## Javascript Object Oriented Programming:

# Blackjack Application


### Setting Up the Blackjack Application Work Environment

Now that we have a grasp of **Object Oriented Programming** in Javascript, let's begin work on our **Blackjack Application**.
Let's first set up our work environment, where we have a few folders and files to make.

1. Put the provided **blackjack** folder in your **AdvancedJavascirpt** folder

### Getting Set Up in the Terminal

2. Via the terminal, **cd** your way to **Documents > AdvancedJavascirpt > blackjack**
3. Once inside **blackjack**, type **ls** to make sure you have an **images** directory.
4. Type **cd images** and then **ls** to make sure you have **blackjack-table.jpg** as well as a **cards** directory
5. Type **cd cards** and then **ls** to make sure you have 52 image files, one per card. File names follow the convention *Value-of-Suit.png,* so *Queen-of-Hearts.png,* etc.
6. Go back to the **blackjack** folder. Type **cd ..** to go to **images** and then **cd ..** again
7. Make a new folder: **mkdir js**
8. Type **cd js**, and make another folder: **mkdir Classes**.
9. Type **cd Classes** to go into the new directory, and make a file: **touch Blackjack.js** This file will contain our **Blackjack Class** with the entire game functionality.
10. Back out to the **blackjack** level with **cd ..** twice, and make a new folder: **mkdir css**
11. Type **cd css** to go into the new directory, and make a file: **touch blackjack.css**

### Separation of Application Logic and Functionality

The **blackjack.css** file will contain all the styles for our **Blackjack Class**. Now, we *could* put all the CSS inside the **Blackjack.js** file, by targeting the **styles.cssText** property of our various dynamic DOM elements. This would offer the advantage of making our Blackjack application totally self-contained—no CSS file required. But in the interest of not jumbling all the code together, which can make the code harder to read and edit, we will go with a dedicated dot-css file, imported into blackjack.html.

### The HTML

Speaking of HTML, our app's markup will comprise one bare-bones file, with a single, empty div in the body, serving as a container for the interface. The only other code in the body will be to import the Blackjack Class and instantiate it. Despite the minimalist HTML, the browser view of the page will be a fully-functional blackjack application!

12. In your editor, make a new file, and save it to your **blackjack** folder.
13. Name the file **blackjack.html**.
14. In **blackjack.html**, add the basic tags: **`<!DOCTYPE html>`**, etc.
15. Give the page a title: **`<title>Blackjack</title>`**
16. In anticipation of our CSS, import that:

```
<link href="css/blackjack.css" rel="stylesheet">
```

17. In the **body**, add this one element, into which the entire Blackjack application will go:

```
<div id="app"></app>
```

18. In anticipation of our Blackjack Class, import it under the div:

```
<script src="js/Blackjack.js">
```

19. Also, in anticipation of using the **Blackjack Class**, instantiate it:

```
<script>
    const blackjack = new Blackjack('app')
</script>
```

### Blackjack Class Constructor Method

An Object of a Class is instantiated by calling the Constructor Method, preceded by the keyword **new,** such as:. **var dateTime = new Date()**. The object has access to all the **methods** and **properties** of the **Class**. Our Blackjack Class Constructor Method takes "**app**" as its argument. This passes the div to the application, for outputting the game to the app div. This is what we have, so far:

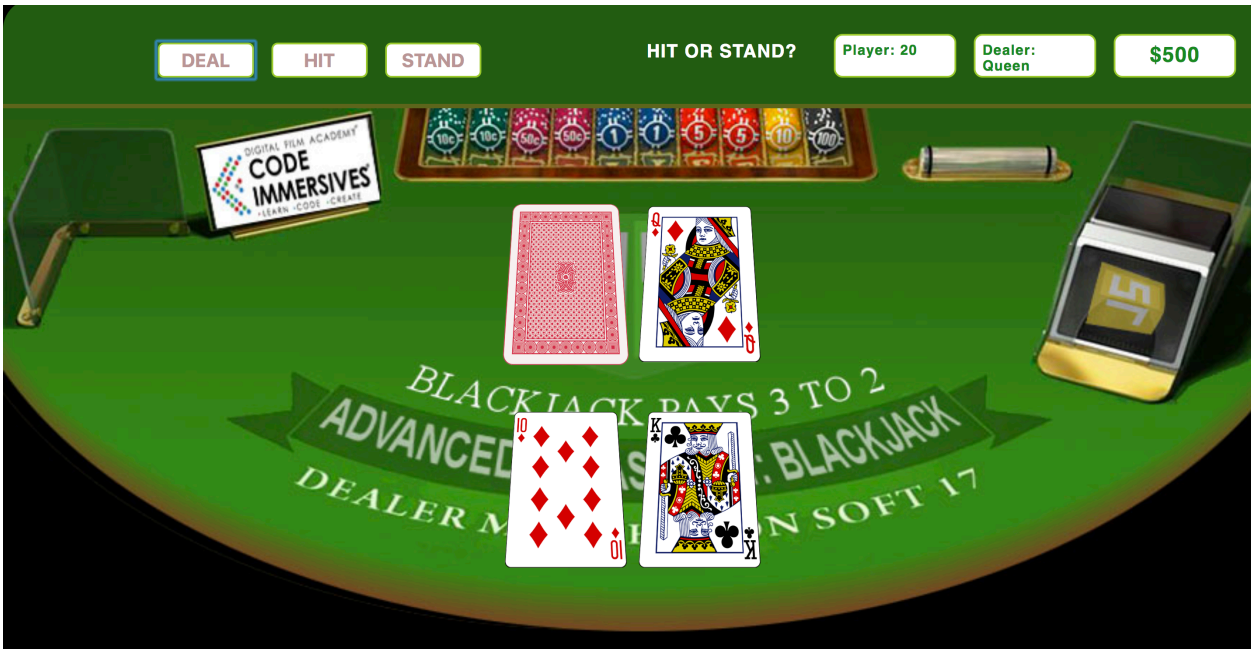
```
1  <!DOCTYPE html>
2  <html lang="en-us">
3  <head>
4      <title>Blackjack</title>
5      <link href="css/blackjack.css" rel="stylesheet">
6  </head>
7  <body>
8      <div id="app"></div>
9      <script src="js/Blackjack.js"></script>
10     <script>
11
12         const blackjack = new Blackjack("app");
13
14     </script>
15 </body>
16 </html>
```

### Blackjack Gameplay Basics

If you are unfamiliar with the rules of blackjack, by all means Google the subject or watch a YouTube video. That said, here are the basic rules and how the game unfolds. (Finer points of blackjack, such as splitting hands, doubling down and taking insurance, will not make it into our game.)

### Deal the Cards

The player and dealer are dealt two cards each, with the player getting the first card. In casinos, cards are dealt from a so-called shoe containing six or even eight decks. (Multiple decks make it harder for players to count cards, that is, to keep track of what cards have been played.) The dealer's second card (the *hole card*) is dealt face-down. All other cards are dealt face-up. Face cards are worth 10. Below, the player has a score of 20. The dealer's score is incomplete; all we know is the dealer has a Queen.




*This is what you are making: Javascript Blackjack Application in action*

### Hit or Stand?

The player is prompted to **HIT** (take a card) **OR STAND** (let the dealer play out their hand). The player decides if taking another card will likely improve his or her hand, without busting (going over 21). Much strategy goes into such decisions, but it is important to understand that these strategies are not reckoned with in our algorithm. It

makes no matter to us whether the player is wise or foolish, cautious or reckless. Our job is to make the cards appear and get scored correctly when the buttons are clicked.

**Playing out one's hand**

Whoever has the higher score without going over 21 (**busting**) wins. The player plays out their hand first, but if they bust, they lose without the dealer having to play out his hand. The player can hit on any hand worth up to 20, although it is ill-advised to hit with a high score, due to the odds of busting. In contrast to the player, who must wrestle with odds and intuition, the dealer makes no decisions. The dealer must hit up to and including 16, and even on "soft 17"—a 17 that includes an Ace that counts as 11.

**Aces are worth 11 or 1**

The Ace is worth either 11 or 1. If a hit puts the score over 21, the player or dealer busts —unless, that is, they have an whose value can be decreased from 11 to 1. Handling the Ace under various scoring scenarios is one of the most challenging parts of the algorithm. Ace scoring is a refinement that we will implement at the end of the project.

**Wagering**

The player receives an initial stake of $500. For our application, the bet will always be $10. Blackjack pays 3-2, so if the player is dealt an Ace and a 10 (or a face card, also worth 10), then the player wins $15.  In the event of a **push** (a tie), the player will neither win or lose their $10. Whenever the player loses a hand, $10 will be deducted from their total. The player is not able to change their bet, although this would certainly be a nice feature to implement for "Extra Credit".

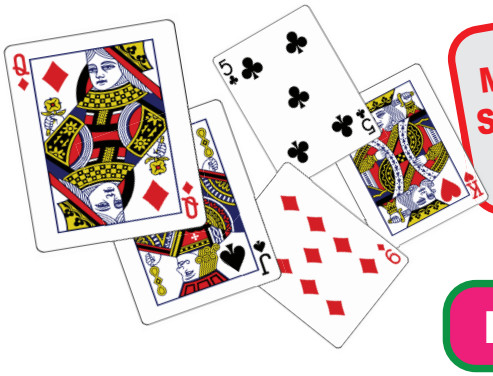
**Shuffling and Dealing**

Casino blackjack is never played from a single deck, but rather from a **shoe** consisting of 6-8 decks. In keeping with that procedure, our initial deck of 52 cards will be cloned into a six-deck cards, which will be shuffled (randomized), and then dealt from a six-deck "shoe". Programmatically, the cards will be objects in an array.

**The Javascript**

Now for the Blackjack Class. We will begin by outputting the DOM: this consists of a background image in the div, plus a button bar above that, which will ultimately house three game play buttons, as well as four child divs for displaying output. The larger app

# Javascript Blackjack Application Algorithm Flow Chart



MAKE & SHUFFLE DECK

First, we need cards! We need all 52. Each card should be an Object so that it can store properties for the file name and card value. Once we have our deck, we will multiply that by 6 to get the multi-deck set for blackjack game play, known as the "shoe". Next we need to shuffle the cards.

DEAL

Cards are dealt to start the game. This requires using the card objects to make card images. For this we use the fileName property of the card object. The Player and Dealer get 2 cards each. Both Player cards are dealt face up. The first Dealer card ("hole card") is dealt face down.

Show Score

After dealing the cards we add up the Player's total and output that. The Dealer's Hole Card is face-down, so the Player only sees the face-up card, the value of which is outputted.

If after the player HITS and they still have less than 21, they face the same choice again: HIT or STAND?

PLAYER'S TURN

HIT

STAND

Unless the Player was dealt Blackjack (21) on their opening hand of 2 cards, it is the Player's turn to go. Player decides to HIT (take another card) or STAND (NOT take another card, in which case it is the Dealer's turn

BUST

Update Score

If a HIT results in the Player's score exceeding 21, the Player loses -- BUSTED!

STAND: Player elects to NOT take another card. By choosing STAND, the Player turns things over to the Dealer, who, makes no decisions, but rather follows the rules of Blackjack. Dealer must hit on 16 and on Soft 17 (score of 17 with an Ace counting as 1)

DEALER'S TURN

Dealer automatically gets another card if their score is less than 17 or if it is Soft 17

if dealer score < 17 or if dealer score is Soft 17, dealer gets another card

if dealer score > 16 and if dealer score i NOT Soft 17, dealer does NOT get another card

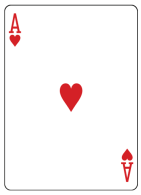
If the additional card results in the Dealer's score exceeding 21, the Dealer loses -- BUSTED!

BUST

Once both Player and Dealer are done taking cards -- and with neither of them having BUSTED -- it is time to compare thieir point totals and announce the WINNER.

WINNER

Aces count for either 11 points or 1 point. If 11 points makes you go Bust, the Ace counts for 1 point. Else, it's 11 points. Scoring Aces is one of the trickier parts of the algorithm



DEAL

div will contain two invisible divs. Appearing in the middle of the interface, these boxes will hold the player's and dealer's cards. This is what the blackjack.html page will look like in the browser after an initial deal of two cards each:

20. Make a new file called **Blackjack.js** and save it to your **js/Classes** directory. Inside the file, declare the **Class** and the required **constructor()** method:

```
class Blackjack {
    constructor() {
    } // close constructor()
 } // close class
```

The **Constructor Method** runs automatically, once, upon instantiation:
**const blackjack = new Blackjack("app")**

**Class Properties and Methods: What is *this* ??**

The constructor method needs to contain the **properties** used in the application. Variables used by more than one method (function) are declared in the constructor. All properties are scoped of the Class, and are preceded by **this**, as **this.playerScore** The **methods** of the Class are written after the constructor and are NOT preceded by this, as **deal() { } .** Methods are function that belong to a Class, so when the method is called, it IS preceded by **this**, as: **this.deal()**

**Passing an Argument to the Constructor Parameter**

Our Blackjack Class has a **parameter**. We set the value of the parameter by passing in an argument during instantiation: **new Blackjack("app")**. The parameter set by the argument needs to be in the constructor. It doesn't matter what we call our argument. We could call it *app*—or *mango*, for that matter. Let's call it **app**:

21. Add this to the constructor. The parameter **app** is a variable (no quotes), whereas the argument **"app"** is a string (quotes):

```
class Blackjack {
   constructor(app) {
   } // close constructor()
 } // close class
```

22. Pass the div into the application as a JS object. Since **app** is a property of the Blackjack Class, it is preceded by the keyword **this**:

```
constructor(app) {
    this.app = document.getElementById(app)
} // close constructor()
```

23. Let's run a test to make sure our HTML and JS files are connected. Add this:

```
this.app = document.getElementById(app)
this.app.innerHTML = 'Hello from the Blackjack Class!'
```

24. Save the JS and HTML files. Publish **blackjack.html** in the browser. You should get: the *Hello* message outputted to an otherwise blank page.

**The Algorithm**

Our application has a lot of steps, a lot of logical progressions, a lot of if-else branching—a lot of code! Before we actually start writing any code, therefore we really should map out our problem-solving as a set of steps. In other words, we need an algorithm. Refer to the flow chart on a previous page, where you will find the major stepping-stones of our application. Without getting into nitty-gritty of how to actually implement the code, have a good look at the algorithm flow chart.


**Algorithm Step 2:**
**Make and Shuffle Deck of Cards**

The 52 card file names follow the same naming convention: Card-of-Suit.png, so: Queen-of-Diamonds.png, etc. To make cards we need images with these file names as the source. That means that we need to get all the file names organized somewhere. Sounds like a job for an array, which sounds like a job for a loop. In fact, we will use a nested for loop to iterate two arrays. One array will contain only the card **values [“Ace”, 2, 3, 4, 5…..”Queen”, “King”]**, while the other array will contain the card **suits [“Clubs”, “Diamonds”, “Hearts”, “Spades”]**.

**A nested for-loop for making 52 card objects**

We will use a nested for-loop to go through the values array 13 times, with each of the 13 iterations also hitting the suits loop 4 times. This will give us 52 total loop iterations. Each time, we will make a **Card Object**, with **properties** of **fileName**, **cardValue** and **numValue** (numeric value). We need the numeric value for each card

to keep score. When we are adding up the points, we need to know that a King is worth 10, etc. The file names will be concatenated. Each of these card objects will be pushed into an object array called **deck**. After we have our deck, we will copy it to get a six-deck *shoe*. We will then shuffle our 312-cards, using the **sort()** method. With our cards randomized, we will be ready for **Algorithm Step 2: Deal the Cards**

But lest we get too far ahead of ourselves, let's take a step back and actually do all the steps necessary to make and shuffle a deck of cards!

1. In your **constructor()**, delete the *Hello* test line, and declare two arrays, one for card values and the other for suits. Set these to **let**, not **this**, since we want them to vanish after using them in our for-loops. These two arrays exist only to concatenate the file names of our images and to make our other card properties.

2. After making the **values** and **suits** arrays, declare an empty **deck** array. This time, we use **this,** because **deck** is used throughout our application, and therefore needs to be a property of the Blackjack Class:

```
class Blackjack {
   constructor(app) {
      this.app = document.getElementById(app)
      let values = ["Ace", 2 , 3, 4, 5, 6, 7, 8, 9, 10,
      "Jack", "Queen", "King"]
      let suits = ["Clubs", "Diamonds", "Hearts", "Spades"]
      this.deck = []
   } // close constructor()
} // close class
```

3. Next make a nested for-loop. Rather than going straight to making 52 objects, let's warm up by just concatenating 52 file names and pushing those strings into the deck array. To make sure it works, output the first array item to the app div:

```
      this.deck = []
      for(let i = 0; i < suits.length; i++) {
         for(let j = 0; j < values.length; j++) {
           let fileName = `${valus[j]}-of-${suits[i]}.png`
```

```
            this.deck.push(card)
          } // for loop
        } // for loop
        this.app.innerHTML = this.deck[0]
```

4. The file names will let us set the source of our card images, which is all very well and good, but these cards have no values assigned to them, so cannot be used to keep score in a card game. For this, we need not just 52 file name strings, but 52 objects, each with multiple properties. Inside the loop change the item being produced from a string to an object, with two properties:

```
for(let i = 0; i < suits.length; i++) {
   for(let j = 0; j < values.length; j++) {
      let card = {
         cardValue: values[j],
         fileName: `${valus[j]}-of-${suits[i]}.png`
      }
      this.deck.push(card)
   } // for loop
} // for loop
```

5. Output a sample to the DOM to make sure the objects made it into the array:

```
this.app.innerHTML=`cardValue: ${this.deck[0].cardValue}
                    fileName: ${this.deck[0].fileName}`
```

6. So we need another property with the numeric value of each card. For keeping score, we cannot add "King" + 8, but we can add 10 + 8. Face cards are worth 10, while Aces have a default value of 11. Each time through the loop, we will look at the length of the array item to determine if it is an Ace or face card. Then we will assign to the string its correct numeric value. Add this to the loop:

```
for(let i = 0; i < suits.length; i++) {
  for(let j = 0; j < values.length; j++) {
    let numVal
    if(values[j].length == 3) { // if Ace
       numVal = 11 // Aces start w default value of 11
    } else if(values[j].length > 3) { // if Jack, Queen, King
       numVal = 10
    } else {
       numVal = values[j] // already a number from 2-10
```

```
        }
        let card = {
            numVal: numVal,
            cardValue: values[j],
            fileName: `${valus[j]}-of-${suits[i]}.png`
        }
        this.deck.push(card)
      } // for loop
    } // for loop
    this.app.innerHTML=`numValue: ${this.deck[0].numValue}
                        cardValue: ${this.deck[0].cardValue}
                        fileName: ${this.deck[0].fileName}`
    } // close constructor()
```

**The Spread Operator**

The **spread operator** provides some novel syntax for copying arrays: the triple-dot. Let's try it, since we need six decks of cards for our casino-grade blackjack shoe:

7. Right after the nested loop, add this **spread operator** expression, and output the **length** of the **deck** array as a test:

```
    } // for loop
    this.deck = [ ...this.deck, ...this.deck, ...this.deck,
    ...this.deck, ...this.deck, ...this.deck ]
    this.app.innerHTML = this.deck.length // 312
```

**The Sort Method for Shuffling the Deck at Random**

The **Array.sort(a, b)** method runs a function on each array item. The first item of the array, **a**, swaps places with the last item, **b**. Then the the next two outer items switch places, and so on. But we don't want to sort—we want to randomize. By making **b** a random value, rather than a sequential one, we can randomize—shuffle!—rather than sort. Call **sort** on the **deck** array. The result will be a randomized deck:

8. Call **sort** on the **deck** array, and then see if the first item has been changed:

```
  this.deck = [ ...this.deck, ...this.deck, ...this.deck,
```

```
...this.deck, ...this.deck, ...this.deck ]
this.deck.sort((a, b) => {
   return 0.5 - Math.random() //
});
this.app.innerHTML = this.deck[0].fileName // random file
```

**The Blackjack App DOM**

The blackjack application requires some eleven DOM elements. With the exception of the app div, all will be made dynamically by the Blackjack Class. These elements are listed below. Each will be styled by our blackjack.css file:

- 1 **app div** — the one element we already have. Filling most of the screen, it is there to hold the blackjack table background image of, as well as the cards.
- 1 **header**, above the "app" div. It will hold the buttons and output.
- 3 **button** elements inside the header. The will say DEAL, HIT and STAND.
  Their object names will be **btnDeal**, **btnHit** and **btnStand**
  These buttons will call our **deal() , hit()** and **stand()** functions.
- 4 divs, also inside the header. Located to the right of the buttons, these divs will display our output, including messages and the current score. Their object names will be **messageBox, playerScoreBox, dealerScoreBox** and **moneyBox**.
- 2 divs inside the app div. These are for holding the dealer and player cards in separate boxes. Their names will be **playerCardBox** and **dealerCardBox.**

1. Remove the output test (**this.app.innerHTML**).
2. Right under the sort method, dynamically make a header element. Instead of using **appendChild,** use the **insertBefore** method, so that we can position the element *before* the first child of the body, which is the app div. This makes the header the first element on the page:
   ```
   // the DOM elements
   this.header = document.createElement('div')
   document.body.insertBefore(this.header,
   document.body.firstChild)
   ```
3. Next, make the DEAL button. Have it call the deal method when clicked, and append the button to the **header.** The big news here is that we have a new twist on the addEventListener method: the function name must be followed by **.bind(this**) to keep the function scoped to the Class object.
   ```
   // the DEAL, HIT and STAND buttons
   ```

```
this.btnDeal = document.createElement('button')
this.btnDeal.innerHTML = "DEAL"
this.btnDeal.addEventListener('click', this.deal.bind(this))
this.header.appendChild(this.btnDeal)
```

4. Repeat this code for the HIT and STAND buttons:

```
this.btnHit = document.createElement('button')
this.btnHit.innerHTML = "HIT"
this.btnHit.addEventListener('click', this.hit.bind(this))
this.header.appendChild(this.btnHit)

this.btnStand = document.createElement('button')
this.btnStand.innerHTML = "STAND"
this.btnStand.addEventListener('click', this.stand.bind(this))
this.header.appendChild(this.btnStand)
```

5. Check the browser. the buttons should be there, but they have no styling. Nor is there any indication that the header of app div even exist. Time for some CSS.
6. Open **blackjack.css**.
7. Add a little CSS to the body. The **overflow:hidden** property will prevent the application interface from scrolling:

```
body {
  background-color: #000;
  text-align: center;
  font-family: sans-serif;
  font-size: 1.25rem;
  font-weight: bold;
  letter-spacing: 1px;
  margin: 0;
  padding: 0;
  overflow: hidden;
}
```

8. Next comes the app div. Add this, and then check the browser:

```
#app {
  width: 1400px;
  margin: 0 auto;
  background-image: url('../images/blackjack-table.jpg');
  background-size: contain;
  background-repeat: no-repeat;
  height:100vh;
```

```
}
```

9. Next, style the **header.** We will use **flex** to vertically center the child elements:

```
header {
  display: flex;
  justify-content: center;
  align-items: center;
  width: 1400px;
  height: 80px;
  margin: 0 auto;
  border-top-left-radius: 35px;
  border-top-right-radius: 35px;
  background-color: #265F12;
  border-bottom: 5px solid #5E651F;
}
```

10. Style the buttons:

```
button {
  width: 90px;
  cursor: pointer;
  margin: 10px;
  color: #B99;
  border-radius: 5px;
  padding: 5px; 15px;
}
```

11. Style the divs in the header that will display scores and other feedback:

```
header div {
  margin: 0 10px;
  width: 150px;
  height: 30px;
  background-color: white;
  float: right;
  color: #265F12;
  padding: 10px;
  border-radius: 5px;
  border: 3px solid yellowgreen;
}
```

12. Style the divs in the app div. These will hold the cards. Give them a temporary border, so that we can see them. Later, we will turn off the border:

```
#app div {
```

```
    width: 1000px;
    height: 160px;
    margin: 50px auto;
    text-align: center;
    padding: 5px 30px;
    border:3px solid red;
    margin: 65px auto 0px auto;
}
```

13. Style the images. The only images are the cards. There is one cool little twist—literally! A **transform:perspective** property that gives the cards that 3D look, like they are lying on the table.

```
img {
    width: 100px;
    height: auto;
    margin: 15px;
    display: inline-block;
    transform: perspective(80px) rotateX(10deg);
}
```

14. We got a little ahead of ourselves with the CSS. Not all of these DOM elements even exist yet. Time to play a little catch up in the JS. Switch to Blackjack.js and add this code inside the constructor. It makes four div boxes in the header:

```
// the score, feedback and money boxes
this.messageBox = document.createElement("div")
this.messageBox.className = "scoreBox"
// override the header div style in blackjack.css
this.messageBox.style.cssText = "width:200px; background-
color:transparent; color:white; border:0; font-size:1.3rem"
this.btnBar.appendChild(this.messageBox)

this.dealerScoreBox = document.createElement('div')
this.dealerScoreBox.innerHTML = “Dealer: "
this.header.appendChild(this.dealerScoreBox)

this.playerScoreBox = document.createElement('div')
this.playerScoreBox.innerHTML = “Dealer: "
this.header.appendChild(this.playerScoreBox)
```

```
this.moneyBox = document.createElement('div')
this.moneyBox.innerHTML = "$500"
this.header.appendChild(this.moneyBox)
```

15. Add the player and dealer card boxes into the app div:

```
// boxes for the cards of the dealer and player
this.dealerCardBox = document.createElement('div')
this.app.appendChild(this.dealerCardBox)
this.playerCardBox = document.createElement('div')
this.app.appendChild(this.playerCardBox)
```

16. Save and check out the browser. All the DOM elements should be there. The buttons don't work yet, but that brings us to our next part…

---


**Algorithm Step 2: Initiate Gameplay**

Next we will deal two cards to each player. This will require us to keep track of the cards and their numeric values. As the deal proceeds, the score of the player and dealer must be tallied. We also have to keep track of whether the player and/or dealer have been dealt any Aces, since the value of an Ace may be either 11 or 1, depending on the score. For this, we need more variables (properties).

**Small Group Algo Brainstorming:** ***How do we make the DEAL button work?***

Get with your group for some algorithm brainstorming. Here is your task in stages: Identify the exact task. Do not exceed the task boundaries. We are only concerned with the DEAL button. Leave the HIT and STAND buttons alone. Refer to the next page for additional guidance:

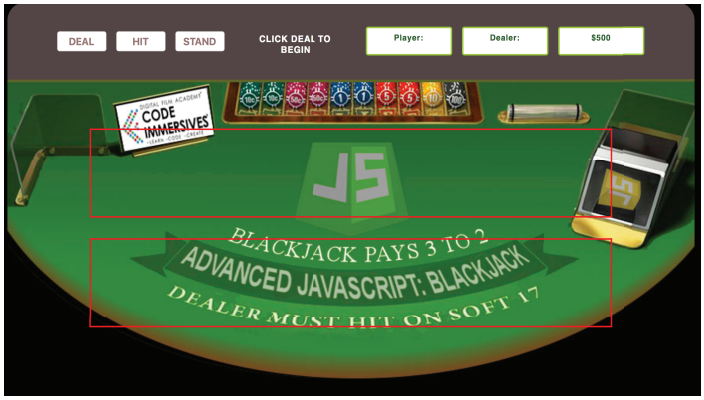
# Javascript Blackjack Application Algorithm Flow Chart

## 1. Prepare for Gameplay

MAKE DECK → MAKE 6-DECK SHOE → SHUFFLE SHOE

## 2. Initiate Gameplay: "CLICK DEAL TO BEGIN"

DEAL




**Algorithm Brainstorming:**
**AFTER we have cards ...**

Dealing involved giving the player and dealer two cards each in order: PLAYER, DEALER, PLAYER, DEALER
The dealer's second card is face-down. The back of the card image is in the folder with all the cards.

How to we do this?
Deal 2 to playerCardBox
Deal 2 to dealerCardBox
2nd dealer card is face-down
Player score is displayed in the divs in the header
Dealer score is NOT displayed--that's a secret for now!
Only the value of the dealer card is outputted.

**Declaring Properties**

What properties (variables) do we need to declare, so that we have cards to deal and can keep score? We have to keep an eye out for Aces, as their score can change from 11 to 1.
List the properties below and explain their purpose. You can always go back add add / edit / remove properties later.
Here is the first one:

```
this.playerScore = 0
```


1. Still in the **constructor method**, declare two new **arrays**, **player** and **dealer**, for storing the two card objects each are dealt. Also declare **number variables** for keeping score and **booleans** for tracking Aces worth 11 (herein called **Ace-11**):

```
        this.playerScore = 0 // for keeping player's score
        this.dealerScore = 0 // for keeping dealer's score
        this.player = [] // array for two cards dealt to player
        this.dealer = [] // array for two cards dealt to dealer
        this.playerAce11 = false // does player have an Ace-11 ?
        this.dealerAce11 = false // does dealer have an Ace-11 ?
        this.soft17 = false // soft 17 is a 17 with an Ace-11
        this.myMoney = 500 // player starts with $500
    } // close constructor()
```

**Algorithm Step 3: Deal the Cards**

For more realistic gameplay, it would be best if the four cards are dealt one at a time, on a slight delay. The best way to time the dealing of each card is with **setTimeout()**. This JS method takes two arguments, a function to call or run right there on the spot, and how long to wait in miliseconds before executing the function. Before we use setTimeout() to deal the cards, let's look at the method in isolation.

**The Javascript setTimeout() Method**

2. Open the **Chrome Console**, so that we can do a **setTimeout()** test.
3. Type this and hit enter. After 3 seconds, it should output *Hello from the setTimeout() method*:

```
setTimeout(sayHello, 3000)
function sayHello() {
    console.log('Hello from the function called
    by the setTimeout() method')
}
```

4. Instead of calling a function by name, you can run an **anonymous function**, right inside the setTimeout() method. Make this change:

```
setTimeout(function() {
     console.log('Hello from the anonymous function
     running inside the setTimeout() method')
 }, 3000)
```

5. Our anonymous function is ideally suited to the more concise syntax of the **fat-arrow function**. Make this change:

```
setTimeout(() => {
     console.log('Hello from the fat-arrow function
     running inside the setTimeout() method')
}, 3000)
```

**Algorithm Brainstorming: Figure out how to Deal the Cards**

Time to work on your own! Get with your coding partners and figure out how to make the four cards. Use for setTimeout() methods, one per card, with steadily increasing milisecond delays, such as: 500, 1000, etc. Here is some scaffolding for the **setTImeout()** method that gets the first card out onto the table. Substitute the correct code for each **CODE#** :

```
// deal player card 1
setTimeout(() CODE1 {
  // store last card object in shoe array at player[0]
  this.player[0] = this.shoe.pop()
  let cardPic = CODE2
  cardPic.src = CODE3
  CODE4.appendChild(CODE5)
  // increment player score
  this.playerScore += this.player[0].CODE6
  this.playerScoreBox.CODE7 = 'Player: '+ this.playerScore
  // if player has an Ace, flip boolean from false to true
  if(CODE8 == "Ace") {
     CODE9 = true
  }
```