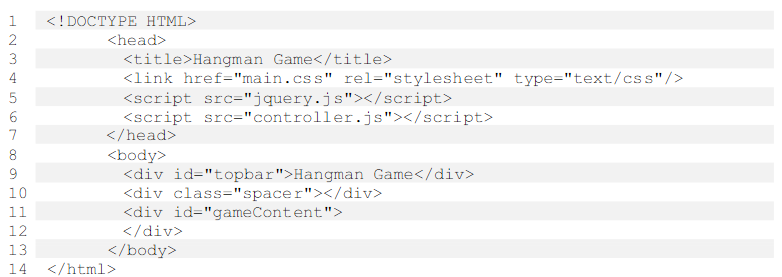
**HTML**

**Step 1**

Open your text editor and type:

<!DOCTYPE HTML>

This is the way that we begin all of our HTML5 documents. Now let’s fill out the document structure:



Believe it or not, that’s our whole html document, except for one line which we will add later to make it mobile-friendly.

The <title> line controls what is shown on the browser tab as the title or description of the page. [3].

The next line tells our page to look for a document called ‘main.css’ which will hold all of our style information (background colors and so on). [4].

The first <script> line tells the page to look for a JavaScript file called ‘jquery’. This is a standard JavaScript ‘extension’ widely used by developers to make JavaScript more powerful. We will not need to write this file ourselves. [5].

The second <script> line tells the page to look for a JavaScript file called ‘controller.js’. This is the file we will use to control the game. [6].

The next few lines, in the <body> of the page name ‘div’s or dividers that we will use to hold the page content. The ‘topbar’ will hold our page title. The ‘spacer’ div will make space on our page. ‘gameContent’ is where the game will be displayed. [8-13].

Notice that most divs have an id, a unique identifier. One of them has a class name instead of an id name. That means we may use it more than once. Save the document as **hangman.html**.

Test it out by opening it in a browser.

Obviously, there will not be much to see yet, just the text Hangman Game at the top…

We will be using JSON which Chrome and some other browsers will not display if offline.

**Step 2**

 So we know that we will need the following files to build our project:

- index.html, which we have already done.

- main.css, which will hold the style information

- jquery.js, which will enhance standard JavaScript

- controller.js, which will control the game

- quizbank.json, which will hold our word database

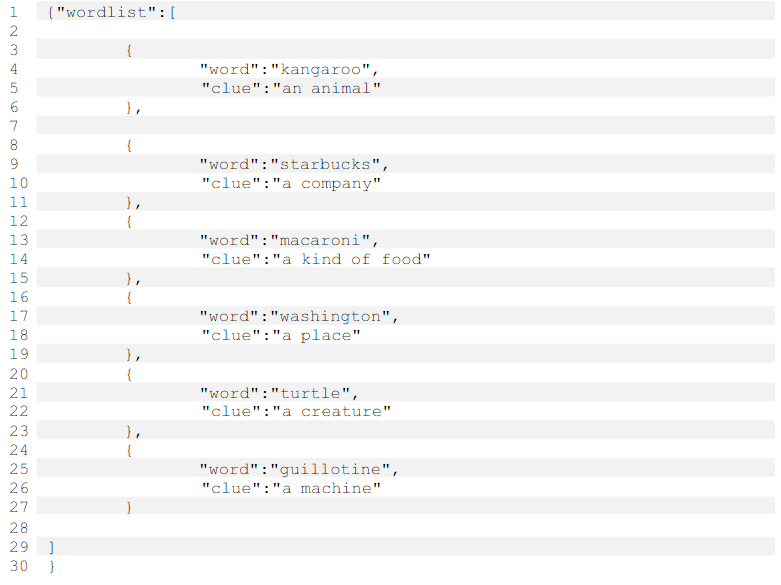
- man.png, which will hold our graphics

All of these files can be written or viewed in a text editor, just like index.html, so there’s no need for any special software aside from Notepad++ or Brackets.

**Step 3**

Build a question database

Create a file in your text editor named **quizbank.json**. Then open it up and type in the following:



This is a JSON file and it holds data in groups. We can use it to hold our question database. You’ll notice that it is easily readable by either machine or human.

Each JSON element holds a word and a clue. However, we will randomize the words before we show them to the user.

Later, if we wish to change or add any words, this is the only file we will need to modify. We can add as many questions as we want – we will program the game to know when to stop. That means our game can be recycled and can even also edited by someone with very little technical expertise.

\*Note that you will need to make sure that you stick very carefully to the format with JSON files. A missing or extra comma, for example, can cause problems. Save the file and continue.

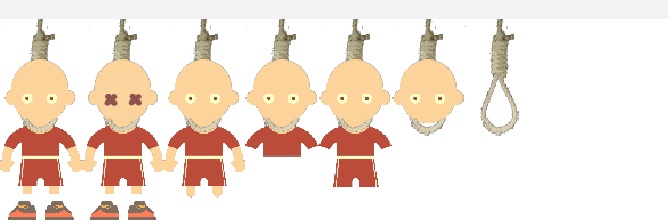
**Step 4**

Gather assets

You will need two external assets.

The first one is the JQuery file. You can get both from the source files in Mr. A’s Webspace. (Jquery.js)

The second asset is the image that we will use for the hanged man. (man.png). It looks like this:

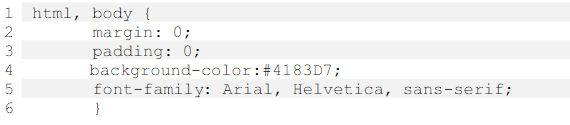


Yes, all the stages of the hanging are in one image file. We will only show one section at a time, like a filmstrip.  You could even customize your own - each segment must be 75px wide and 200px tall and the image file should be 525px by 200px with the filename man.png.

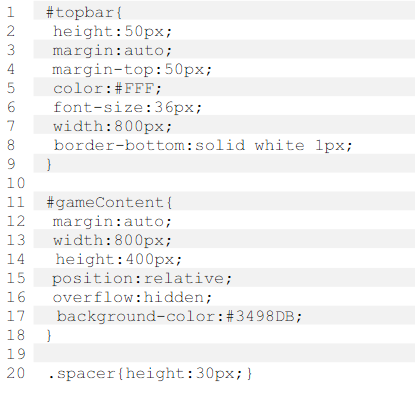
**Step 5**

Build a style sheet

We will use a CSS file to control the look and feel of our game. Later if we need to modify the look and feel, we need only modify the CSS file. Create a document named **main.css.**



This tells us that we will use a shade of blue as the background color for the entire page. Then unless we override it, we will use Arial font, or Helvetica or sans-serif if Arial is not available. We set margin and padding to 0 so that the page will not have any extra spacing.



#topbar will be used simply to display the text Hangman Game above the game area. We want the height of this section of the page to be 50px at a margin of 50px from the top and so on.

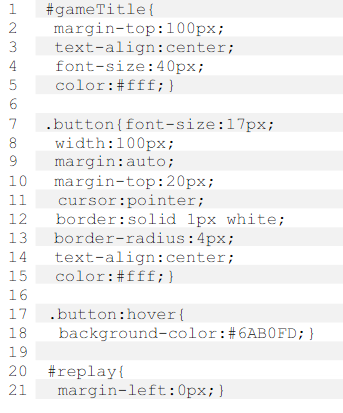
#gameContent controls the style for our game area. We want our game area to be800 pixels wide, 400 pixels in height and centered horizontally on the page

(margin:auto). Anything that ‘sticks out’ of this area will be hidden (overflow:hidden).

We will give it a background color that is slightly lighter than the rest of the page.

Setting these elements at a width of 800px will cause issues if viewed on a mobile device, but don't worry, we'll address this later.

.spacer, is used to set some space between the title and the game area. In CSS, we refer to an ID name by a hashtag (#) and we refer to a class name by a dot (.). Remember, we expect to reuse class names for multiple elements, but the ID name should refer to a unique element. Some standardized elements (body/html) are referred to directly without a hashtag or dot.

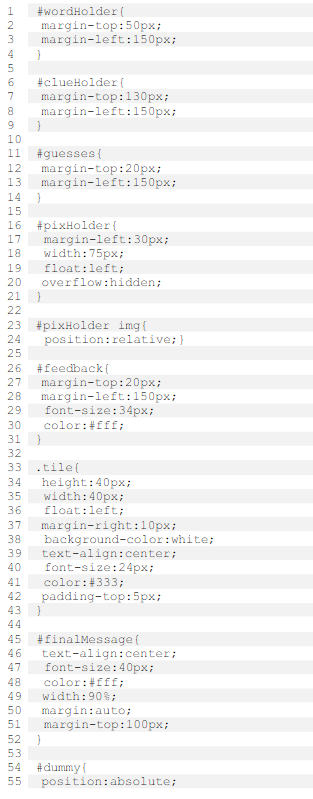


Next we style the game title which will be shown in the game area. We set it at a margin of 100px from the top of the game area (not the page), center it, set the font size to 40 and the color to white.

Then we need to define a style for a simple button. We will use this as a PLAY button at the beginning of the game and also as a REPLAY button at the end of the game. Note that we give it a border for effect. Line 11 ensures that the hand pointer is displayed when our button is moused over, signalling to the user that it is clickable. Line 13 sets slightly rounded edges for the border. Lines 17-18 set an effect when the button is moused over - a change in background color.

Finally, on lines 20-21, we add a little extra code for our REPLAY button. We want itto be on the left, not centered like the PLAY button.

Let’s finish off our CSS:





This is the style information for the gameplay:

#wordHolder is the area on the screen which will hold the blank spaces where the word will go.

#clueHolder will display the current clue

#guesses will be the area that displays letters that have already been guessed.

#pixHolder will hold the image of the hanged man. Note that the width is 75px and any overflow is hidden, so that only one segment will be shown at a time. Lines 23-34 dictate that the position of any image within pixHolder will be set relative to pixHolder (instead of relative the entire page or relative to the game area).#feedback will hold the feedback message where we inform the player that he has won the game or that he has run out of guesses.

.tile dictates the style that we will apply to the individual blanks - where each letter will be displayed. We use a class (.) instead of a div (#) because we will have more than one of them.

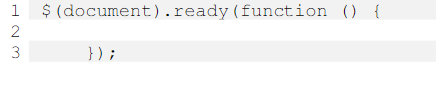
#finalMessage dictates the style for our final message of the game - to tell the player that he has finished all of the words.

#dummy has a special purpose to ensure the game works on mobile devices – we will come back to this later. That will do for the CSS for now. We will add some more later to ensure the game works on a mobile phone. You can open the index.html file in a browser and see how the page looks.

**Step 6**

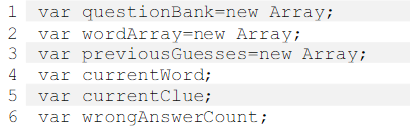
 Creating the JavaScript code

Create a document called **controller.js**, save it and open it with Notepad++ or Brackets.



This is how we begin and all of our code will fall between these two lines . It basically says that we will do the following as soon as the document is ‘ready’ in the browser.

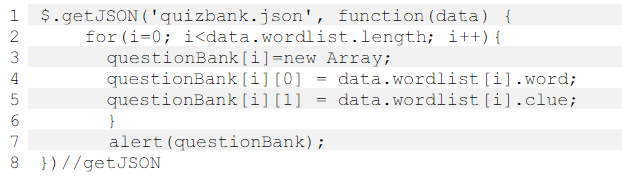
We’ll begin by defining the various variables we will need to use throughout the game:



We will always try to keep our variables ‘human and machine readable’. For example, we can see the variable currentClue refers to the current clue. Following this principle makes it really easy to read your code.

The first three variables are arrays. An array is a series of variables. For example, if a=["tiger","lion","panther"] then a[0] is tiger, a[1] is lion and so forth. We use an array to hold our question bank, which we will read from our JSON document. We will use wordArray to split up and hold the letters of the current word. previousGuesses will contain all the letters previously guessed by the user. We need the variable wrongAnswerCount to track how many wrong answers have been entered. These are the global variables - used throughout the code. We will also have some local variables (used in only one function) which we will define later.

Right then, we need to get the data from our JSON file into a useable format in controller.js. Here’s how:



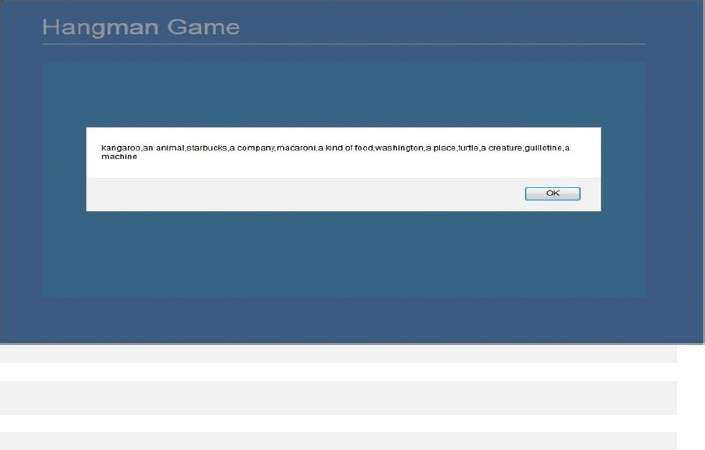
We use the command $.getJSON to read our JSON data and call a function to format it. The $.getJSON() command is an example of a function from JQUERY. Without our JQUERY extension, we would not be able to use this code. Code that begins with a ‘$’ references JQUERY.

 We loop through all the JSON elements inside the wordlist element, i.e. all of our words and clues. Then we use the data to form an array of information called questionBank. We are using a two-dimensional array here:

One dimensional array: questionBank = ["cat","dog","fox"]; (questionBank[1]= "dog")

Two dimensional array: questionBank=[["cat", "dog", "fox"], ["lion", "tiger", "zebra"],["kangaroo", "koala", "wallaby"]]; (questionBank[1][2]= "zebra")

Line [7], alert(questionBank); is only for testing; it will display the contents in an alert(pop up) window so that we can ensure everything is working so far. Now would be a good time to test the app. Launch the website in a browser of your choice. You should see a pop up window displaying the contents of the database, as shown below:



**Step 7**

Displaying the game screens

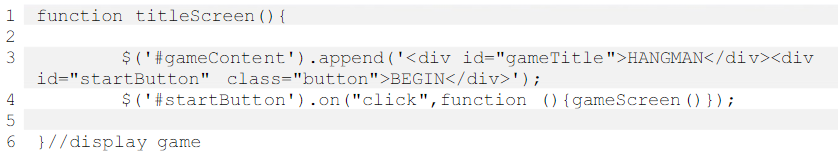
Find the line that says alert(questionBank);

Delete it and in its place add:titleScreen();

This will call a function which creates a simple title screen.

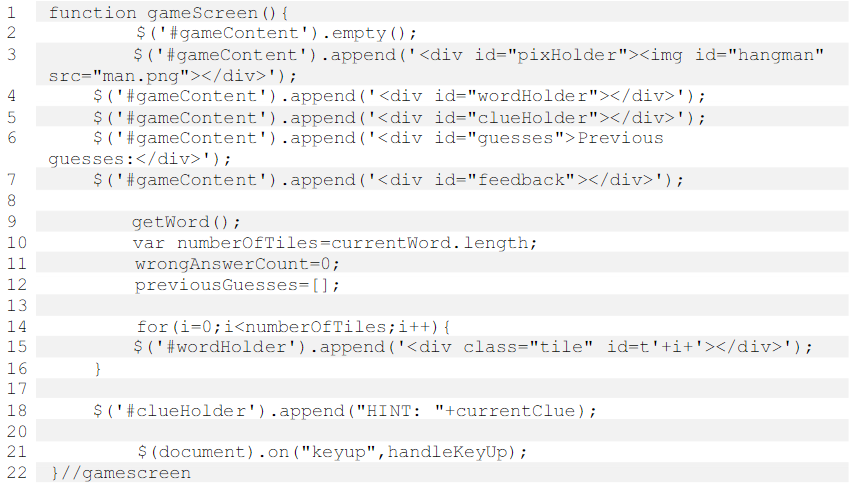
We will write this function now.

It can be placed immediately after the previous code that ends with the line:})//getJSON



What we are doing here is adding content to the empty container named #gameContent - a title and a start button. The styles for these elements have already been put together in our CSS file.

Next, we add a 'click handler' to the start button. It says, 'when the button is clicked, call the function gameScreen()'. The gameScreen function will handle functionality during the game. We will write this function now.



There we go. The first thing is to empty the #gameContent container, thus removing our 'intro screen'. [2].

Next, we add the sections that will make up our game: #pixHolder, containing our graphic - man.png, then the areas that will hold our blanks, clue, previous guesses and feedback [3-7].

Next, we call a function getWord(), which will choose a random word from our databank. [9]

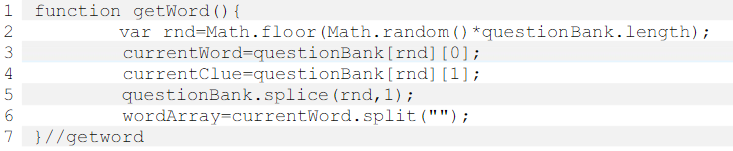
We will write this function in a moment. getWord will generate the variable currentWord and we use the length of this word to determine how many blank tiles to display. We reset the number of wrong answers to 0 and we make sure our previousGuesses array is empty [10-12].

Lines 14-16 add blank tiles to the screen and gives each tile an identifier: t0, t1, t2, t3...

Line 18 adds our clue to the screen.

Line 20 adds a 'listener' to our document (web page). When a key up is sensed – that is, when a user presses and releases a key - the function handleKeyUp is called. We will write this function presently.

Now there are two functions that we have called in our code, but not yet written. Let's do the first one:

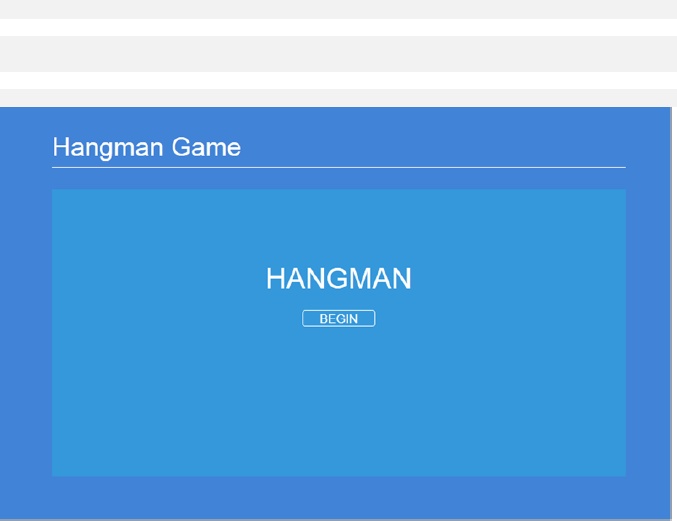


We start by defining a variable rnd and using it to select a random variable between 0 and the length of our questionBank. With this variable, we set the current word andcurrent clue. [2-4]

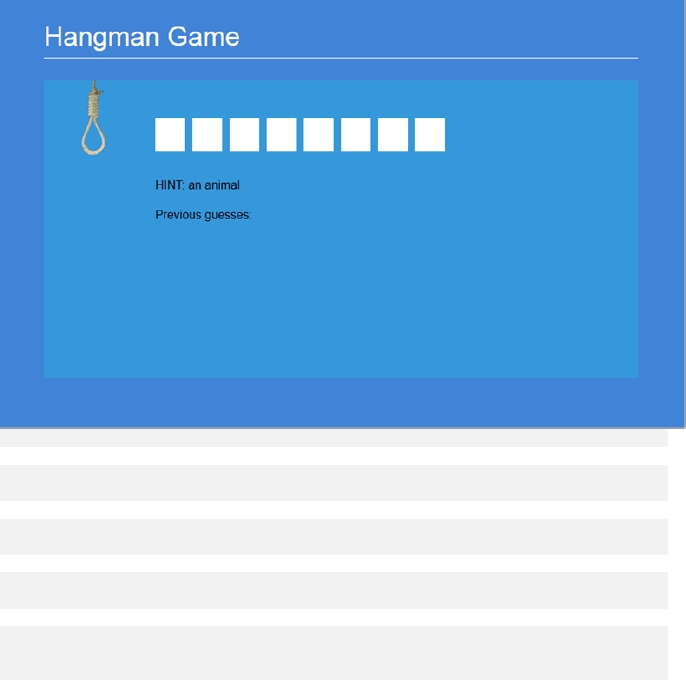
We then splice the questionBank array, meaning we remove the word and clue that we have just chosen. [5]

Finally, we form an array containing the letters of the current word by splitting it.

Now would be a good time to test our game. You should see this:



...and when you click BEGIN, you should see this:



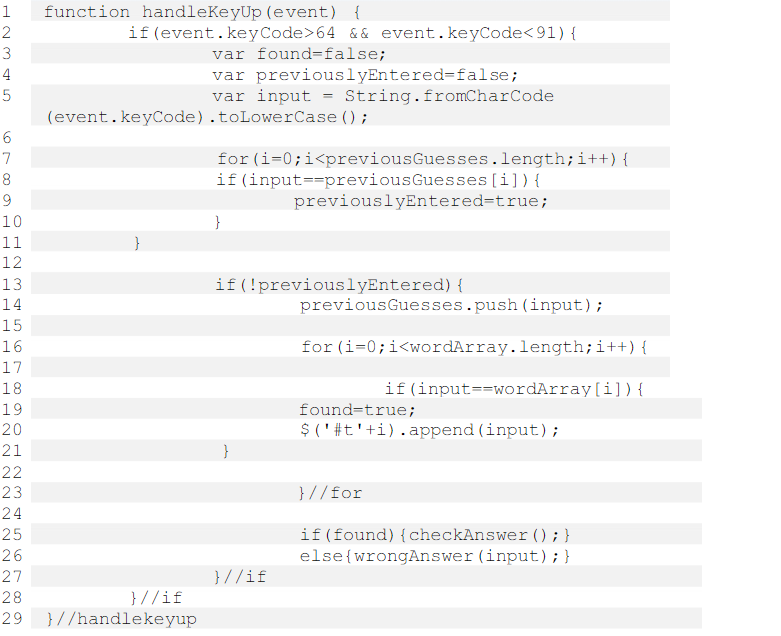
Try refreshing the page and clicking again. You should see a different word each time.

If it's not working, go back and check the code. Otherwise, let's add the functionalitythat we need when the user presses a key:

**Step 8**

Adding functionality

We have already added code to listen for a keypress, but we haven't added code to deal with it. Let's do that now:



On line 1, we begin the function and indicate that it is acting upon an event, in this case a key press.

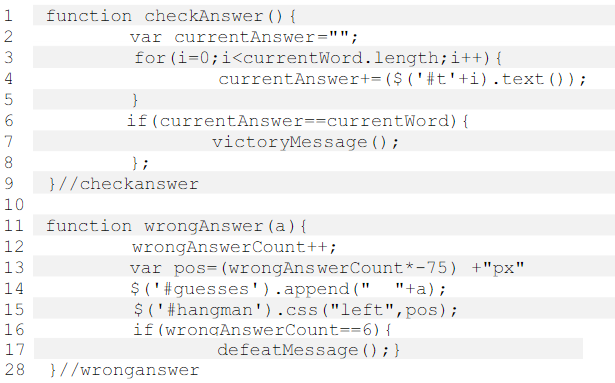
Key presses are distinguished by codes in Javascript, where a=65, b=66... z=90.Hence, in line 2, we ignore any other key presses, such as a comma or a number.

Lines 3-5 define some local variables. The variable found will help us determine if the submitted letter is found in the word. previouslyEntered will help us determine if the letter has been previously submitted. The variable input converts the code to a letter and ensures that it is lower case.

Lines 7-11 check whether the letter has been entered before. Line 13 dictates that the rest of the function should only be carried out if the letter has not already been input. If the user enters a letter twice, the second entry will simply be ignored. Line14 adds the current letter to the list of previosuly entered letters.

Lines 16-21 check whether the letter appears in the word. If so, the letter is added to the appropriate tile, which we can reference using the jquery shorthand $('#t'+i).

Line 25 specifies a function checkAnswer() - this will check whether the entire word has been solved. Line 26 specifies that if the letter was not found in the word, we should call a function wrongAnswer() and send it the variable 'input'. We still need to write these two functions. However, you can test the game again now and you should see that correct letters can be entered.



On lines 1-9 above, we have the function checkAnswer() which declares a local variable currentAnswer and then uses it to store the user input so far by adding to it the text from each of our answer tiles. If the current answer is the same as the current word from our database, then we use the function victoryMessage() to display a victory message [6-8].

If a wrong answer was inputted, we use the function wrongAnswer(), which labels the inputted letter as 'a' [11]. We increase the value of the variable wrongAnswerCount by one [12].

We use the number of wrong answers to recalculate the position of our hanged man image [13] and then apply the new position [15]. You can observe that we do this by changing the CSS rule that applies to the div#hangman. This technique of changing the CSS rule on the fly can come in very handy for simple games such as this.

Meanwhile, we display the incorrect guess on the screen [14]. We add a couple of spaces so that the incorrect guesses are not clumped together. Finally, we know that if the user has entered six incorrect guesses, the hanging is complete and the user has lost the round. Therefore we need to display a message using the function defeatMessage(). [16-17]



The first function displays our victory message with a button to click when the player is ready to continue. First we remove the key handler that we instituted [2]. next we display the message and button in the area we previously designated as '#feedback'[2]. Then we write some code to act on the CONTINUE button when clicked [4-8].This code will check whether we have any questions left in the databank [5] and if so, will restart the game[6]. If not, the final page will be displayed [7].

Line 11-19 essentially mirror the first function, but display a message saying that the player has lost and also displaying the answer that the player failed to guess. When all the words have been used up, we send the player to a final page which is described above in lines 21 - 24. We simply clear everything off the game area and display a message.

Test the game at this point - everything should be working.