_LAST_BALL;
            } else {
                rollingFrame++;
                addFrame (firstBallInFrame + ball);
                state = ROLLING_FIRST_BALL;
            }
        } else {
            System.out.println ("Invalid state: " + state);
            System.exit (1);
        }
        return frameScores;
    }

    private void addFrame (int toAdd) {
        totalScore = totalScore + toAdd;
        if (frameScores.length < lastFrameNumber) {
            int [ ] temp = new int [frameScores.length+1];
            for (int k=0; k<frameScores.length; k++) {
                temp[k] = frameScores[k];
            }
            temp[frameScores.length] = totalScore;
            frameScores = temp;
        }
    }
}
```