Learning Objectives:

1. Become able to conceptualize and program, test, debug and perfect simple algorithms that involve and utilize combinations of the following:
   - An int array, confusing because the entries are small non-negative numbers, so an index and an entry must be carefully distinguished.
   - Searching that distinguishes failure from success, and use of the location where the item is found in the case of success.
   - Index calculations related to directions on a line (left and right).
   - Decisions for (a) validating user input, (b) checking situations involving the array content, and (c) testing if a calculated index is within the array bounds.
   - Counting and decision making based on a count.
2. Become able to program such algorithms in Java.
3. Become able to plan the project and solve multiple problems to complete a 150-250 line Java program.

Specification:

Your program will be a Java Application named OneDimCheckersSolitaire. It simulates a game board consisting of 10 positions in a line. Each position holds an integer from 1 to 9 inclusively or the integer 0. The integers 1 to 9 represent distinct game pieces called checkers; no two checkers are the same number. Zero in a position means it is empty, that is, there is no checker in it. Each non-empty position holds exactly one checker.

At the beginning, the user is prompted to specify how many checkers to start with by typing an integer from 2 to 9 inclusively. The program places consecutively checkers numbered 1, 2, ..., up to the number of checkers to start (nStart) in the first nStart positions. If the user types an integer not in this range, a message is printed and the user is prompted to try again. (A crash is OK if the input is badly formatted as an integer.) When a valid number is provided, the program prints the starting board, with one space between each number. So, if say the user starts with 8 checkers, the program prints:

```
1 2 3 4 5 6 7 8 0 0
```

The program prompts for, inputs and performs moves that the user inputs. A move is expressed by the checker number, a space and then one of the letters R or L. A move means that the player or computer tries to move the checker with the given number in the direction, Right or Left, given by the letter (R or L respectively).

The movement of a checker follows the following rules:

- If the position right next to the checker's current position in the given direction R or L is in the board and is empty, the checker is moved there. So the current position becomes empty.
If that position (indicated above) in not-empty and the position one more step in the given R or L direction is in the board and is empty, the checker "jumps" the checker in that position. This means (a) That checker is moved from its original position, so the original position becomes empty, (b) the "jumped" checker is removed from the board, so its number is gone permanently and the position becomes empty, and (c) the moving checker is placed in the previously empty position.

It is OK if the program crashes if the user inputs something other than an integer number, a space and then a single letter.

If the user input is an integer, a space and then a letter, the program reprints them on the next line without the space.

The program tries to make the given move. If it is impossible according to the rules for any reason, including the checker or the destination position or the position to be jumped not being on the board, the program prints

```
Bad Move
```

and then repeats the prompt and the following steps.

If the move is possible, the program performs it and prints the new board configuration.

If there is only one checker left, the program prints End of Game and exits. So, the last configuration it prints has exactly one checker.

Example (user input is printed in **boldface**):

```
Number of checkers: **8**
1 2 3 4 5 6 7 8 0 0
Move (number space [RL]): **8 L**
8L
Bad Move
Move (number space [RL]): **8 R**
8R
1 2 3 4 5 6 7 0 8 0
Move (number space [RL]): **6 R**
6R
1 2 3 4 5 0 0 6 8 0
```

**Some Ingredients:**

1. Construct and refer to the board array: static int[ ] board = new int[10];
2. Inputting: import java.util.*; Construct and refer to a Scanner:

```
static Scanner sc = new Scanner(System.in);
```

Declare and input an integer:
```
int myInt;
myInt = sc.nextInt();
```

Declare and input a string, for which the user should type one letter:
```
String myInString;
myInString = sc.next();
```
3. Test if a string equals another string, such as a single letter string "R":
   
   if( myInString.equals("R") )
   {
      // code for does equal case
   } 
   else
   {
      // code for does not equal case
   }

4. Boolean operators: ! (not), && (and), || (two vertical bars, or) Read about them and use them to shorten your code. Also, ! may be applied to a parenthesized expression. Remember too that == compares things like numbers (but not Strings!) for equality and != is its negation ("not equals").

5. Loop that finishes either after a scan is complete or after an item is found:
   
   boolean FOUND = false;
   int i = 0;
   while( !FOUND && i < 10 )
   {
      if ( board[ i ] is what you are looking for )
      {
         FOUND = true;
      }
      else
      {
         i = i + 1
      }
   }

   FOUND equals (is) true if the item is found, and in that case, i equals the position where it was found. So, if FOUND is false (test with say if( !FOUND ), equivalently, if (FOUND==false) ), then the program should ignore i and do something different from what it would do when the item is found.

6. If you have an idea of what you want to program but don't know how, or your attempts are unsuccessful (with syntax, runtime or semantic errors you don't quickly account for and fix), SEEK HELP before spending excessive time (say 20 minutes.)

7. You might organize your program like those in the first half or so of Downey's book. Declare appropriate variables like the count of checkers, the board array reference, and other references like the reference to the Scanner as static inside the class, so that different methods can share them. Make all the methods static. Remember that execution starts in the main method, which must be declared public static void main(String a[ ] )  The only leeway you have here is in the name "a" for the argument list parameter.

HOWEVER, if you want to practice making the game into a class that can be instantiated multiple times or perhaps be adapted into an applet eventually, that is ok if you can get it to work (no extra credit).