Minesweeper

This is a modified and simplified version of the one described at ZetCode.com. Here we use a two dimensional array to represent the mine field.

Minesweeper is a popular board game shipped with many operating systems by default. The goal of the game is to sweep all mines from a mine field. If the player clicks on the cell which contains a mine, the mine detonates and the game is over.

The Minesweeper game is controlled solely by mouse. We react to left and right mouse clicks.

Further a cell can contain a number or it can be blank. The number indicates how many mines are adjacent to this particular cell. We set a mark (red flag) on a cell by right clicking on it. This way we indicate, that we believe, there is a mine. To remove a flag, right click on it.

There are 15 images used in this game:

A cell can be surrounded by maximum of 8 mines, so we need numbers 1..8. We need images for an empty cell, a mine, a detonated mine (game is over), a marked cell (flag), a covered cell and finally for a wrongly marked cell (not implemented here). The size of each of the images is 20x20 px. There is a Class for each one (One, Two, Three, Four, Five, Six, Seven, Eight, Empty, Mine, Boom, Flag, Cover).

A mine field is represented by a two dimensional array of numbers, 16 rows and 16 columns, 256 cells.

```java
private final int ROWS = 16;
private final int COLS = 16;
public int [][] cell = new int [ROWS][COLS];
```
Each cell in the field has a specific number. For example 0 denotes an empty cell. Number 10 is used for a cell cover as well as for a mark. A mine cell has number 9.

We’ll use constants to improve readability of the code.

```java
private final int COVER_FOR_CELL = 10;
private final int MARK_FOR_CELL = 10;
private final int EMPTY_CELL = 0;
private final int MINE_CELL = 9;
private final int COVERED_MINE_CELL = MINE_CELL + COVER_FOR_CELL;
private final int MARKED_MINE_CELL = COVERED_MINE_CELL +
                                    MARK_FOR_CELL;
```

A cell with a number 2, meaning it is adjacent to two mines. The numbers are added. For example, a covered mine has number 19, 9 for the mine and 10 for the cell cover etc. A cell that is marked with a flag would have a value from 20 to 29.

**Task 0**
We need to create all these images. Here is the code to create the image representing the first tile One:

```java
public class One extends Actor {

    public One() {
        // create a 20x20 empty image
        GreenfootImage tile = new GreenfootImage(20,20);

        Font myFont = new Font("Courier", Font.BOLD,20);
        tile.setFont(myFont);
        // fill the tile with color
        tile.setColor(Color.lightGray);
        tile.fill();
        // frame the tile with a square
        tile.setColor(Color.gray);
        tile.drawRect(0, 0, 20, 20);
        // display the number one (in bold)
        tile.setColor(Color.blue);
        tile.drawString("1",5,16);

        setImage(tile);
    }

    public void act() {
    }
}
```
If you double click on One in the right panel, you should see the above code. Open Two in the editor and you will see code similar to the above except that public One is replaced by public Two. Complete the code by essentially duplicating the code from one, but you need to select a different colour when the call to setColor is made and you need to replace the “1”, with “2” for the first parameter.

Next you need to create the rest of the images by creating the subclasses named Three, Four, Five, Six, Seven, and Eight. Subclasses are created by right-clicking on Actor and selecting “New Subclass”. Give the class a name (i.e. Three) and click OK. Do that for all of the subclasses identified above. Edit each subclass and add the appropriate public method that has the same name as the class. Model it after the methods One and Two above. To check how you are doing you can right click on an Actor and create a new object and place it on the stage.

**Task 1**

Your first task is to randomly place 40 mines in the field of 16x16=256 cells, and display them. This is done in the populate method of the MineWorld class.

```java
private final int MINES = 40;
```

Here is pseudocode of one way of doing it:

Set loop counter to zero
while (j < MINES)
    pull a random number 0-256
determine the cell row and column
    int r=j/rows;
    int c=j%cols;
    if cell[r][c] does not have a mine
        add a mine
    increment i

```java
int i=0;
while (i<MINES)
{
    int j=Greenfoot.getRandomNumber(ROWS*COLS);
    int r=j/ROWS;
    int c=j%COLS;
}```
```java
//System.out.println("j="+j+" r="+r+" c="+c);
if (cell[r][c] != MINE_CELL)
{
    cell[r][c] = MINE_CELL;
    addObject(new Mine(), c*20+10, r*20+10);
    i++;
}
```

20 indicates the width (and height) of each image. 10 indicates half an image. The center of the image is placed at the coordinates you specify. Perhaps, it is better to use a constant like `private int CELL_WIDTH=20;` instead of 20.

Notice the commented out print statement. It displays the values of the calculated row and column to verify that we are doing it right.

**Task 2**

Write method `revealAll()` to display the board shown above with all mines revealed. Here is how in pseudocode:

```java
public void revealAll()
{
    /*
     * Loop for each row r
     *   Loop for each column c
     *     revealCell(r,c)
     */
}
```

Here is part of the code for `public void revealCell(int r, int c)`:

```java
public void revealCell(int r, int c)
{
    if (cell[r][c]==MINE_CELL  )
    {
        addObject(new Mine(), c*20+10, r*20+10);
    }
    else if (cell[r][c]== EMPTY_CELL)
    {
        addObject(new Empty(), c*20+10, r*20+10);
    }
    ...  
    // Repeat for all cases (One, Two, Three, Four, Five, Six, Seven, Eight).
```
Task 3

Write code to display the numbers in the cells adjacent to mines, indicating the number of adjacent mines.

Showing these numbers ensures that your code works correctly before we cover all tiles to start the game.

Here is the algorithm in pseudocode:
For each cell that has a mine at row \( r \) and column \( c \)
Loop for all rows \( r-1, r, r+1 \)
Loop for all columns \( c-1, c, c+1 \)
If the cell has no mine
Add 1 to the contents of the cell

Careful here not to exceed array bounds. Rows must be \(( > 0 \text{ and } < \text{ROWS})\) and columns \(( >0 \text{ and } < \text{COLS})\).

We progress left to right as we add 1 to cells neighbouring a mine. Back in Act, call the method `revealAll()` to display the game board to this point.

Task 4

Next we need to cover all cells in preparation to start the game. Write code to add `COVER_FOR_CELL` to each cell in the field.

Call `revealAll()` back in Act to display what is now the covered board.
Task 5

To the act method, the code to handle mouse events needs to be added. Clicking with the left mouse key should reveal what is under the cover. Clicking with the right mouse key should place a flag to indicate the possibility of a mine. We mark cells with a flag to mark them as mines and so to avoid clicking on them with the left key. The game is over if we accidentally click with the left mouse key on a cell that has a mine.

Here is code snippet written to experiment with and to understand how to handle the mouse in Greenfoot. You might want to try it before proceeding.

```java
if (Greenfoot.mouseClicked(null))
{
    MouseInfo mouse = Greenfoot.getMouseInfo();
    int x = mouse.getX();
    int y = mouse.getY();
    int btn=mouse.getButton();

    int c = x/20;
    int r = y/20;
    System.out.println("x="+x+"    y="+y+"    button="+btn+"    r="+r+"    c="+c+" is "+cell[r][c]);
}
```

The two constants:
private final int RIGHT_BUTTON = 3;
private final int LEFT_BUTTON = 1;

have been declared so that they can be used to check if btn is the right or the left mouse button.

```java
if (btn == LEFT_BUTTON)
{
    if (cell[r][c] == COVERED_MINE_CELL)
    {
        System.out.println("Booooooooooom!!");
        addObject(new Boom(), c*20+10, r*20+10);
        uncoverMines();
        // display “Game is Over” or whatever you like
    }
    else if (cell[r][c]> MINE_CELL)
    {
        cell[r][c] -= COVER_FOR_CELL;
    }
```
Here if we left click on a mine cell, the game is over and we should reveal the locations of all mines.

If the clicked cell is a covered cell, then remove the cover by subtracting

\text{COVER\_FOR\_CELL}

if (btn == RIGHT\_BUTTON)
{
    if (cell[r][c]>=10 && cell[r][c] <=19)
    {
        cell[r][c] += MARK\_FOR\_CELL;
    }
    else if (cell[r][c]>=20 && cell[r][c] <=29)
    {
        cell[r][c] -= MARK\_FOR\_CELL;
    }
}

In the above code, upon right clicking an unmarked cell to add a flag, we add \text{MARK\_FOR\_CELL} to the content of the cell. Right clicking on a cell already marked with a flag we need to remove the flag by subtracting \text{MARK\_FOR\_CELL} from the content of the cell.

Remember, a flag could be incorrectly placed on an empty covered cell which has 20 in it, or a covered cell with a number (one (21), covered two (22), ..., covered eight (28), or correctly on a mine (29).

Call \text{revealCell(r,c)} to reveal the contents of a clicked cell. Now that the clicked cell has been revealed, we need to check if it is empty and if it is empty we need to reveal adjacent empty cells.

The above code could be enhanced by using constants instead of literal numbers.
Task 6

Uncovering an empty cell, is to be followed by uncovering all adjacent empty cells.

Here we clicked on the top left cell (row 1 column 1) which happened to be empty. Adjacent empty cells and the first line of marked cells are uncovered. Cells that have a mine should not be uncovered.

We use a recursive method `revealEmptyCells(int r, int c)`

```java
if (cell[r][c] == EMPTY_CELL) {
    System.out.println("r=\"+r+\"  c=\"+c+\" is empty");
    revealEmptyCells(r, c);
}
```

Similar to the algorithm described in Task 3, for cells adjacent to mines, to display the number of adjacent mines. We instead check if the adjacent cell is also empty.

If the cell adjacent to our empty cell is also empty, we try to find an adjacent cell with a number (not a mine). We then recursively call our `revealEmptyCell(r,c)` where `r` and `c` are the row and column of the empty cell we just found.

```java
public void revealEmptyCells(int r, int c) {
    for (int nr=r-1; nr<=r+1; nr++)
    {
        for (int nc=c-1; nc<=c+1; nc++)
        {
            //|   |    |   |
            //|   | x |   |
            //|   |    |   |
            if (nr>=0 && nc>=0 && nr<rows && nc<cols)
            {
                if (cell[nr][nc] == COVER_FOR_CELL)
                {
                    cell[nr][nc] = COVER_FOR_CELL;
                    revealCell(nr, nc);
                }
            }
        }
    }
}
```
revealNumberCells(nr,nc);
revealEmptyCells(nr,nc);
}
}
}
}

**Task 7**

Write the code for `revealNumberCells(nr,nc)`.

**Task 8**

Other enhancements could also include:

We did not implement it in our example, but in the real Minesweeper game, hitting a mine would reveal all other mines plus all wrongly flagged mines. That is, cells wrongly flagged believing they contained but did not actually have a mine.

Also, in the real game, the number of remaining mines is displayed. A happy face is displayed if all flags were correctly flagged, and an unhappy face if we click and detonate a mine.

You can add more difficult levels like fields with more cells and more or less mines.

Enjoy the game but beware, it can be addictive.