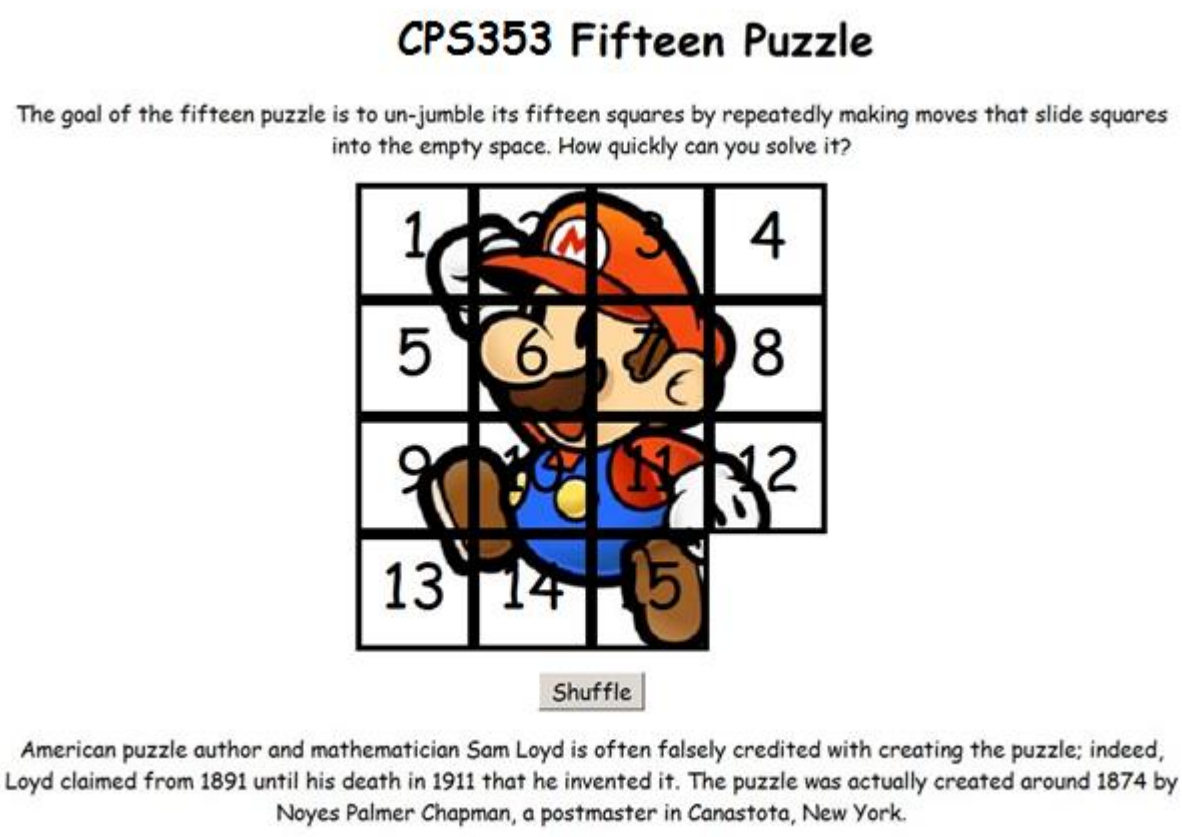
# CPS353: Internet Programming
# Homework Assignment 6: Fifteen Puzzle
# Due Wednesday, November 18 by the start of class

This assignment is about JavaScript's Document Object Model (DOM) and events. You'll write the following page:




**Background Information:**

The Fifteen Puzzle (also called the Sliding Puzzle) is a classic game consisting of a 4x4 grid of numbered squares with one square missing. The object of the game is to arrange the tiles into numerical order by repeatedly sliding a square that neighbors the missing square into its empty space.

You will write the CSS and JavaScript code for a page **fifteen.html** that plays the Fifteen Puzzle. You will also submit a **background image** of your own choosing, displayed underneath the tiles of the board. Choose any (wholesome) image you like, so long as its tiles can be distinguished on the board. Turn in the following files:

- **fifteen.js**, the JavaScript code for your web page
- **fifteen.css**, the CSS styles for your web page
- **background.jpg**, your background image, suitable for a puzzle of size 400x400px

You will not submit any **.html** file, nor directly write any HTML code. You will be provided with the HTML code to use, which should not be modified. (Download the **.html** file to your machine while writing your JavaScript code, but your code should work with the provided files unmodified.) You will write JavaScript code that interacts with the page using the DOM. To modify the page's appearance, write appropriate DOM code to change styles of on-screen elements by setting classes, IDs, and/or style properties on them.

*(Tip for playing the game: First get the entire top and left sides into proper position. That is, maneuver squares number 1, 2, 3, 4, 5, 9, and 13 into their final positions. Now never touch those squares again. This leaves a 3x3 board left to be solved, which is much easier.*

**Appearance Details:**

All text on the page is displayed in a "cursive" font family, at a default font size of 14pt. Everything on the page is centered, including the top heading, paragraphs, the puzzle, and the Shuffle button.

In the center of the page are **fifteen tiles** representing the puzzle. The overall puzzle occupies 400x400 pixels on the page, horizontally centered. Each puzzle tile occupies a total of 100x100 pixels, but all four sides have a 5px black border. This leaves 90x90 pixels of area inside each tile.

Each tile displays a number from 1 to 15, in a 40pt font. When the page loads, initially the tiles are arranged in their correct order with the missing square in the bottom-right. Each tile displays part of the image **background.jpg**, which you should put in the same folder as your page. The portion of the image displayed in each tile is related to that tile's number. The "1" tile shows the top-left 100x100 portion of the image. The "2" tile shows the next 100x100px of the background that would be to the right of the part shown under the "1" tile, and so on.

(Your **background image** appears on the puzzle pieces when you set it as the `background-image` of each piece. By adjusting the `background-position` of each div, you can show a different part of the background on each piece. One confusing thing about `background-position` is that the x/y values shift the background behind the element, not the element itself. The offsets are the negation of what you may expect. For example, if you wanted a 100x100px `div` to show the top-right corner of a 400x400px image, set its `background-position` property to `-300px 0px` .)

Centered under the puzzle tiles is a **Shuffle** button that can be clicked to randomly rearrange the tiles of the puzzle.

All other style elements on the page are subject to the preference of the web browser. The screenshots in this document were taken on Windows in Firefox, which may differ from your system.

**Behavior Details:**

When the mouse button is pressed on a puzzle square, if that square is next to the blank square, it is moved into the blank space. If the square does not neighbor the blank square, no action occurs. Similarly, if the mouse is pressed on the empty square or elsewhere on the page, no action occurs.

When the mouse **hovers** over a square that can be moved (neighbors the blank spot), its border and text color should become `red`. Once the cursor is no longer hovering on the square, its appearance should revert to its original state. Hovering over a square that cannot be moved should have no effect. (Use the: hover CSS pseudo-class.)

When the **Shuffle** button is clicked, the tiles of the puzzle are randomized. The tiles must be rearranged into a solvable state. Note that some puzzle states cannot be solved. For example, it has been proven that the puzzle cannot be solved if you switch only its 14 and 15 tiles. You should generate a random, valid, and solvable puzzle state by starting with a solved puzzle, and then repeatedly choosing a random neighbor of the missing tile and sliding it onto the missing tile's space. A few hundred such random movements should produce a shuffled board. Your algorithm should be relatively efficient; it should not animate each tile's movement, and if it takes more than a second to run or performs a large number of unnecessary tests and calls, you may lose points. For full credit, your shuffle code should thoroughly rearrange the tiles as well as the position of the blank square.

The game is not required to take any particular action when the puzzle has been won. You can decide if you'd like to pop up an `alert` box congratulating the user or add any other optional behavior to handle this event.

**Development Strategy:**

Here is a recommended development strategy for this assignment:

1. Make the fifteen **puzzle pieces appear** in the correct positions without any background behind them.
2. Make the correct parts of the **background** show through behind each tile.
3. Write the code that **moves a tile** when it is clicked from its current location to the empty square's location. Don't worry initially about whether the clicked tile is next to the empty square. Your code should include logic to optionally animate the tile's movement.
4. Write code to determine whether a **square can move** or not (whether it neighbors the empty square). Implement the highlight when the user's mouse hovers over tiles that can be moved. You must keep track of where the empty square is at all times.
5. Write the **shuffling** algorithm. You will probably find it easiest to first implement the code to perform a single random move – that is, randomly picking one square that neighbors the empty square and moving that square to the empty spot.

   Get it to do this one time when Shuffle is clicked, then work on doing it many times in a loop.

**Implementation Hints:**

1. Use **absolute positioning** to set the x/y locations of each puzzle piece. The overall puzzle area must use a `relative` position in order for the x/y offsets of each piece to be relative to the puzzle area's location.

2. Convert a string to a number using `parseInt`. This also works for strings that start with a number and end with nonnumeric content. For example, `parseInt("123four")` returns 123.
3. You will need to incorporate **randomness** in your shuffle algorithm. You can generate a random integer from 0 to *N*, or randomly choose between *N* choices, by saying `parseInt(Math.random() * N)` .
4. Do not explicitly make a `div` to represent the empty square. Keep track of where it is, either by row/column or by x/y position, but don't create an actual element for it. Also, don't store your puzzle squares in a two-dimensional array. This might seem like a good structure because of the 4x4 appearance of the grid, but it is difficult to keep such an array up to date as the squares move.
5. Avoid redundant code by creating **helper functions**. You should consider writing functions for common operations, such as moving a particular square, or for determining whether a given square currently can be moved. The `this` keyword can be helpful for reducing redundancy.
6. At some point you will find yourself needing to get access to the DOM object for the square at a particular row/column or x/y position. Consider writing a function that accepts a row/column as parameters and returns the DOM object for the corresponding square. It may be helpful to you to give an `id` to each square, such as `"square_2_3"` for the square in row 2, column 3, so that you can more easily access the squares later in your JavaScript code. (If any square moves, you will need to update its `id` value to match its new location.)

Use these `id`s in appropriate ways. Don't break apart the string `"square_2_3"` to extract the 2 or 3 from it. Instead, use the `id`s in the opposite direction: To access the DOM object for the square at row 2, column 3, make an `id` string of `"square_2_3"`. Use jQuery to find the piece with the right x/y position. ⅄ ⅄ Use jQuery to create your solution. It will make effects like animating the moving tiles straightforward. The code to load the jQuery libraries is already included in the HTML.

**Implementation and Grading:**

For full credit, your CSS code must pass the **W3C CSS validator**.

Minimize global variables, avoid redundant code, and use parameters and return values properly. Make extra effort to minimize **redundant code**. Capture common operations as functions to keep code size and complexity from growing. You can reduce your code size by using the `this` keyword in your event handlers.

Some **global variables** are allowed, but it is not appropriate to declare too many; values should be local as much as possible. If a particular value is used frequently throughout your code, declare it as a global "constant" variable named `IN_UPPER_CASE` and use it as a constant throughout your code. Do not store DOM element objects, such as those returned by the `$` or `$$` or `document.getElementById` functions, as global variables. As a reference, the professor's solution has only four global variables – the empty square's current row and column (initially both are set to 3), the number of rows/cols in the puzzle (4), and the pixel width/height of each tile (100). The last two are essentially constants.

Your JavaScript code should have adequate **commenting**. The top of your file should have a descriptive comment header describing the assignment, and each function and complex section of code should be documented.

**Format your code** to be readable and maintainable. Properly use whitespace and indentation. Use good variable and function names. Keep single lines of code from exceeding 100 characters in length. For reference, the professor's .js file has roughly 100 lines (60 "substantive"), and the CSS file has roughly 35 lines.

Separate content (HTML), presentation (CSS), and behavior (JS). Your JS code should **use styles and classes from the CSS** rather than manually setting each style property in the JS. For example, rather than setting the style of a DOM object, instead, give it a `className` and put the styles for that class in your CSS file. Style properties related to x/y positions of tiles and their backgrounds are impractical to put in the CSS file, so those can be in your JS code.

Use **unobtrusive JavaScript** so that no JavaScript code, `onclick` handlers, etc. are embedded into the HTML code.

Zip your three files (js, css, and background image) for your completed assignment and send them to the professor by Wednesday, November 18th before class.