Bowling Scorekeeper

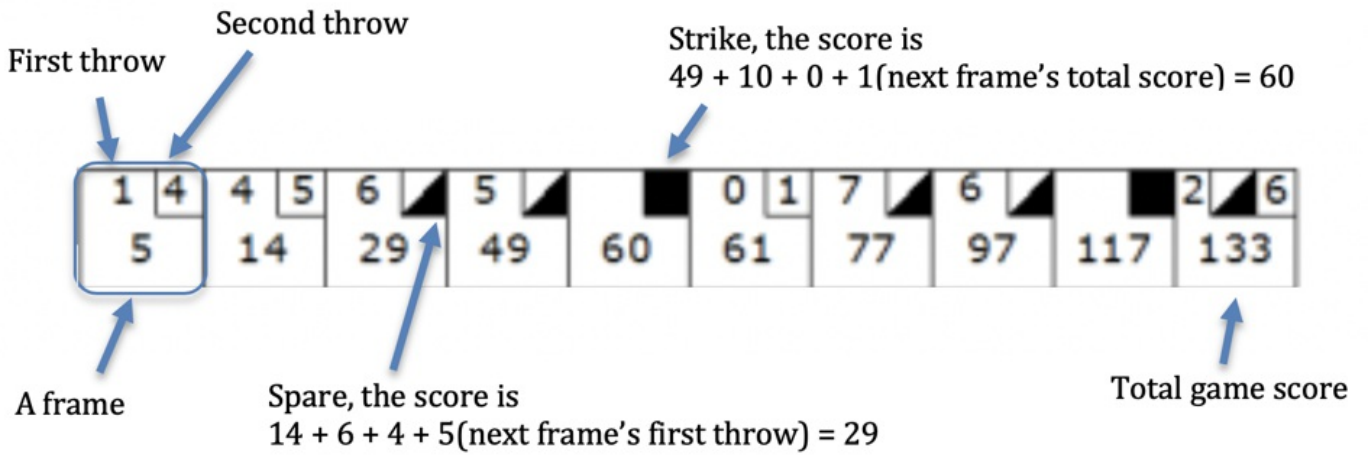
The objective is to TEST an application that can calculate the score of a single bowling game.

- There is no graphical user interface.
- You work ONLY with JUnit test cases in this project.
- You have ONE HOUR to work on this project (if needed, you may exceed this limit by up to 20%).
- You are free to consult/use any online resources, including documentations, tutorials, Q&A sites, and any Eclipse built-in tools or plug-ins.

Project Template
You are provided with a completed project that contains three classes: `Frame`, `BowlingGame` and `BowlingException`, each contains some fields and methods. DO NOT CHANGE the names and functionalities of the existing fields and methods.

You are expected to create JUnit test cases to verify the behavior of this implementation as thorough as possible based on the following description of a bowling score keeper. Your program should throw `BowlingException` in all error situations.

Bowling Score Keeper Task Description
The game consists of 10 frames as shown below. In each frame the player has two opportunities to knock down 10 pins. The score for the frame is the total number of pins knocked down, plus bonuses for strikes and spares.



A spare is when the player knocks down all 10 pins in two throws. The bonus for that frame is the number of pins knocked down by the next throw. So in frame 3 of the example game below, the score is 10 (the total number knocked down) plus a bonus of 5 (the number of pins knocked down on the next throw).

A strike is when the player knocks down all 10 pins on his first try. The bonus for that frame is the value of the next two throws.

In the tenth frame, a player who rolls a spare or strike is allowed to have bonus throws to complete the frame. However, no more than three balls can be rolled in tenth frame.

Examples
Click for Detailed Rules and Examples of Scoring a Game of Bowling

Online Bowling Game Score Calculator

Project Requirements

- Req 1: A frame consists of two throws.
- Req 2: The possible points for the first throw range from 0 to 10.
- Req 3: The possible points for the second throw range from 0 to 10.
- Req 4: The score of a frame is the sum of two throws and it ranges from 0 to 10.
- Req 5: A frame is a *strike* if the first throw is 10 points.
- Req 6: A strike frame receives bonus score of the value of the next two throws.
- Req 7: A frame is a *spare* if its score is 10 and it is not a strike.
- Req 8: A spare frame receives bonus score of the value of the next throw.
- Req 9: A bowling game consists of ten frames.
- Req 10: The score of a game is the sum of all frames.
- Req 11: If the tenth frame is a spare, there is one bonus throw.
- Req 12: If the tenth frame is a strike, there are two bonus throws.
- Req 13: The bonus throw(s) is added to the game score.

Checklists

Important Note: You are *required* to use the following checklists to improve the quality of your test suite. The screen recording you submit should demonstrate your compliance with this requirement.

Test Case Checklist

Each test case *should*:

- ☐ be executable (i.e., it has an **@Test** annotation and can be run via "Run as JUnit Test")
- ☐ have at least one assert statement or assert an exception is thrown. Example assert statements include: **assertTrue**, **assertFalse**, and **assertEquals** (click for tutorials). For asserting an exception is thrown, use **assertThrows** in JUnit 5 (click for tutorials).

- [ ] evaluate/test only one method

Each test case *could*:

- [ ] be descriptively named and commented
- [ ] If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using **@BeforeEach**)
- [ ] If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior.

Test Suite Checklist

The test suite *should*:

- [ ] have at least one test for each requirement
- [ ] appropriately use the setup and teardown code (e.g., **@BeforeEach**, which runs before each **@Test**)
- [ ] contain a fault-revealing test for each bug in the code (i.e., a test that fails)
- [ ] For each requirement, contain test cases for:
    - [ ] Valid inputs
    - [ ] Boundary cases
    - [ ] Invalid inputs
    - [ ] Expected exceptions

To improve the test suite, you *could*:

- [ ] measure code coverage using an appropriate tool, such as EclEmma (installation, tutorial). Inspect uncovered code and write tests as appropriate.