Domineering

By the time you are done with this activity, you should be able to:

- use multidimensional arrays.
- develop a simple graphic user interface (GUI).
- write individual methods, manually testing each one.

Playing the Game

Included in the Domineering module is a file Domineering.jar. This is a compiled version of the working game. To play it, use a terminal to navigate to the directory containing the file and type this on the command line:

```
java -jar Domineering.jar
```

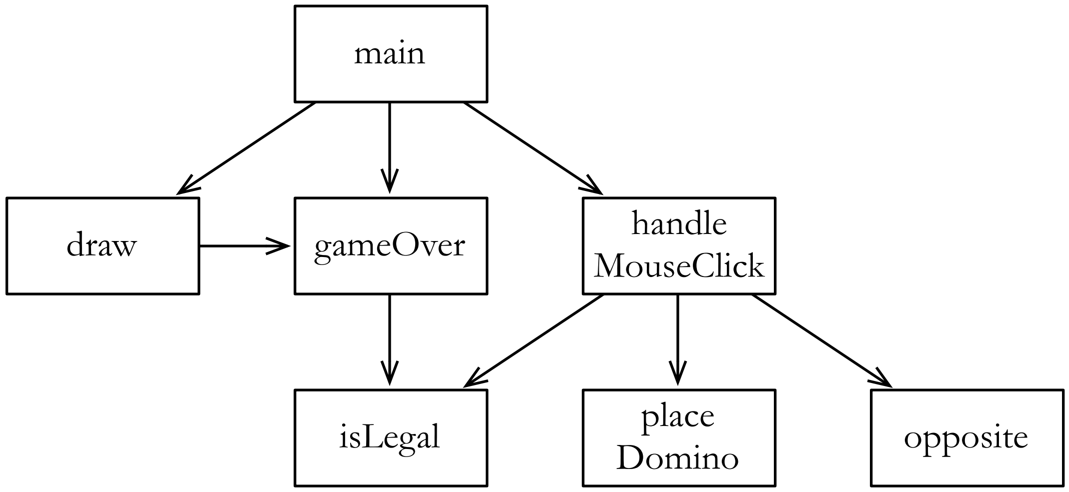
This is a two-player game. Play once or twice with another student so that you understand the rules.

Program Structure

You have been given an incomplete skeleton file Domineering.java. Your job is to complete the program so that it behaves *exactly* like Domineering.jar. As a slight bit of creative freedom, you may change the displayed colors. If you do so, choose an aesthetically pleasing palette and keep in mind that some users cannot distinguish, e.g., red from green.

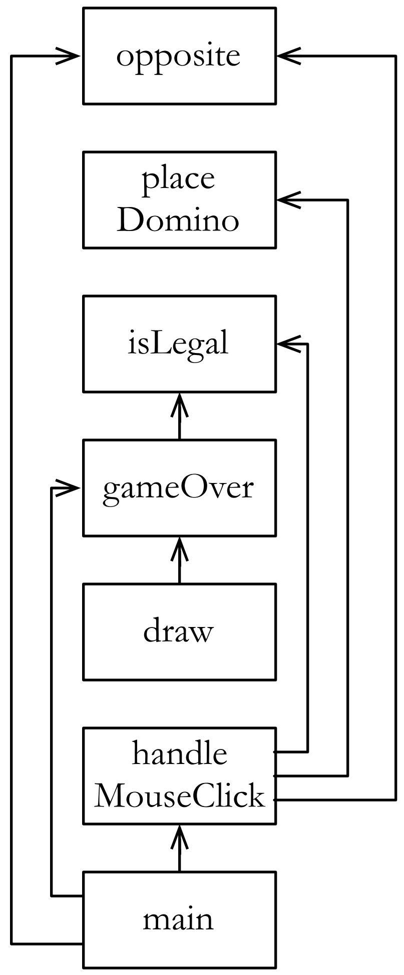
Take notes as you work. What bugs and conceptual difficulties did you encounter? How did you overcome them? What did you learn?

The program is broken up into seven methods. The diagram below shows which methods call which other methods. For example, `main` calls `gameOver`, which in turn calls `isLegal`.



This diagram helps us determine which methods to write first. For example, we won't be able to get `handleMouseClick` to work properly until `isLegal`, `placeDomino`, and `opposite` are working. We should complete the methods in an order that will avoid any arrows pointing from earlier methods to later ones.

One reasonable solution is to write `opposite` first, then `placeDomino`, and so on, as shown below.



opposite

This is a very simple method. Read the Javadoc comment at the top of the method and implement it. (Hint: if you're clever, you can do it without an if statement.)

Before you start, you will want to delete the line

```
return false;
```

That line is just there temporarily because a method with a return type of boolean would not compile if it was empty.

Before moving on to the next method, you want to be confident that this method is working correctly. Perhaps you can tell that it's correct just by looking at it, but even experienced programmers occasionally make mistakes this way. It's better to test the method by running it.

Add this line in the `main` method:

```
StdOut.println(opposite(true));
```

What is printed in the console when you run the program? Is it what you expected? How will you test that `opposite(false)` returns the correct value?

Testing may seem like unnecessary work, especially for such a simple method. Sooner or later, though, you're going to run into a bug. If you've thoroughly tested all of your previous methods, you'll have confidence that the bug is somewhere in the method you just wrote. This vastly reduces the amount of time you'll have to spend debugging.

placeDomino

This method is also relatively simple. To avoid any need for conversion this program uses x, y coordinates throughout.

To test this method, you might put code like this in `main`:

```
boolean[][] board = new boolean[8][8];
placeDomino(1, 2, board, false);
StdOut.println(board[1][2]);
StdOut.println(board[2][2]);
StdOut.println(board[3][2]);
StdOut.println(board[1][3]);
```

Before running the modified program, *write down* the values you expect it to print. Doing this first forces you to consider how the program is *supposed to* behave and not simply accept whatever values it prints.

This domino was placed horizontally. How will you test the placement of a vertical domino?

Don't forget to remove the testing code when you're done. Otherwise, the game will start with a domino already on the board!

isLegal

This method is slightly more complex, especially when it comes time to test it. In writing the method itself, the Boolean operators `&&`, `||`, and `!` may come in handy.

You may want to use `placeDomino` in your testing. Be sure that your tests include:

- A case where the placement is legal.
- A case where square x, y is occupied by a previous domino.
- Cases where the second square of the domino overlaps a previous domino. Test this for both vertical and horizontal placement.
- Cases where the second square of the domino is off the board. Test this for both vertical and horizontal placement.

gameOver

Can you see how calling one of your previous methods would make this easier to write?

How will you test this method?

draw

Use the draw method from TicTacToe.java (in the Tic-Tac-Toe module) as a template. Start with

```
StdDraw.clear();
```

and end with:

```
StdDraw.show();
```

The expression `(x + y) % 2` will be useful in determining which color to draw each square of the board. Figure out the value of this expression for several similar values of *x* and *y* to see why.

To test this one, call it in the `main` method. Does it look right? What if you place some dominoes before drawing?

handleMouseClick

Recall from the earlier diagram that this method calls `isLegal`, `placeDomino`, and `opposite`.

You can use the version from TicTacToe.java as a template, but notice a subtle difference. In that game, trying to make an illegal move resulted in passing (not playing). Passing would never be beneficial in *Tic-Tac-Toe*, but it would be in *Domineering*. The `handleMouseClick` method in Domineering.java therefore has the added responsibility of returning a boolean indicating whose turn it is next: the opposite player in case of a successful move, the same player in case of an attempted illegal move.

A method like this that depends on user input is difficult to test without having the entire program running. That's why we've done no user input in other methods and saved this method for close to the end.

Having written it (and hoping it works right), we can move on to the last method!

main

Again, you can use the version from TicTacToe.java as a template.

Don't forget to start with `StdDraw.enableDoubleBuffering()` and set the graphic scale to -1.5 to 8.5.

Be sure to update the local variable `verticalToPlay` with the result of a call to `handleMouseClick`:

```
verticalToPlay = handleMouseClick(board, verticalToPlay);
```

Now it's time to test out the whole program by playing the game!

Did you find that debugging went faster when you broke the program down into methods and tested each method as you went?

This work was supported by the Google Education and Relations Fund's CS Engagement Small Grant Program grant #TFR15-00411.

The principal investigator, Peter Drake, wishes to thank the following for their useful comments: the members of the CS-POGIL project, specifically Clif Kussmaul and Helen Hu; Maggie Dreyer; and various anonymous reviewers.