JavaScript For The Hobbyist

By Kirk Burleson

There's a few things you need to do before we get started learning JavaScript. Namely, downloading the code to go along with this book, and getting your programming environment set up. To get the code, follow these steps:

1) Go here: https://github.com/kirkBurleson/yahtzee.
2) On the right side of the page, scroll down until you see a button to download the Yahtzee program as a zip file. Click the button.
3) Unzip the file (usually by double clicking it.)
4) Double click the "index.html" file inside the Yahtzee folder.
5) Play the game by double clicking the "index.html" file.

Since you'll want to type in the code for Chapter 1, rename the "main.js" file to "main_old.js" after your done playing with the game. You won't be able to run the game correctly from Internet Explorer because it disables the images and javascript. You must edit the "index.html" file by adding the "Mark of the Web." I tried to get this to work but couldn't. I can only recommend that you stop using IE and use FireFox, Chrome, or Safari. Really, anything but IE. I'm not at all against Microsoft, but you'll find as you develop for web browsers that IE plays by its own rules and you soon get tired of dealing with it. Google for it if you must get it to work.

Now, about the development environment. You'll need a code editor and a web browser to run the code. I use Sublime Text 2 for my editor and I test with FireFox, Chrome, and Safari. If you use a Mac then Safari is available, but don’t worry about it if you don’t have a Mac.

All you do is open your editor and create a new document named “main.js.” The zip file you downloaded will have all the other files needed. Remember the new file needs to be located with the other files. That’s why you should rename the original “main.js” file. Alright, open the editor, create a new file and let’s get started!

You’re going to learn JavaScript by typing in each line of a program I wrote, after which I’ll give commentary on the code you just typed in. This is not meant to teach every nuance of JavaScript, but rather to get you started and having fun by creating something meaningful right out of the gate. You’ll know more than enough JavaScript to create your own programs after we make this game. You should play the game in your browser before starting this.

```
var bit4 = bit4 || {};
```

First we declare a variable named “bit4” by using the “var” keyword. The equal sign means we want to set the value of the variable; remember, a variable exists to hold one value at a time and that value can be changed whenever we want. The next part says “if bit4 has already been declared then set our variable to be that value, otherwise set the value to a brand new empty object.” You might be wondering how bit4 could already be declared if we just started our program. It would come from another script loaded into the browser. In this case bit4 is a name I chose to be unique so I can use it for a namespace to hold all the code for the Yahtzee game. This will keep the Yahtzee variables and functions from being seen by other scripts, such as jQuery or some other library that we didn't create. Remember, namespaces keep code separated from each other. If you look at "index.html" you'll see that we load another script named "dice.js", and that starts with the same line of code.

Now that you know what a namespace is and how to declare a variable, what's that double bar looking thing in between bit4 and {}; that's the "default" operator. Read it like, "if bit4 is a true value then use it, otherwise use what follows the default operator." If you're a C# programmer then think null coalescence operator, but instead of defaulting on null it defaults on any false value. The false values are null, 0, '', undefined, NaN, and false. By the way, '' is an empty string (a sequence of 1 or more characters) and can be written as "". Also, NaN stands for "Not a Number." Let's move on to the empty curly braces. That's called an object literal and that's one of the ways to create objects in JavaScript. Think of objects as containers. One last thing I need to mention is that statements in JavaScript should end with a semicolon. You'll see code without it, but it's not good to do that because the JavaScript parser will insert semicolons where it thinks they should go (you won't actually see the change on screen.) There are documented situations where it will guess wrong and will cause your program to act incorrectly. Use semicolons!

```
bit4.yahtzee = bit4.yahtzee || {};
```

Ok, the star of the show here is the “dot” operator. That’s what we use to access properties on objects. Let me explain. A property is a variable that lives inside an object. So yahtzee is an object just like bit4 except it lives inside bit4 and cannot be seen outside of it. It’s another namespace. Think of bit4 as our company name and yahtzee as one of the many games we make. If we made a Pacman game it would look like “bit4.pacman = bit4.pacman || {};” and the code for each game would be separated into their own namespaces. That means we could load both games into the same Web page and they wouldn’t interfere with each other. Objects and properties are very important in JavaScript and we’ll see a lot more of them and how to work with them as we go. Remember that properties are variables but they are not declared using the “var” key word.

(function () {

Here we have a function that’s wrapped in parentheses. How come? Because we want to execute the function immediately so we can set all the properties of the yahtzee object. The function we have here is called an anonymous function because it has no name. By wrapping it in parentheses, we turn it into a function expression that can be executed immediately. After the function runs, all the code will exist inside the yahtzee object namespace and will be hidden from other scripts. If you look at the last line of code you’ll see the closing curly brace, left right parentheses, and a closing right parentheses. The “()” is what tells JavaScript to run the function immediately. Sow what’s a function? A function is a unit of code that does something. We put that code into a function so we can run it more than once from different parts of our program. So why are we placing all the yahtzee code into a self invoking function if it will only be ran one time? That’s a good question! The answer is simply that I preferred to do it that way. Another way to add properties to the bit4.yahtzee object would be to remove the code from the function. And another way is if we knew that there was not a yahtzee object already declared in a previously loaded script we could place all the code into the empty curly braces during the yahtzee object declaration; this option would require changing the equal signs into colons.

Here’s an example of multiple ways to create and add properties to an object:

```
test.game = {
 maxTurns: 5,
 gameFinished: false
};
```

```
test.game = {};
test.game.maxTurns = 5;
test.game.gameFinished = false;
```

Notice when we use colons and when we use the equals sign. Let’s move on.

```
var y = bit4.yahtzee;
```

Here we’re defining a variable named “y” and setting it to equal “bit4.yahtzee.” This will save us a lot of typing when we start assigning properties to the yahtzee object. It’s simply for convenience.

```
y.hi = 0;
```

We’re adding a property variable to the bit4.yahtzee object namespace and assigning it the value zero. This is the high score. All variables that are not assigned a value are given a value of undefined. Undefined is one of the variable types in JavaScript; other types are boolean (true or false), number, string (zero or more characters enclosed in single or double quotes), and null. Technically these are known as data types.

```
y.intervalID = null;
```

This property is used to hold the ID of the interval we’ll use later for the messaging system. We have to save this id so we can stop the interval when the game ends. We assign null as the value. This is good practice when you know a value will be assigned later. FireFox will flag a warning if you leave it unassigned. But some people leave it undefined because it’s less code and arguably may look better. You’ll define your own style as you program over the years.

```
y.gameOver = false;
```

Here we’re assigning a boolean value. The game is over if this equals true;

```
y.gameState = "roll";
```

Here we assign a string. The gameState property keeps track of what the player can do at any time in the game. “roll” means the player can roll the dice by clicking the roll button.

```
y.rounds = 0;
```

Which round of the game are we on? The game is over when this equals y.maxRounds. I define a round as applying a score. So if the player rolls 3 times he or she must apply the score. If the dice are 1 1 3 1 6 then a 3 can be applied to the “ones” spot or a 6 can be applied to the “sixes” spot and so on.

```
y.maxRounds = 13;
```

This is the number of rounds until the game ends. 13 is the number of spots that die rolls can be applied to. This is not a constant because it will be incremented if a bonus yahtzee is scored. A constant is a value that cannot be changed once it is assigned a value. Some JavaScript engines will allow this, but others will treat it the same as the “var” keyword. At the time of this writing FireFox 28 flags it as a warning, but still treats it as read-only. Chrome 33 treats it as read-only as well.

```
y.maxRolls = 3;
```

This is how many times the player can roll the dice before being made to apply a score.

```
y.rolls = y.maxRolls;
```

Set current rolls left. It counts down to zero.

```
y.grandTotal = 0;
```

This is the total of all scores. It displays in the upper right corner of the game.

y.diceToBeRolled = [0,1,2,3,4,5];

The square brackets create an array. An array is a variable that holds multiple values. How does it do that when a variable can only hold one value at a time? Remember objects? Remember how we added properties to the object? That’s how arrays work, but they’re special objects in that the keys are numbers and there are built in properties and methods we can use to make it easier to work with. In the array above, the first element is indexed at 0 and has a value of 0. It’s the same with the rest of them in that the values are the same as the index. We can set array elements to any type we want and each element can hold a different type of value.
y.diceToBeRolled[0] = “hello”; would set the first element to the string “hello”. We could also put a value in the 99th element, skipping everything between it and 5 and that would be fine with JavaScript. The values of the indexes in between will be set to undefined. So what are we using this for? This keeps track of which dice should be rolled when the player clicks the roll button. Any element with a 0 will be ignored. The first element is a spacer so we can start our index with a 1 instead of 0. We do this so we don’t have to subtract 1 to match a die to an array element. The first die goes with the first element and the second die goes with the second element.

y.dieValues = [0,0,0,0,0,0];

Another array! Arrays are used a lot so make sure you take time to play with them and learn how to use them. Once again we have our spacer element so we can match dice to elements without subtracting 1. This array holds the values of the rolled dice.

```
y.upper = {};
y.lower = {};
```

Here we're creating two more objects that correspond to the scoring spots on the board. The original Yahtzee game has upper and lower sections while our game places these sections on the left and right sides of the screen. The left side is the Upper section and the right side is the Lower section.

```
y.upper.scores = {
	ones:   {score: 0, scored: false},
	twos:   {score: 0, scored: false},
	threes: {score: 0, scored: false},
	fours:  {score: 0, scored: false},
	fives:  {score: 0, scored: false},
	sixes:  {score: 0, scored: false},
	bonus:  0,
	total:  0 };

y.lower.scores = {
	threeOfKind:  {score: 0, scored: false},
	fourOfKind:   {score: 0, scored: false},
	fullHouse:    {score: 0, scored: false},
	smStr:        {score: 0, scored: false},
	lgStr:        {score: 0, scored: false},
	yahtzee:      {score: 0, scored: false},
	bonusYahtzee: {score: 0, scored: false},
	chance:       {score: 0, scored: false},
	total:        0 };
```

Here we’re adding a property to the sections in the form of an object called “scores.” Notice how we’re defining the properties using the colon instead of the equal sign? You’ll see a lot of this so get used to it! All the properties except “bonus” and “total” have object literals for values. See them? {score: 0, scored: false}. These are used to keep track of what score the player has put on which target. The “score” property keeps the value scored and the “scored” property keeps track of if that target has been scored. We need the “scored” property because sometimes a 0 is scored.

```
y.startMsgSystem = function () {
```

Here we’re assigning a function to the variable startMsgSystem. This function defines the message system that tells the player what to do. Let’s look at the guts of this function.

```
y.msgPointer = document.getElementById("msgBox");
```

Here we grab the element that will show the current status of the game. “document.getElementById()” is very important because it’s how we find the html element on the page. The html page is kept in memory by the browser as a tree structure; this structure is called the DOM or Document Object Model. If you studied data structures then you probably remember trees, if not, think of a family tree. A parent has children and those children each have children and so on. Each html element is a parent and or a child. When we call getElementById(), the DOM starts at the top of the page where the html tag is and proceeds down the tree examining each tag (element) looking for an ID match. Remember to give your html elements an ID.

```
y.intervalID = setInterval(function () {
```

This is a very interesting piece of code. We call the “setInterval” function, but what does it do? It runs a function at a given interval; in this case we pass it an anonymous function and the number 1000 which means 1 second. So this interval will run our function every 1 second. The “setInterval” function will return us an ID that we’ll use later to stop the interval. Remember, intervals will continue to run until they are stopped in code. Let’s look at the function we’re passing into the interval.

```
var state = y.gameState;
```

First we capture the game state into a local variable.

```
if (state === "roll") {
```

We haven’t seen the “if” statement yet. The if statement allows us to run code based on the boolean value of the expression inside the “if” statement’s parentheses. Remember that boolean means “true” or “false”. Here we’re testing if the state is equal to the string “roll”; if it is then we’ll run the code inside the curly braces of the if statement.

```
y.msgPointer.innerHTML = "Roll the dice!";        }
```

You may not be familiar with “innerHTML” so I’ll explain. It’s used to set the inside value of block elements such as <span> or <div>. In this case msgPointer is a variable that holds the html element for the state messages. We simply set the inside of the span element to be what we want the player to see. So if the state of the game is “roll”, set the message to “Roll the dice!”;

```
else if (state === "apply") {
```

The “else” statement defines the code to run if the last “if” statement’s test was false. The “else” has to follow an “if” statement and cannot be used without one. In the code above another “if” statement will be tested inside the “else” code. These are called “if else ladders” or “if chaining.” Note that the code after the “else” can be any code you want to run and does not have to be another “if” statement.

```
y.msgPointer.innerHTML = "Apply the score!"; }
```

So if the state of the game is “apply”, set the message to “Apply the score!”;

```
else if (state === "game_over") {
        y.msgPointer.innerHTML = "GAME OVER";
```

Nothing new here.

```
        clearInterval(y.intervalID); }
```

This is interesting. If the game is over we need to stop the message system. We do that by calling the “clearInterval” function with the ID we saved from calling “setInterval.” See how we passed the ID into the function by placing it inside the function’s parentheses?

```
}, 1000);
```

The number 1000 is the second argument to the “setInterval” function and represents the milliseconds to wait before running the function we passed into it. It equates to 1 second.

```
};
```

Every beginning curly brace must be matched by an ending curly brace. Note the curly braces define a block of code. Functions use them, “if” statements use them, as well as many other constructs. They define code that should be ran together as a unit, such as when an “if” statement tests true or an “else” statement when the “if” tests false.

See the strings of text in the function? “Roll the dice!”, “Apply the score!”, and “GAME OVER” are the messages the player can see at any time at the bottom of the screen. These messages correspond to the current state of the game. This function starts off by grabbing the html element on the game screen so we can change what it shows later. We then call a function named “setInterval” and pass it a function. What do we mean by “pass it a function?”

Let’s recap on functions. A function contains code that we want to run multiple times. If we didn’t put that code into a function, we would have to repeat the same code in different parts of the program. So it allows us to write the code once and use it many times. We can pass data into a function; this data is called the function arguments. y.startMsgSystem is a function that takes no arguments, but the setInterval function takes 2 arguments: a function and a number. The number 1000 is how many milliseconds to wait before it will run the code passed in as the function. 1000 equates to 1 second. The function we passed in is called an anonymous function because it has no name. We could have declared a function elsewhere and just passed in the name of it instead of passing in the whole definition of the function. You will see it done like this so you should get used to seeing it; although I think it’s more readable the other way. Still with me? Let’s keep going!

```
y.saveHighScore = function (score) {
      if (localStorage) {
            localStorage.setItem("bit4_yahtzee_hs",
score); };
};
```

It’s a good idea to give your functions a descriptive name like we did here. This function does just what its name says; it saves a high score. See how we pass the score into the function? We don’t want to have to compute the score because we want to try to do just one thing in a function. It’s ok to call out to another function to do some calculation, but try to make each function do just one thing. Now, let’s look at “localStorage.” Local Storage is provided by Web browsers that implement html5 and allows us to save data as “key-value” pairs. If you play the game, starting on the second game you’ll see a high score. It will even be there if you close the browser and reopen it.

But don’t count on it for very important data. Use a database or some other cacheing technology such as Redis. On we go.

```
y.fetchHighScore = function () {
      if (localStorage) {
            return
localStorage.getItem("bit4_yahtzee_hs"); }
      else {
            return null; }
};
```

This function gets the high score stored by the browser through LocalStorage. If there’s no high score then the function will return null, which means non-existent. Remember, null is without value. It is different than zero or blank string. This function uses the “else” statement without another “if” statement following it. You see it? Earlier we wrote an “else” that had an “if” right after it. This is called an “if-else”, but not an “if-else ladder.” We could have left out the “else” and just returned null, so try it out in your projects.

```
y.loadHighScore = function () {
      var score;
      score = y.fetchHighScore();
      if (score === null) {
            return; }
      else {
            y.hi = score;
            document.getElementById("hi").innerHTML =
y.hi; }
};
```

Loading the high score means getting the score out of Local Storage and assigning it to the html element so it can be seen. There’s nothing new here, but you’ll notice we call our “fetchHighScore” function that we went over earlier. Did you notice that we combined “getElementById” and “innerHTML” on the last line? This is called chaining function calls. JavaScript will use the return value of the first function to call the second function on. These chains can get quite long so try to get used to seeing them.

```
y.resetRollsCounter = function () {
	y.rolls = y.maxRolls;
	document.getElementById("rollsLeft").innerHTML = 
y.rolls;
};
```

The player can roll the dice 3 times before having to place a score. Instead of 3 we use “maxRolls” so we don’t have to hunt down all the 3s if we want to make it 5 or some other value. I actually had to change this while testing for yahtzees. They happen so rarely that I increased the maxRolls to 20 so I could easily get one. Using numbers like the 3 in your code is called “hard coding” or using a “magic number.” It’s hard coded because the number can only be itself and cannot change like a variable can. It’s “set in stone.” It’s also a magic number because we don’t know what it represents. The variable name tells us what it’s used for, but the number 3 tells us little about what it represents. In the case of passing the 1000 to the “setInterval” function earlier, it would be easier to read if we assigned 1000 to a descriptive variable and passed that in, but I think that function is well known enough that it won’t cause much confusion. You’re free to decide for yourself.

```
y.gameIsOver = function () {
	return (y.rounds) === y.maxRounds;
};
```

This function simply checks if rounds is equal to maxRounds. Remember rounds keeps track of how many times the player has applied dice scores to the board. A round starts with the first roll of the dice and ends when the player applies dice values (score) to a board pattern, such as "full house" or "large straight."

```
y.resetDiceContainer = function () {
	y.diceToBeRolled = [0,1,2,3,4,5];
};
```

This function simply resets our data structure that keeps track of which of the dice will be rolled. The first element of the array is just there so we can index starting with 1 instead of 0. Any number other than 0 means that die will be rolled. We could have initialized the elements with all ones or any other number but 0.

```
y.resetDiceValueContainer = function () {
	var namespace = y;
	namespace.dieValues[0] = 0;
	namespace.dieValues[1] = 0;
	namespace.dieValues[2] = 0;
	namespace.dieValues[3] = 0;
	namespace.dieValues[4] = 0;
	namespace.dieValues[5] = 0;
};
```

Here we're reseting the data structure that holds the value of each die.

```
y.clearDiceImages = function () {
        document.getElementById("d1").src =
"images/diceBlank.gif";
        document.getElementById("d2").src =
"images/diceBlank.gif";
        document.getElementById("d3").src =
"images/diceBlank.gif";
        document.getElementById("d4").src =
"images/diceBlank.gif";
        document.getElementById("d5").src =
"images/diceBlank.gif";
};
```

This function sets the dice elements' "src" property to an empty image. Notice the file name.

```
y.unmarkDice = function () {
        document.getElementById("d1").className =
"unfrozen";
        document.getElementById("d2").className =
"unfrozen";
        document.getElementById("d3").className =
"unfrozen";
        document.getElementById("d4").className =
"unfrozen";
        document.getElementById("d5").className =
"unfrozen";
};
```

Here we're changing the class name of the die elements so they can be rolled. When the player wants to save a die from being rolled next time, he or she clicks the die image and javascript will change the class name of the element to be "frozen." The player knows it is frozen because a black border will appear around the die image. Of course "unfrozen" reverses that behavior.

```
y.clearForNewGame = function () {
        var upper = y.upper.scores,
        lower = y.lower.scores;
        upper.ones.score = 0;
        upper.ones.scored = false;
        upper.twos.score = 0;
        upper.twos.scored = false;
        upper.threes.score = 0;
        upper.threes.scored = false;
        upper.fours.score = 0;
        upper.fours.scored = false;
        upper.fives.score = 0;
        upper.fives.scored = false;
        upper.sixes.score = 0;
        upper.sixes.scored = false;
        upper.bonus = 0;
        upper.total = 0;
```

Here we reset the score to zero and we set the "scored" property to be false. The "scored" property will be set to true when the player puts a score on one of these targets. In our game these correspond to the left side of the board. Many people would insist that we re-factor the code and rename it "left" instead of "upper." I agree, but originally it was at the top of the board and that's why I named it "upper." That would be a nice exercise for the reader, but I'd wait until I finished the book!

```
        lower.threeOfKind.score = 0;
        lower.threeOfKind.scored = false;
        lower.fourOfKind.score = 0;
```

```
	lower.fourOfKind.scored = false;
	lower.fullHouse.score = 0;
	lower.fullHouse.scored = false;
	lower.smStr.score = 0;
	lower.smStr.scored = false;
	lower.lgStr.score = 0;
	lower.lgStr.scored = false;
	lower.yahtzee.score = 0;
	lower.yahtzee.scored = false;
	lower.bonusYahtzee.score = 0;
	lower.bonusYahtzee.scored = false;
	lower.chance.score = 0;
	lower.chance.scored = false;
	lower.total = 0;
```

Same thing going on here, except it’s named “lower.”

```
	y.resetRollsCounter();
	y.resetDiceContainer();
	y.clearDiceImages();
	y.unmarkDice();
	y.gameState = "roll";
	y.gameOver = false;
	y.rounds = 0;
	y.maxRounds = 13;
	y.grandTotal = 0;
	y.resetDiceValueContainer();
```

Here we’re just resetting our variable to keep state. State means the current state of the game. How many rounds into the game are we? How many times has the player rolled the dice? Those kinds of things.

```
	document.getElementById("gt").innerHTML = 
"000";
	document.getElementById("ones").innerHTML = 
"-";
        document.getElementById("twos").innerHTML =
"-";
        document.getElementById("threes").innerHTML =
"-";
        document.getElementById("fours").innerHTML =
"-";
        document.getElementById("fives").innerHTML =
"-";
        document.getElementById("sixes").innerHTML =
"-";
        document.getElementById("bonus").innerHTML =
"0";

document.getElementById("upperTotal").innerHTML =
"0";

        document.getElementById("3kind").innerHTML =
"-";
        document.getElementById("4kind").innerHTML =
"-";
        document.getElementById("fh").innerHTML = "-";
        document.getElementById("smst").innerHTML =
"-";
        document.getElementById("lgst").innerHTML = "-";
        document.getElementById("y").innerHTML = "-";
        document.getElementById("xy").innerHTML = "-";

y.addClassName(document.getElementById("xydiv"),
"hidden");
        document.getElementById("ch").innerHTML = "-";
        document.getElementById("lowerTotal").innerHTML
= "0";
```

We're resetting the board values to dashes so the player knows a score can be applied to them. The 3rd line from the bottom adds a "hidden" class name to the "extra yahtzee" target. It won’t be visible until the player has gotten a first Yahtzee.

```
y.addClassName(document.getElementById("playAgain"),
"hidden");
```

Here we hide the “play again” button by adding the “hidden” class name to the element.

```
	y.startMsgSystem();
```

When the game is over the message system is stopped. We restart it here for the new game.

```
};

y.updateHiScore = function () {
	var score =
Number(document.getElementById("gt").innerHTML);
```

We haven’t seen this yet. See the “Number” function? That’s a built in JavaScript constructor for creating numbers. What’s a constructor? A constructor is a function the creates (constructs) an object and returns it. It’s good practice to capitalize the first letter of a constructor so readers know it’s a constructor and not a normal function. We’re passing the constructor a number in string form (“33” instead of 33). Another way to convert a string number to a number is to use the “+” operator. So, writing “+(document.getElementById(“gt”).innerHTML)” would do the same thing. You can try it by opening FireFox, open the development tools (command + option + I on a mac), click the “>” icon and at the bottom of the box type this: +”33” and press return. It should print out

33 with no quotes. Any way, the idea is to update the high score on the board. Here we're just grabbing the text from the "grand total" or "gt" element.

```
          if (score > y.hi) {
                    y.hi = score;
                    document.getElementById("hi").innerHTML =
y.hi;
                    y.saveHighScore(y.hi); } };
```

This is where we're actually updating the high score and saving it. If the grand total is higher than the high score then the grand total becomes the new high score. And we call "saveHighScore" passing in the score to do the saving.

```
y.clearForNextRoll = function () {
          y.resetRollsCounter();
          y.resetDiceContainer();
          y.resetDiceValueContainer();
          y.clearDiceImages();
          y.unmarkDice();
          y.gameState = "roll";

          if (y.gameIsOver()) {
                    y.gameOver = true;
                    y.gameState = "game_over";
                    y.rolls = 0; // this will lock the game down
                    y.updateHiScore();


y.removeClassName(document.getElementById("playAgai
n"), "hidden"); }
};
```

We must call this function after the player applies a score. So we reset some data structures, clear the dice images, and check if the game is over. If it is over, we set some state variables to change the game message, update the high score, and unlock the hidden “play again” button so the player can play again. Nothing you haven’t seen already.

```
y.roll = function () {
        var i, die, roll;

        if (y.rolls === 0) {
                return;}

        // roll the dice
        for (i = 1; i < y.diceToBeRolled.length; i++) {
                if (y.diceToBeRolled[i] !== 0) {
                        die = document.getElementById("d" +
i);
                        roll = bit4.dice.roll1(6);
                        y.dieValues[i] = roll; // save die value
for quick lookup later
                        die.src = "images/dice" + roll +
".gif"; } }

        // adjust roll count
        y.rolls--;
        document.getElementById("rollsLeft").innerHTML =
y.rolls;

        if (y.rolls === 0) {
                y.gameState = "apply";
        }
};
```

This is the roll function that determines the value of each die and the image to show for it. Let’s take a look at what’s going on.

Here we see I've declared variables on one line separated by commas. I don't mind doing this if the names are short or if there are few variables, but some people get anal about consistency and change it to one variable per line. Do what you like. Next we check that rolls equals zero, if it does then we bail out because for some reason the player is not allowed to roll the dice at this time; it could be the game is over or the player should apply a score before rolling again. So the player is allowed to roll the dice so we start by iterating the "diceToBeRolled" array. Remember, this array holds zeros in the elements for dice that do not get rolled because the player clicked to save it. If the value doesn't equal zero then we grab the die element, generate a random number using our own library function to roll dice, save the rolled value into a data structure, and finally change the image on the dice. After we've iterated all the dice we decrement the roll count and update the value on the screen through the "getElementById" function. Lastly, we check if that was the last roll of the dice. If it is we change the game message to tell the player to apply a score. So what is decrementing? The 2 dashes after "y.rolls" means minus one from this variable. Two plus signs would mean add one to this variable. When you place these operators before a variable as in "--y.rolls" it means minus one from this variable before it's used in a computation or other manner. This is useful in a "while" loop so we don't have to decrement the control variable separately from the point of testing. Like this: var x = 10; while (--x) { do something here }. So, we set "x" equal to 10 and we want some code to run as long as “x” is true. Remember, 0 is considered false, so the “while” statement will minus one from “x” before it tests “x” for true or false. If we had put the double dashes after “x” then the “while” would test “x” before subtracting one from it. It’s an alternative to the “if” statement and they both test for true or false. On the end of the “for” loop you’ll see how we increment the “i” variable using double plusses. In that case it doesn’t matter if we did it before or after the variable. One more thing is the “not equals” sign (!==). The exclamation point means “not”, so we use it in place of the first equal sign. Let’s move on.

```
y.keepThisDie = function (img, num) {
        var id = +num;
        if (y.rolls === y.maxRolls) {
                return; }

        if (y.diceToBeRolled[num] === id) {
                y.diceToBeRolled[id] = 0;
                y.removeClassName(img, "unfrozen");
                y.addClassName(img, "frozen");
                return; }

        if (y.diceToBeRolled[id] === 0) {
                y.diceToBeRolled[id] = id;
                y.removeClassName(img, "frozen");
                y.addClassName(img, "unfrozen"); }
};
```

This function marks a die so it doesn’t get rolled on the next roll. How does it do that? We pass in the image element and the die number, convert the number into a “number” type using the “+” operator (remember that from earlier?) Then we check to see if the player is allowed to click the dice. If "y.rolls" equals "y.maxRolls" then the game is over and there's no point doing any of this. Otherwise, we check to see if we're "freezing" the die or "unfreezing" it. This function handles both and some would say that it's bad design. You could probably change the name to mention toggling, but I don't see a problem in handling both cases in the same function. Your opinion may be different.

```
y.containsClassName = function (e, name) {
        var i, classNames;

        classNames = e.className.split(/\s/);
        for (i = 0; i < classNames.length; i++) {
                if (classNames[i] === name) {
                        return true; } }

        return false;
};
```

Here we're checking that an html element has a particular class name assigned to it. The new thing here is the assignment to "classNames." We're calling a built in string function named "split" on the element's "className" property. But what are we passing into it? The "/\s/" is a regular expression. Regular expressions are used for pattern matching inside strings of text. In this case the "\s" means a space from the space bar. We're saying start at the beginning and look at every character in the string, and if you see a space put the characters up to that point into an array, then keep looking. The "split" function will fill an array with what it finds and return it. Once we have an array of all the class name words we can see if one of them equals “name.” If we find it we return true otherwise return false. Regular expressions are put inside slashes (/). Whole books have been written on them so take a look around the Internet to read up on them.

```
y.removeClassName = function (e, name) {
    var i, len, classNames;

    if (y.containsClassName(e, name) === false) {
        return; }

    classNames = e.className.split(/\s/);
    for (i = 0, len = classNames.length; i < len; i++) {
        if (classNames[i] === name) {
            classNames.splice(i, 1);
            e.className = classNames.join(" ");
            return; } }
};
```

Now we’re going to remove a class name. Notice we use the “containsClassName” function we just looked at to see if the class name is in it? If it’s not, we exit the function, otherwise we remove the class. So we iterate the array of names and when we find it we use the Array function “splice” to remove the element that contains the name. The “splice” method takes the index of the starting element to remove and a number representing how many elements to remove; in this case we just want one element removed. Another interesting Array function is “join.” This function will join every element in the array into one big string and each element will be separated by the string argument you pass into the function. We’re passing in a blank character so each class name will be separated by a space. That’s how css classes are written on html elements. Take a look at the “index.html” file of this project and you’ll see what I’m talking about. One more thing, notice how we return after we call the “join” function? That’s called an early return. We do that because there’s no reason to keep looking through the array if we found the one class name we were looking for. Remember to return early where possible.

```
y.addClassName = function (e, name) {
        if (y.containsClassName(e, name) === false) {
                if (e.className === "") {
                        e.className = name; }
                else {
                        e.className = e.className + " " +
name; } }
};
```

Here we just add a class name. See how we just tack it on to the “className” property of the element with a space in front of it? Try to keep things simple until you need complexity.

```
y.getDiceTotal = function () {
        var container = y.dieValues;
        return container[1] +
                container[2] +
                container[3] +
                container[4] +
                container[5]; };
```

Here we just return the total of all the dice values. We use the “container” variable so we only have to dereference the “dieValues” property of “y” one time. After that we just index into a local variable. Note that JavaScript arrays are just special cases of objects with properties. The indexes are turned into strings so they can act as property names. But using arrays instead of objects gives us access to built in Array functions like “join” so it’s worth it if you have numeric indexes. Don’t think too much into that one and just use arrays with numeric indexes and objects with text indexes.

```
y.updateGrandTotal = function () {
        var total;
        total = y.calculateUpperTotal() +
y.calculateLowerTotal();
        document.getElementById("gt").innerHTML =
total; };
```

Well, the grand total is the sum of the upper and lower scores. We calculate it and assign the total to the “innerHTML” of the grand total element.

```
y.calculateUpperTotal = function () {
        var total, bonus, scores;

        bonus = 0;
        scores = y.upper.scores;
        total = scores.ones.score +
                scores.twos.score +
                scores.threes.score +
                scores.fours.score +
                scores.fives.score +
                scores.sixes.score;

        if (total >= 63) {
                bonus = 35;
                scores.bonus = bonus;

        document.getElementById("bonus").innerHTML =
bonus;
        }

        return total + bonus;
};
```

This shows how we calculate the upper total. First we add up all the upper score targets and then we compare that total to the number 63. If it’s equal or greater than 63 we add a bonus of 35 to the score and then apply the bonus to the bonus element’s “innerHTML” property so it will be visible to the player. Finally we return the total + bonus.

```
y.calculateLowerTotal = function () {
        var total, scores;

        scores = y.lower.scores;
        total = scores.threeOfKind.score +
                scores.fourOfKind.score +
                scores.fullHouse.score +
                scores.smStr.score +
                scores.lgStr.score +
                scores.yahtzee.score +
                scores.bonusYahtzee.score +
                scores.chance.score;

        return total;
};
```

We calculate the lower section score here. Notice it doesn’t have a bonus so we just add up the targets and return the total.

```
y.updateGUIscore = function (name, score) {
        var element;
        element = document.getElementById(name)
        element.innerHTML = score;
};
```

This is a function used to show the score of a score target. We pass in the name and the score and assign it to the element's "innerHTML" property. That's pretty simple.

```
y.applyLowerSectionPoints = function (name) {
        var i, fullhouse, smStraight, score, count,
scoreElement,
                updateLowerTotal, hasYahtzee;

        if (y.rolls === y.maxRolls) {
                return; }
```

We've seen this before. We return if there are no more rolls left. Remember, "rolls" doesn't keep track of how many times the player has rolled the dice, but tracks how many times the player has applied a score to the board. It sounds like a candidate for refactoring to a more descriptive name. Maybe "turns" would be more appropriate. I'll leave that to you.

```
        updateLowerTotal = function () {
                var total = y.calculateLowerTotal();

        document.getElementById("lowerTotal").innerHTML
= total; };
```

This is an inner function (meaning it lives inside another function) and simply updates the lower total score after a player has applied a score to a lower section target. There's nothing new here.

```
        hasYahtzee = function () {
```

```
        return (count[1] === 5 || count[2] === 5 ||
            count[3] === 5 || count[4] === 5 ||
            count[5] === 5 || count[6] === 5); };
```

Here’s another inner function that just checks if a yahtzee has been rolled. Remember, an inner function has access to it’s outer function’s variables as shown here with the “count” variable. They all have to have the same value for it to be a yahtzee.

```
    score = 0;
    count = [0,0,0,0,0,0,0];
    // how many of each die values do we have?
    for (i = 1; i < y.dieValues.length; i++) {
        count[y.dieValues[i]]++; }
```

Here we are storing the number of times each value (1 - 6) has been rolled. This data structure will be used to figure out if the player has a small straight or full house or whatever.

```
    switch (name) {
    case "3kind":
        if (y.lower.scores.threeOfKind.scored ===
true) {
            return; }
```

First we check to see if the player has already placed a score on this target. If so we return. We’ll repeat this check for every target.

```
        for (i = 1; i < count.length; i++) {
            if (count[i] >= 3) {
                score = y.getDiceTotal(); }
            y.lower.scores.threeOfKind.score =
score;
            y.lower.scores.threeOfKind.scored =
true; }
    break;
```

So the only thing we need to do is check if the player has 3 of any die value. We do that by iterating over our "count" data structure. Remember when we stored the number of times a value had been rolled? If there's an element in that array that has a 3 or above then we can get the total of the dice by calling "getDiceTotal" and assign it to the variable "score" for later use. We then update the "threeOfKind" object to store the score and mark it as being scored. We then break out of the switch statement. A switch statement takes a value and has a "case" statement that contains code to run if the value passed in, matches the case statement name. I like switch statements, but you may run into people who dislike them. One famous architecture and design pattern consultant named Robert "uncle Bob" Martin dislikes them, and so many of his followers may comment on there use in your code. Many of these followers can be down right zealous in there approach to software development. My advice is to look at different types of design principles and make up your own mind.

```
    case "4kind":
        if (y.lower.scores.fourOfKind.scored ===
true) {
            return; }

        for (i = 1; i < count.length; i++) {
            if (count[i] >= 4) {
                score = y.getDiceTotal(); }
        y.lower.scores.fourOfKind.score = score;
        y.lower.scores.fourOfKind.scored = true; }
    break;
```

This function is much like the last except we’re checking that one of the array elements has a value of 4 or more.

```
    case "fh":
        if (y.lower.scores.fullHouse.scored ===
true) {
            return; }

        fullhouse = 0;

        for (i = 1; i < count.length; i++) {
            if (count[i] === 2) {
                fullhouse += 2; }
            if (count[i] === 3) {
                fullhouse += 3; } }

        if (fullhouse === 5) {
            score = 25; }

        y.lower.scores.fullHouse.score = score;
        y.lower.scores.fullHouse.scored = true;
    break;
```

This one’s different because we need 2 of one number and 3 of another. First we declare a temporary variable to hold our computations. If we find an array element with a value of 2 then we add 2 to the variable “fullhouse.” At the same time we look for a 3. Next we look to see if “fullhouse” equals 5; if ti does then we have a Full House and set the “score” variable to 25, which is the value of a Full House.

```
    case "smst":
        if (y.lower.scores.smStr.scored === true) {
            return; }

        if (count[3] > 0 && count[4] > 0) {
            if ((count[5] > 0 && count[6] > 0) ||
                (count[1] > 0 && count[2] > 0)
||
                (count[2] > 0 && count[5] > 0)) {
                score = 30; } }

        y.lower.scores.smStr.score = score;
        y.lower.scores.smStr.scored = true;
    break;
```

So how do we figure out if the player indeed has a Small Straight? Since we have our data structure, it’s easy! There’s only 3 combinations to check: “1 2 3 4”, “2 3 4 5”, or “3 4 5 6.” That’s it! They each share “3 4” so we look for those first. If the “count” array doesn’t contain a zero in those elements then we can go to the next test. If elements 5 and 6 or 1and 2 or 2 and 5 are not zero then we have a Small Straight and we can give “score” a value of 30. Let’s talk about the double &. That means “and” and it’s known as a boolean operator. In other words, it return a true or false. We use it for testing expressions that must be looked at together. In our case the 3rd element must be greater than 0 and the 6th element must be greater than 0. See how both tests have to pass? The double bars mean “or.” If we used the || operator instead of the && operator, then only one of the 2 tests would have to be true. You can see that when we test “5 6”, “1 2”, and “2 3” after we’ve tested for “3 4.” You’ll write a lot of “if” statements with these operators, so get used to them!

```
	case "lgst":
		if (y.lower.scores.lgStr.scored === true) {
			return; }

		if (count[2] > 0 && count[3] > 0 && count[4] 
> 0 && count[5] > 0) {
			if (count[1] > 0 || count[6] > 0) {
				score = 40; } }

		y.lower.scores.lgStr.score = score;
		y.lower.scores.lgStr.scored = true;
	break;
```

The Large Straight is just like the Small Straight except we need 5 in a row instead of 4. So we have to have a “2 3 4 5” first and then we check for a 1 or a 6. If all is good we give “score” a value of 40.

```
	case "y":
		if (y.lower.scores.yahtzee.scored === true) {
			return; }

		if (hasYahtzee() === true) {
			score = 50;


y.removeClassName(document.getElementById("xydiv"), 
"hidden");
		}

		y.lower.scores.yahtzee.score = score;
		y.lower.scores.yahtzee.scored = true;
	break;
```

This one’s very easy because we just call “hasYahtzee” to see if the player really does have a Yahtzee. If so, we set “score” to 50, but we have to do one more thing. We have to show the Extra Yahtzee target now that the first Yahtzee has been achieved. We use the “removeClassName” function to do that.

```
	case "xy":
		if (hasYahtzee() === false) {
			return; }

		score = 100;
		y.lower.scores.bonusYahtzee.score +=
score;
		y.lower.scores.bonusYahtzee.scored = true;
		y.maxRounds++;
	break;
```

Here’s the code to handle an Extra Yahtzee. All we need to know is if the player has a Yahtzee and we do that by calling the “hasYahtzee” function. If all is good we set “score” to 100 and increase the “maxRounds” variable. We do that because “maxRounds” is the maximum number of times a player can apply a score to the Yahtzee board. Since the Extra Yahtzee is not included in the original 13 score targets, we must add one to it if a score is applied to the Extra Yahtzee. Notice we add to the “bonusYahtzee.score” property instead of just assigning the value. That’s because the player can get more than one Extra Yahtzee.

```
	case "ch":
		if (y.lower.scores.chance.scored === true) {
```

```
            return; }

        score = y.getDiceTotal();
        y.lower.scores.chance.score = score;
        y.lower.scores.chance.scored = true;
    break;
```

Chance is where you apply the score to when there’s nothing better. We just tally all the dice and assign it to “score.”

```
    }

    y.rounds++;
    y.updateGUIscore(name, score);
    updateLowerTotal();
    y.updateGrandTotal();
    y.clearForNextRoll();
```

Here we just update various scores and increment the “rounds” property. The final thing we do is clear state for the next round.

```
};

y.applyUpperSectionPoints = function (name) {
    var i,
    score,
    updateUpperTotal,
    addScoreToTotals,
    scoreElement,
    clearForNextRoll;

    if (y.rolls === y.maxRolls) {
        return; }

    updateUpperTotal = function () {
        var total = y.calculateUpperTotal();
```

```
document.getElementById("upperTotal").innerHTML =
total; };

        score = 0;

        switch (name) {
        case "ones":
                if (y.upper.scores.ones.scored === true) {
                        return; }

                for (i = 1; i < y.dieValues.length; i++) {
                        if (y.dieValues[i] === 1) {
                                score++; }}
```

We do the same thing we did with the "applyLowerSectionPoints" function except these "case" statements are simple to calculate. We just iterate the "dieValues" array and increment "score" for each "1" we find.

```
                y.upper.scores.ones.score = score;
                y.upper.scores.ones.scored = true;
        break;

        case "twos":
                if (y.upper.scores.twos.scored === true) {
                        return; }

                for (i = 1; i < y.dieValues.length; i++) {
                        if (y.dieValues[i] === 2) {
                                score += 2; } }
```

This is the same as "ones" except we're adding 2 to "score" every time we find a two. The rest of the "case" statements work the same so I won't comment on them.

```
    y.upper.scores.twos.score = score;
    y.upper.scores.twos.scored = true;
break;

case "threes":
    if (y.upper.scores.threes.scored === true) {
        return; }

    for (i = 1; i < y.dieValues.length; i++) {
        if (y.dieValues[i] === 3) {
            score += 3; }}

    y.upper.scores.threes.score = score;
    y.upper.scores.threes.scored = true;
break;

case "fours":
    if (y.upper.scores.fours.scored === true) {
        return; }

    for (i = 1; i < y.dieValues.length; i++) {
        if (y.dieValues[i] === 4) {
            score += 4; }}

    y.upper.scores.fours.score = score;
    y.upper.scores.fours.scored = true;
break;

case "fives":
    if (y.upper.scores.fives.scored === true) {
        return; }

    for (i = 1; i < y.dieValues.length; i++) {
        if (y.dieValues[i] === 5) {
            score += 5; }}

    y.upper.scores.fives.score = score;
```

```
        y.upper.scores.fives.scored = true;
    break;

    case "sixes":
        if (y.upper.scores.sixes.scored === true) {
            return; }

        for (i = 1; i < y.dieValues.length; i++) {
            if (y.dieValues[i] === 6) {
                score += 6; } }

        y.upper.scores.sixes.score = score;
        y.upper.scores.sixes.scored = true;
    break;
    }

    y.rounds++;
    y.updateGUIscore(name, score);
    updateUpperTotal();
    y.updateGrandTotal();
    y.clearForNextRoll();
```

This is the same thing we did in the lower section code.

```
};

y.startMsgSystem();
y.loadHighScore();
```

These two functions are called to start the messaging system and load the previous high score.

```
}());
```

Well, that does it for the code commentary.

Some Things to Know

There are some topics we didn’t mention before because we just didn’t use them. Now I want to go over a few things so you’ll have a better understanding of JavaScript.

Closure: If I define a function inside another function then the inner function will have access to the outer function’s variables. That’s all there is to closure. In the example below, the set_age function “closes” around the age variable. Take a look at this code:

```
var person = (function () {
        var age = 3;
        var set_age = function (num) {
                age = num; };

        return set_age;
}());
```

Now since we made this an immediately invoked function, person will equal the set_age function. The set_age function takes a number and sets age to it. The outer function is no longer available to code, but the inner function can access the age variable; that’s closure and it’s very useful! So, the parent function is gone but the inner function still has access to the parent’s variables. That’s all there is to it! We could also return an object that has properties that access the parent’s variables.

Hoisting: JavaScript uses functional scope instead of block scope. Block scope means variables can only be seen by code in the same block. So an “if” statement would define its own block inside its curly braces and variables declared inside it would not be available outside it. With functional scope, all those variables inside the “if” statement get “hoisted” to the top of the containing function (the function the code sits in.) That’s not the whole story though. The variables get hoisted to the top of the function and are assigned as undefined. Then your assignment takes place where you originally declared the variable. All this happens in memory and you never see this actually happen, but you can see the result if you run this code:

```
var test = function () {
      console.log(x);
      var x = 50;
      console.log(x);
};
test();
```

If you run this code you’ll see undefined and then 50. The variable x got hoisted above the first console.log and assigned the value undefined. After that, x got assigned the value 50. If you comment out the “var x = 50” line then you get an error saying x is not defined. This is why it’s good practice to declare your variables at the top of functions.

Variables aren’t the only things that get hoisted in JavaScript. Functions are hoisted as well. But there’s two kinds of functions, declarations and expressions. And they get hoisted in different ways. Function declarations are hoisted along with the definition while function expressions are hoisted like variables. So how do we tell the difference? If the first word of a line is "function" then it's a declaration. In other words declarations only happen inside and outside functions and never as an argument to another construct, such as another function or an "if" statement. Run this code:

```
var test = function () {
      s();
      function s () {
            console.log(10); }

      s2();
      var s2 = function () {
            console.log(22); };
};

test();
```

It prints 10, but says s2 is not a function. Function expressions cannot be called before they're defined. Function declarations can because the definition gets hoisted along with the name. So the above code looks like this to the parser:

```
var test = function () {
      var s;
      var s2;

      s = function s () {
            console.log(10); };
      s();

      s2();
      s2 = function () {
```

```
                    console.log(22); };
    };

    test();
```

I use expressions, but you’ll see a bunch of code using declarations. Now you know how they work.

Functions: A function in JavaScript is called a first class object. That means you can pass it around to other functions just like other variable types. It's main use is to put code in one place so it doesn't have to be retyped each time you want to use that code. Put the code in a function and you can call it many times from any where in your program. A function can take arguments that become local variables to the function's code. It also gets 2 invisible arguments called “this” and “arguments.” You don't have to specify the argument names in a function because the “arguments” object contains everything that was passed in to the function. This is very different from Java that requires named parameters for its arguments. The “this” object is the “object of invocation.” That means the context of how the function was called. If you just call the function then the “this” object will be the global environment that JavaScript is running in; which is the Window object of the browser. A method is a function that's assigned to an object's property. So if you call a method then “this” will be the containing object and you can use it to access other properties on the object.

A constructor function is one that will add properties to the “this” object and return it by default if no other object is returned. In order to have this behavior, a constructor must be called using the “new” key word. If the “new” key word is not used then the function will be an ordinary function and “this” will be the global object or undefined. Using “new” creates a new “this” object and that's why it's important not to forget it. In order to give a visual clue that a programmer should call a function with the “new” key word, a constructor is capitalized. Now you know...

Inheritance: In JavaScript, inheritance is handled through prototypes. You might remember that a prototype is a secret link to another object that every object, including functions, have. An object gets a prototype when it is created or when an object is assigned to the prototype property of an object. If we create an empty object like this:

```
var obj = {};
```

Obj will have a prototype that points to Object.prototype, which is built into JavaScript and is given to every object literal. Object.prototype contains common object methods like “toString.” Likewise, unless you change it, functions will have a prototype of Function.prototype. There's also an Array.prototype.

Ok, so how do we use the prototype to implement inheritance? Look at this code:

```
var inherit = (function () {
        var Proxy = function () {};
        return function (Child, Parent) {
                Proxy.prototype = Parent.prototype;
                Child.prototype = new Proxy();
                Child.prototype.constructor = Child; };
}());

var Parent = function Parent (name) {
        this.name = name || “unknown”;
        this.owns_car = true; };
Parent.prototype.sayName = function () {
        console.log(this.name); };

var Child = function Child (name, age) {
        Parent.call(this, name);
        this.age = age; };

inherit(Child, Parent);
```

So we have 2 constructors, a Parent and a Child. We know they're constructors because we capitalized the first letters of the names. Make sure you use named function expressions or the object's constructor name will be blank. Look at this:

```
var Parent = function Parent () {};
```

Using the “Parent” name twice makes a named function expression. Leaving off the second “Parent” will create an anonymous function without a name. First, Parent adds a “name” property to its “this” object. If a name is not supplied the name will be “unknown.” Remember, constructors return “this.”

Remember, all properties inside the Parent constructor will be copied onto the Child's “this” object. That means we can put an array on the Parent constructor and each Child will get its own copy of it.   We want the Child objects to inherit everything on the Parent's “this” so inside the Child constructor we “borrow” the Parent's constructor and pass in the Child's “this” object for the Parent constructor to use. That's what the “call” method is doing; it also passes in one parameter, “name.” Doing that will add the “name” property onto the Child's “this” object. Keep reading that until you get it. Remember, the Parent's prototype is where we put anything that should be shared. That usually means just methods that operate with “this.” If you put an array on a prototype, it will be shared!

What's the “inherit” method? That takes the place of “Child.prototype = Parent.prototype.” That would create a shared prototype and that means changes are seen by all! Instead, the “inherit” function uses a “proxy” object to stand between the Parent and Child so the Child won't share the Parent's prototype. You would use the code like this:

```
var kirk = new Parent(“Kirk”);
var kim = new Child(“Kim”, 12);
kim.sayName();
```

If you run that you will see “Kim.” I think you're ready to go forth and create objects in your own hobby programs.

There’s so much more to JavaScript than I presented here and I encourage you to visit YouTube and search for Douglas Crockford, Nicholas C. Zakas, and Stoyan Sefanov. Each of these people are masters of JavaScript and you’ll learn a great deal from their videos and books.

After reading this booklet you should have enough knowledge that you won’t be lost when you encounter other code. I just hope you got something out of this book and it inspires you to learn more about JavaScript and make some hobby programs!

Good luck to you.

Kirk Burleson
Kirk.burleson@yahoo.com