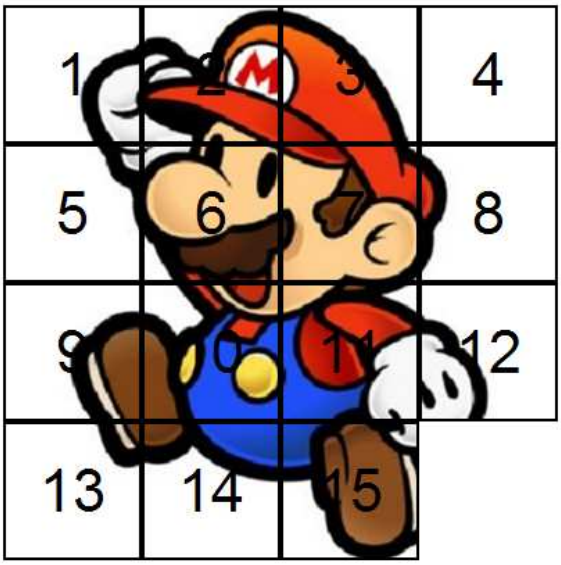
University of Washington, CSE 190 M, Spring 2009
Homework Assignment 6: Fifteen Puzzle
due Wednesday, May 13, 2009, 11:30pm electronically

This assignment is about JavaScript's Document Object Model (DOM) and event handling. You must match in appearance and behavior the following web page (between but not including the thick black lines):

CSE 190 M Fifteen Puzzle

The goal of the fifteen puzzle is to un-jumble its fifteen squares by repeatedly making moves that slide squares into the empty space. How quickly can you solve it?



Shuffle

American puzzle author and mathematician Sam Loyd is often falsely credited with creating the puzzle; indeed, Loyd claimed from 1891 until his death in 1911 that he invented it. The puzzle was actually created around 1874 by Noyes Palmer Chapman, a postmaster in Canastota, New York.

Background Information:

The "Fifteen puzzle" (more generally called the Sliding Puzzle) is a simple 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 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 400x400px

You will not submit any .html or .css file, nor directly write any XHTML or CSS code in this assignment.

Instead, we will provide you with the XHTML and CSS code to use, which should not be modified. (You should download these files to your machine while you're writing your JavaScript code, but your code should work with the provided files unmodified.) To modify the page's behavior and content, write JavaScript code that interacts with the page using the DOM. To modify its appearance, write appropriate DOM code to change styles of onscreen elements by setting classes, IDs, and/or style properties on them.

Appearance Details:

In the center of the page is a set of tiles representing the playable Fifteen Puzzle game. Each tile is 100x100 pixels in total size, including a 2px black border around all four of its sides. (This leaves 96x96 pixels visible area inside the tile.) Each tile displays a number from 1 to 15 in 32pt text using the default sans-serif font available on the system. 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 also display a chunk of the image `background.jpg`, assumed to be located in the same folder as your page. (A CSS class named `puzzlepiece` has been created for you that represents these styles, but nothing on the existing page uses this class. If you want any onscreen elements to use this class, you'll have to set it using 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 piece of the background on each of the 15 puzzle pieces. One confusing thing about the `background-position` property is that the x/y offsets shift the background behind an element, not the element itself; this means that 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 background image, you would set its `background-position` to `-300px 0px`.

Centered underneath the puzzle tiles is a Shuffle button that can be clicked to randomly rearrange the tiles of the puzzle. The last content on the page is a right-aligned paragraph containing two links to the W3C validators and JSLint. These are the same images and links as used in the previous assignments. The images should not have borders.

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, 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 coler 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` exists that represents 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.

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 states of the puzzle 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 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 shuffle 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.

The game is not required to take any particular action when the puzzle has been won, that is, when the tiles have been rearranged into the correct order.

Extra Features:

In addition to the previous requirements, you must also complete **at least one of the following additional features**. If you want to complete more than one, that is fine, but only one is required.

- **End-of-game notification**: Provide some sort of visual notification when the game has been won; that is, when the tiles have been rearranged into their original order. An `alert` is not sufficient; you should modify the appearance of the page. You may display an image(s) if you like, but there is no way to turn them in, so put them on your Webster space and use full path URLs when linking to them.
- **Ability to slide multiple squares at once**: Make it so that if you click aany square in the empty square's row or column, even ones more than one spot away from the empty square, all squares between it and the empty square slide over. (Much more pleasant to play!) If you do this extra feature, make it so that all movable squares (including ones several rows or columns away from the empty square) highlight on hover as described before.
- **Animations and/or transitions**: Instead of each tile immediately appearing in its new position, make them animate. You can do any sort of animation or other styling you like, as long as the game ends up in the proper state after a reasonable amount of time.
- **Game time**: Keep track of the game time elapsed in seconds and the total number of moves, and when the puzzle has been solved, display them along with the best time/moves seen so far.
- **Multiple backgrounds**: Provide several background images (at least 4) to choose from. The game should choose a random background on startup, and should have a UI (such as a `select` box) by which the player can choose a different image while playing. Host your additional backgrounds on the Webster server and link to them using full path URLs.
- **Different puzzle sizes**: Place a control on the board to allow the game to be broken apart in other sizes besides 4x4, such as 3x3 or 6x6.

Near the top of your JS or HTML file, put a comment saying which extra feature(s) you have completed. If you implement more than one, comment which one you want us to grade (the others will be ignored). Regardless of how many additions you implement, the main behavior and appearance should still work as specified. You may not modify the XHTML/CSS files; each feature must be done through JavaScript. If you have a different idea for an addition to the program, please ask us and we may approve it.

Development Strategy and Hints:

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, 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.

Here are some hints to help you solve particular aspects of the assignment:

- Many students have redundant code on this program 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 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 to you to give `id` values 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. (Such `id` values would need to change as squares move.) You should use these `id`s in appropriate ways. It's not very wise to try to break apart the string `square_2_3` to extract the number 2 or 3 from it to determine that square's row or column. 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 the proper `id` to find it. Another approach is to store the puzzle pieces into 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 into a 2-D array. This might seem to be an appropriate structure because of the 4x4 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 assignment online from the turnin area of the course web site. For reference, our solution has roughly 125 lines of JavaScript including blank lines and comments.

You should make extra effort to minimize redundant code. If you are not cautious about avoiding redundant code, capturing common operations as functions, etc., it is easy for your code size and complexity to grow. 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 (when the Recommended options are checked). Follow reasonable stylistic guidelines similar to those from a CSE 14x programming assignment. For example, utilize parameters and return values properly, correctly use indentation and spacing, and place a comment header on top of the file, atop every function explaining that function's behavior, and on complex code sections.

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 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 only use material discussed in the textbook or lectures unless given permission by the instructor.

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

Implement the behavior of each onscreen control using JavaScript event handlers defined in your script file. 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.

You may be able to find Fifteen Puzzle games written in JavaScript on the internet. Please do not look at any such games. Using or borrowing ideas from such code is a violation of our course academic integrity policy. We have done searches to find several such games and will include them in our similarity detection process.

Part of your grade will come from successfully uploading your files to the **Webster** server at the URL:

```
https://webster.cs.washington.edu/your_uw_netid/hw6/fifteen.html
```

Please do not place a solution to this assignment online on a publicly accessible web site.