Building *Snake* in JavaScript

James Danielsen – Version 1.4

16 November 2022

TABLE OF CONTENTS

1.1 GETTING STARTED

When starting a big coding project, it is important to do as much planning as possible. The more that you can clarify and write down the logical parts of your program, the easier the coding and development will be.

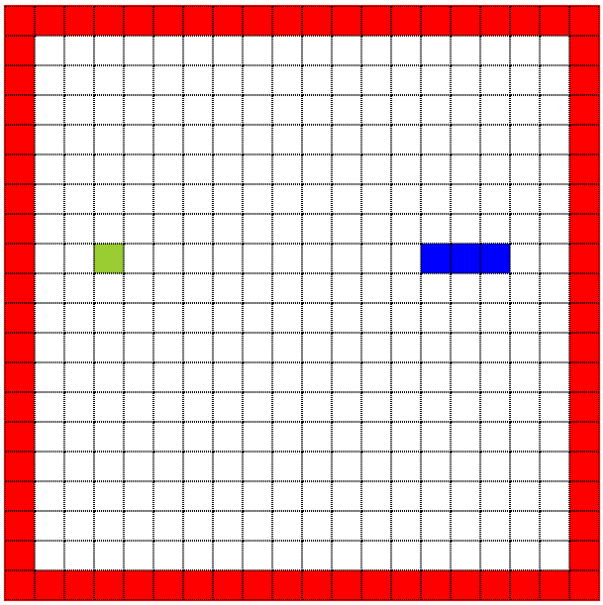


Figure 1: A Snake Board

Let's take a basic look at what the snake game would require:

1. A board or level made up out of different coloured squares
2. A snake that can be displayed on the board
3. A way to link input from the user to the movement of the snake
    a. Keys "←", "↑", "↓", "→" to control movement
4. A way to generate a random "apple" and display that apple on the board
5. Logic that comprises the rules of the game
    a. What happens when the snake hits an obstacle
    b. What happens when the snake eats the apple
    c. Any restrictions on the directions in which the snake can move, depending on its current direction
6. A coding loop that runs every few split-seconds (a fraction of a second) to implement the user input in (3) and the logic in (5)

When designing the snake game, we should try and complete each step before going onto the next step. We should not try and design every part of the game at the same time. Moreover, for each step, we need to break that step into a series of smaller sub steps. In this way, we want to slowly develop our code one small step at a time – adding functionality in small steps and testing our code at each step. Let's start with the first step.

******

To help you build the Snake project, I have provided you with the basic HTML document, **snake.html**, a basic CSS (style) file, **snake.css**, and a basic HelperLibrary JavaScript file,

HelperLibrary.js. You can write your own code in the **Snake.js** file. The HelperLibrary will work with the HTML document and allow you to focus on writing JavaScript, rather than focusing on HTML. The basic functions in the HelperLibrary are:

1. ***ClearGrid()*** -> This will delete all the blocks in the Grid element
2. ***AddBlock(x, y, color)*** -> This will add a element to the Grid.
    a. The x and y parameters are the x-coordinate and the y-coordinate of the block. These two coordinates are used to create an id for the block. This helps us to find the block again if need to change its color. x and y are integers.
    b. The color is the color of the block. Color is a string. The available colors are "white", "red", "blue", and "green". In this project we will use red for wall tiles, white for empty tiles, blue for snake tiles, and green for apple tiles
3. ***ChangeBlockColor(x, y, color)*** -> This will change the color of the block at the coordinates x and y:
    a. The x and y are based on how we create the coordinate system for the tiles (we'll learn more about that in a moment)
    b. The color is a string just like in (2) above

1.2 The Board

Creating the board requires a lot of sub steps. Let's take a look at some of those steps:

1. We need a **window with set dimensions** (height and width)
2. We need to decide on the **size of the blocks (tiles)** that will make up our board (height and width). This size will give us the **tile-width** and the **tile-height** of our game board. For example, if our window is 800 pixels by 800 pixels and our block is 40 pixels by 40 pixels then our board will be 20 tiles wide and 20 tiles high. The board in *Figure 1: A Snake Board* above, for example, is 20 tiles by 20 tiles.
3. We need to build a function called ***BuildBoard*** that will:
    a. Create a variable that holds all the information we need to store about the board
    b. This will have to store the tile information for our 20 by 20 grid of tiles
    c. The variable should have a *schema*; i.e. a map that tells us for example that the character "W" means a "wall" tile
4. We need to build a function ***DrawBoard*** that will:
    a. Access the HTML DOM and build the tiles stored in the board variable
    b. To achieve (a) above, **DrawBoard** needs to **loop** through the **tile-width** and the **tile-height** of our board, **appending elements** until the window in (1) is completely filled
    c. Which tiles are built in (a) above should be determined by some kind of **map** of the level. This map is the schema that we have mentioned in (3.c) above. The different types of tiles should be determined by the HTML style class given to the element

*****

The steps above can be broken down into a series of exercises:

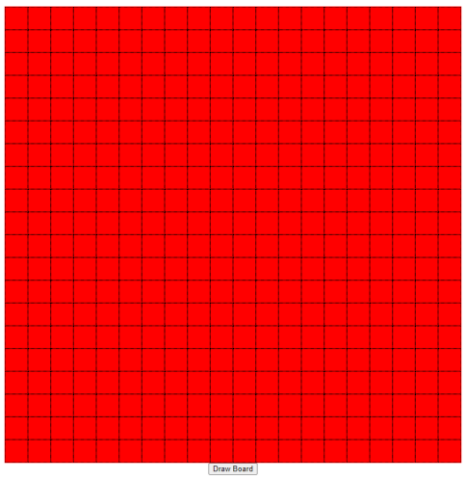
1. Use HTML to create a element with height 800 pixels and width 800 pixels. Give the div a red border. Give this element an ***id*** of "grid_id" and a class of "grid". [This step has already been done for us in the snake.html document]

2. Using HTML, create a button with the text "Draw Board". When this button is clicked, call a function called ***DrawBoard***. To test your code, add an alert in the ***DrawBoard*** function. [This step has already been done for us in the snake.html and snake.js documents]

3. Update your code in (2) above to add a block to the grid whenever the button is pressed. To do so, use the ***AddBlock*** function. For now, do not worry about the x and y parameters of the function – you can use any integers you want. Remove or comment out the alert – it is no longer needed.

4. Using a **loop**, update your code in (3) above such that clicking the button will add a row of 20 blocks. Use your loopcounter variable as the first parameter in the ***AddBlock*** function (pass your loopcounter variable into the function as the first parameter; the second parameter can still be set to 0).

5. Using a **nested loop**, update your code in (4) above such that you tile the whole grid with blocks. You should then have 400 blocks. See image below. Pass both an x parameter and a y parameter to the ***AddBlock*** function.



6. Currently, clicking the "Draw Board" button will keep adding the board to the HTML page. Update your ***DrawBoard*** function so that it clears the contents of the grid before the blocks are drawn. Use the premade ***ClearGrid()*** function to do this. Test this by repeatedly clicking the "Draw Board" button. The board should not be drawn more than once.

7. In this exercise, we will start drawing the board using information stored in a variable. Create a global variable called *Board*:
    a. The *Board* variable is an array with the following structure (containing 20 string variables, alternating between the value "W" and the value "."):

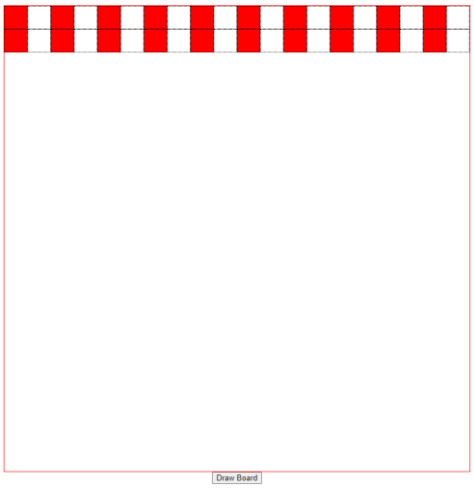
    Board =
    ["W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", "."]

b. Update your ***DrawBoard*** function to iterate through *Board* and then build a red "wall" block if the element in *Board* is "W" and a white empty block if the element is "." You will need to use a simple conditional if-statement to do this.
c. The output should be a series of alternating red and white blocks
d. Test your code by changing the length and contents of the *board* variable

8. Update your code in (7) such that:
    a. The *Board* variable is a **nested array** with the structure:

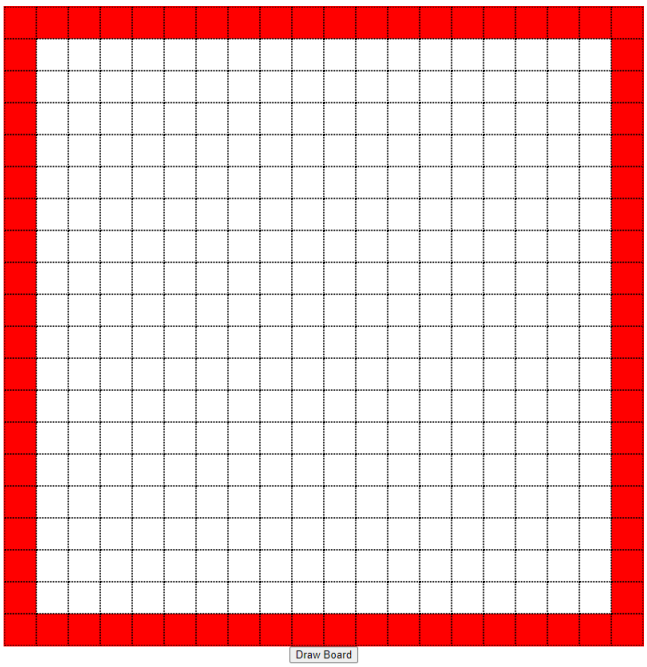
```
[
        ["W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", "."],
        ["W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", ".", "W", "."]

]
```

    b. Your function should iterate through *Board* in the same way as in (6) above, adding a new block for each element in the array.
    c. The output should be two rows, each containing a series of alternating red and white blocks. See image below.



    d. Pass an *x-parameter* and *y-parameter* to the ***AddBlock*** function according to the loopcounter variables in your nested loop.
    e. Change the length and contents of the *Board* variable to further test your code. Adding a third array to *Board*, for example, should result in a third row of blocks being printed.

9. Update your *Board* variable such that it contains <u>all</u> the information to build the board in *Figure 2: Empty board* below. Use the **map schema** of using "W" to represent a wall and using "." to represent an empty tile. Running the ***DrawBoard*** function should therefore draw the board!



10. Figure 2: Empty board

1.3 The Snake

In this step we need to think about storing, adding, and updating information about the snake:

1. We need a variable to keep track of the coordinates of each of the segments of the snake
2. We need a variable to keep track of the direction in which the snake is moving (anticipating Step 3 where the user controls the movement of the snake)
3. We need to implement a code loop whereby:
    a. We can make the snake move in the direction in which it is set
    b. The board updates to show the snake moving

*****

The steps above can be broken down into a series of exercises:

1. Create an array variable called *Snake* that stores the location of the head of the snake. This variable will have the structure [10,10]. The first 10 represents the x-coordinate on our board, and the second 10 represents the y-coordinate. Update your code in the ***DrawBoard*** function such that the snake is also drawn on the board using the color blue. Do this by doing the following steps and updating the *Board* variable before the board is drawn:
    a. Find the x-coordinate and the y-coordinate in the *Board* variable
    b. Change the value of that element from "." to "S" (for Snake)
    c. Update the ***DrawBoard*** function so that an "S" character is drawn in blue
    d. When you run your code, the block at coordinates 10,10 should be blue

2. Update your code in (1) above. Make *Snake* a **nested array** that stores the location of all of the segments of the snake. The structure of *Snake* will be something like [[10,10], [11,10], [12,10]]. Iterate through this array in your ***DrawBoard*** function and update the *Board* variable so that all of the snake's segments are drawn.

3. Create a function called ***MoveSnake*** which adds a new segment to the snake. To do this we need to decide which segment is the head of the snake:

   [[10,10], [11,10], **[12,10]**]

   In the **nested array** above, let us assume that the last element ([12,10]) represents the head of the snake. Let's also assume the snake is moving to the right. Thus, to "move" the snake we need to:
   a. Get the current x-coordinate of the snake head (12)
   b. Get the current y-coordinate of the snake head (10)
   c. Append a new segment (a new array) with the **updated coordinates** of where the snake has "moved". The snake is moving right, we need to add 1 to the x-coordinate.
   d. The coordinates for the new snake segment will therefore be [13,10]. Append (add) this element to the end of the Snake array.
   e. Using HTML, add a button to the HTML page. The text on the button should be "Move Snake". Attach the ***MoveSnake*** function to the button.
   f. Make sure to call the ***DrawBoard*** function inside and at the end of the ***MoveSnake*** function (otherwise the board will not update).
   g. Test your code by clicking the button. The snake should get longer every time you click the button.

4. Create a global variable called *Direction* to store the direction in which the snake is moving. Use "up", "down", "left", and "right" to indicate direction. According to the information stored in the *Snake* variable in (2) above, our snake is moving to the right. Thus, set *Direction* to "right".

5. Update your ***MoveSnake*** function such that it checks the *Direction* variable and moves the snake up, down, left, or right. ***MoveSnake*** must use an if-statement to check what the current direction is and then add a new segment to *Snake* according to the direction:
   a. To do this, update your code to change the x or y-coordinate depending on the direction the snake is moving:
      i. If the snake is moving *right*, then we add 1 to the x-coordinate
      ii. If the snake is moving *left*, then we subtract 1 from the x-coordinate
      iii. If the snake is moving *up*, then we subtract 1 from the y-coordinate
      iv. If the snake is moving *down*, then we add 1 to the y-coordinate
   b. Append (add) a new element to *Snake* that contains the coordinates of the new snake segment
   c. To test your code, click the "Move Snake" button. You should see an extra segment added according to the *Direction* variable. Change the value of the *Direction* variable to test the different directions. (Note that you won't immediately see any change with the left direction because the new segment will be 'hidden' by an existing segment. Keep clicking the button to see the change.)

6. Using HTML, add four buttons to the HTML page:

a. An “Up” button, a “Down” button, a “Right” button, and a “Left” button
b. Attach each button to a new ***UpdateDirection*** function. Pass a string to the ***UpdateDirection*** function that contains the new direction (‘up’, ‘down’, ‘left’, or ‘right’). For example, onclick=”UpdateDirection(‘right’)”.
c. Inside the ***UpdateDirection*** function, set the value of the global variable *Direction* to the value of the string parameter passed to the function. Then call the ***MoveSnake*** function.
d. To test your code, click the buttons. The snake should move according to the buttons you press.

7. Update the ***MoveSnake*** function so that the snake moves without getting longer. It should stay only 3 segments long. To do this we need to update the Snake variable and the Board variable.
    a. First, in the **MoveSnake** function and after you’ve added a new segment, remove the first element in the *Snake* array. You can use the shift method to remove the first element in the *Snake* array.
    b. Test your code. You will see that the snake will still be growing longer and longer on the board. To fix this, we need to update the *Board* variable by changing the “S” that represents the tail of the snake to a “.” that represents an empty space.
    c. Do this by getting the x-position and y-position of the tail of the snake (the first array stored in the Snake array). Find this x-position and y-position in the Board variable and set the value of the element to “.”
    d. Test your code. The snake should now move around without getting longer.

1.4 USING THE KEYBOARD

In the previous step, we used buttons to move the snake. But using the buttons is not easy. In this step we need to use the keys to allow the user to control the snake quickly and easily:

1. We need a way to collect keyboard input
2. We need to update the *Direction* variable based on the key that the user has pressed
3. We need to check the logic of how our snake moves such that the snake cannot turn back on itself

*****

The steps above can be broken down into a series of exercises:

1. Create a new function called ***StartGame***. This function will hold all the logic we need to setup and start the game. Inside the ***StartGame*** function, call the ***DrawBoard*** function. Change the “Draw Board” button on the HTML page like so:
    a. Change the text to “Start Game”
    b. Change the onclick function so that clicking the button calls the ***StartGame*** function
    c. Test that the button works as expected

2. Create a new function called ***Tick***. Call the ***MoveSnake*** function inside ***Tick***. The Tick function will be where the main logic of our program loops. We call it Tick because it will run every few split-seconds, just like the ticking of a clock.

3. In the beginning of your code, declare a global variable called *Timer* (you do not have to assign it a value). Inside the ***StartGame*** function, use the JavaScript ***setInterval*** function to call the ***Tick*** function every 1000 milliseconds. Test your code by clicking the "Start Game" button. You should see the snake moving to the right of the screen every 1 second.

4. Create a function called ***KeyPressed(event)***. This is the function that will be called whenever a user presses a key. The event parameter will be automatically passed to the function. Update your StartGame function by adding the following code:

   document.addEventListener("keydown", KeyPressed);

   This JavaScript code tells the computer to call the ***KeyPressed*** function whenever a key is pressed. Inside the ***KeyPressed*** function write the parameter *event.keyCode* to the console. Then run your code and press any key. Different numbers will be printed to the console depending on which keys you pressed. For example, pressing the left arrow key will give a code of 37. Pressing the right arrow key will give a code of 39. Make a note of the codes for the up and down arrow keys.

5. Update the ***KeyPressed*** function so that it updates the global *Direction* variable based on the key pressed. You will need to use if-statements and the keycodes you found in (4) above to implement this logic:
   a. Pressing "←" should change *Direction* to 'left'
   b. Pressing "↑" should change *Direction* to 'up'
   c. Pressing "↓" should change *Direction* to 'down'
   d. Pressing "→" should change *Direction* to 'right'

   To test your code simply run it and press the arrow keys. The snake should move accordingly.

6. Remove the up, down, left, and right buttons from the HTML page. They are no longer needed.

7. Update the logic of your program such that the user **cannot immediately reverse direction of the snake**. For example, if the snake is moving to the right then the user cannot make it immediately go left. She must first move up or down before she can go left. Similarly, if the user is moving upwards, she cannot immediately change the direction such that the snake is moving downwards.

1.5 Adding the Apple

In this step we need to generate an apple such that the snake can grow:

1. We need to create a function to generate an apple on the map, with the condition that the apple cannot be placed on a section of wall

*****

The steps above can be broken down into a series of exercises:

1. Declare a global *Apple* variable. You do not have to assign a value to it.
2. Create a function ***CreateApple***. The function should randomly select a tile on the board and update the *Board* variable with an "A" for apple. You can follow these substeps:
    a. Store the location of the apple inside the *Apple* variable (we might need this information later).
    b. Update your ***DrawBoard*** function such that an "A" on the board will be rendered as a green tile.
    c. Test your code by calling the ***CreateApple*** function before you draw your first board. One of the blocks should appear as a green tile.

3. Update your function in (1) such that your code checks that the random coordinates you create do not fall on a "wall" section of the *Board* variable. For example, the coordinates [0,0] checked against the *Board* variable, would return "W" – representing a wall obstacle. If you checked the coordinates [1,1], however, then your code would return "." – an empty tile. Use a loop to make sure your apple lands on an empty tile.

1.6 Collision Detection

In this step we need to check whether our snake is colliding with any obstacles, either with a wall or with itself:

1. We need to find the coordinates of the tile in which the snake will move to *next*
2. We need to check those coordinates against three things:
    a. The type of tile at those coordinates as stored in our *Board* variable. If the tile is an obstacle, then the snake dies and the game ends
    b. Whether or not the tile contains a snake segment. If the snake is already in that location, then the snake runs into itself and dies and the game ends
    c. Whether or not the tile contains an apple. If so, then the snake needs to grow, and a new apple needs to be created.

*****

The steps above can be broken down into a series of exercises:

1. Inside the ***Tick*** function, write some code that will print to the console the current coordinates of the head of the snake segment. As the game runs, these coordinates should continuously print to the console.

2. Update your code in (1) above such that the coordinates displayed are not those of the head of the snake, but rather the coordinates of the next tile in which the snake will move. For example, if the snake is moving right and the coordinates of the head are [12,10] then your code should print [13,10]. If the snake were moving up, then the coordinates would be [12,9]. [Hint: you can always slow down the game by changing the second parameter of the ***setInterval*** function. This might make it easier to follow the information printed to the console]

3. Update your code in (2) above such that the type of tile is printed to the console, rather than the coordinates. For example, the code should print "." if the tile in which the snake will move next is an empty tile but will print "W" if the next tile is a wall.

4. Update your code such that a collision with a wall will kill the snake and end the game. To do this, make a function called ***GameOver***. Inside this function, use the JavaScript function ***clearInterval*** to clear the *Timer* variable and stop the ***Tick*** function from being called.

5. Update your code such that a collision with one of the snake segments will kill the snake and end the game.

6. Update your code such that a collision with an apple will cause the snake to grow. [Hint: this could be done by simply delaying for one loop the step of removing the last segment]. If the snake gets the apple, your code should generate a new apple on the board.