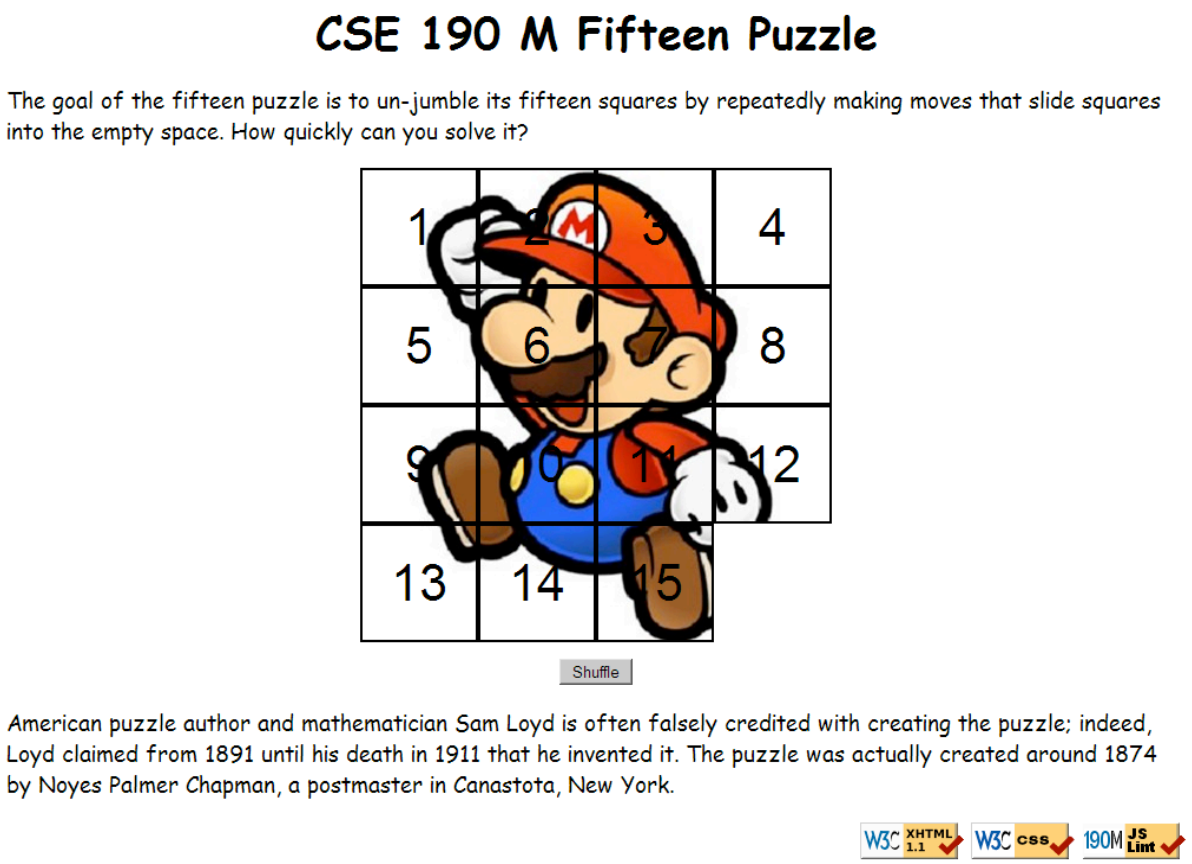
University of Washington, CSE 190 M
Homework Assignment 6: Fifteen Puzzle

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



Background Information:

The "Fifteen puzzle" (more generally called the Sliding Puzzle) is a simple classic game consisting of a 4×4 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 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 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
- **background.jpg**, your background image, suitable for a puzzle of size 400×400px

You will not submit any **.html** or **.css** file, nor directly write any XHTML or CSS code. We will provide you with the XHTML and CSS 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/left sides into proper position. That is, put squares number 1, 2, 3, 4, 5, 9, and 13 into their proper places. Now never touch those squares again. Now what's left to be solved is a 3×3 board, which is much easier.)

Appearance Details:

In the center of the page is are **fifteen tiles** representing the puzzle. Each tile is 100×100 pixels in size, including a 2px black border on all four sides. This leaves 96×96 pixels area inside the tile. Each tile displays a number from 1 to 15, in 32pt text in the system's default sans-serif font. When the page loads, initially the tiles are arranged in their correct order, top to bottom, left to right, with the missing square in the bottom-right. The tiles display part of the image **background.jpg**, located in the same folder as your page. (A CSS class `puzzlepiece` has been created for you that represents these styles. If you want any elements to use this class, set it on the elements using JS DOM code.)

Your **background image** appears on the 15 puzzle pieces because it is set as the `background-image` of each puzzle piece. By adjusting the `background-position` on each div, you'll be able to show a different portion of the background image inside each of the 15 puzzle pieces. One confusing thing about the `background-position` property is that the x/y offsets shift the *background with respect to the element*, not the other way around. For instance, consider the second tile on the first row. Because it is located immediately to the right of the first, its background image will need to be shifted *left* one space (not right!) in order to shift the second square's portion of the background into view—meaning for a tile size of 100×100px, its offset will need to be `-100px 0px`. Thus in general, for each tile on the board, the `background-position` offset will need to be negative (or zero), not positive.

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 XP in Firefox 3.5, 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.

Whenever the mouse cursor **hovers** over a square that can be moved (one that neighbors the blank square), its appearance should change. Its border color should change from black to red. Its text should become underlined and should become drawn in a green color of #006600. (A CSS class `movablepiece` is defined in the provided CSS file that incorporates these styles.) Once the cursor is no longer hovering over the square, its appearance should revert to its original state. Hovering the mouse over a square that cannot be moved has no effect. Note that technically a piece is movable by virtue of its position with respect to the blank space, not the fact that it has been hovered over—meaning if you want to mark a piece as being movable, this should occur prior to hovering the mouse over the piece. (The provided CSS code makes use of the `:hover` pseudo-class to ensure a tile marked as movable only changes appearance when it is hovered over.)

When the **Shuffle** button is clicked, the tiles of the puzzle are randomized. The tiles must be rearranged into a solvable state. Note that many puzzle states are not solvable; for example, it has been proven that the puzzle cannot be solved if you switch only its 14 and 15 tiles. We suggest that you generate a random valid solvable puzzle state by 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; 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. *(Hint: At any given time, the 2, 3, or 4 neighbors of the empty square are easily calculable if you know the row and column of the empty square. Choose a random existing neighbor.)*

The game is not required to take any particular action when the puzzle has been won. (An extra credit feature exists, documented in the Extra Features spec, to notify the user when they've solved the puzzle.)

Development Strategy:

Past students claim that this program is hard! We suggest the following development strategy:

- Make the fifteen **puzzle pieces appear** in the correct positions without any background behind them.
- Make the correct parts of the **background** show through behind each tile.
- 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.
- Write code to determine whether a given **square can move** or not (whether it neighbors the empty square). Use this to implement the highlighting style that occurs when the user's mouse hovers over tiles that can be moved. This will require you to keep track of where the empty square is at all times.
- Write the **shuffling** algorithm. We suggest first implementing the code to perform a single random move; that is, picking one random square that neighbors the empty square and moving that square to the empty spot. First get it to do that only once when Shuffle is clicked, then work on repeating the single random action many times.

Implementation Hints:

- Many students have redundant code because they don't create **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 (see book section 8.1) can be helpful for reducing redundancy.
- 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. We suggest you write a function that accepts a row/column as parameters and returns the DOM object for the corresponding square. It may be helpful for 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.)

  You should use these `id`s in appropriate ways. It's not good to try to break apart the string `"square_2_3"` to extract the number 2 or 3 from it. Instead, use the `id`s in the opposite direction: If you need to get access to the DOM object for the square at row 2, column 3, build an `id` string of `"square_2_3"` to find it. Another approach is to store the pieces in an array, or to use `$$` to find the piece with the right x/y position.
- You can convert a string into a number using the `parseInt` function. This also works for strings that end with non-numeric content. For example, `parseInt("20percent")` returns 20.
- You can generate a random number from 0 to N, or randomly choose between N choices, by saying `Math.floor(Math.random() * N)`.
- We suggest that you 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.
- We suggest that you not store your squares in a 2-D array. This might seem to be a good structure because of the 4×4 appearance of the grid, but it will be difficult to keep such an array up to date as the squares move.

Implementation and Grading:

Submit your work from the course web site. For reference, our .js file has roughly 99 lines (57 "substantive").

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.

For full credit, your JavaScript code should pass the provided **JSLint** tool with no errors reported. You should follow reasonable style guidelines similar to those of a CSE 14x programming assignment. In particular, minimize global variables, avoid redundant code, and use parameters and return values properly.

Some **global variables** are allowed, but it is not appropriate to declare lots of them; values should be local as much as possible. If a particular constant value is used frequently throughout your code, declare it as a global "constant" variable named `IN_UPPER_CASE` and use the constant throughout your code.

You should separate content (XHTML), presentation (CSS), and behavior (JS). As much as possible, your JS code should **use styles and classes from the CSS** rather than manually setting each style property in the JS.

For full credit, you must write your code using **unobtrusive JavaScript**, so that no JavaScript code, `onclick` handlers, etc. are embedded into the XHTML code.

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 similarly to the examples from class. Properly use whitespace and indentation. Use good variable and method names. Avoid lines of code or comments more than 100 characters wide.

Do not place a solution to this assignment on a public web site. Please do not look for JavaScript Fifteen Puzzle games on the internet. Using or borrowing ideas from such code is a violation of our academic integrity policy. We have done our own searches to find several such games and will include them in our similarity detection process.

Put your files on the **Webster** at: **https://webster.cs.washington.edu/*your_uwnetid*/hw6/fifteen.html**