Rev. 08Aug2019

Exercise 47

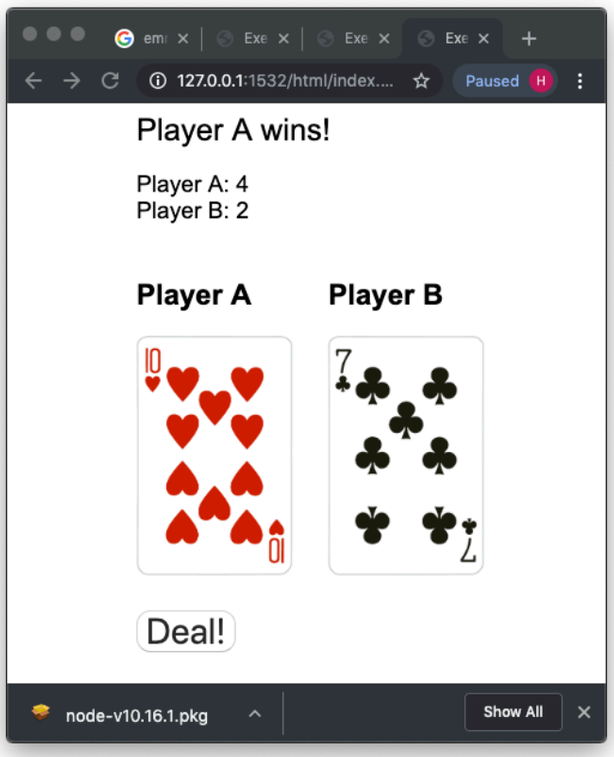
BUILDING A GAME OF WAR

In this exercise, we'll look at building the card game, war. Download and unzip the accompany code into your **exercises** folder. As with the last exercise, you won't be writing any code, but rather following along with the code I wrote. Let me explain once more why I'm asking you to do this.

If you decide to take up a career of programming (and if you got this far in the free lessons, you should *strongly* consider doing so), you're going to most likely start off working on an existing code base rather than starting from scratch. To be successful at this, you know to be able to read others' code and while *you* will be disciplined to add meaningful comments to your code, others may not. I also want to give you one of the greatest tools for peering into code, called `dump.js`.

So, for this exercise, I'll show you my code, discuss what it does — and then ask you to provide comments. The point is to get you experience reading code and making your own comments to help you understand it.

The project is the card game of war:



What do we call a list of steps needed to perform a task?

In the last exercise, I went into some depth about the need to think through the steps needed to perform a task. I know from teaching others that this is something students shy away from. It's *hard* thinking through all the steps, then considering how your code should react in a variety of situations. It's just easier to start writing code and hoping you'll figure it out as you go.

As tempting as it is to take the easy route, it's almost always a mistake. And, depending on the size of the project you're working on, it can be a fatal mistake. Programming leads and hiring managers are used to seeing code that isn't well thought-out. It's proof of how desperately programmers are needed that they're willing to often overlook this gap in the candidate's skill set. But you won't be in that group! And to ensure that, for the following exercises, we're going to first put together the plan for the project instead of immediately jumping into code.

How do we approach the problem? By listing all the things that your app needs to do. This is a first pass. We'll definitely refine this by combining certain items, removing some that it turns out we won't need, *etc*. But let's get started.

Before you go onto the next page, I want you to think about all the things a software version of war needs to do. What needs to be stored? What actions need to happen? Write those down for yourself. Then, when you think you're done, go onto the next page and you can compare your list with mine.

INITIAL LIST OF THINGS THE SOFTWARE NEEDS TO DO

A. create a deck of cards

B. shuffle the deck

C. deal a card to each player

D. store which player got which card

E. compare the two cards to see who wins

F. keep track of the score

G. let the user play again

H. show an image of each card to the player

I. Display the winner of the game

J. Display the ongoing score

Does your list look like mine? Did you add some things that I forgot? Do you forget some things that I added?

One of the most important things to learn from this exercise is that the steps are *not* written in technical language. They're written in plain English (or whatever the language you're most familiar with). That's very important. The steps detail *what* should happen, not *how* it should happen. The "how" is an implementation detail that you can think about when you start to write code.

And speaking of writing code, it's time to look at my code...

MY SOLUTION

The HTML provided is quite simple and no effort was taken to give the project any styling flare: all of our focus will be on the logic needed to make the game work. We'll start from the top and work downward. As you see fit, add comments to my original code to make sense of them. At the end of this exercise, you should be able to explain to another coder what's going on.

This is "A" from my list of steps

```
// create a deck of cards
let suits = ['C','D','H','S']
let pips = [2,3,4,5,6,7,8,9,10,11,12,13,14]
```

I've created arrays to hold all possibilities for a card. You know that playing cards have suits. Did you know that the *values* of each card are called *pips*? Instead of giving the face cards (Jack, Queen, King, and Ace) their own pips, I've translated them into values. This way, when comparing two cards against each other, I won't have to translate on the fly.

This is part of the "A", comment, but if you wanted to place a "sub-comment" about starting with an unshuffled deck, I think that would be fine.

```
let unshuffledDeck = () => {
  let arr = []
  suits.forEach( suit => {
    pips.forEach( pip => {
      arr.push(pip + suit)
    })
  })
  return arr
}
```

I start by creating an *un*shuffled deck. To do so, I use a *nested loop*. I first loop over the `suits`. Then, inside each iteration of that array, I loop over the `pips`. I concatenate the current pip and the current suit and add them to an array that I'll return when the nested looping is done.

This is "B" from my list of steps

```
let shuffle = array => {
  array.sort(() => Math.random() - 0.5);
  return array
}
```

This function uses JavaScript's array's `sort` function. Confession time: I googled "Javascript card shuffle algorithm" and found this quite impressive bit of code. I say "impressive" because it does so much with such a small amount of code. In case you're wondering how it does its magic, the `sort` method for an array takes an optional function that compares two elements in the array and decides which element has precedence over another. The clever author of the code used this to his benefit by writing a function that

```
let shuffledDeck = shuffle(unshuffledDeck())
```

Now, I can create a shuffled deck by passing the *un*shuffled deck into the `shuffle` function.

This is "C" from my list of steps. Whenever you have some block of logic (like dealing a card), consider turning that into a function.

```
let dealCard = deck => {
  let dealtCard = shuffledDeck.shift()
  return dealtCard
}
```

This function accepts a deck and deals a returns a card by calling `shift` on the function. The `shift` function removes the first item in an array. (I knew there was a function that did this, but had to look up the name of the function...)

This is "F" from my list of steps.

```
let wins = {
  playerA: 0,
  playerB: 0
}
```

I want a way of keeping score between two players.

This is "D" from my list of steps.

```
let currentDeal = {
  playerA: null,
  playerB: null
}
```

I want a way of keeping track of which player is dealt which card.

Next, I need a way of dealing to both players. This one is long, so I've put it on the next page.

This is "C" from my list of steps.

```
let deal = () => {
  let playerAcards = document.getElementById('playerA')
  playerAcards.innerHTML = '<h2>Player A</h2>'
  currentDeal.playerA = dealCard(shuffledDeck)
  let img = document.createElement('img')
  let imgSrc = document.createAttribute('src')
  imgSrc = `../images/cards/${currentDeal.playerA}.png`
  img.setAttribute('src', imgSrc)
  playerAcards.appendChild(img)

  let playerBcards = document.getElementById('playerB')
  playerBcards.innerHTML = '<h2>Player B</h2>'
  currentDeal.playerB = dealCard(shuffledDeck)
  img = document.createElement('img')
  imgSrc = document.createAttribute('src')
  imgSrc = `../images/cards/${currentDeal.playerB}.png`
  img.setAttribute('src', imgSrc)
  playerBcards.appendChild(img)
}
```

This is "H" from my list of steps.

This deals cards to each player and displays them on the screen.

This is "E" from my list of steps.

```
let evaluateWinner = currentDeal => {
  let regex = /[0-9]*/
  let playerA = parseInt(regex.exec(currentDeal.playerA)[0])
  let playerB = parseInt(regex.exec(currentDeal.playerB)[0])
  let winner = "TIE"
  if (playerA > playerB) {
    winner = 'Player A'
    wins.playerA = wins.playerA + 1
  }
  if (playerB > playerA) {
    winner = 'Player B'
    wins.playerB = wins.playerB + 1
  }

  document.getElementById('winner').innerHTML = winner + ' wins!'
}
```

This is "I" from my list of steps.

I happened to choose to use a *regular expression* ("regex" for short) but that's an implementation detail. There are different ways of accomplishing the same end.

This is "J" from my list of steps.

```
let showScore = wins => {
  let message = `Player A: ${wins.playerA} <br />Player B: ${wins.playerB}`
  document.getElementById('score').innerHTML = message
}
```

Letting the player know the score.

```
let newDeal = () => {
  if (shuffledDeck.length > 1) {
    deal()
    evaluateWinner(currentDeal)
    showScore(wins)
  }
}
```

This is one I forgot to list on my intitial list of steps: we need to stop when there are no more cards in the deck.

This is "G" from my list of steps.

```
document.getElementById('deal').addEventListener('click', newDeal)
```

And with that, we're done with the code.

At the risk of overtasking your patience, let me urge you again *not* to be discouraged by the thought that you couldn't write this code. I'm guessing the first steps you took as a child were not models of grace and precision? That you weren't setting any speed records? That climbing stairs was still a few months down the line?

The secret to mastery is to enjoy what you're learning, not allowing yourself to be discouraged by what you still do not know.

Oh, I almost forgot: I promised to give you an amazing tool to help you in your learning of programming. The tool is a file called `dump.js`. I've already linked it to the HTML page, but let me show you how it's used.

- Open `index.js`. Let's say you're having a hard time imagining what `shuffledDeck` *looks like*. Since creating mental models is very important in programming, you want to have a sense of various data structures. The `dump` function is *ideal* for this.

There's nothing special about the div or the id. We just need some place to direct the dump function to place the results.

In the HTML, I created a `<div>` element with an `id` of `dump`. Let's use that element to have the dump function display a way of looking at shuffledDeck.

At the end of index.js, place this code:

```
document.getElementById('dump').innerHTML = dump(shuffledDeck);
```

You should see a representation of the array that is `shuffledDeck`.

Let's make it more complex. Type this code at the bottom of `index.js` and then `dump` it.

The dump function is particularly useful when you're dealing with a complex data structure.

```
const henryVIII = {
  yearBorn: 1491,
  yearDied: 1547,
  yearsInPower: '1509 - 1547',
  marriages: [
    {
      spouse: 'Catherine of Aragorn',
      children: ['Henry', 'Mary']
    },
    {
      spouse: 'Anne Boleyn',
      children: ['Elizabeth']
    },
    {
      spouse: 'Jane Seymour',
      children: ['Edward']
    },
    {
      spouse: 'Anne of Cleves',
      children: []
    },
    {
      spouse: 'Catherine Howard',
      children: []
    },
    {
      spouse: 'Catherine Parr',
      children: []
    }
  ]
}
```