JavaScript Absolute Beginner's Guide Part 1 and 2

Part 1: The Basic Stuff

Chapter 2: Values and Variables

- JavaScript code is placed between 2 script tags in an html document.

  ```
  <script> alert("This is an alert!") </script>
  ```

Variable Naming Rules:

```
- Identifiers cannot be keywords.
- Identifiers can contain alphabets and numbers.
- Identifiers cannot contain spaces and special characters, except the
underscore (_) and the dollar ($) sign.
- Variable names cannot begin with a number.
```

Variable Declaration:

```
var winners = 2;
var name = "Chief";
var isEligible = false;
var not_declared;
var $;
var __$abc;
```

List of Keywords

```
break case catch class const continue debugger default
delete do else enum export extends false finally
for function if implements import in instanceof interface
let new package private protected public return static
super switch this throw true try typeof var
void while with yield
```

Chapter 3: Functions

Most Basic Function:

```
function sayHello() {
    alert("hello!");
}
```

Funciton that takes an argument:

```
function showDistance(speed, time) {
    alert(speed * time);
 }
```

Function that returns:

```
function getDistance(speed, time) {
    var distance = speed * time;
    return distance;
}
```

Chapter 4: Conditional Statements

If Statements:

```
var safeToProceed = true;
    if (safeToProceed) { alert("You shall pass!");
} else {
    alert("You shall not pass!");
}

function amISpeeding(speed) {
    // If / Else If
    if (speed >= speedLimit) {
        alert("Yes. You are speeding.");
    } else {
        alert("No. You are not speeding. What's wrong with you?");
    }
}

// If / Else-If / Else
if (position < 100) {
    alert("Do something!");
    } else if ((position >= 100) && (position < 300)) {
        alert("Do something else!");
    } else {
        alert("Do something even more different!");
}
```

Conditional Operators in JavaScript:

```
==                equal to
>=                greater than or equal to
>                 greater than
<=                less than or equal to
<                 less than
!=                not equal to
&&                and
||                or
!==               not equal value or not equal type
===               equal value and equal type
```

- Reference: https://www.w3schools.com/jsref/jsref_operators.asp

Switch Statement:

```
switch (expression) {
    case value1:
        statement;
        break;
    case value2:
        statement;
        break;
    case value3:
        statement;
        break;
    default:
        statement;
}
```

Switch V.S. If-Else Statement:

```
var number = 20;

if (number > 10) {
    alert("yes");
} else {
    alert("nope");
}
```

```
switch (number > 10) {
    case true:
        alert("yes");
        break;
    case false:
        alert("nope");
        break;
}
```

Chapter 5: Loops

The `for` Loop:

```
var count = 10;

function saySomething() {
    document.writeln("hello!");
}

for (var i = 0; i < count; i++) {
    saySomething();
}
```

- Output: The word "hello!" will be repeated ten times across your page.

Stopping a Loop Early:

```
for (var i = 0; i < 100; i++) {
    document.writeln(i);
    if (i == 45) {
        break;
    }
}
```

Skipping an Iteration:

```
var floors = 28;
for (var i = 1; i <= floors; i++) {
    if (i == 13) {
        // no floor here
        continue;
    }
    document.writeln("At floor: " + i + "<br>");
}
```

Going Backwards:

```
for (var i = 25; i > 0; i--) {
    document.writeln("hello");
}
```

You Don't Have to Use Numbers:

```
for (var i = "a"; i !="aaaaaaaa"; i += "a") {
    document.writeln("hmm...");
}
```

Array Iteration:

```
var myArray = ["one", "two", "three"];
for (var i = 0; i < myArray.length; i++) {
    document.writeln(myArray[i]);
}
```

`for(; boolean;)` Loops:

```
var i = 0;
var yay = true;

for (; yay;) {
    if (i == 10) {
        yay = false;
    }
    i++;
    document.writeln("weird");
}
```

The `while` Loop:

```
var count = 0;
while (count < 10) {
    document.writeln("looping away!");
    count++;
}
```

`do...while`

```
var count = 0;

do {
    count++;
    document.writeln("I don't know what I am doing here!");
} while (count < 10);
```

- The main difference between a while loop and a do...while loop is that the contents of a while loop could never get executed if its conditional expression is false from the very beginning.

  ```
  while (false) {
      document.writeln("Can't touch this!");
  }
  ```

- With a do...while loop, because the conditional expression is evaluated only after one iteration, your loop's contents are guaranteed to run at least once.

  ```
  do {
      document.writeln("This code will run once!");
  } while (false);
  ```

- Note that `break` and `continue` work in `while` loops as well.

Chapter 6: Timers

Delaying with `setTimeout`

- The `setTimeout` function enables you to delay executing some code. The way you use it is pretty simple. This function makes it possible for you to specify what code to execute and how many milliseconds (aka 1/1000 of a second) to wait before the code you specified executes.
  - Putting that into JavaScript, it looks something like this: `var timeID = setTimeout(someFunction, delayInMilliseconds);`
- Further Example:

```
function showAlert() {
    alert("moo");
}
var timeID = setTimeout(showAlert, 5000);
```

- If you ever wanted to access this `setTimeout` timer again, you need a way to reference it. By associating a variable named `timeID` with our `setTimeout` declaration, we can easily accomplish that.
    - The main reason you would want to do this is so that later you could cancel the timer using the `clearTimeout` function and passing the timeout ID as the argument: `clearTimeout(timeID);`
- using `setTimeout` to detect whether a user is idle: https://www.kirupa.com/html5/detecting_if_the_user_is_idle_or_inactive.htm

Looping with `setInterval`

- `setInterval` doesn't just execute the code once. It keeps on executing the code in a loop forever.
- Here is how you would use the setInterval function: `var intervalID = setInterval(someFunction, delayInMilliseconds);`
- If you wish to cancel the looping, you can use the appropriately named `clearInterval` function: `clearInterval(intervalID);`
- Aside: For the longest time, `setInterval` was the primary function you had for creating animations in JavaScript. To get something running at 60 frames a second, you would do something that looks as follows:

```
// 1000 divided 60 is the millisecond value for 60fps
window.setInterval(moveCircles, 1000/60);
```

    - Using the `setInterval` function in this way doesn't guarantee that frames won't get dropped when the browser is busy optimizing for other things. To avoid your animation code from being treated like any other generic piece of code, you have the `requestAnimationFrame` function.

Animationg Smoothly with `requestAnimationFrame`

- The way you use this function starts off a bit similar to setTimeout and setInterval: `var requestID = requestAnimationFrame(someFunction);`
    - The only real difference is that you don't specify a duration value. The duration is automatically calculated based on the current frame rate, whether the current tab is active or not, whether your device is running on battery or not, and a whole host of other factors that go beyond what we can control or understand.
- In practice, you'll rarely make a single call to `requestAnimationFrame` like this. Key to all animations created in JavaScript is an animation loop, and it is this loop that we want to throw `requestAnimationFrame` at. The result of that throw looks something as follows:

```
var requestID;
```

```
function animationLoop() {
    // animation-related code

    requestID = requestAnimationFrame(animationLoop);
}

// start off our animation loop!
animationLoop();
```

- To stop a `requestAnimationFrame` function, you have the `cancelAnimationFrame` function: `cancelAnimationFrame(requestID);`

Chapter 7: Variable Scope

Global Scope

```
var counter = 0;
function returnCount() {
    return counter;
}
```

- The variable `counter` is declared outside of the function `returnCounter` and is therefore a global variable.
- Something is considered global in JavaScript when it is a direct child of your browser's `window` object. That is a more precise way of saying "declared outside of a function." You can verify this pretty easily by checking if counter and `window.counter` point to exactly the same thing: `alert(window.counter == counter);`
    - The answer is going to be **true**. The reason is that you are referring to the exact same thing.
    - Realizing that global variables live under your `window` object should help you understand why you can access a global variable from anywhere in your document.
-

Local Scope

```
function setState() {
    var state = "on";
}
setState();
alert(state) // This will not work
```

- In this example, the `state` variable is declared inside the `setState` function, and accessing the state variable outside of that function doesn't work. The reason is that the scope for your `state` variable is local to the `setState` function itself.
- **JavaScript doesn't support block scoping.**
  - Meaning that this code...

```
function checkWeight(weight) {
    if (weight > 5000) {
        var text = "No free shipping for you!";
        alert(text);
    }
    alert(text); // how did it know??!
}
checkWeight(6000);
```

  - is identical to this code in the mind of JavaScript...

```
function checkWeight(weight) {
    var text;
    if (weight > 5000) {
        text = "No free shipping for you!";
        alert(text);
    }
    alert(text);
}
checkWeight(6000);
```

  - Notice where the variable `text` is declared in both examples.

Part 2: It's an Object-Oriented World

Chapter 11: Types, Primitives, and Objects

Basic JavaScript Types

Type	What It Does
String	The basic structure for working with text
Number	Allows you to work with numbers
Boolean	Comes alive when you are using true and false

Type	What It Does
Null	Represents the digital equivalent of nothing...or moo :P
Undefined	While sorta similar to null, this is returned when a value should exist but doesn't...like when you declare a variable but don't assign anything to it
Object	Acts as a shell for other types including other objects

Some Pre-Defined JavaScript Objects

Type	What It Does
Array	Helps store, retrieve, and manipulate a collection of data
Boolean	Acts as a wrapper around the `boolean` primitive; still very much in love with true and false
Date	Allows you to more easily represent and work with dates
Function	Allows you to invoke some code among aother estoeric things
Math	Helps you work better with numbers
Number	Acts as a wrapper for the `number` primitive
RegExp	Provides a lot of functionality for matching patterns in text
String	Acts as a wrapper around the `string` primitive

```
// an array
var names = ["Jerry", "Elaine", "George", "Kramer"];
var alsoNames = new Array("Dennis", "Frank", "Dee", "Mac");

// a round number
var roundNumber = Math.round("3.14");

// today's date
var today = new Date();

// a boolean object
var booleanObject = new Boolean(true);

// infinity
var unquantifiablyBigNumber = Infinity;
```

```
// a string object
var hello = new String("Hello!");
```

Chapter 12: Strings

Accessing Individual Characters:

```
var vowels = "aeiou";
alert(vowels[2]);  // Would output 'i'
```

Making Substrings out of Strings:

The `slice` Method:

- The `slice` method allows you to specify the start and end positions of the part of the string that you want to extract:

  ```
  var theBigString = "Pulp Fiction is an awesome movie!";
  alert(theBigString.slice(5, 12));
  ```

  - In this example, we extract the characters between index positions 5 and 12.
  - If you specify a negative start position, you start position is the count of whatever you specify starting from the end of the string:

    ```
    var theBigString = "Pulp Fiction is an awesome movie!";
    alert(theBigString.slice(-14, -7));
    ```

The `substr` Method:

- The next approach we will look at for splitting up your string is the `substr` method. This method takes two arguments as well:
  `var newString = substr(start, length);`
- Example:

  ```
  var theBigString = "Pulp Fiction is an awesome movie!";
  alert(theBigString.substr(0, 4)); // Pulp
  alert(theBigString.substr(5, 7)); // Fiction
  alert(theBigString.substr(5)); // Fiction is an awesome movie!
  ```

Splitting a String with `split`:

```
var days = "Monday,Tuesday,Wednesday,Thursday,Friday, Saturday,Sunday";
var splitWords = days.split(",");
```

```
alert(splitWords[6]); // Sunday
```

Finding Something Inside a String:

- If you ever need to find a character or characters inside a string, you can use the `indexOf`, `lastIndexOf`, and `match` methods.

`indexOf`

```
var question = "I wonder what the pigs did to make these birds so angry?";
alert(question.indexOf("pigs")); // 18
alert(question.indexOf("z")); // -1
```

- Because what I am looking for does exist, the indexOf method lets me know that the **first occurrence** of this word occurs at the 18th index position.
- If I look for something that doesn't exist, like the letter z in this example, a –1 gets returned for:

`lastIndexOf`

```
var question = "How much wood could a woodchuck chuck if a
woodchuck could chuck wood?";
alert(question.lastIndexOf("wood")); // 65
```

- **NOTE:** There is one more argument you can specify to both `indexOf` and `lastIndexOf`. In addition to providing the characters to search for, you can also specify an index position on your string to start your search from:

  ```
  var question = "How much wood could a woodchuck chuck if a woodchuck
  could chuck wood?";
  alert(question.indexOf("wood", 30)); // 43
  ```

`match`

- With the `match` method, you have a little more control. This method takes a Regex as its argument:

  ```
  var phrase = "There are 3 little pigs.";
  var regexp = /[0–9]/;
  var numbers = phrase.match(regexp);
  alert(numbers[0]); // 3
  ```

Upper and Lower Casing Strings:

```
var phrase = "My name is Bond. James Bond.";
alert(phrase.toUpperCase()); // MY NAME IS BOND. JAMES BOND.
```

```
alert(phrase.toLowerCase()); // my name is bond. james bond.
```

Chapter 13: When Primitives Behave Like Objects

```
var game = "Dragon Age: Origins";
alert("Length is: " + game.length);

var gameObject = new String("Dragon Age:Origins");

alert(typeof game); // string
alert(typeof game.length); // number
alert(typeof gameObject); // Object
```

Chapter 14: Arrays

Initialization:

```
var graoceries = [];  // empty init

// filled init
var groceries = ["Milk", "Eggs", "Frosted Flakes", "Salami", "Juice"];
```

Accessing Array Values in a Loop:

```
for (var i = 0; i < groceries.length; i++) {
    var item = groceries[i];
}
```

Adding Items to your Array:

```
// add items to end of your array
groceries.push("Cookies");



// add items to start of your array
groceries.unshift("Bananas");
```

- Note that, when you are adding items to your array using unshift or push, **the returned value from that method call is the new length of your array.**

Removing Items from the Array:

```
// removes last item from array and returns it
var lastItem = groceries.pop();

// removes first item from array and returns it
var firstItem = groceries.shift();

// return a portion of the array
var newArray = groceries.slice(1, 4);
```

Finding Items in the Array:

- To use `indexOf`, I pass in the element I am looking for along with the index position to start from:

  ```
  groceries.indexOf("eggs", 0);
  ```

Merging Arrays:

```
var good = ["Mario", "Luigi", "Kirby", "Yoshi"];
var bad = ["Bowser", "Koopa Troopa", "Goomba"];

var goodAndBad = good.concat(bad);
alert(goodAndBad);
```

- In this example, because the concat method returns a new array, the goodAndBad variable ends up becoming an array that stores the results of our concatenation operation. The order of the elements inside goodAndBad is good first and bad second.

Chapter 15: Numbers

- In JavaScript, all numbers are converted into 64-bit floating point numbers.
- These are all valid examples of numbers:

```
var stoodes = 3;
var pi = 3.14159;
var color = 0xFF;
var massOfEarth = 5.9742e+24;

var temperature = -42;
```

Operators:

Expression	What It Does

Expression	What It Does
i ++	Increments i by 1 (i = i + 1)
i --	Decrements i by 1 (i = i – 1)
i += n	Increments i by n (i = i + n)
i -= n	Decrements i by n (i = i – n)
i *= n	Multiplies by n (i = i * n)
i /= n	Divides i by n (i = i / n)
i %= n	Finds the remainder of i when divided by n (i = i % n)

- Examples:

  ```
  var total = 4 + 26;
  var average = total / 2;
  var doublePi = 2 * 3.14159;
  var removeItem = 50—25;
  var remainder = total % 7;
  var more = (1 + average * 10) / 5;
  ```

Order of Operations:

1. Parenthesis
2. Exponents
3. Multiply
4. Divide
5. Add
6. Subtract

Special Values

Infinity

- You can use the `Infinity` and `-Infinity` values to define infinitely large or small numbers:

  ```
  var reallyBigNumber = Infinity;
  var reallySmallNumber = -Infinity;
  ```

  - The chances of you having to use `Infinity` are often very slim. Instead, you will probably see it returned as part of something else your code does. For example, you will see `Infinity` returned if you divide by 0.

NaN

- The `NaN` keyword stands for "Not a Number", and it gets returned when you do some numerical operation that is invalid. For example, NaN gets returned in the following case: `var oops = Math.sqrt(-1);`

The Math Object

The Constants

Property	What It Stands For
Math.E	Euler's constant
Math.LN2	Natural logarithm of 2
Math.LN10	Natural logarithm of 10
Math.LOG2E	Base 2 logarithm of E
Math.LOG10E	Base 10 logarithm of E
Math.PI	3.14159...
Math.SQRT1_2	Square root of 1/2
Math.SQRT2	Square root of 2

Rounding Numbers

Function	What It Does
Math.round()	Returns a number that is rounded to the nearest integer. You round up if your argument is greater than or equal to .5. You stay at your current integer, if your argument is less than .5.
Math.ceil()	Returns a number that is greater than or equal to your argument
Math.floor()	Returns a number that is less than or equal to your argument

```
Math.floor(.5); // 0
Math.ceil(.5); // 1
Math.round(.5); // 1

Math.floor(3.14); // 3
Math.round(3.14); // 3
Math.ceil(3.14); // 4

Math.floor(5.9); // 5
Math.round(5.9); // 6
Math.ceil(5.9); // 6
```

Trigonometric Functions

Funciton	What It Does
Math.cos()	Gives you the cosine for a given argument
Math.sin()	Gives you the sine for a given argument
Math.tan()	Gives you the tan for a given argument
Math.acos()	Gives you the arccosine for a given argument
Math.asin()	Gives you the arcsine for a given argument
Math.atan()	Gives you the arctan for a given argument

```
Math.cos(0); // 1
Math.sin(0); // 0
Math.tan(Math.PI / 4); // 0.9999999999999999
Math.cos(Math.PI); // -1
Math.cos(4 * Math.PI); // 1
```

Powers and Square Roots

Function	What It Does
Math.pow()	Raises a number to a specified power
Math.exp()	Raises the Euler's constant to a specified number
Math.sqrt()	Returns the square root of a giving argument

```
Math.pow(2, 4) //equivalent of 2^4 (or 2 * 2 * 2 * 2)
Math.exp(3) //equivalent of Math.E^3
Math.sqrt(16) //4
```

Getting the Absolute Value

- If you want the absolute value of a number, simply use the Math.abs() function:

  ```
  Math.abs(37) //37
  Math.abs(-6) //6
  ```

Random Numbers

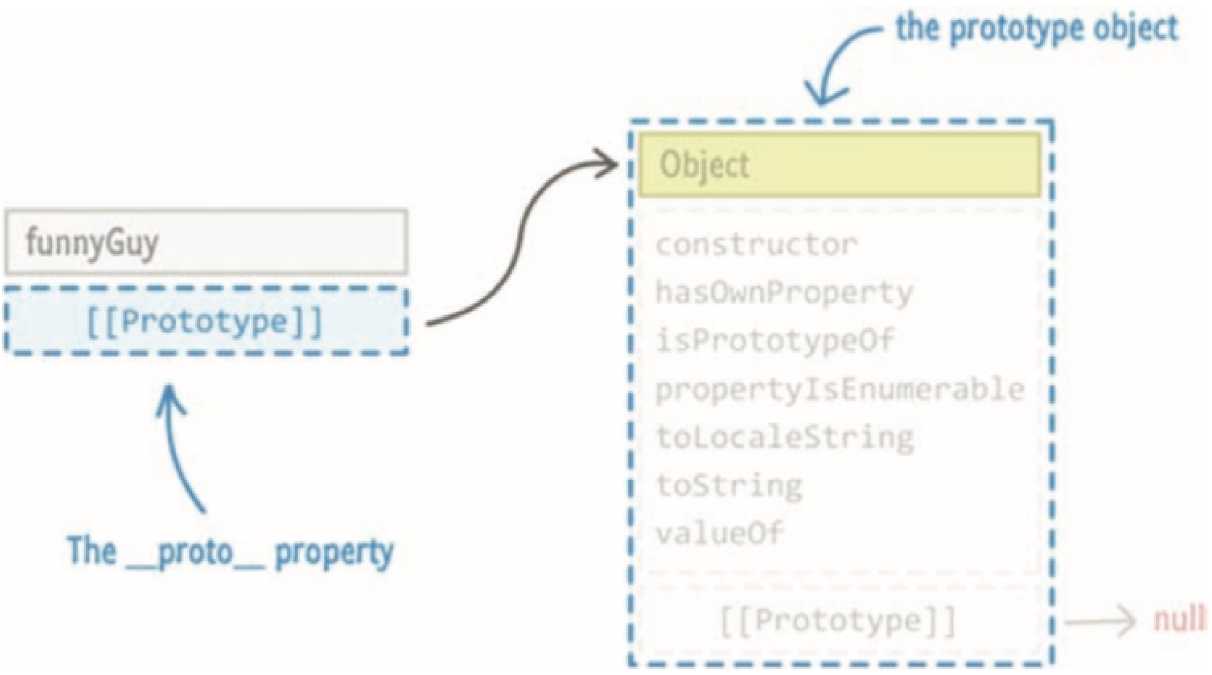
- To generate a somewhat random number between 0 and a smidgen less than 1, you have the `Math.random()` function. This function doesn't take any arguments, but you can simply use it as part of a mathematical expression:
  `var randomNumber = Math.random() * 100;`

Chapter 16: A Deeper Look At Objects

Creating Objects

```
var funnyGuy = {};
```

- You can initialize an object by using two curly bracket: `{}`
  - In the above example, `funnyGuy` is simply an empty object. While there may not be any properties we defined on it, there is a special internal property that exists called `__proto__` and often visualized as `[[Prototype]]` that points to the internally defined `Object`.



  - Because it is "derived" from our Object type, you can access any properties the Object contains through funnyGuy itself.
  - For example, I can do something like this:

```
var funnyGuy = {};
funnyGuy.toString(); // [object Object]
```

Specifying Properties

- Example of specifying some properties called `firstName` and `lastName` using Dot Notation:

```
var funnyGuy = {};
funnyGiy.firstName = "Conan";
funnyGuy.lastName = "O'Brien";
```

- Example of specifying some properties called `firstName` and `lastName` using Square Bracket Notation:

```
var funnyGuy = {};
funnyGuy["firstName"] = "Conan";
funnyGuy["lastName"] = "O'Brien";
```

- Example of specifying some properties called `firstName` and `lastName` using Literal Notation:

```
var funnyGuy = {
    firstName: "Conan",
    lastName: "O'Brien"
};
```

- Example of creating a method called `getName` on `funnyGuy` that will return the value of the `firstName` and `lastName` properties:

```
var funnyGuy = {
    firstName: "Conan",
    lastName: "O'Brien",

    getName: function() {
        return "Name is: " + this.firstName + " " + this.lastName;
    }
};
```

  - To call this `getName` property, which is a funciton whose body returns a string the includes the value of our `firstName` and `lastName` properties, you do this:

    ```
    alert(funnyGuy.getName()); // Name is: Conan O'Brien
    ```

Creating Custom Objects

- Lets say we want to create 3 objects that all contain a `getName` property function... We can do this using the `Object.create` method, which creates a new object and allows you to specify what the newly created object's prototype will be.

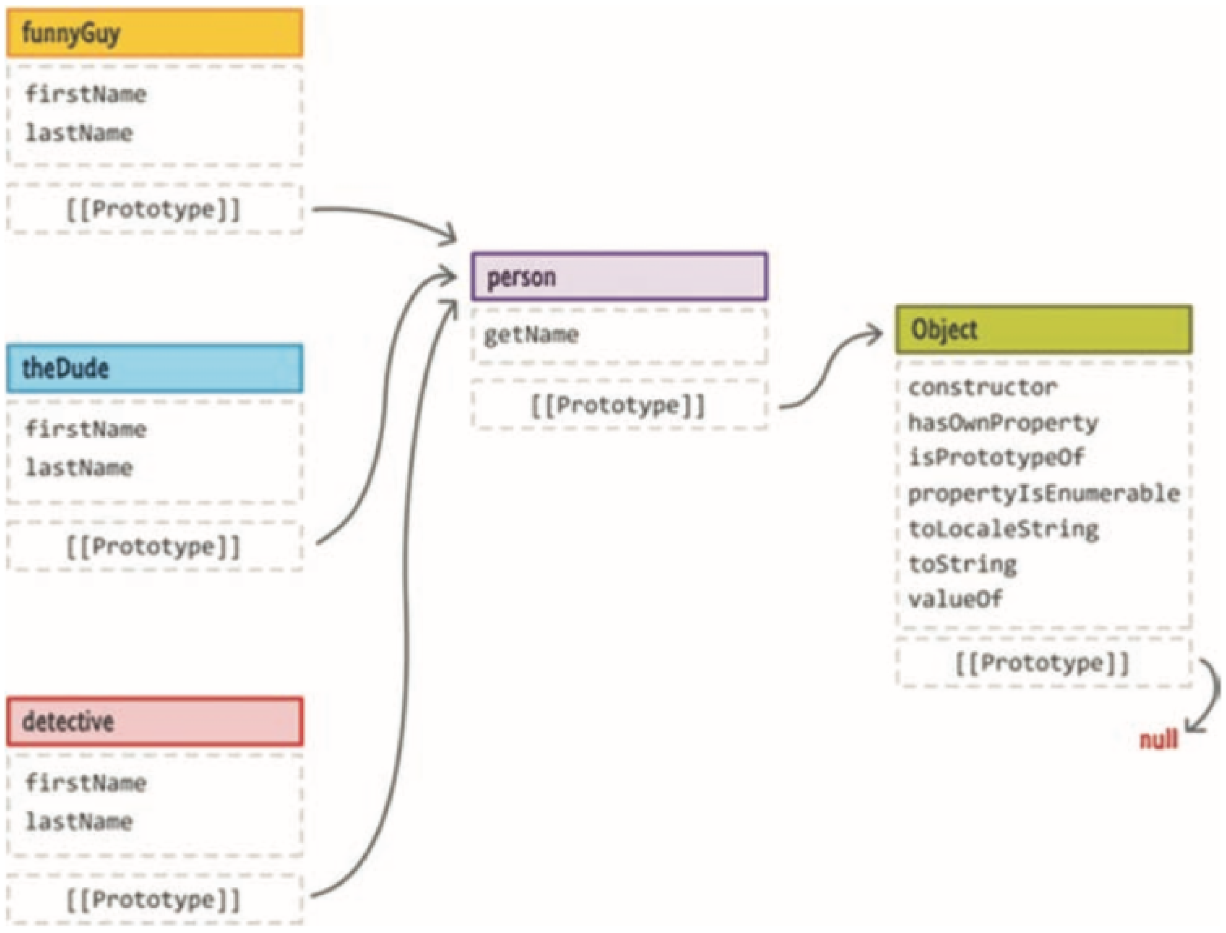
```
var person = {
    getName: function () {
```

```
        return "Name is " + this.firstName + " " + this.lastName;
    }
};
var funnyGuy = Object.create(person);
funnyGuy.firstName = "Conan";
funnyGuy.lastName = "O'Brien";

var theDude = Object.create(person);
theDude.firstName = "Jeffrey";
theDude.lastName = "Lebowski";

var detective = Object.create(person);
detective.firstName = "Adrian";
detective.lastName = "Monk";
```

- Notice in the `getName` function that it refers to properties using the `this` keyword, which will grab the properties from the object that is calling the function.



Chapter 17: Extending Built-In Objects

- Below is an example of a function that can be used to shuffle the contents of an array:

```
function shuffle(input) {
    for (var i = input.length - 1; i >= 0; i--) {

        var randomIndex = Math.floor(Math.random() * (i + 1));
        var itemAtIndex = input[randomIndex];

        input[randomIndex] = input[i];
        input[i] = itemAtIndex;
    }
    return input;
}
```

- This function can be used by passing in an array as a parameter to the function:

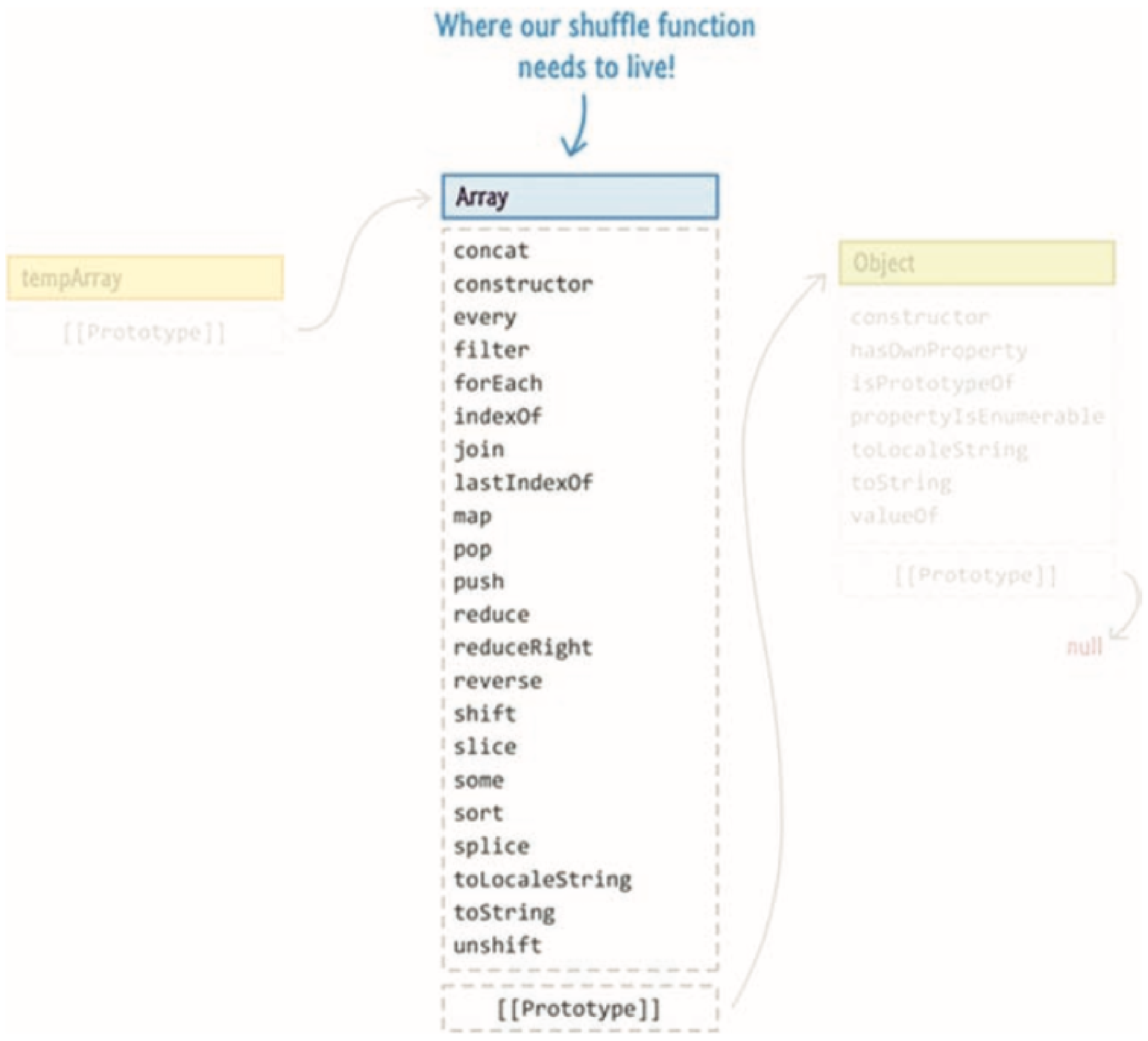
  ```
  var tempArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  shuffle(tempArray);
  ```

- Now imagine you want to have this funciton accessible to any and every `Array Object` within your program. To do this you must Extend the built in `Array Object`'s functionality.

```
Array.prototype.shuffle = function () {
    var input = this;
    for (var i = input.length - 1; i >= 0; i--) {
        var randomIndex = Math.floor(Math.random() * (i + 1));
        var itemAtIndex = input[randomIndex];
        input[randomIndex] = input[i];
        input[i] = itemAtIndex;
    }
    return input;
}
```

- This new built in `Array Object` funciton can be used like so:

  ```
  var tempArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  tempArray.shuffle();
  ```



- **Note:** You can use this same method to override the default implementation of built in functions.
  - For Example, overridding the `slice` function's default behavior:

    ```
    Array.prototype.slice = function () {
        var input = this;
        input[0] = "This is an awesome example!";
        return input;
    }
    ```

- **Note:** Extending functionality and overridding functions is somewhat controversial and should be done with care because *you don't control the built-in object's future.*

Chapter 18: Booleans, === and !== Operators

- Boolean primitives:

  ```
  var sunny = false;
  var traffic = true;
  ```

- *Behind every primitive there is an Object-based representation lurking in the shadows*
- The way you create a new boolean Object is by using the new keyword, the `Boolean` constructor name, and an initial value:

  ```
  var boolObject = new Boolean(false);
  var anotherBool = new Boolean(true);
  ```

  - The initial values you can pass in to the Boolean constructor are commonly `true` and `false`, but you can pretty much pass anything in there that will result in the final evaluation being `true` or `false`.
  - However, note that you should probably stick to boolean primitives 99% of the time.
- There is one major advantage the Boolean constructor provides, and that advantage revolves around being able to pass in any arbitrary value or expression as part of creating your Boolean object:

  ```
  var boolObject = new Boolean(arbitrary expression);
  ```

- The Boolean function allows you to pass in arbitrary values and expressions while still returning a **primitive boolean value of true or false**.
  - The main difference in how you use it compared to the constructor approach is that you don’t have the new keyword.
- The values you can pass in to return false are null, undefined, empty/nothing, 0, an empty string, NaN, and (of course) false:

  ```
  var bool;
  bool = Boolean(null);
  bool = Boolean(undefined); bool = Boolean();
  bool = Boolean(0);
  bool = Boolean("");
  bool = Boolean(NaN);
  bool = Boolean(false);
  ```

  - In all of these examples, the bool variable will return false.
- . To return true, you can pass in a value of true or ANYTHING that results in something other than the various false values you saw earlier:

  ```
  var bool;
  bool = Boolean(true);
  bool = Boolean("hello");
  bool = Boolean("Liam Neesons" + "Bruce Willie"); bool = Boolean(new
  ```

```
Boolean()); // Inception!!! bool = Boolean("false"); // "false" is a
string
```

- In these examples, the bool variable will return a true.

Strict Equality and Inequality Operators

- Here is an example:

```
function theSolution(answer) {
    if (answer == 42) {
        alert("You have nothing more to learn!");
    }
}

theSolution("42"); //42 is passed in as a string
```

  - In this example, the expression `answer == 42` will evaluate to true.
    - This works despite the 42 you are passing in being a string and the 42 you are checking against being a number.
  - With the `==` and `!=` operators, this is expected behavior. The value for the two things you are comparing is 42. To make this work, JavaScript forces the two different yet similar values to be the same under the hood.
    - This is formally known as **type coercion**.
- `===` and `!==` respectively **check for both value and type** and do not perform any type coercion.

Chapter 19: Null and Undefined

Null

- The `null` keyword is also sort of a primitive that fills a special role in the world of JavaScript. It is an explicit definition that stands for no value.
  - It is quite popular, for the advantage of null lies in its definitiveness.
  - Instead of having variables contain stale values or mystery `undefined` values, setting it to null is a clear indication that you want the value to not exist.
- This advantage is important when you are writing code and want to initialize or clear a variable to something that represents nothing.
  - Here is an example:

```
var test = null;
if (test === null) {
    test = "Peter Griffin";
} else {
    test = null;
}
```

- Checking for the value of null is pretty easy as well:

```
if (test === null) {
// do something interesting...or not
}
```

Undefined

- To represent something that isn't defined, you have the `undefined` primitive.
- You see `undefined` most commonly when you try to access a variable that hasn't been initialized or when accessing the value of a function that doesn't actually return anything.
- Here is a code snippet that points out undefined in a few of its natural habitats:

```
var myVariable;
alert(myVariable); // undefined

function doNothing() {
    // watch paint dry
    return;
}
var weekendPlans = doNothing();
alert(weekendPlans); // undefined
```

- The safest way to perform a check for `undefined` involves typeof and the `===` operator:

```
var myVariable;
if (typeof myVariable === "undefined")
{
    alert("Define me!!!");
}
```

- This ensures that you will perform a check for undefined and always return the correct answer.

Chapter 20: Immediately Invoked Function Expressions

- If you give your function a name, you can re-use those grouped statements a whole bunch of times, as in the example that follows:

```
function areYouLucky() {
// Pick a random number between 0 and 100 var foo =
Math.floor(Math.random() * 100);
if (foo > 50) {
alert("You are lucky!");
} else {
alert("You are not lucky!");
} }
 // calling this function!
areYouLucky();
```

- Anonymous Functions are functions that are not named, and they look as follows:

```
// anonymous function #1
var isLucky = function () {
    var foo = Math.round(Math.random() * 100);
    if (foo > 50) {
        return "You are lucky!";
    } else {
        return "You are not lucky!";
    }
};

var me = isLucky();
alert(me);

// anonymous function #2
window.setTimeout(function () {
    alert("Everything is awesome!!!");
}, 2000);
```

- These anonymous functions can only be called if they are associated with a variable as shown in the first example.
  - They can also be called if they are provided as part of another function (such as `setTimeOut`) that knows what to do with them.

Immediately Invoked Function Expression (IIFE)

- To be very blunt, an IIFE is nothing more than a function (surrounded by a whole bunch of parentheses) that executes immediately.
- Below is a simple IIFE:

```
(function() {
var shout = "I AM ALIVE!!!";
alert(shout);
})();
```

  - Adding the `()` generally means you intend for whatever preceded the `()` to execute immediately.
    - When you do this, JavaScript will still complain because this is also not valid syntax.
    - *The reason is that you need to tell JavaScript that this function you want to execute is actually an expression.* The easiest way to do that is to wrap your function inside another pair of parentheses, as seen above, `(function() { .... })`

Writing an IIFE That Takes Arguments

- The general structure for creating an IIFE that takes arguments looks as follows:

```
(function (a, b) {
    /* code */
})(arg1, arg2);
```

- Just like with any function call, the order of the arguments you pass in maps to the order of the arguments your function will take.
- Here is a slightly fuller example:

```
(function (first, last) {
    alert("My name is " + last + ", " + first + " " + last + ".");
})("James", "Bond");
```

- Reminder: JavaScript uses **Lexical Scoping**, not block scoping.
  - Lexical Scoping means that variables declared using var inside a block such as an if statement or loop will actually be accessible to the entire enclosing function:

```
function scopeDemo() {
    if (true) {
        var foo = "I know what you did last summer!";
    }
    alert(foo); // totally exists!
}
scopeDemo();
```

Avoiding Code Collisions

- One of the biggest advantages of IIFEs is their ability to insulate any code from outside interference.
    - This is important if you are writing code that will be used widely in someone else's application.
    - You want to ensure that any existing (or new) code doesn't accidentally clobber your variables or override functions and methods.
        - The way to ensure that such accidents don't happen is to wrap all of your code inside an IIFE.
- For example, here is some code for a content slider wrapped into an IIFE:

```
(function() {
    // just querying the DOM...like a boss!
    var links = document.querySelectorAll(".itemLinks");
    var wrapper = document.querySelector("#wrapper");

    // the activeLink provides a pointer to the currently displayed item
    var activeLink = 0;

    // setup the event listeners
    for (var i = 0; i < links.length; i++) {
        var link = links[i];
        link.addEventListener('click', setClickedItem, false);

        // identify the item for the activeLink
        link.itemID = i;
    }

    // set first item as active
    links[activeLink].classList.add("active");

    function setClickedItem(e) {
        removeActiveLinks();
        var clickedLink = e.target;
        activeLink = clickedLink.itemID;

        changePosition(clickedLink);
    }

    function removeActiveLinks() {
        for (var i = 0; i < links.length; i++) {
            links[i].classList.remove("active");
        }
    }
    // Handle changing the slider position as well as ensure
    // the correct link is highlighted as being active function
changePosition(link) {
        var position = link.getAttribute("data-pos");
        wrapper.style.left = position;

        link.classList.add("active");
    }
})();
```

- As highlighted in this example, just add the first and last line that wraps everything you are doing into a function that gets immediately executed. You don't have to make any additional modifications.
    - Because all of the code inside the IIFE runs in its own scope, you don't have to worry about someone else creating their own copies of things found in your code and breaking your functionality.

Closures and Locking in State

- *Closures store their outer values by referencing them. They don't directly store the actual values.*
- Let's take a look at an example, Let's say we have a function called `quotatious`:

```
function quotatious(names){
    var quotes = [];

    for (var i = 0; i < names.length; i++) {
        var theFunction = function (){
            return "My name is " + names[i] + "!";
        }
        quotes.push(theFunction);
    }
    return quotes;
}
```

- `quotatious(names)` takes an array (for now, just think of it as a collection of values separated by a comma) of names and returns an array of functions that print out the name when called.
- To use this function, add the following lines of code:

```
//our list of names
var people = ["Tony Stark", "John Rambo", "James Bond", "Rick
James"];

//getting an array of functions
```

```
var peopleFunctions = quotatious(people);

//get the first function
var person = peopleFunctions[0];

//execute the first function
alert(person());
```

        - When the last line with the alert statement is executed, what would you expect to see? Because our person variable is getting the first item from the array of functions returned by `peopleFunctions`, it would seem reasonable to expect `My name is Tony Stark!` to appear.
        - What you would actually see is an alert displaying `My name is undefined!`
        - Your `theFunction` function relies on the value of `names[i]`. The value of `i` is defined by the for loop (in the scope of the parent function), and the for loop is outside of the `theFunction`'s scope. What you have is basically a closure with the `theFunction` function and the variable `i` being an outer variable.
            - See more detailed explanation on pages 209-212
    - To avoid getting this cumbersome error, our solution is to lock in the value of our outer variable, the troublesome `i`, inside the closure. This will involve the assistance of an IIFE.
- Below is the modified version of the `quotatious` function that does what we want it to do:

```
function quotatious(names) {
    var quotes = [];

    for (var i = 0; i < names.length; i++) {
        (function(index) {
            var theFunction = function() {
                return "My name is " + names[index] + "!";
            }
            quotes.push(theFunction);
        })(i);
    }
    return quotes;
}
```

- **NOTE:** Sometimes referencing outer variables is what you want! There may be times where you want your closure to work with the updated value of an outer variable, so in those cases, don't use an IIFE to lock-in the value.

Making Things Private

- In JavaScript, you don't have an easy, built-in way to limit the visibility of variables and properties you end up creating.
    - This means it is difficult to have parts of your app hidden from other parts of your app.
- Take a look at the following example that highlights this problem:

```
var weakSauce = {
    secretCode: "Zorb!",

    checkCode: function (code) {
        if (this.secretCode == code) {
            alert("You are awesome!");
        } else {
            alert("Try again!");
        }
    }
};
```

    - You have this object called `weakSauce` that has these two properties called `secretCode` and `checkCode`. This is a very naive password checker. When you pass in a code value to the `checkCode` function, it will check if what you entered matches the value of `secretCode`.
        - If it matches, you are awesome will get displayed.
        - If it doesn't, you will see Try again! appear instead.
    - Nothing prevents me, you, or someone who uses this code from checking the value of `secretCode` directly by using the following JavaScript snippet:

    ```
    var bar = Object.create(weakSauce); alert(bar.secretCode); // sigh :(
    ```

    - This reveals why the `secretCode` property should be hidden to everything except the internals of the `weakSauce` object.
        - In other languages, you could add an access modifier like `private` to `secretCode`, and that would be the end of this entire conversation. JavaScript isn't like other languages.
- By taking advantage of the local scope IIFEs create, you can selectively choose what to expose or what not to expose. Here is a version of the same example that works as we would want it to:

```
var awesomeSauce = (function () {
    var secretCode = "Zorb!";
    function privateCheckCode(code) {
        if (secretCode == code) {
            alert("You are awesome!");
        } else {
            alert("Try again!");
```

```
        }
    }

    // the public method we want to return
    return {
        checkCode: privateCheckCode
    };
})();
```

- By only returning an object that contains the `checkCode` property, that internally references the `privateCheckCode` function (which is a closure), you get all the functionality you originally had.
- This method of essentially making an object private in JavaScript is known as the **Revealing Module Pattern**.
    - Good resource for additional reading: https://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatternjavascript

- **NOTE:** REMEMBER, YOUR JAVASCRIPT CODE IS AN OPEN BOOK! While you may have just learned about a way to keep the internals of your code private, it is only private to the typical code you write. Even with obfuscation and other techniques, if your browser can see your code, then anybody else with a few extra seconds to spare can as well.