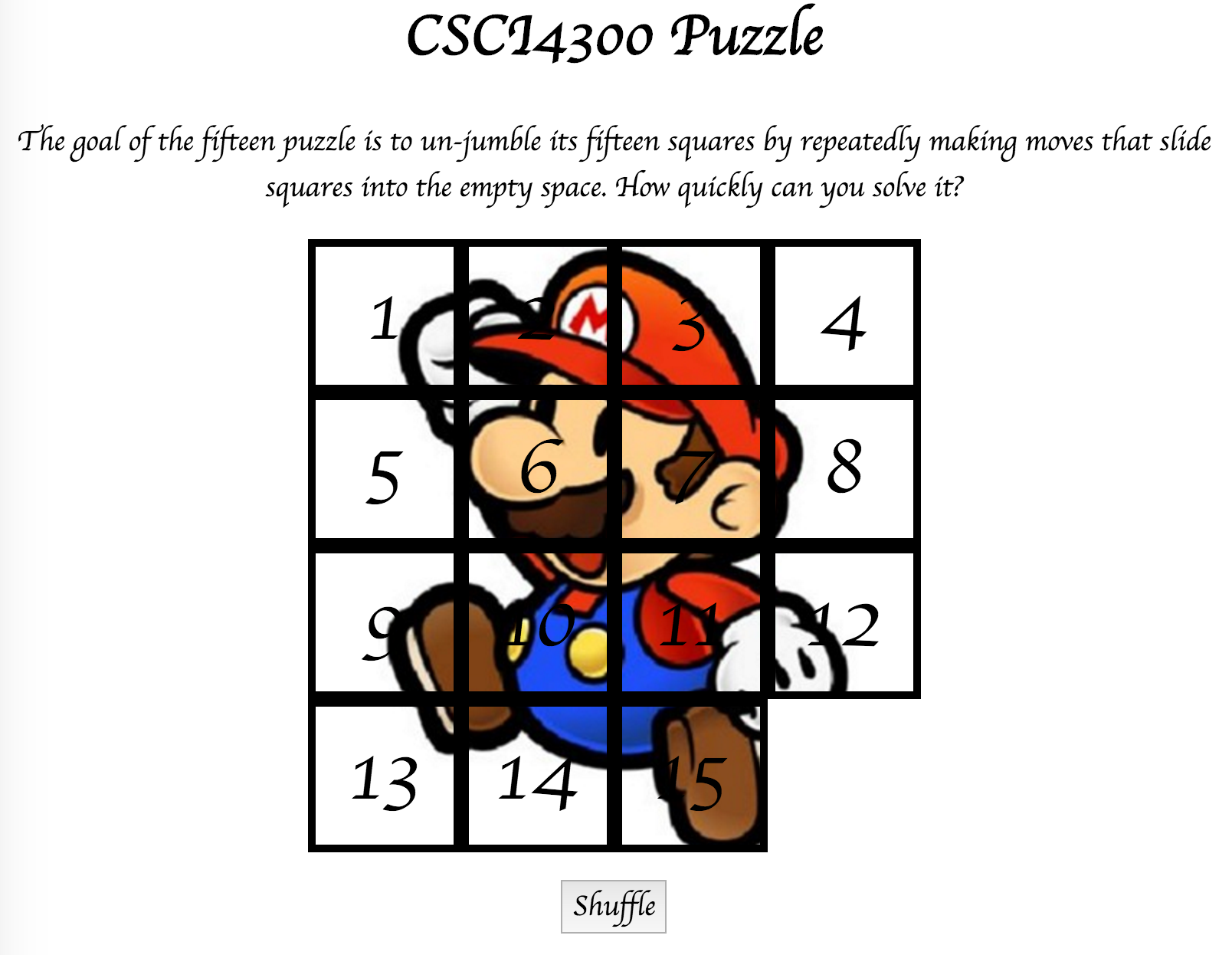
# CSCI 4300: Web Programming

**Spring 2019**

## Project 4: Advanced JavaScript: DOM and Events

Due: March 4th (11:59 pm)

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 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 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 image you like, so long as its tiles can be distinguished on the board. Turn in the following files:

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

You will not directly write any HTML or CSS code. We will provide you with the HTML code which should not be modified. CSS is also provided, but you can modify it if you prefer. 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 3x3 board, which is much easier.)*

## Appearance Details:

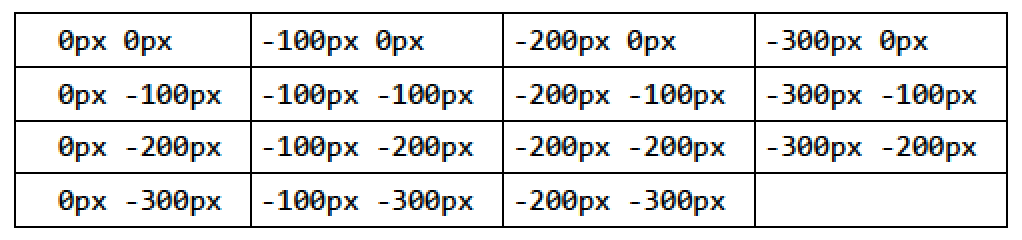
All text on the page is displayed in a "cursive" font family, at a default font size of 14pt. Everything on the page is centered, including the top heading, paragraph, the puzzle, and the Shuffle button.

In the center of the page is 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.

The HTML file given to you contains the fifteen div elements and you are not supposed to modify the HTML file. Initially the page should appear with the puzzle in its properly arranged order like in the screenshot on the first page of this spec, with 1 at top-left, 4 at top-right, 13 at bottom-left, the empty square at bottom-right, and so on.

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.) The following is a complete listing of the exact background-position values each location on the board should have:



Centered under the puzzle tiles is a **Shuffle** button that can be clicked to randomly rearrange the tiles of the puzzle. See the "Shuffle Algorithm" section of this spec for more details about implementing the shuffle behavior.

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

## Playing the Game:

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.)

The game is not required to take any particular action when the puzzle has been won.

## Development Strategy:

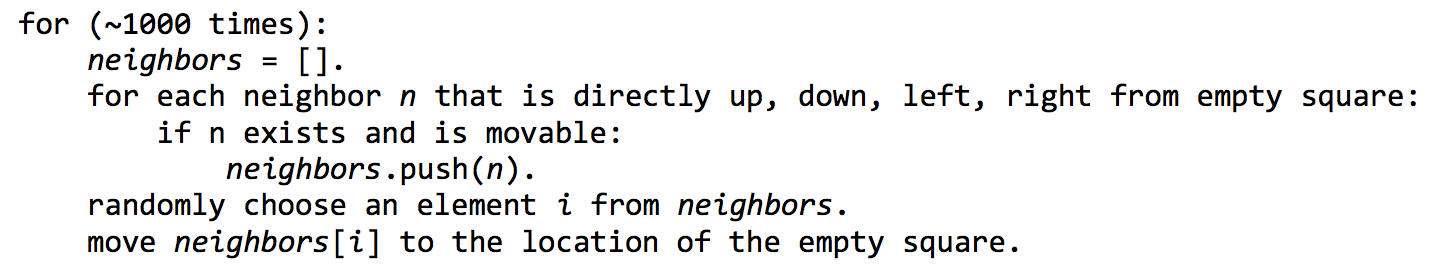
We suggest the following development strategy:

* 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 **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.

## Shuffle Algorithm:

Centered under the puzzle tiles is a Shuffle button that can be clicked to randomly rearrange the tiles of the puzzle. When the Shuffle button is clicked, the puzzle tiles are rearranged into a random ordering so that the user has a challenging puzzle to solve.

The tiles must be rearranged into a solvable state. Some puzzle states are not solvable; for example, the puzzle cannot be solved if you swap only its 14 and 15 tiles. Therefore your algorithm for shuffling cannot simply move each tile to a completely random location. 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. Roughly 1000 such random movements should produce a well-shuffled board. Here is a rough pseudo-code of the algorithm we suggest for shuffling:



Notice that on each pass of our algorithm, it is guaranteed that one square will move. One can write an algorithm that randomly chooses any one of the 15 squares and tries to move it; but this is a poor way to shuffle because many of the 15 squares are not neighbors of the empty square. Therefore the loop must repeat many more times in order to shuffle the elements effectively, making it slow and causing the page to lag. This is not acceptable.

Your shuffle algorithm will need to incorporate randomness. You can generate a random integer from 0 to **K**, which is helpful to randomly choose between **K** choices, by writing, parseInt(Math.random() \* K). We suggest first implementing code to perform a single random move; that is, when Shuffle is clicked, randomly pick one square near the empty square and move it. Get it to do this once, 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.
* Many students have bugs related to not setting their DOM style properties using proper units and formatting. The string you assign in your JS code must exactly match what would have been in the CSS file. For example, if you want to set the size of an element, a value like 42 or "42" will fail, but "42px" or "42em" will succeed. When setting a background position, "42 35" will fail but "42px 35px" will succeed. And so on.
* 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.
* 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. 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.)

## Implementation and Grading:

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. You should not use any external JavaScript frameworks or libraries such as jQuery to solve this assignment. Your .js file must run in strict mode by putting **"use strict";** at the top. 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 **this** keyword in your event handlers.

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 sections of code should be documented. Properly use whitespace and indentation. Use good variable and method names. Avoid lines of code more than 100 characters wide. For reference, our .js file has roughly 100 lines, and our CSS file has roughly 30 lines.

**Do not place your solution on a public web site. Submit your own work and follow the course misconduct policy.**

## How to Submit:

Submit your ".zip" file using ELC. Only team leaders need to make a submission. **Every student needs to submit a peer-evaluation form within 24 hours of the project submission deadline**.