# Question 1: A Major Class With 2D Arrays (70%)

In this question, you will complete a program to play **Flipper**: a game somewhat similar to Reversi (and Othello).

## *The Game Rules for Flipper*

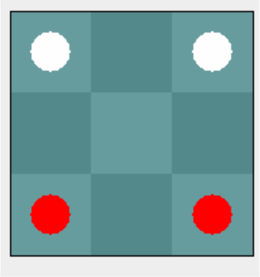
**Goal**: To have the greatest number of pegs on the board at the end of the game.

**Setup**: An $n \times n$ board is used. The pegs in the top left and right corners start with player one's colour. The pegs in the bottom left and right corner start with the other player's colour.

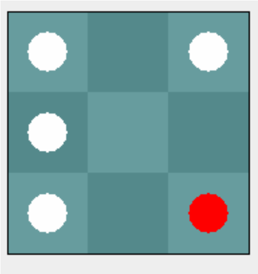
**Play**: Each player takes turns making a move. A move is when a player places a peg of their colour in an empty square directly adjacent (horizontally, vertically or diagonally) to where another one of their pegs is located. If there are any pegs of the opponents colour directly adjacent (horizontally, vertically or diagonally) to the newly paced peg, those pegs switch colour.

**Finishing**: The game stops immediately when the next player does not have a valid move. This can result from the board being completely filled, all of the player's pegs being eliminated, or all of the player's pegs being completely surrounded.

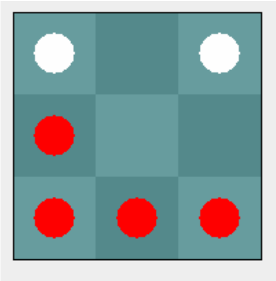
**Demonstration**: Here is a sample game played on a game board of size 3.



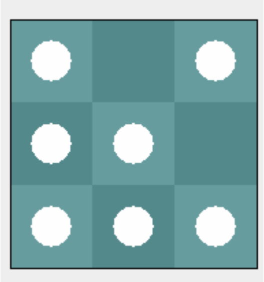
**Initial setup.**



**First move**: White places a peg directly below the top left white peg. This turns red's peg in the bottom left into a white peg.

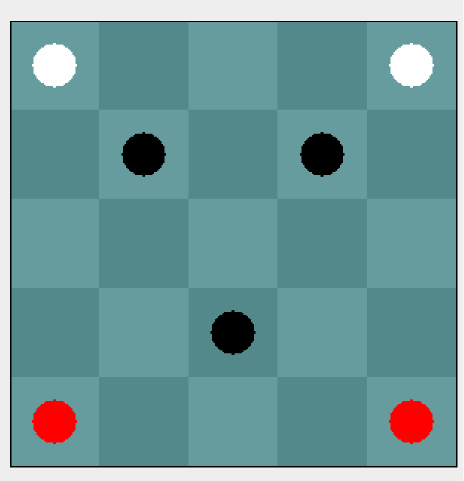


**Second move:** Red puts a peg in the middle square on the bottom row, turning both neighbouring white pegs red.



**Third move**: White places a peg in the middle square turning all of red's pegs white and winning the game.

**Variation**: To make the game more interesting, some of the squares can be made unusable by placing a black peg in the square. Neither player can place a peg in these squares, nor can the black peg ever switch colours. Placing more of these unusable pegs near player one's starting pegs can help mitigate the advantage of going first. The following setup has 3 unusable squares on it.



## *Program Design*

The program is divided into three parts:

- `GameDriver.java` or `GameVariationDriver.java` – The program driver
- `Game.java` – Implements the game rules, and manages the board

We have supplied the implementation for the drivers. Your task is to complete the other file which implements the flipper game.

**The Game Engine Class**
`Game.java` contains the basic functionality for the game. It maintains the state of the game in a 2-D array, manages a board object and implements all of the rules for the game. The following is a Class diagram for the class:

| Game |
|---|
| `- Board pegboard`<br>`- int[][] pegs`<br>`- int turn` |
| `+ Game( int size )`<br>`+ Game( int size, Coordinate[] squares )`<br>`+ void playGame()`<br>`- void drawBoard()`<br>`- boolean isGameOver()`<br>`- Coordinate getValidMove()`<br>`- void performMove( Coordinate move )`<br>`- void displayFinalMessage()`<br>`- boolean isValidMove( int row, int col )`<br>`- int numPegs( int player )` |

Note that outside of the constructors there is only one public method which plays the game. All the rest of the methods are private helper methods that take care of one portion of the gameplay.

Some of the methods are already completed for you. You should not modify them. You need only implement the remaining methods. Read the pre/post conditions to help you understand them. You are allowed (and encouraged) to make more helper methods.

Note that this might seem like a very big task. However, stepwise refinement has been used in the design of the class to break the big problem (playing the game) into smaller, more manageable pieces. You have to write several small pieces of code, but, in the end, you will have a fun program completed.

### The Driver Classes

`GameDriver.java` runs the program using the basic rules, with no unusable squares. It simply prompts the user for a single number corresponding to the size of the game board, and then creates the `Game` object and starts the game.

`GameVariationDriver.java` runs the program using the rules variation that allows for unusable squares. After reading in the size of the game board, it prompts for the number of unusable squares, and then reads in the coordinates (row then column, indexed starting from zero) for each square. For instance the input to get the board configuration shown in the picture from the rules section would be:

```
5    ←———— size
3    ←———— number of unusable squares
1 1  ⎫
1 3  ⎬———  coordinates of each unusable square
3 2  ⎭
```

The driver then creates the `Game` object with those unusable squares before starting the game.

## *Program Implementation*

You must use the **public** method signatures as given in the tables above. This ensures we can properly test your work. You may not add any additional **public** methods to the classes you modify. You are free to add as many **private** helper methods as you require.

You must leave the provided drivers alone. If you do modify them, the files you submit may not work with the original drivers.

You should develop your solution in stages, rather than attempting to put it together in one shot. A reasonable start would be to get the game to display the initial starting position.

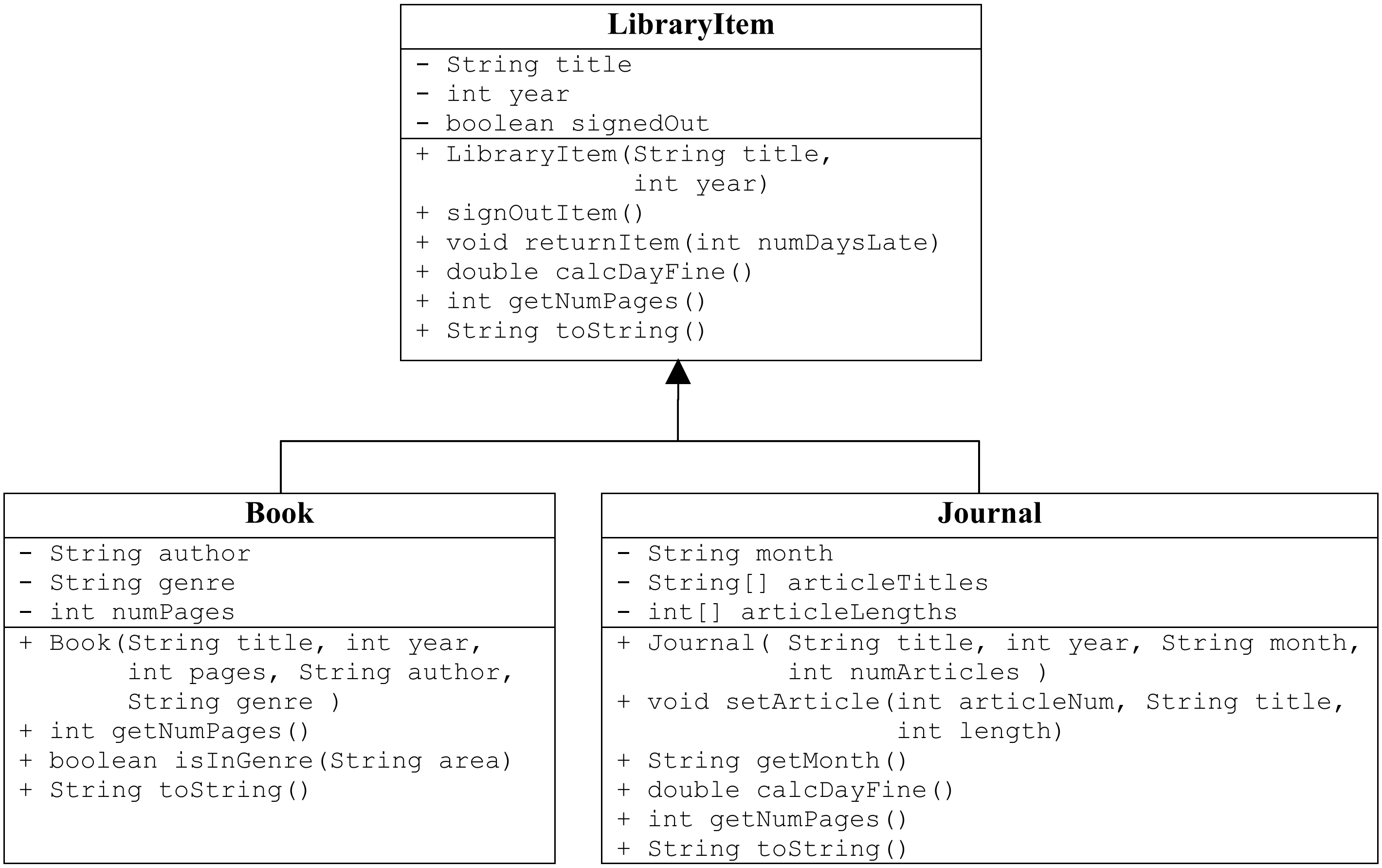
We have given you two drivers for the game so that you can initially get the game working with the basic rules. It is then a smaller jump to add in the extra functionality required to have the game support the unusable square variation. You would be better off submitting a proper solution to the basic game than an unworking solution to the variation.

## *Submission*

**Files to submit**: `Game.java`

## Question 2: Inheritance (30%)

A library wants to track two types of items in its collection, Books and Journals.  Both are very similar (for instance they both have titles, and publication years, and they both can be borrowed or returned) but do have some differences (for instance journals have a publication month, and a list of articles in that issue).  The following is a class diagram showing the relationship of these two classes with a super class called **`LibraryItem`**.



**LibraryItem**

```
- String title
- int year
- boolean signedOut
```

```
+ LibraryItem(String title,
              int year)
+ signOutItem()
+ void returnItem(int numDaysLate)
+ double calcDayFine()
+ int getNumPages()
+ String toString()
```

**Book**

```
- String author
- String genre
- int numPages
```

```
+ Book(String title, int year,
       int pages, String author,
       String genre )
+ int getNumPages()
+ boolean isInGenre(String area)
+ String toString()
```

**Journal**

```
- String month
- String[] articleTitles
- int[] articleLengths
```

```
+ Journal( String title, int year, String month,
           int numArticles )
+ void setArticle(int articleNum, String title,
                  int length)
+ String getMonth()
+ double calcDayFine()
+ int getNumPages()
+ String toString()
```

Note that all of the public methods from the **`LibraryItem`** class are inherited by the two subclasses, but some methods are overridden by both (the **`toString`** and **`getNumPages`** operations), while some are only overridden in one (the **`calcDayFine`** method is only overridden in **`Journal`**).  Both subclasses also introduce methods which are unique to that class (the **`isInGenre`** method in **`Book`**, and the **`getMonth`** and **`setArticle`** methods in **`Journal`**).

You are given complete implementations of **`LibraryItem`** and **`Book`**. You need to complete and submit the class **`Journal`**. You are given a very basic starter file for that class, including documentation for all methods (but not method signatures), and must complete all the methods listed in the class diagram.

You are also given a class called **`Library`** which maintains an array of **`LibraryItem`**s. You must complete the three methods that are not implemented in the class. You are not allowed to modify any of the other methods, or add new instance variables or create any new methods. We have provided a **`TestLibrary`** class to call the various **`Library`** methods.

Note that you must complete the methods so that they work for any collection of **`LibraryItem`** objects, not just the one set created in the provided **`TestLibrary`** class. In particular, do not assume that the **`Journal`** objects are in specific positions of the array when writing the last method.

On the following page is the expected output for the provided **`TestLibrary`** class.

**Files to Submit**: **`Journal.java`**, **`Library.java`**

```
> java TestLibrary
Expect: <no output should appear>
Actual:

Expect: Overdue fees: 1.25
Actual: Overdue fees: 1.25

Expect: Overdue fees: 2.0
Actual: Overdue fees: 2.0

Expect: 4 items should be printed
Actual:
Available
Harry Potter and the Goblet of Fire, Pages: 734
2000
Rowling, J.K.

Available
Journal of CS, Pages: 800
2007-Feb
  1. Java is Better Than Scheme! (1-400)
  2. Scheme is Better Than Java! (401-800)

Signed Out
Harry Potter and the Deathly Hallows, Pages: 784
2007
Rowling, J.K.

Available
Programming Scheme, Pages: 550
2006
Smith, John
```

1st Library Item
(A book)

2nd Library Item
(A Journal)

3rd Library Item
(A book)

4th Library Item
(A book)

The four Library
Items that are
over 525 pages

```
Expect: 2 books should be printed
Actual:
Available
Harry Potter and the Goblet of Fire, Pages: 734
2000
Rowling, J.K.

Signed Out
Harry Potter and the Deathly Hallows, Pages: 784
2007
Rowling, J.K.


Expect: 2 journals should be printed
Actual:
Available
Journal of Math, Pages: 35
2007-Feb
  1. Why is pi Good? (1-10)
  2. Reading the cosins (11-30)
  3. Three-sided Squares? (31-35)

Available
Journal of CS, Pages: 800
2007-Feb
  1. Java is Better Than Scheme! (1-400)
  2. Scheme is Better Than Java! (401-800)
```