

(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 3x3 board, 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, the Shuffle button, and the W3C buttons at bottom.

In the center of the page there 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 5px on all four sides are occupied by a 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. Which part of the image is displayed by 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 in Chrome on a Mac, 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 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.

When the puzzle has been solved, display an alert box congratulating the user. The text in this box must be "Congratulations! You solved the puzzle!" (autolab will be looking for this as an exact text match).

Development Strategy:

Past students have said that this program is hard! Here is a suggested 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 **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.
- ⤴ 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 in a loop.

Implementation Hints:

- ^ 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.
- ^ Convert a string to a number using `parseInt`. This also works for strings that start with a number and end with non-numeric content. For example, `parseInt("123four")` returns 123.
- ^ 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)`.
- ^ 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 also suggest that you not store your puzzle squares in a 2-D array. This might seem like a good 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.
- ^ 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 can be helpful for reducing redundancy.
- ^ At some point you will find yourself needing to get access to the DOM (or jQuery) 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/jQuery 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 ids in appropriate ways. Don't break apart the string "square_2_3" to extract the 2 or 3 from it. Instead, use the ids in the opposite direction: To access the DOM/jQuery object for the square at row 2, column 3, make an id string of "square_2_3". Or use jQuery functions to find the piece with the right x/y position (you might want to check out position and offset).

Implementation and Grading:

For full credit, your CSS code must pass the **W3C CSS validator**. Your .js file must run in **JavaScript strict mode** by putting "use strict"; at the top.

Follow reasonable style guidelines similar to previous programming assignments. In particular, 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 the constant throughout your code. Do not store DOM/jQuery element objects as global variables. As a reference, our own solution has only four global variables: the empty square's 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 similarly to the examples from class. Properly use whitespace and indentation. Use good variable and method names. Avoid lines of code more than 80 characters wide. For reference, our .js file has roughly 100 lines (60 "substantive"), and our 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.

Copyright © Marty Stepp / Jessica Miller, licensed under Creative Commons Attribution 2.5 License. All rights reserved.