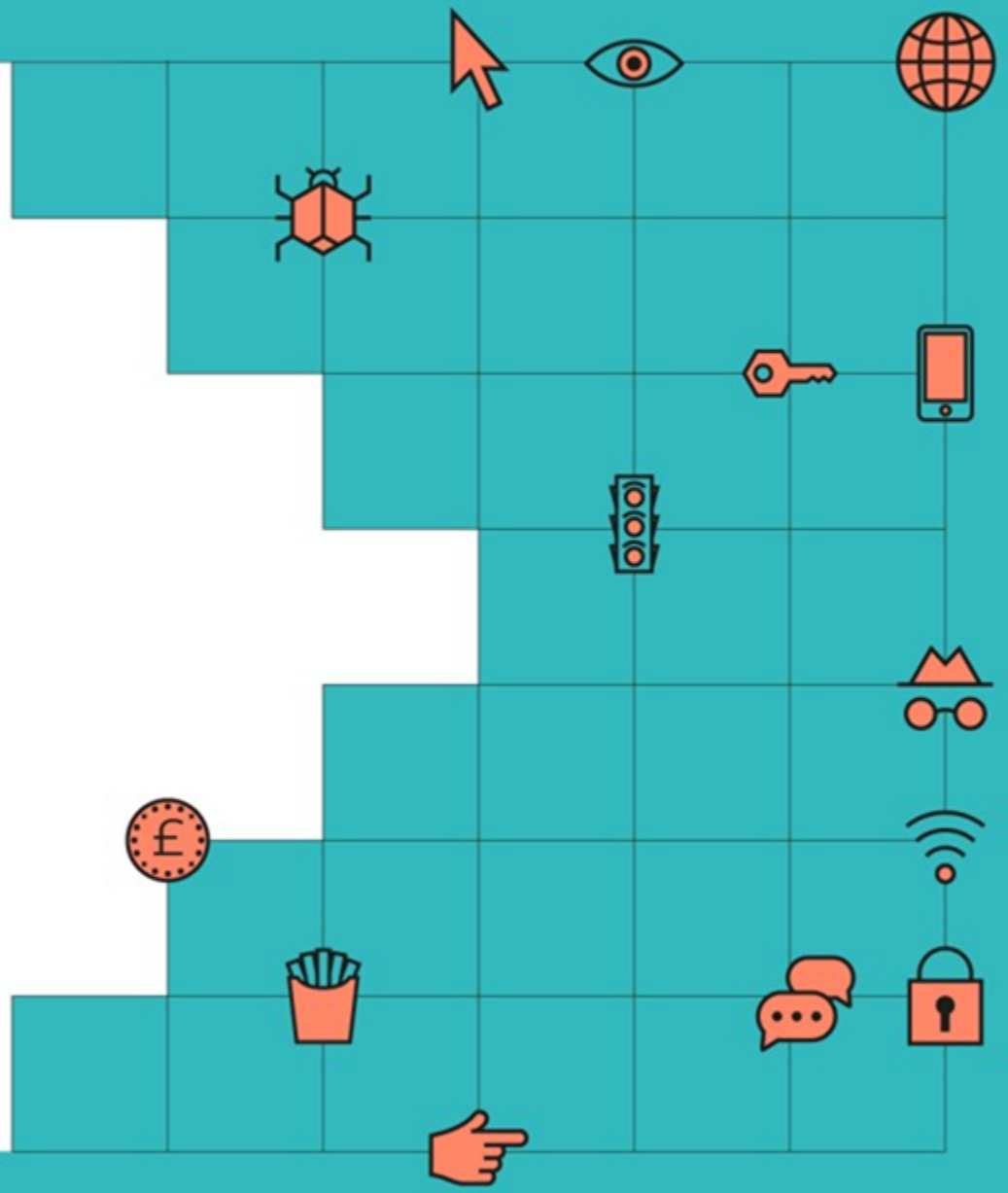
[EM]POWER

How to programme a card matching game

CGI



Learning Objectives

These instructions will guide you on how to create a simple memory matching card game, whilst at the same time teaching you techniques used in Computer Programming using web based programming tools and languages.

What do you need before you start

Access to a Code Editor

This can be something as simple as a text pad editor such as Notepad or Notepad++.

However, if you would like a smoother and more integrated experience it is advised to install something like Visual Studio Code (VS Code) for easier viewing and editing of code. VS Code is available for Windows, Linux and Mac at https://code.visualstudio.com/Download

Access to Web Browser

Any web browser such as Google Chrome, Mozilla Firefox or Microsoft Edge.

What do you need before you start

Source Training Materials

In the project material provided, you will see a ‘game’ folder, within this there are two folders:

starter

solution

This tutorial will start by explaining the contents of the ‘starter’ folder and then you will be building up the code to create the Memory Match Game. Once complete this will have the same content as the ‘solution’ folder.

If at any point you feel stuck or need help, then please look at the solution to help point you in the right direction.

IMPORTANT: Remember this is only there to help you if you are **really** stuck.
Something to keep in mind when solving problems, is that if you try to solve the problem first, you will learn a lot more by doing this rather than looking at the solution first.
It is ok to make mistakes here also, so do not be worried if the application does not work properly on your first attempt.

Introduction

Throughout the creation of this game, we will be using three core files each with their own concept, and it is important to understand the difference between each of these files.

These are as follows:

HTML

- HTML stands for ‘Hyper Text Markup Language’.
- *HTML* is the standard mark-up language for creating web pages.
- *HTML* describes the structure of a web page and it consists of a series of elements. It tells the browser what elements to display and where to display them.
- All HTML files have a **.html** extension at the end of the file name.

CSS

- CSS stands for ‘Cascading Style Sheets’.
- CSS is the language used for describing the presentation of web pages, including colours, layout and fonts.
- All CSS files have a **.css** extension at the end of the file name.

JavaScript

- JavaScript is one of many programming languages for the web.
- It is used to control what happens when someone interacts with a web page, for example: clicking a button, loading a page, etc.
- All JavaScript files have a **.js** extension at the end of the file name.

What is the game all about?

You may have played this game before on a phone, tablet or PC. The aim of the game is to match all cards in the grid by finding matching pairs until all the cards have been successfully matched.

Here is a screenshot of the game in this tutorial part way through game play:

However, have you ever wondered how someone would write something like this and all of the other much more complex games you have played?

Let’s Start

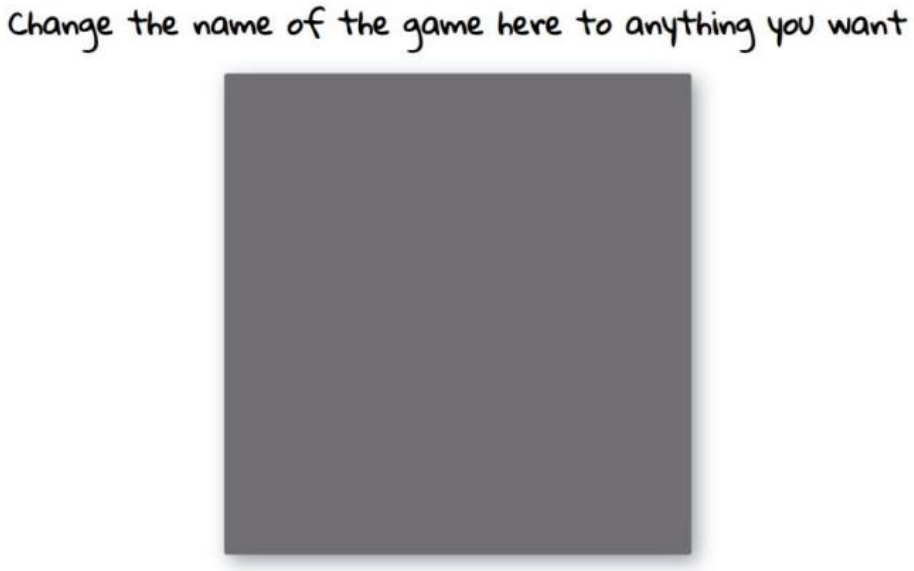
Load the game

To load the game, you will need to open your Web Browser, e.g. Google Chrome, Microsoft Edge, Firefox.

Once there find the location of the **index.html** file in the ‘starter’ folder and type the file location including the **index.html** file name into the address bar. e.g. c:\users**username**\Documents\MemoryMatch\game\starter\index.html.

IMPORTANT, please swap **username** for your user name.

When you load the page in your web browser from the ‘starter’ folder for the first time, you will not be able to click on anything; this is what we are going to build up as we go. You should see the following:



Let’s change something

Now that you know where the **index.html** for the ‘starter’ folder is located, open the **index.html** in the text editor of your choice, e.g. Notepad, Notepad++, Visual Studio Code etc.

Locate the text, which is above the grey box in the picture above, and change that to something else of your choice, it does not matter what you enter here. An example could be **Memory Match Game**, **Memory Matcher** or **Joe’s awesome memory game**.

Once you have made the change please make sure that you save the changes to the file. You will need to do this at all stages throughout the tutorial as you build up the game.

Now either hit the refresh/reload button in the web browser or press the F5 key on the keyboard. You should now see the text change to the text you entered. **Pretty cool, don’t you think!**

Before we make any serious changes, let us understand what we are starting with.

We have three files in the ‘starter’ folder, these are:

app.css

This file contains the stylesheet language to control the presentation of the elements that will appear on the web page. E.g. font, colours etc.

For this tutorial, this file is complete. You will be only making a few changes to alter how things are displayed.

app.js

This file contains the programming language that will decide what to do when the user of the game clicks buttons on the web page.

It could be that the user gets a match and we need to let them know that they have a match, or it could be that they do not have a match yet and so they need to try again.

We start by declaring a placeholder to hold all of the cards in our deck of cards. This is done by declaring it as a **constant**, which means the deck will never change, even though the cards within it will get re-shuffled on each play.

```
// deck of all cards in game
const deck = document.getElementById("card-deck");
```

Index.html

This file contains the mark-up language that will control where elements on the page are displayed. It also pulls in any other resources that are needed to allow these elements to be shown to the user.

These resources as well as some others include both the **app.css** and **app.js** files. See if you can spot the references to both of these files in the **index.html** file?

We start with a basic layout that currently shows the empty grey box in the screenshot above, this is essentially empty the **card-deck** declared in the **app.js** file above.

```
<head> 1
  <link rel="stylesheet prefetch" href="https://maxcdn.bootstrapcdn.com/font-awesome/4.6.1/css/font-awesome.min.css">
2 <link rel="stylesheet prefetch" href="https://fonts.googleapis.com/css?family=Coda">
  <link rel="stylesheet prefetch" href="https://fonts.googleapis.com/css?family=Gloria+Hallelujah|Permanent+Marker">
  <link rel="stylesheet" href="app.css"> 3
</head>
<body> 4
  <div class="container"> 5
    <header>
6     <h1>Change the name of the game here to anything you want</h1>
    </header>
    <ul id="card-deck" class="deck"> 7
    </ul>
  </div>
  <script src="app.js"></script> 8
</body>
```

1. The head is the first thing that is loaded into the browser.
2. External resources containing pictures and fonts to be displayed in the application.
3. This is a reference to the Cascading Style Sheet we have control over. CSS.
4. The body is loaded after the head has been loaded.
5. We need an area to hold the grid that will display the cards.
6. The header is displayed at the top of the page.
7. We need a list of cards, to keep a track of the types of cards we will be using.
8. This is a reference to the JavaScript file we have control over.

```
<head> 1
  <link rel="stylesheet prefetch" href="https:/
2 <link rel="stylesheet prefetch" href="https:/
  <link rel="stylesheet prefetch" href="https:/
  <link rel="stylesheet" href="app.css"> 3
</head>
<body> 4
  <div class="container"> 5
    <header>
6     <h1>Change the name of the game here
    </header>
    <ul id="card-deck" class="deck"> 7
    </ul>
  </div>
  <script src="app.js"></script> 8
</body>
```

Please note that wherever you see a ***/*** in a html file, this means it is the end of that block, e.g. <head>…….<***/***head> is the entire ‘head’ element.

Where are the cards?

Let us load some cards.

For the cards in this game, we have a selection of pictures to use:

1. Diamond
2. Bicycle
3. Plane
4. Anchor
5. Bolt
6. Cube
7. Leaf
8. Bomb

In order to make matching pairs, we need two of each picture. Therefore, this will need to be sixteen cards.

In the html file, we can now add cards to the ‘card-deck’ element. Look below to get started and read the key below to understand the elements.

Add the two diamond cards and the other fourteen cards.

```
<ul id="card-deck" class="deck">
            <li class="card" type="diamond">
              <div class="fa fa-diamond"></div>
            </li>
            <li class="card" type="diamond">
              <div class="fa fa-diamond"></div>
            </li>
</ul>
```

Key:

- **ul** => unordered list – this means that we can randomise the position of the cards later on in the JavaScript code.
- **li** => list item - an entry in the list
- **div** => placeholder – in this instance it will hold the picture to be displayed
- **class** => reference to the CSS file for displaying an element, in this instance it is the '**.deck .card'** part entry in the CSS file, see corresponding highlighted entries above.
- **type** = > description of card to be used by JavaScript to identify a matching set.

Once you have added all sixteen cards then save the .html file and refresh the web page, .e.g press the F5 key.

You should see sixteen black boxes now, representing the sixteen cards.

Where are the pictures

If you look in the CSS file, you will see entries under **‘.deck .card.match**’, ‘**.deck .card.open**’ and ‘**.deck .card.show**’

See what happens when you add, save and refresh (F5) the following to any of the cards:

```
<li class="card match" type="diamond">
    <div class="fa fa-diamond"></div>
</li>
<li class="card open show" type="diamond">
    <div class=" fa fa-diamond"></div>
</li>
```

It does not matter if you leave these in or not, this is because the JavaScript code we write later on will overwrite these values when we play the game.

Let us change some pictures

At the top of the html file, you will see link to a resource at https://maxcdn.bootstrapcdn.com/font-awesome/4.6.3/css/font-awesome.min.css

Copy this link into the address bar in your web browser and go this location. **IMPORTANT**. Do not worry about the fact that this is not very readable; this is used by the web page to retrieve the pictures.

When in this web page, go into the search option, e.g. press Ctrl + F on the keyboard. Type one of the known eight pictures, e.g. **fa-diamond**. You will see it highlighted.

Now see if you kind find some other interesting pictures in there, e.g. **fa-microphone**, **fa-recycle**, etc. The list goes on and on with lots of other pictures.

When making the changes in your html file just make sure that you change both matching pairs. If you do not do this then the game will not work properly.

For example, here is a change to use a microphone picture.

```
<li class="card match" type="microphone">
    <div class="fa fa-microphone"></div>
</li>
```

Let us make the game actually do something!

Before getting started, we should understand how to look at the JavaScript code whilst it is running.

In order to work out if there are any issues in our JavaScript code, we sometimes need the ability to look at what is actually happening '***behind the scenes***' when the application is running.

Understanding what is happening in this way makes use of a technique known as **debugging,** this can be done in various ways.

The debugging technique we will use in this tutorial will involve writing out information to the screen and then making decisions based on that.

Try this out.

1. Go to your app.js file and below the line that declares the ‘deck‘, use a **console.log(deck);** statement.

Don’t forget the ending semi-colon, all lines will need this.

```
// deck of all cards in game – anything we a // prefix is treated as a comment and will not affect the running of the application
const deck = document.getElementById("card-deck");
```

```
console.log(deck);
```

2. Go to the web browser and press the F12 key, this will give you visibility of the ‘developer tools’.

3. Look for a tab named ‘**Console’** and click on this.

4. Now refresh the page. Each time you refresh the page, information will be written to the console and you can use this to look at details about your application.

5. To hide the ‘developer tools’, press the F12 key again.

NOTE: Please note that you can leave as many console.logs in the code as you wish, however you may want to clean them up after a while as the output does start to grow.

Let us make some JavaScript code changes

Declare all of our variables

Add the cards we created in the html file to the deck

```
// cards array holds all cards
let cards = [...document.getElementsByClassName("card")];
```

In the above code, this will look in the currently loaded document, in this case, our html file.

It will then find all elements with a class name of ‘card’ and then it will then load the cards into an **array** of cards.

NOTE:
1. Please note that an array is a way of saying it is a list.
2. The syntax with three dots above is known in JavaScript as a ‘spread operator’ and allows loading of elements into an array.

If you wish, you can add a console.log at any time to see what these look like when the code is running. e.g. **console.log(cards);**

Start the game with zero moves

```
// declaring move variable
let moves = 0;
```

We need to tell the game that we are starting, so create a counter to track how many moves we have made.

Create variables to track how many matched and opened cards we have

```
// declaring variable of matchedCards
let matchedCard = document.getElementsByClassName("match");
```

This will find all elements in the html file, in our case the cards, with a class name of ‘match and then it will then load these matched cards into an array called ‘`matchedCard`’

```
// array for opened cards
let openedCards = [];
```

This will create an empty array, essentially a list, to hold any cards that are open at any one time.

Create a shuffle card function

```
// shuffles cards
function shuffle(array) {
    let currentIndex = array.length;

    while (currentIndex !== 0) {
        randomIndex = Math.floor(Math.random() * currentIndex);
        currentIndex -= 1;
        temporaryValue = array[currentIndex];
        array[currentIndex] = array[randomIndex];
        array[randomIndex] = temporaryValue;
    }

    return array;
};
```

This will create a function that will take an array as input, in this instance the cards array, and shuffle them using a randomiser method.

NOTE: A function is a contained set of logic that is called from other areas in the code to run specific tasks.

Test the shuffle card function

To test this, directly below this code call the **shuffle** command and return the first two cards.

```
let shuffledCards = shuffle(cards);

console.log(shuffledCards);
console.log(shuffledCards[0].childNodes[1]);
console.log(shuffledCards[1].childNodes[1]);
```

Each time you press F5 to refresh the screen you will see different cards in the console for the first two cards in the cards array.

Something to note here is that indexes in computer programming commonly start at 0 rather than 1.

IMPORTANT: once you have tested the above code, please delete this as this could break the application if is left in.

Create a start game function

```
// function to start a new play
function startGame(){

    // shuffle deck
    cards = shuffle(cards);

    // add cards to the deck
    for (var i = 0; i < cards.length; i++){

        [].forEach.call(cards, function(item) {
            deck.appendChild(item);
        });
    }
}
```

This will create a function that will:

1. Shuffle the cards, as described in the previous step.
2. Add each shuffled card to the deck, by going through the array of shuffled cards.

Tell the application to start the game as soon as we load the page

```
// shuffles cards when page is refreshed / loads
document.body.onload = startGame();
```

This will look at the currently loaded document, in this case our html file. It will then run the **startGame** function from the previous step.

To look at the shuffled deck add a console.log.

```
console.log(deck);
```

Add event listener to display a card when the user clicks a square

```
// loop to add event listeners to each card
for (var i = 0; i < cards.length; i++){
    let card = cards[i];
    card.addEventListener("click", displayCard);
};
```

Event listeners exist to listen and fire a command based on an action, usually performed by the user.

In the instance above the event listener attached to each card, fires the displayCard function when a user clicks on any card. We will create this function next.

Create a display card function

```
// toggles open and show class to display cards
function displayCard(){
    this.classList.toggle("open");
    this.classList.toggle("show");
    this.classList.toggle("disabled");
};
```

The above will change the class associated against each card and use the settings in the CSS file to display the card that was clicked.

If you look in the .css file you will find three entries that relate to these settings:

`.deck .card.open` – This controls things like the colour of the card, the transformation action, the cursor etc.

`.deck .card.show` – This controls the size of the font, in this case because we are using a special font type that are images, it controls the size of the image.

`.deck .card.disabled` – This disables the card from being clicked more than once. Please note in order for the game to work correctly we will change this behaviour later on.

Try this out

Try experimenting with some of these settings to see how it changes the behaviour and display of the cards.

For example:

Change the font size in the `.deck .card.show` settings

IMPORTANT: In order to see these changes you will need to save the file and refresh the web page, e.g. press F5

Now each card you click will turn over and will be displayed, you will see that there are eight pairs of cards in random order if you turn them all over.

Add an event listener to check if opened card is a match to the previous card

Add the below line in bold to your existing event listener.

```
// loop to add event listeners to each card
for (var i = 0; i < cards.length; i++){
    let card = cards[i];
    card.addEventListener("click", displayCard);
    card.addEventListener("click", checkCard);
};
```

In the instance above the new event listener will call a function to check if the opened card is a match to the previously opened card. We will create this function next.

IMPORTANT: If you save the file and refresh the web page at this point you will not be able to click on all of the cards, as it requires the next function to be in place for the event listener to work.

Create a check card function

```
// add opened cards to OpenedCards list and check if cards are match or not
function checkCard() {
    openedCards.push(this);
    const len = openedCards.length;
    if(len === 2){
        moves++;
        if(openedCards[0].type === openedCards[1].type){
            matched();
        } else {
            unmatched();
        }
    }
};
```

This will take the current card and add it to the array of ‘openedCards’ that has been declared at the top of the JavaScript file.

It will then check to see if the number of cards in the ‘openedCards’ is equal to 2 cards, if so it will do two things:

- increase the number of moves made by one. (This keeps track of how many moves has been made)
- perform a check to see if the two cards selected match each other. We will need to write the ‘matched’ and ‘unmatched’ functions next.

To see what is happening here, try writing out some console.logs as described previously at the bottom of the function.

For example, see highlighted line below.

```
function checkCard() {
    openedCards.push(this);
    const len = openedCards.length;
    console.log('Opened Cards Length: ' + openedCards.length);

    if(len === 2){
        moves++;
        if(openedCards[0].type === openedCards[1].type){
            matched();
        } else {
            unmatched();
        }
    }

};
```

Remember to save the file and refresh the page before running this.

Each time you click a card you will see the openCards length increase.

IMPORTANT: Do not worry about the error message when hitting the second card in the count, we are yet to write the ‘matched’ and ‘unmatched’ functions, which is causing this to error.

Once finished with this step you can leave this console.log statement in or take it out, as it will not affect the application. It is generally considered good practice to remove console.logs when they are no longer required. This is to keep the code clean.

Create matched and unmatched functions

Match function

```
// when cards match
function matched(){
    openedCards[0].classList.add("match", "disabled");
    openedCards[1].classList.add("match", "disabled");
    openedCards[0].classList.remove("show", "open");
    openedCards[1].classList.remove("show", "open");
    openedCards = [];
}
```

If a match is successful, this will do the following:

1. For both of the matched cards, it will apply the '**match'** and '**disabled'** classes from the .css file. See .deck .card.match and .deck .card.disabled in the .css file.
2. For both of the matched cards, it will remove the currently applied '**show'** and '**open'** classes. See .show and .deck .card.open in the .css file.
3. It will reset the openCards array back to an empty array, ready for the player to find the next set of matching cards.

Unmatched function

```
// when cards don't match
function unmatched(){
    openedCards[0].classList.add("unmatched");
    openedCards[1].classList.add("unmatched");
    disable();
    setTimeout(function(){
        openedCards[0].classList.remove("show", "open", "unmatched");
        openedCards[1].classList.remove("show", "open", "unmatched");
        enable();
        openedCards = [];
    },1500);
}
```

If a match is unsuccessful, this will do the following:

1. For both of the unmatched cards, it will apply the '**unmatched'** class from the .css file. See .deck .card.unmatched in the .css file.
2. It will call the disable function, which will stop the user clicking on the two selected cards. This will allow the rest of the function to run without interference from the user clicking the cards again. We will need to write this '**disable'** function next.
3. After a one and half second interval, represented by 1,500 milliseconds, it will do the following:
    i. For both of the unmatched cards, it will remove the currently applied '**show'**, '**open'** and '**unmatched'** classes. See .show, .deck .card.open and .deck .card.unmatched in the .css file. This resets the cards so they can be tried again for another match.
    ii. It will then call the enable function, so that user can now click one of the two unmatched cards again to try a different match. We will need to write this '**enable**' function next.
    iii. It will reset the openCards array back to an empty array, ready for the player to find the next set of matching cards.

Create disable and enable functions

Disable function

```
// disable cards temporarily
function disable(){
    Array.prototype.filter.call(cards, function(card){
        card.classList.add('disabled');
    });
}
```

This will apply the '**disabled'** class from the .css file to all of the cards. This is to ensure that whilst the code is being ran to process unmatched cards, the user cannot click on any cards. See .deck .card.disabled in the .css file.

Enable function

```
// enable cards and disable matched cards
function enable(){
    Array.prototype.filter.call(cards, function(card){
        card.classList.remove('disabled');
        for(var i = 0; i < matchedCard.length; i++){
            matchedCard[i].classList.add("disabled");
        }
    });
}
```

This will do two things:

1. Firstly, it will remove the currently applied '**disabled'** class from the .css file for all cards. This will allow the user to now click on any card.
2. Secondly it will re-apply the 'disabled' class to all existing matched cards, so that they cannot be clicked on.

Let the user know how they can restart the game

Add a footer to the html file to notify the user on how to restart the game. Please see bolded footer tag below.

```
<div class="container">
        <header>
            <h1>Change the name of the game here to anything you want</h1>
        </header>

        <ul id="card-deck" class="deck" >
                ……….
        </ul>
        <footer>
                <h1>Refresh or press F5 to restart game</h1>
         </footer>
    </div>
```

Now let us play!

Congratulations, we have now finished coding the Memory Match game, and you can now challenge your memory in getting all the matches in as few moves as possible! Good luck!

Credits

Please note that the original source code for this tutorial has come from https://github.com/sandraisrael/Memory-Game-fend

The aim of this tutorial has been to take this content and apply a structured breakdown to be used as a teaching aid.

Further development

See if you can make this game with your own style, think of things like:

- Changing the pictures
- Changing the colours of the cards
- Changing font sizes

If you would like to take this game to the next level, including:

- Tracking number of moves
- Keeping a score rating
- Adding a timer

In the source materials for this project there is a folder named ‘advanced_features’. This contains the same three files we have been working with containing additional code in the .html and .js files.

If you are up to the challenge, see if you can work out how things are behaving in here, this would be a good time to use console.logs for debugging purposes. Good luck!

Have further questions about this activity?

You can contact Craig.murch@cgi.com

[EM]POWER

If you've enjoyed these EmPower activities then ask your teacher to sign you up to the 2022 CyberFirst Girls' Competition at ncsc.gov.uk/cyberfirst/girls-competition

Registration opens 18th October 2021.