Building a Minesweeper Game

So, how to create a minesweeper? Let's first break down our goal to a few smaller tasks:

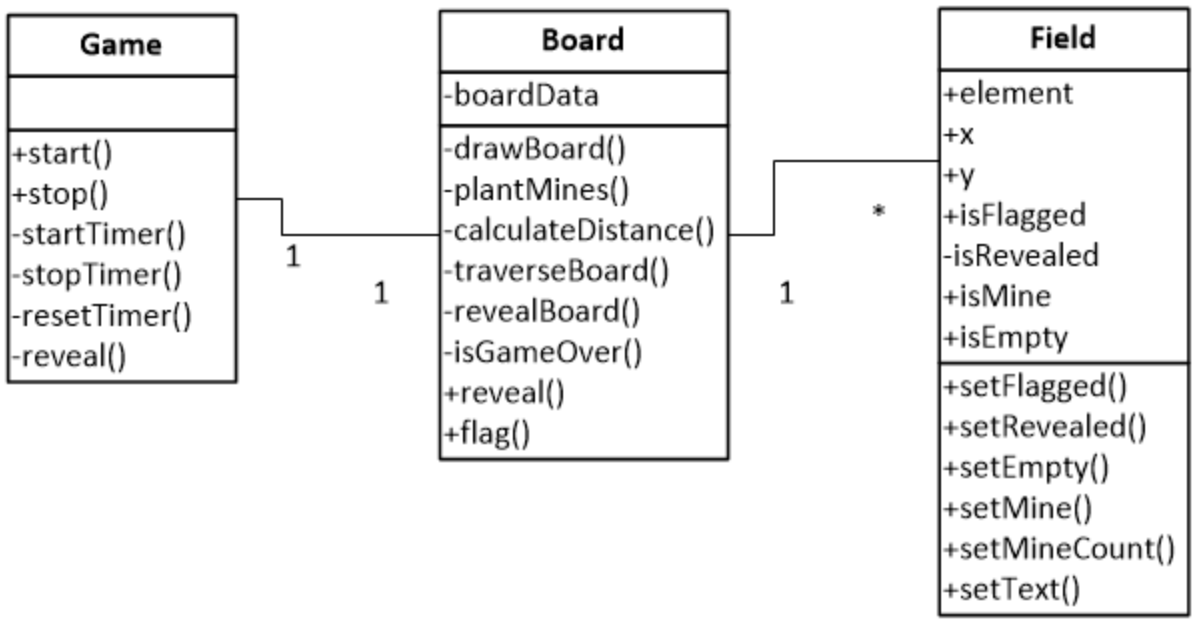
1. Define rules
2. Structure game
3. Draw the board
    1. Draw fields
    2. Plant mines
    3. Calculate distances
4. Find a way to traverse the board
5. Implement the "reveal" logic

Defining the rules

1. Goal of the game is to find all mines on the board.
2. You reveal mines by clicking at board field.
3. If you reveal a field with mine you lose the game.
4. If you reveal field without a mine it will show exact number of mines surrounding that field.
5. If you reveal field without number it means that there are no mines in its surroundings. In that case board will reveal all connected empty fields with its surroundings.
6. You can flag field by right-clicking it.
7. If you click on a revealed field and you already flagged all mines around that field, board will reveal rest of the hidden fields. Of course, if you misplaced flags you will reveal a field with a mine and lose the game.

Structuring the game

It's very important to define game structure before we proceed with the coding. That way we will think through about all classes, they responsibilities and relations. You can see how I structured this game at the class diagram below:



- Field class represents single field on the board and provides all operations on a field that we need.
- Board class is responsible for drawing a board, planting mines, calculating mine distance, traversing the board and revealing fields.
- Game class is responsible for the gameplay (difficulty levels, timer, starting and stopping a game).

You can see below what this structure looks like in code editor:

```
  1 (function ($)
  2 {
  3     var Game = function (gameElement)...;
108
109     var Field = function (element, x, y)...;
157
158     var Board = function (element, dimension, mines)...;
409
410     // export jquery plugin
411     $.fn.minesweeper = function ()
412     {
413         Game(this);
414
415         return this;
416     };
417
418 }(jQuery));
```

For the class creation I will use "Functional Inheritance" pattern described in Douglas Crockford's book named "Javascript - The Good Parts".

Game markup

Next I'm going to create HTML markup necessary for the game:

```
<div id="game" class="minesweeper">
<div>
    <h2>
        <span>Your time:</span>
        <span class="timer">0</span>
    </h2>
</div>
<div>
    <span>Level:</span>
    <select class="level">
        <option value="easy">Easy</option>
        <option value="beginner">Beginner</option>
        <option value="intermediate">Intermediate</option>
        <option value="advanced">Advanced</option>
    </select>
    <button class="newGame" type="button">New game</button>
</div>
<div class="board"></div>
</div>
```

And finally to initialize the game I'm simply going to call jQuery plugin over this markup:

```
$(document).ready(function ()
{
    $('#game').minesweeper();
});
```

Ok now the fun stuff. We defined the structure and now we are ready to implement the game.

Drawing the board

I'm going to draw the board using series of div elements floated to left. To break a row I will add another div element with clear: left style.

Also I'm going to create an in memory representation of the board using a matrix. It is easier and faster to traverse a matrix than the DOM. For this to work I must maintain the binding between in memory representation and the DOM, if one gets updated the other must update as well. I was tempted here to use knockoutjs library for the two-way binding, but for the sake of simplicity I decided to do it manually.

To draw the board we need to do three things:

1. Draw fields
2. Plant mines
3. Calculate distances

drawing fields:

```
function drawBoard()
{
    var i, j, fieldElement;

    for (i = 0; i < dimension; i++)
    {
        boardData[i] = [];

        for (j = 0; j < dimension; j++)
        {
            fieldElement = $('<div class="field hidden" />')
                .appendTo(element);

            boardData[i][j] = Field(fieldElement, i, j);

            fieldElement.data('location', { x: i, y: j });
        }

        $('<div class="clear" />').appendTo(element);
    }
}
```

In the code above you can notice a few things. First, I'm creating a matrix in class variable "boardData", then I'm appending a div element for the field to the board element. And finally I'm

storing custom data in DOM element under name "location". This data will hold position of the field within boardData matrix.

planting mines:

```
function getRandomNumber(max)
{
    return Math.floor((Math.random() * 1000) + 1) % max;
}

function plantMines()
{
    var i, minesPlanted = 0, x, y;

    while (minesPlanted < mines)
    {
        x = getRandomNumber(dimension);
        y = getRandomNumber(dimension);

        if (!boardData[x][y].isMine)
        {
            boardData[x][y].setMine(true);
            minesPlanted++;
        }
    }
}
```

Planting mines is dead simple: find a random field in the matrix and call its setMine method, repeat operation until all mines are planted. SetMine method (I'll show it later) sets flag that current field is a mine and updates DOM element giving it a specific class for a mine.

calculating distances:

```
function calculateDistance()
{
    var i, j;

    for (i = 0; i < dimension; i++)
        for (j = 0; j < dimension; j++)
```

```
            {
                var field = boardData[i][j];

                if (!field.isMine)
                {
                    var mines = traverseBoard(field,
                        function (f) { return !!f.isMine; });

                    if (mines.length > 0)
                    {
                        field.setMineCount(mines.length);
                    }
                    else
                    {
                        field.setEmpty(true);
                    }
                }
            }
    }
```

Now it's time to fill the board with some numbers. Idea is to process every field which is not a mine, get its surrounding fields which are mines and update field text to number of mines surrounding that field. The trickiest part here is to find surrounding fields which are mines. For that job I made a function named "traverseBoard" which I will describe later in more detail.

Traversing the board

Board traversal function is the heart of minesweeper game. The goal of this function is to get all surrounding fields for observed field on the board and to apply filtering function over it. The function simply moves through the matrix in all directions and returns surrounding fields.

```
    function traverseBoard(fromField, condition)
    {
        var result = [];

        condition = condition || function () { return true; };
```

```
        // traverse up
        if (fromField.x > 0)
        {
            result.push(boardData[fromField.x - 1][fromField.y]);
        }

        // traverse down
        if (fromField.x < dimension - 1)
        {
            result.push(boardData[fromField.x + 1][fromField.y]);
        }

        // traverse left
        if (fromField.y > 0)
        {
            result.push(boardData[fromField.x][fromField.y - 1]);
        }

        // traverse right
        if (fromField.y < dimension - 1)
        {
            result.push(boardData[fromField.x][fromField.y + 1]);
        }

        // traverse upper left
        if (fromField.x > 0 && fromField.y > 0)
        {
            result.push(boardData[fromField.x - 1][fromField.y - 1]);
        }

        // traverse lower left
        if (fromField.x < dimension - 1 && fromField.y > 0)
        {
            result.push(boardData[fromField.x + 1][fromField.y - 1]);
        }
```

```
    // traverse upper right
    if (fromField.x > 0 && fromField.y < dimension - 1)
    {
        result.push(boardData[fromField.x - 1][fromField.y + 1]);
    }

    // traverse lower right
    if (fromField.x < dimension - 1 && fromField.y < dimension - 1)
    {
        result.push(boardData[fromField.x + 1][fromField.y + 1]);
    }

    return $.grep(result, condition);
}
```

The reveal logic

And finally the reveal logic, the function that executes when user clicks on a field. There are two modes for accessing this functions: manual and automatic. Manual mode is when reveal action is invoked as a result to user interaction, and automatic mode is when it's invoked automatically through recursion.

The logic:

- If the field is flagged or already revealed then we do nothing.
- If the field is a mine then we trigger gameover event.
- If the field is already revealed and clicked by user then we check if all surrounding mines are already flagged, if yes then we reveal all hidden surrounding fields.
- And finally we handle situation when user clicked on the empty field (no mines in surrounding). In this situation we recursively reveal all empty fields in surrounding area.

```
obj.reveal = function (field, auto)
{
    // do not reveal flagged and revealed fields in auto mode
    if (field.isFlagged || (auto && field.isRevealed)) { return; }

    if (field.isMine)
    {
```

```
            revealBoard();
            $(obj).trigger('gameover');
            return;
        }
        else if (field.isRevealed && !auto)
        {
            var flaggedMines = traverseBoard(field,
                function (f) { return f.isFlagged; });

            if (field.mineCount === flaggedMines.length)
            {
                var hiddenFields = traverseBoard(field,
                    function (f) { return !f.isRevealed && !f.isFlagged; });

                for (var i = 0; i < hiddenFields.length; i++)
                {
                    obj.reveal(hiddenFields[i], true);
                }
            }
        }
        else
        {
            field.setRevealed(true);
            field.setFlagged(false);

            if (field.isEmpty)
            {
                var area = traverseBoard(field);

                for (var i = 0; i < area.length; i++)
                {
                    if (area[i].isEmpty || !area[i].isMine)
                    {
                        obj.reveal(area[i], true);
                    }
                }
```

```
            }

            if (isGameOver())
            {
                $(obj).trigger('win');
                return;
            }
        }
    };
```

And that's it! I described only the main logic behind this game and left out a few helper functions and the gameplay but that's not very interesting anyway.