Assignment 1: Conway's Game of Life

This assignment uses the example of Conway's Game of Life to practice some object-oriented programming basics.

Generic Rubric

- 40% task completion: all requirements implemented
- 30% code correctness: code is correct, demonstrated with unit tests
- 20% style: code is well-organized, follows coding conventions, and is easy to read
- 10% documentation: code is adequately documented

Task 1 (25 points)

Implementation

Implement a class `Matrix` that has a `private` field `data` to hold a two-dimensional array of `int`. Implement public setters and getters for the `data` field. Implement a `public` method that returns the number of rows in the matrix (first dimension of the array), and another `public` method to return the number of columns (second dimension of the array). Implement a constructor that takes the number of rows and the number of columns as arguments and initializes the array field; make sure to check that there is at least 1 row and 1 column in the matrix (raise an `IllegalArgumentException` if it doesn't). Implement an overloaded constructor that takes a two-dimensional array as argument and initializes the field `data`; make sure that the argument is not `null` and has at least 1 row and 1 column (raise an `IllegalArgumentException` if it doesn't).

Unit Testing

Implement a `MatrixTests` class using JUnit and test that

- the constructor reports illegal arguments
- the row and column methods return the expected number of rows and columns

Task 2 (25 points)

Implementation

Implement an `interface` named `MatrixPrinter` that defines the signature of a method with a `Matrix` argument and returns a `String`. Implement a class `MatrixOutlinePrinter` that prints the outline of a matrix. The example below shows the outline of a matrix with 2 rows and 4 columns. Implement printing the content of a row of the matrix in its own `protected` method (here it prints a space per cell, later we will override this behavior).

```
+----+
|    |
|    |
+----+
```

Implement a class `BoolMatrixPrinter` that extends `MatrixOutlinePrinter` and prints a matrix whose values are either `0` or `1`. Override the `protected` method that prints a row in order to print values; make sure to check that the matrix indeed contains only `0` or `1` (raise an `IllegalArgumentException` if it doesn't). The example below shows output for a matrix with 2 rows and 4 columns; the first and second row have the same entries `0`, `0`, `1`, and `0`.

```
+----+
|  # |
|  # |
+----+
```

Unit Testing

Implement a class `MatrixPrinterTests` using JUnit and test that

- the `OutlineMatrixPrinter` prints an outline of the appropriate size when handed a matrix with 3 rows and 4 columns. Use the `assertEquals` functionality of JUnit to check for the expected `String` result.
- the `BoolMatrixPrinter` raises an exception if handed an illegal matrix (that contains numbers other than `0` or `1`).
- the `BoolMatrixPrinter` correctly turns non-empty matrices into strings. Test with 2 different matrices of your choice. Use the `assertEquals` functionality of JUnit to check for the expected `String` results.

Task 3 (25 points)

Implement a `Shape` class that extends class `Matrix` and adds a private field `name`. Implement a constructor that takes a name and a two-dimensional matrix whose values are only `0` and `1`; check this requirement in the constructor (raise an `IllegalArgumentException` if it doesn't).

Showcase your class by creating two instances of the `Shape` class: a Beehive and a Boat. Print the instances using a `BoolMatrixPrinter`.

Task 4 (25 points)

Implementation

1. Implement a class `GameOfLife` that extends `Matrix` to represent a game field.
2. Implement a method `addShape` that takes a shape instance and a coordinate to place a shape onto the game field; check in the method that the shape fits fully onto the game field before placing it (raise an `IllegalArgumentException` if it doesn't).
3. Implement a method `step` following the game rules to initialize the next generation of the game in a local two-dimensional array; once computed, change the inherited `data` field using its setter.

Showcase your game class by creating at least the following two instances of the `Shape` class in a `main` method: a Blinker and a Glider. Any of the states of the dynamic shape can be used as starting shape. For example, you can instantiate a Blinker using any of the two following shapes:

```
+---+      +---+
| # |      |   |
| # |      |###|
| # |      |   |
+---+      +---+
```

Add any additional number and type of shapes that you like.

Add the shape instances to an instance of `GameOfLife` and implement an infinite `while` loop that calls `step` and then uses an instance of the `BoolMatrixPrinter` to print the game state to a `String` and then to the console using `System.out.println`. You should see an animation with the Blinker oscillating between its two states, and the Glider moving over the game field.

Unit Testing

Implement a class `GameOfLifeTests` using JUnit; initialize a game with a Blinker and a Glider and test that

- after one step the game field changed as expected
- after 10 steps the game field changed as expected

Submission Guidelines

Submit the following 9 files

- Matrix.java
- MatrixTests.java
- MatrixPrinter.java
- MatrixOutlinePrinter.java
- BoolMatrixPrinter.java
- MatrixPrinterTests.java
- Shape.java
- GameOfLife.java
- GameOfLifeTests.java