Memory Game

Part 3: Setting up click events

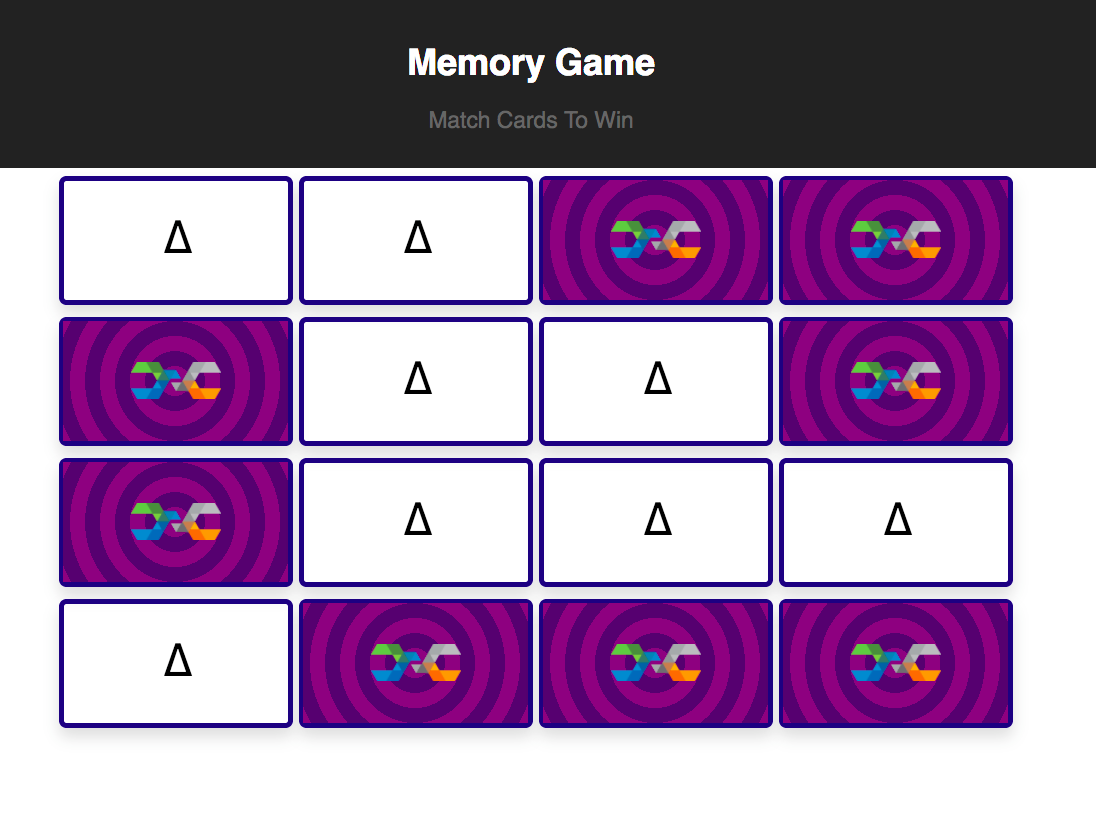
### Objective

We’re going to be coding a simple memory game using ReactJS as our framework and create-react-app as the foundation.

If you haven’t played a memory game before, the rules are quite simple. The game starts with 16 face down cards on the table. On the other side of each card is a easily recognizable symbol, and there are 8 pairs of symbols arranged randomly among the 16 cards. The player will flip 2 cards at a time, trying to flip over 2 cards with the same symbol. If the player is successful, the cards are remain flipped over. Otherwise, the cards are flipped back over. The player keeps trying to flip over pairs until all of the cards on the board are face up. The goal of the game is to match all the cards in as few flips as possible.

### Your task for part 3:

If you finished part 2, you should have an app that renders something like this:



Clicking each card should toggle its flip animation. Now we’re at the point where the *visual* mechanics of our application are in place, but now we need to top it off by implementing the *logic* of the game. Changes in our game state are going to control the flipping animation, which means we need to edit some of our existing code.

### Let’s do it!

# Step 1 - Setting up the initial application state

As it stands, our application doesn’t implement the logic of “memory game”. For small games like this, it’s typical to hold all of the game logic and state in the top level component - in our case, the **App** component. Once once we set up the game logic in App.js,, the App component can dictate how its MemoryCard children will render by using **props**. (More on props later).

1. Write a basic constructor for your App in App.js
   1. Use MemoryCard.js as a reference
   2. Don’t forget the call to **super()**
2. Initialize the app state with 2 properties: **deck** and **pickedCards**
   1. Again, use MemoryCard.js as a reference.
   2. For now, set **deck** and **pickedCards** to empty arrays

At this point, our App state is almost set up. The only thing that’s missing is filling up the **deck** array with the initial set of 16 “cards” (I put cards in quotes here, because in reality the cards are just basic Javascript objects). We’ll write a helper function that can generate a deck of memory cards for us.

1. Define a function named **generateDeck** between the import statements and the class definition (Somewhere around line 5);
2. The first line of the function should define a variable called **symbols**
   1. Set **symbols** equal to an array containing these characters:
      1. ∆, ß, £, §, •, $, +, ø
3. Next, create an empty array named **deck**.
4. Since we want to push 16 “cards” into **deck**, write a for-loop that will loop 16 times
5. In the for loop, let’s push an object into the **deck**
   1. The object we push should represent a card
   2. The object should have two properties:
      1. **isFlipped**
      2. **Symbol**
   3. Set **isFlipped** to false
   4. Set **symbol** to **symbols[i%8**]
6. Copy-paste the shuffle function from stackoverflow after **generateDeck**’s closing curly-brace
   1. <https://stackoverflow.com/questions/6274339/how-can-i-shuffle-an-array>
7. Back in **generateDeck,** make a call to **shuffle(deck)**
8. Finally, just return **deck**

We’ve just written a function that returns an array of cards! It has nothing to do with React, it’s just a basic JS function. Now it’s time to use it in our App constructor:

1. In the constructor where you set **this.state = …**, make the **deck** property point to a call to **generateDeck()**.
2. That’s it!

Alrighty, our App has a brain now! It can keep track of which cards are in the deck, as well as which cards the user is selecting as they play the game.

# Step 2- Mapping state to Components

We’ve created a basic representation of the game using plain javascript, and we dumped it into App’s state. Unfortunately, it’s not useful to us unless we can show the state to the screen. Yes, our App is currently displaying 16 MemoryCards to the screen, but those cards have nothing to do with the game state. Let’s change that!

1. Before the return statement of the render function, create a variable called **cardsJSX**
   1. We’re going to hold an array of <MemoryCard /> elements in this array
2. Set **cardsJSX** equal to **this.state.deck.map();**
   1. Remember what .map() does? It spits out an array with elements that correspond to the elements from the input array (in this case, the input is this.state.deck)
3. Inside the call to **.map(),** write an anonymous function that takes **card** and **index** as parameters
   1. Make sure to use the ES6 syntax for anonymous functions, the “fat arrow” operator
   2. (param1, param2) => { ... }
   3. Make this anonymous function return <MemoryCard />

So at this point, **cardsJSX** should be an array that holds 16 <MemoryCard /> components. We can use .slice(startIndex, endIndex), do render each group of 4 MemoryCards inside of App’s JSX.

1. Inside the return statement of the render function, you probably have 4 groups of 4 <MemoryCard /> tags, wrapped in an empty div.
2. Replace each grouping of 4 with the following (but keep the <div> dividers):
   1. {cardsJSX.slice(0,4)}
   2. {cardsJSX.slice(4,8)}
   3. {cardsJSX.slice(8,12)}
   4. {cardsJSX.slice(12,16)}

Ok great! Visually, nothing should change about your application, but now we have this one-to-one mapping of memorycard state objects to <MemoryCard /> components. Inside of the .map() call, we now have an opportunity to pass down information from the App component to each MemoryCard component

# Step 3- Passing props to each MemoryCard

Right now, each MemoryCard is hard coded to display a ∆ character. Instead, we want the App component to tell each MemoryCard what symbol to display.

On top of that, the MemoryCards are in control of their own isFlipped state. We’re going to change it so that the App component controls whether or not a MemoryCard is flipped.

Introducing: Props! Props look almost identical to HTML attributes (attributeName=”value”), but they’re custom to React. They allow your custom components to pass down state information to children components. In our case, we want to pass down **symbol** and **isFlipped** using props

1. Inside the .map() callback in App.js, add two props to the <MemoryCard /> component in the return statement: **symbol** and **isFlipped**
2. Don’t forget that the anonymous function in .map() is supposed to have two parameteres, **card** and **index.** The card parameter holds the information we need for the next step
3. Set the **symbol** prop equal to **card.symbol**, and set the **isFlipped** prop equal to **card.isFlipped**
   1. symbol={card.symbol}
   2. isFlipped={card.isFlipped}

That should do it for now - At this point, App.js’s render function is telling each MemoryCard which symbol it should display. Let’s change MemoryCard.js to make use of this new information.

1. Change the MemoryCard constructor to take one parameter: **props**
2. Pass this parameter into the **super()** call
3. In the render function, find the ∆ symbol in the JSX
4. Replace it with {this.props.symbol}

Much like this.state, **this.props** is a React reserved variable that holds all of the properties that are passed down from parent components. Since we added a **symbol** prop to each MemoryComponent in App.js, we can use **this.props.symbol** inside our class to reference this prop. We’re using it here to dynamically decide how our MemoryCard will render.

Verify that each card has a different symbol on its front side.

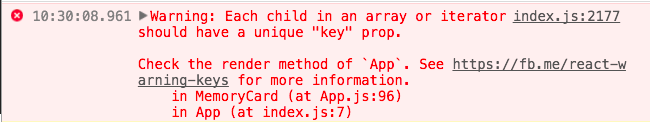
Now, we can make use of **this.props.isFlipped**. This prop is how the App component is going to communicate to its MemoryCards whether or not they should flip over.

1. In your render function, you should already have this:
   1. **If (this.state.isFlipped) { … }**
2. Change it to this:
   1. **If (this.props.isFlipped) { …}**

Now the App component is in control! Notice that your cards won’t flipped when clicked anymore. Why is that?

# Step 4- Make React stop yelling at you by using “key”

You might notice that we’re getting an error in the Chrome console:



The issue lies back in App.js where we build our array of MemoryCard components. This is one of those JSX gotchas - React need you to use a “key” prop so that it can differentiate between identical sibling components. We can set the “key” prop to whatever value we want, as long as it’s unique. In our case, we’ll use the **index** parameter in .map()’s anonymous function.

1. Add another prop to the <MemoryCard /> called **key**
2. Set it equal to {index}
   1. Remember, the anonymous function in .map() should have index as its second parameter

This should fix the console error.

# Step 5- Implement the pickCard method

In the game of “memory game”, there’s a lot to consider when picking cards:

1. You can’t pick cards that are already face up
2. Picked cards get flipped over
3. If two picked cards share the same symbol, they stay face up
4. Otherwise, they get turned back over

Let’s implement these rules in a **pickCard** method in App.js

1. Add a new member function to the App component called **pickCard**
2. Have picked card take a parameter called **cardIndex**
   1. We need to know which card is getting picked, after all!
3. Write an if-statement that checks if the card we’re trying to pick is already flippedOver
   1. The card we’re trying to flip can be accessed by **this.state.deck[cardIndex]**
   2. If the card is in fact already flipped over, just **return;** and end code execution
4. Next, create a variable named **cardToFlip**
   1. Set it equal to **{...this.state.deck[cardIndex]}**
   2. We can’t just set it to this.state.deck[cardIndex] because we’ll be modifying its isFlipped property later. React doesn’t let us change state directly, only with .setState(), so using the spread operator like this will create a carbon-copy of the card we’re trying to flip
5. Change the isFlipped property of **cardToFlip** to **true**

Ok, so at this point we have a variable that represents the card we just flipped. Let’s come back to this variable later, because we have other work to do.

React won’t let us mutate state directly, we have to use setState() instead. Our state has two arrays that we want to push to, but unfortunately arrays are pass-by-reference, so we can’t just do this.state.deck.push(whatever). Instead, we need to create copies of these arrays to use as “scratch paper”

1. Create a new variable called **newPickedCards**
2. Set it equal to **this.state.pickedCards.concat(cardIndex);**
   1. This is a clever use of the .concat() array function. Concat returns a brand new array with all the old contents, but it also inserts **cardIndex** in one fell swoop
3. Create a new variable called **newDeck**
4. Set it equal to **this.state.deck.map();**
   1. .map() also returns a brand new array. This is our clever way of copying the deck.
5. Add an anonymous function to .map() that takes **card** and  **index** as a parameter
6. Inside this anonymous function, return **card**
7. Before **return card;** add an if statement that checks if **cardIndex == index**
   1. Remember, **cardIndex** is the index card that’s being picked, and **index** is just an iterator for the .map() function
   2. Inside this if block, return **cardToFlip**
   3. Essentially, we’re making a copy of the **deck** array, but swapping out the card that we just flipped

Finally, now that we have the updated version of **deck** and **pickedCards** saved to new variables, we can make the call to setState();

1. At the end of the pickCard() function, call this.setState();
   1. Pass in this object as a parameter:
      1. **{deck: newDeck, pickedCards: newPickedCards}**

That’s it for pickCard() for now! We’re going to revisit it in a little bit so that we can make it actually compare two picked cards.

# Step 6- Pass pickCard down to the memory card components

pickCard() is now an action in our game. We need to make it so that clicking on a memory card will call pickCard() for that given card. But there’s an issue… we wrote pickCard() inside of App.js, but the onClick handler lives in MemoryCard.js! What do?

Turns out that we can also use props to pass down functions to child components.

1. Add yet another prop to <Memory Card /> inside of the render function - call it **pickCard**
2. Set this new prop equal to:
   1. **{this.pickCard.bind(this, index)}**
   2. We’re passing down an entire function to the MemoryCard!
   3. bind(this, index) is making sure that the function remembers which App instance it is, and it’s also passing down the cardIndex for when the function gets called
3. In MemoryCard.js, find the onClick attribute in the render function
   1. Instead of this.clickHandler.bind(this), make it equal to **this.props.pickCard**

If you hit save, you should be able to click cards and flip them over! Now, our MemoryCard is totally dependant on the App. In fact, there’s nothing in the render function that’s making use of this.state anymore, so we can delete a few things:

1. In MemoryCard.js, delete the clickHandler() function
2. Delete the entire constructor

Now our MemoryCard is back to being kind of “dumb”. But no worries! The App component is now the brains of the operation, it’s dictating how our MemoryCard should behave via **this.props**

# Step 7- Complete the game logic

We’re able to flip cards now, but we need the App component to unflip any pairs of cards that don’t match.

1. Back in App.js, add an if-block before the call to **setState()** in the **pickCard()** function
2. Make the if-block check if the number of selected cards is equal to 2
   1. Hint: The newPickedCards variable holds the list of selected cards
3. Inside the if-block, create two variables, **card1Index** and **card2Index**
   1. Set them equal to the first and second elements of newPickedCards, respectively
4. Now, nest another if-block inside of here. Have it check to see the symbol of the card at **card1Index** doesn’t match the symbol of the card of **card2Index**
   1. This line will end up being a bit long.
   2. Remember to reach inside of the **newDeck** array to extract a card to inspect
5. For now, just type a comment that says “unflip cards” inside the inner if-block
6. Finally, inside the outer if-block, empty newPickedCards
   1. I.e. set newPickedCards equal to an empty array

We’re so close!! The only logic left to implement is “unflipping” two mismatched cards. Let’s write a function that’ll do that for us.

1. In App.js, write a new function called unflipCards
   1. Make it take two parameters: **card1Index** and **card2Index**
2. In the first lines of the function, create two variables called **card1** and **card2**
3. Set them equal to **{...this.state.deck[card1Index]}** and **{...this.state.deck[card2Index]}**
   1. Again, this weird syntax is used to clone objects. Otherwise, **card1** and **card2** would hold references to the cards in **this.state**, and changing them would make react complain about not usingsetState()
4. Set the **isFlipped** property of **card1** and **card2** to false.
5. INCOMING: Here comes the least hand-holdy instruction in this document!
6. Write a .map() loop over **this.state.deck** that is saved to a variable called **newDeck**
   1. Have it return each card, but inject **card1** and **card2** in the appropriate locations
   2. Use the .map() loop in **pickCard()** as a reference!!
7. Finally, make a call to **this.setState()**, passing in **{deck: newDeck}** as a parameter

That’s it for the **unflipCards()** method. Let’s call it back in **pickCard()**

1. Replace your earlier “unflip cards” comment with a call to **this.unflipCards(card1Index, card2Index)**

Does the game work now? Don’t be silly!

# Step 8- Delaying the call to unflipCards();

While our game logic is sound, we forgot about one thing - time! When a user selects two cards that don’t match, we want them to be able to see that they made a wrong choice, at least for a second. Unfortunately, our code just immediately unflips cards, which isn’t a nice user experience.

To fix this, we need to delay our call to **unflipCards();**. We can do that using the built in **setTimeout()** function.

1. In the **pickCards()** function, delete the line that says **unflipCards(card1Index, card2Index)**
2. Replace it with a call to **setTimeout();**
3. **setTimeout();** takes two parameters - a callback function, and a delay length
   1. First parameter: **this.unflipCards.bind(this, card1Index, card2Index)**
   2. Second parameter: **1000**

Whew! Now we can breathe. setTimeout is going to wait 1000 milliseconds before calling this.unflipCards(). PLEASE ask me to come over and talk through it with you if you’re unsure of what you just wrote - this won’t be the last setTimeout() you write!

Now, bask in the glory of your application! You’ve officially written your first functioning React app!