Full Stack Web Dev

**Command Line**

**NAVIGATION**

**ls**

The first command we’re going to look at is ls. A *command* is a directive to the computer to perform a specific task. When you type ls, the command line looks at the directory you are in, and then “lists” all the files and directories inside of it.

In the terminal, the first thing you see is $. This is called a *shell prompt*. It appears when the terminal is ready to accept a command.

**pwd**

The next command we’re going to look at is pwd, which stands for “print working directory.” It outputs the name of the directory you are currently in, called the *working directory*.

Together with ls, the pwd command is useful to show where you are in the filesystem.

**cd I**

Our next command is cd, which stands for “change directory.” Just as you would click on a folder in Windows Explorer or Finder, cd switches you into the directory you specify. In other words, cd changes the working directory.

Let’s say the directory we change into is **2015**:

$ cd 2015

When a file, directory, or program is passed into a command, it is called an *argument*. Here the 2015 directory is an argument for the cd command.

The cd command takes a directory name as an argument and switches into that directory.

**cd II**

Instead of using cd twice in order to move from **2015** to **memory**, we can use it once and give it a longer argument:

$ cd jan/memory

To navigate directly to a directory, use cd with the directory’s path as an argument. Here, cd jan/memory navigates directly to the **memory** directory.

To move up one directory, we use cd ..:

$ cd ..

Here, cd .. navigates up from **jan/memory/** to **jan/**.

**mkdir**

Now that we can traverse the existing filesystem, let’s try editing it by making directories (folders) through the command line. The command for that is mkdir.

The mkdir command stands for “make directory”. It takes in a directory name as an argument and then creates a new directory in the current working directory.

Another way to make a new directory from your current position is by using a / to combine arguments.

Use:

$ mkdir media/tv

to create a new directory named **tv** inside **media** from the directory **blog.**

It should look like this:

$ pwd

/home/ccuser/workspace/blog/2014/dec

$ ls

media monitor.txt mouse.txt

$ mkdir media/tv

$ ls

media monitor.txt mouse.txt

$ cd media

$ ls

tv

**touch**

Now we know how to create directories through the command line, but how do we create new files?

We can do this using the command touch:

$ touch keyboard.txt

The touch command creates a new file inside the working directory. It takes in a filename as an argument and then creates an empty file with that name in the current working directory.

Here we used touch to create a new file named **keyboard.txt**.

**Helper Commands**

Now that we’ve covered the basics of navigating your filesystem from the command line, let’s look at some helpful commands that will make using it easier!

clear is used to clear your terminal, which is useful when it’s full of previous commands and outputs. It doesn’t change or undo your previous commands, it just clears them from the view. You can scroll upwards to see them at any time.

tab can be used to autocomplete your command. When you are typing the name of an existing file or directory, you can use tab to finish the rest of the name.

The up and down arrows (↑ and ↓) can be used to cycle through your previous commands. ↑ will take you up through your most recent commands, and ↓ will take you back through to the most recent one.

**Review**

Congratulations! You’ve learned five commands commonly used to navigate the filesystem from the command line. What can we generalize so far?

* The *command line* is a text interface for the computer’s operating system. To access the command line, we use the terminal.
* A *filesystem* organizes a computer’s files and directories into a tree structure. It starts with the *root directory*. Each parent directory can contain more child directories and files.
* From the command line, you can navigate through files and folders on your computer:
  + pwd outputs the name of the current working directory.
  + ls lists all files and directories in the working directory.
  + cd switches you into the directory you specify.
  + mkdir creates a new directory in the working directory.
  + touch creates a new file inside the working directory.
* You can use helper commands to make navigation easier:
  + clear clears the terminal
  + tab autocompletes the name of a file or directory
  + ↑ and ↓ allow you to cycle through previous commands

***Bonus***

1. echo "Hello Command Line" >> hello\_cli.txt to create a new file named **hello\_cli.txt** and add Hello Command Line to that file. When you type ls, you should see the following:

$ ls  
hello\_cli.txt

1. cat hello\_cli.txt to print the contents of the **hello\_cli.txt** file to the terminal. You should see something like:

$ cat hello\_cli.txt  
Hello Command Line

**Intro to JavaScript**

**Console**

When we write console.log() what we put inside the parentheses will get printed, or logged, to the console.

console.log(5);

**Comments**

Programming is often highly collaborative. In addition, our own code can quickly become difficult to understand when we return to it— sometimes only an hour later! For these reasons, it’s often useful to leave notes in our code for other developers or ourselves.

As we write JavaScript, we can write comments in our code that the computer will ignore as our program runs. These comments exist just for human readers.

Comments can explain what the code is doing, leave instructions for developers using the code, or add any other useful annotations.

There are two types of code comments in JavaScript:

1. A *single line comment* will comment out a single line and is denoted with two forward slashes // preceding it.

// Prints 5 to the console  
console.log(5);

You can also use a single line comment to comment after a line of code:

console.log(5);  // Prints 5

1. A *multi-line comment* will comment out multiple lines and is denoted with /\* to begin the comment, and \*/ to end the comment.

/\*  
This is all commented   
console.log(10);  
None of this is going to run!  
console.log(99);  
\*/

You can also use this syntax to comment something out in the middle of a line of code:

console.log(/\*IGNORED!\*/ 5);  // Still just prints 5

Data Types

*Data types* are the classifications we give to the different kinds of data that we use in programming. In JavaScript, there are seven fundamental data types:

* *Number*: Any number, including numbers with decimals: 4, 8, 1516, 23.42.
* *String*: Any grouping of characters on your keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: ' ... ' or double quotes " ... ". Though we prefer single quotes. Some people like to think of string as a fancy word for text.
* *Boolean*: This data type only has two possible values— either true or false (without quotes). It’s helpful to think of booleans as on and off switches or as the answers to a “yes” or “no” question.
* *Null*: This data type represents the intentional absence of a value, and is represented by the keyword null (without quotes).
* *Undefined*: This data type is denoted by the keyword undefined (without quotes). It also represents the absence of a value though it has a different use than null.
* *Symbol*: A newer feature to the language, symbols are unique identifiers, useful in more complex coding. No need to worry about these for now.
* *Object*: Collections of related data.

The first 6 of those types are considered *primitive data types*.

**Arithmetic Operators**

1. Add: +
2. Subtract: -
3. Multiply: \*
4. Divide: /
5. Remainder: %

**String Concatenation**

Operators aren’t just for numbers! When a + operator is used on two strings, it appends the right string to the left string:

console.log('hi' + 'ya'); // Prints 'hiya'  
console.log('wo' + 'ah'); // Prints 'woah'  
console.log('I love to ' + 'code.')  
// Prints 'I love to code.'

This process of appending one string to another is called *concatenation*. Notice in the third example we had to make sure to include a space at the end of the first string. The computer will join the strings exactly, so we needed to make sure to include the space we wanted between the two strings.

**\*\*A note about +**

When using + with only integers, it will add the values together. *However,* if using at least one String value with any number of integers and the + operator, JavaScript will automatically convert everything to a String.

1 + 1 = 2 // add two integers, will do addition

“$” + 3 + 4 = $34 //will concatenate and convert everything to strings

+ can also be used an operand that can convert strings to numbers: +”42” = 42

Also: Number(‘42’) = 42

Also: parseInt(“42”, 10) = 42

+”3” + (+”4”) = 7

**Properties**

When you introduce a new piece of data into a JavaScript program, the browser saves it as an instance of the data type. Every string instance has a property called length that stores the number of characters in that string. You can retrieve property information by appending the string with a period and the property name:

console.log('Hello'.length); // Prints 5

The . is another operator! We call it the *dot operator*.

**Methods**

Methods are actions we can perform. We *call*, or use, these methods by appending an instance with:

* a period (the dot operator)
* the name of the method
* opening and closing parentheses

E.g. 'example string'.methodName().

console.log('hello'.toUpperCase()); // Prints 'HELLO'  
console.log('Hey'.startsWith('H')); // Prints true

**Built-in Objects**

In addition to console, there are other [objects built into JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects).

For example, if you wanted to perform more complex mathematical operations than arithmetic, JavaScript has the built-in Math object.

The great thing about objects is that they have methods! Let’s call the .random() method from the built-in Math object:

console.log(Math.random()); // Prints a random number between 0 and 1

This method returns a random number between 0 (inclusive) and 1 (exclusive).

To generate a random number between 0 and 50, we could multiply this result by 50, like so:

Math.random() \* 50;

The example above will likely evaluate to a decimal. To ensure the answer is a whole number, we can take advantage of another useful Math method called Math.floor().

Math.floor() takes a decimal number, and rounds down to the nearest whole number. You can use Math.floor() to round down a random number like this:

Math.floor(Math.random() \* 50);

In this case:

1. Math.random generates a random number between 0 and 1.
2. We then multiply that number by 50, so now we have a number between 0 and 50.
3. Then, Math.floor() rounds the number down to the nearest whole number.

If you wanted to see the number printed to the terminal, you would still need to use a console.log() statement.

Math.round(); rounds to the nearest whole number.

**VARIABLES**

*Create a Variable: var*

There are a few general rules for naming variables:

* Variable names cannot start with numbers.
* Variable names are case sensitive, so myName and myname would be different variables. It is bad practice to create two variables that have the same name using different cases.
* Variable names cannot be the same as keywords. For a comprehensive list of keywords check out [MDN’s keyword documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Keywords).

*Create a Variable: let*

* As mentioned in the previous exercise, the let keyword was introduced in ES6. The let keyword signals that the variable can be reassigned a different value.

Another concept that we should be aware of when using let (and even var) is that we can declare a variable without assigning the variable a value. In such a case, the variable will be automatically initialized with a value of undefined.

*Create a Variable: const*

The const keyword was also introduced in ES6, and is short for the word constant.  A const variable cannot be reassigned because it is constant. If you try to reassign a const variable, you’ll get a TypeError.

Constant variables must be assigned a value when declared. If you try to declare a const variable without a value, you’ll get a SyntaxError.

*String Interpolation*

In the ES6 version of JavaScript, we can insert, or *interpolate*, variables into strings using *template literals*. Check out the following example where a template literal is used to log strings together:

const myPet = 'armadillo';  
console.log(`I own a pet ${myPet}.`);  
// Output: I own a pet armadillo.

Notice that:

* a template literal is wrapped by backticks ` (this key is usually located on the top of your keyboard, left of the 1 key).
* Inside the template literal, you’ll see a placeholder, ${myPet}. The value of myPet is inserted into the template literal.
* When we interpolate `I own a pet ${myPet}.`, the output we print is the string: 'I own a pet armadillo.'

One of the biggest benefits to using template literals is the readability of the code. Using template literals, you can more easily tell what the new string will be. You also don’t have to worry about escaping double quotes or single quotes.

*typeof operator*

While writing code, it can be useful to keep track of the data types of the variables in your program. If you need to check the data type of a variable’s value, you can use the typeof operator.

The typeof operator checks the value to its right and *returns*, or passes back, a string of the data type.

const unknown1 = 'foo';  
console.log(typeof unknown1); // Output: string  
   
const unknown2 = 10;  
console.log(typeof unknown2); // Output: number  
   
const unknown3 = true;   
console.log(typeof unknown3); // Output: boolean

Let’s break down the first example. Since the value unknown1 is 'foo', a string, typeof unknown1 will return 'string'.

**CONDITIONAL STATEMENTS**

*If...Else Statements*

if...else statements allow us to automate solutions to yes-or-no questions, also known as binary decisions.

*Logical Operators*

In JavaScript, there are operators that work with boolean values known as *logical operators*. We can use logical operators to add more sophisticated logic to our conditionals. There are three logical operators:

* the logical *and* operator (&&) also known as the guard operator
* the *or* operator (||)
* the *not* operator, otherwise known as the *bang* operator (!)

*Truthy and Falsy*

Let’s consider how non-boolean data types, like strings or numbers, are evaluated when checked inside a condition.

Sometimes, you’ll want to check if a variable exists and you won’t necessarily want it to equal a specific value — you’ll only check to see if the variable has been assigned a value.

Here’s an example:

let myVariable = 'I Exist!';  
   
if (myVariable) {  
   console.log(myVariable)  
} else {  
   console.log('The variable does not exist.')  
}

The code block in the if statement will run because myVariable has a *truthy* value; even though the value of myVariable is not explicitly the value true, when used in a boolean or conditional context, it evaluates to true because it has been assigned a non-falsy value.

So which values are *falsy*— or evaluate to false when checked as a condition? The list of falsy values includes:

* 0
* Empty strings like "" or ''
* null which represent when there is no value at all
* undefined which represent when a declared variable lacks a value
* NaN, or Not a Number

Here’s an example with numbers:

let numberOfApples = 0;  
   
if (numberOfApples){  
   console.log('Let us eat apples!');  
} else {  
   console.log('No apples left!');  
}  
   
// Prints 'No apples left!'

The condition evaluates to false because the value of the numberOfApples is 0. Since 0 is a falsy value, the code block in the else statement will run.

*Truthy and Falsy Assignment*

Truthy and falsy evaluations open a world of short-hand possibilities!

Say you have a website and want to take a user’s username to make a personalized greeting. Sometimes, the user does not have an account, making the username variable falsy. The code below checks if username is defined and assigns a default string if it is not:

let defaultName;  
if (username) {  
  defaultName = username;  
} else {  
  defaultName = 'Stranger';  
}

If you combine your knowledge of logical operators you can use a short-hand for the code above. In a boolean condition, JavaScript assigns the truthy value to a variable if you use the || operator in your assignment:

let defaultName = username || 'Stranger';

Because || or statements check the left-hand condition first, the variable defaultName will be assigned the actual value of username if is truthy, and it will be assigned the value of 'Stranger' if username is falsy. This concept is also referred to as *short-circuit evaluation*.

As soon as a truthy value is found, the remaining options are not checked. Therefore, experienced programmers often:

* Put the code most likely to return true *first* in OR operations, and false answers first in AND operations.
* Place the options requiring the most processing power last, just in case another value returns true and they do not need to be run.

*Ternary Operator*

In the spirit of using short-hand syntax, we can use a *ternary operator* to simplify an if...else statement.

Take a look at the if...else statement example:

let isNightTime = true;  
   
if (isNightTime) {  
  console.log('Turn on the lights!');  
} else {  
  console.log('Turn off the lights!');  
}

We can use a *ternary operator* to perform the same functionality:

isNightTime ? console.log('Turn on the lights!') : console.log('Turn off the lights!');

In the example above:

* The condition, isNightTime, is provided before the ?.
* Two expressions follow the ? and are separated by a colon :.
* If the condition evaluates to true, the first expression executes.
* If the condition evaluates to false, the second expression executes.

Like if...else statements, ternary operators can be used for conditions which evaluate to true or false.

*Unary Operator*

A unary operator returns a result with just one operand. Here you can see an if statement checking for the presence of an element. If the element is found, the result is truthy, so the first set of code is run. If it is not found, the second set is run instead.

if (document.getElementById(‘header’)) {  
  //Found: do something  
} else {  
  //Not found: do something else  
}

Those new to JavaScript often think the following would do the same:

if (document.getElementById(‘header’) == true)

but *document.getElementById(‘header’)* would return an object which is a truthy

value but it is **not** equal to a Boolean value of true.

*Type Coercion & Weak Typing*

If you use a data type JavaScript did not expect, it tries to make sense of the operation rather than report an error.

JavaScript can convert data types behind the scenes to complete an operation. This is known as *type coercion*. For example, a string ‘1’ could be converted to a number 1 in the following expression: (‘1’ > 0). As a result, the above expression would evaluate true.

JavaScript is said to use **weak typing** because the data type for a value can change. Some other languages require that you specify what data type each variable will be. They are said to use **strong typing**.

Type coercion can lead to unexpected values in your code (and also cause errors). Therefore, when checking if two values are equal, it is considered better to use strict equals operators === and !== rather than == and != as these strict operators check that the value and data types match.

\*\*NaN is a value that is counted as a number. You may see it when a number is expected, but is not returned, e.g., (‘ten’/2) results in NaN.

*Review*

Way to go! Here are some of the major concepts for conditionals:

* An if statement checks a condition and will execute a task if that condition evaluates to true.
* if...else statements make binary decisions and execute different code blocks based on a provided condition.
* We can add more conditions using else if statements.
* Comparison operators, including <, >, <=, >=, ===, and !== can compare two values.
* The logical and operator, &&, or “and”, checks if both provided expressions are truthy.
* The logical operator ||, or “or”, checks if either provided expression is truthy.
* The bang operator, !, switches the truthiness and falsiness of a value.
* The ternary operator is shorthand to simplify concise if...else statements.
* A switch statement can be used to simplify the process of writing multiple else if statements. The break keyword stops the remaining cases from being checked and executed in a switch statement.

**FUNCTIONS**

*Function Declarations*

In JavaScript, there are many ways to create a function. One way to create a function is by using a function declaration.

A function declaration consists of:

* The function keyword.
* The name of the function, or its identifier, followed by parentheses.
* A function body, or the block of statements required to perform a specific task, enclosed in the function’s curly brackets, { }.

We should also be aware of the hoisting feature in JavaScript which allows access to function declarations before they’re defined.

Notice how hoisting allowed greetWorld() to be called before the greetWorld() function was defined! Since hoisting isn’t considered good practice, we simply want you to be aware of this feature.

JavaScript only hoists declarations, not initializations. If a variable is declared and initialized after using it, the value will be undefined.

*Parameters and Arguments*

When declaring a function, we can specify its parameters.

When calling a function that has parameters, we specify the values in the parentheses that follow the function name. The values that are passed to the function when it is called are called arguments. Arguments can be passed to the function as values or variables.

*Default Parameters*

One of the features added in ES6 is the ability to use *default parameters*. Default parameters allow parameters to have a predetermined value in case there is no argument passed into the function or if the argument is undefined when called.

Take a look at the code snippet below that uses a default parameter:

function greeting (name = 'stranger') {  
  console.log(`Hello, ${name}!`);  
}  
   
greeting('Nick') // Output: Hello, Nick!  
greeting() // Output: Hello, stranger!

* In the example above, we used the = operator to assign the parameter name a default value of 'stranger'. This is useful to have in case we ever want to include a non-personalized default greeting!
* When the code calls greeting('Nick') the value of the argument is passed in and, 'Nick', will override the default parameter of 'stranger' to log 'Hello, Nick!' to the console.
* When there isn’t an argument passed into greeting(), the default value of 'stranger' is used, and 'Hello, stranger!' is logged to the console.

By using a default parameter, we account for situations when an argument isn’t passed into a function that is expecting an argument.

*Return*

When a function is called, the computer will run through the function’s code and evaluate the result of calling the function. By default that resulting value is undefined.

To pass back information from the function call, we use a return statement. To create a return statement, we use the return keyword followed by the value that we wish to return. Like we saw above, if the value is omitted, undefined is returned instead.

When a return statement is used in a function body, the execution of the function is stopped and the code that follows it will not be executed.

The return keyword is powerful because it allows functions to produce an output. We can then save the output to a variable for later use.

\*\*If there is no expression ( return; ), then the return value is undefined, except for constructors, whose default return value is *this*.

*Helper Functions*

We can also use the return value of a function inside another function. These functions being called within another function are often referred to as *helper functions*. Since each function is carrying out a specific task, it makes our code easier to read and debug if necessary.

We can use functions to section off small bits of logic or tasks, then use them when we need to. Writing helper functions can help take large and difficult tasks and break them into smaller and more manageable tasks.

*Function Expressions*

Another way to define a function is to use a *function expression*. To define a function inside an expression, we can use the function keyword. In a function expression, the function name is usually omitted. A function with no name is called an *anonymous function*. A function expression is often stored in a variable in order to refer to it.

To declare a function expression:

1. Declare a variable to make the variable’s name be the name, or identifier, of your function. Since the release of ES6, it is common practice to use const as the keyword to declare the variable.
2. Assign as that variable’s value an anonymous function created by using the function keyword followed by a set of parentheses with possible parameters. Then a set of curly braces that contain the function body.

To invoke a function expression, write the name of the variable in which the function is stored followed by parentheses enclosing any arguments being passed into the function.

variableName(argument1, argument2)

Unlike function declarations, function expressions are not hoisted so they cannot be called before they are defined.

*Arrow Functions*

ES6 introduced *arrow function syntax*, a shorter way to write functions by using the special “fat arrow” () => notation.

Arrow functions remove the need to type out the keyword function every time you need to create a function. Instead, you first include the parameters inside the ( ) and then add an arrow => that points to the function body surrounded in { } like this:

const rectangleArea = (width, height) => {  
  let area = width \* height;  
  return area;  
};

*Concise Body Arrow Functions*

JavaScript also provides several ways to refactor arrow function syntax. The most condensed form of the function is known as *concise body*. We’ll explore a few of these techniques below:

1. Functions that take only a single parameter do not need that parameter to be enclosed in parentheses. However, if a function takes zero or multiple parameters, parentheses are required.

**Zero parameters**

const functionName = () => {};

**One parameter**

const functionName = paramOne => {};

**Two or more parameters**

const functionName = (paramOne, paramTwo) => {};

1. A function body composed of a single-line block does not need curly braces. Without the curly braces, whatever that line evaluates will be automatically returned. The contents of the block should immediately follow the arrow => and the return keyword can be removed. This is referred to as implicit return.

**Single-line Block**

const sumNumbers = number => number + number;

**Multi-line Block**

const sumNumbers = number => {

const sum = number + number;

return sum; **--RETURN STATEMENT**

};

So if we have a function:

const squareNum = (num) => {  
  return num \* num;  
};

We can refactor the function to:

const squareNum = num => num \* num;

Notice the following changes:

* The parentheses around num have been removed, since it has a single parameter.
* The curly braces { } have been removed since the function consists of a single-line block.
* The return keyword has been removed since the function consists of a single-line block.

*Immediately Invoked Function Expression (IIFE)*

Pronounced “iffy”, these functions are not given a name. Instead, they are executed once as the interpreter comes across them.

Below, the variable called area will hold the value returned from the function (rather than storing the function itself so that it can be called later).

var area = (function() {

var width = 3;

var height = 2;

return width \* height;

} ());

The ***final parentheses*** (shown in purple) after the closing curly brace of the code block tell the interpreter to call the function immediately.

The ***grouping operators*** (shown on blue) are parentheses there to ensure the interpreter treats this as an expression.

\*\*You may see the final parentheses in an IIFE placed *after* the closing grouping operator but it is commonly considered better practice to place the final parentheses *before* the closing grouping operator, as shown in the code above.

*When To Use Anonymous Functions And IIFEs*

They are used for code that only needs to run once within a task, rather than repeatedly being called by other parts of the script. For example:

* As an argument when a function is called (to calculate a value for that function).
* To assign the value of a property to an object.
* In event handlers and listeners to perform a task when an event occurs.
* To prevent conflicts between two scripts that might use the same variable names.

IIFEs are commonly used as a wrapper around a set of code. Any variables declared within that anonymous function are effectively protected from variables in other scripts that might have the same name.

**SCOPE**

*Review: Scope*

Let’s review the following terms:

* **Scope** is the idea in programming that some variables are accessible/inaccessible from other parts of the program.
* **Blocks** are statements that exist within curly braces {}.
* **Global scope** refers to the context within which variables are accessible to every part of the program.
* **Global variables** are variables that exist within global scope.
* **Block scope** refers to the context within which variables that are accessible only within the block they are defined.
* **Local variables** are variables that exist within block scope.
* **Global namespace** is the space in our code that contains globally scoped information.
* **Scope pollution** is when too many variables exist in a namespace or variable names are reused.
* An inner function has access to the variables and parameters of functions that it is contained within.
* This is known as **Static Scoping** or **Lexical Scoping**

**REVIEW: JavaScript Part I**

*Douglas Crockford: JavaScript Programming Language notes*

Key Ideas

* Load and go delivery
* Loose typing
* Objects as general containers
* Prototypal inheritance - powerful
* Lambda
* Linkage through global variables – bad idea

Numbers

* Only one number type (no integers)
* 64-bit floating point
* IEEE-754(aka “Double”)
* Does not map well to common understanding of arithmetic:
* 0.1 + 0.2 = 0.3000000000000…4

\*\*A good practice when dealing with cents is to multiply your value times 100, do your arithmetic, and then scale it back down. This is a quirk of JavaScript not being able to process floating point numbers with high precision.

NaN

* Special number: Not a Number
* Result of undefined or erroneous operations ( a number divided by zero)
* Toxic: any arithmetic operation with NaN as an input will have NaN as a result.
* NaN is not equal to anything, including NaN. ( NaN === NaN will return false) also when asked for a typeOf it states that it is a number. NaN is neither greater than nor lesser than when compared to another NaN.

Number function

Number(value)

* Converts the value into a number. (input field like text)
* It produces NaN if it has a problem.
* Similar to + prefix operator.

parseInt Function

parseInt(value, 10)

* Converts the value into a number.
* It stops at the first non-digit character.
* The radix(10) should be required.

parseInt(“08”) === 0

parseInt(“08”, 10) === 8

If you don’t use it, it will stop after 0 thinking the number is an octal number, and the next number is non-octal. Suggested to always use a radix.

Math

* Math object is modeled on Java’s Math class.
* It contains:

abs absolute value

*floor integer*

log logarithm

max maximum

pow raise to a power

random random number

round nearest integer

sin sine

sqrt square root

Strings

* Sequence of 0 or more 16-bit characters

UCS-2, not quite UTF-16; JavaScript was created before Unicode got to that level of maturity. Generally not a problem since the extra characters are rarely used.

No awareness of surrogate pairs

* No separate character type; like in Java (char)

Characters are represented as strings with a length of 1

* Strings are immutable
* Similar strings are equal(==)
* String literals can use single or double quotes
* String.*length*
* The *length* property determines the number of 16-bit characters in a string.

String function

String(value)

* Converts value to a string

String Methods

|  |  |  |  |
| --- | --- | --- | --- |
| * charAt | * concat | * indexOf | * lastIndexOf |
| * match | * replace | * search | * slice |
| * split | * substring | * toLowerCase | * toUpperCase |

Boolean and Boolean function

Only has two values: true or false

Boolean(value)

* returns *true* if value is truthy
* returns *false* if value is falsy
* Similar to !! prefix operator

Null and Undefined

null

* A value that isn’t anything

undefined

* A value that isn’t even that
* The default value for variables and parameters
* The value of missing members in objects

Dynamic Objects

* Unification of Object and Hashtable
* *new Object()* produces an empty container of name/value pairs
* A name can be any string, a value can be any value except *undefined*
* members can be accessed with dot notation or subscript notation
* No hash nature is visible (no hash codes or rehash methods)

Loosely Typed

* Any of these types can be stored in a variable, or passed as a parameter to any function
* The language is not “untyped”. It has types, but they are loosely used.

C

* JavaScript is syntactically a C family language
* It differs from C mainly in its type system, which allows functions to be values

Identifiers

* Starts with a letter or \_ or $
* Followed by zero or more letters, digits, \_ or $
* By convention, all variables, parameters, members, and function names start with lower case
  + Except for constructors which start with upper case
* Initial \_ should be reserved for implementations
* $ should be reserved for machines: program generator and macroprocessors

Bitwise

& | ^ >> >>> <<

* The bitwise operators convert the operand to a 32-bit signed integer, and turn the result back into 64-bit floating point.

Break Statement

* Statements can have labels.
* Break statements can refer to those labels.

*loop:* for(;;) {

*…*

if(…){

**break** *loop;*

}

…

}

The break statement will break out of the specified label completely.

For statement

* Iterate through all of the members of an object:

for ( var *name* in *object* ) {

if ( *object*.hasOwnProperty( *name* )) {

//within the loop

//*name* is the key of current member

// *object[name]* is the current value

}

}

\*\*To avoid going through all the keys that the name of the object inherits from, the if statement is more specific.

Try statement

* The JavaScript implementation can produce these exception names:
  + ‘Error’
  + ‘EvalError’
  + ‘RangeError’
  + ‘SyntaxError’
  + ‘TypeError’
  + ‘URIError’

With statement

* Intended as a short-hand
  + with (o) {

foo = null;

}

* Ambiguous and Error-prone
* Don’t use it.

\*\*Not specific enough to use

Objects

* Everything else is objects
* Objects can contain data and methods
* Objects can inherit from other objects

Collections

* An object is an unordered collection of name/value pairs
* Names are strings
* Values are any type, including other objects
* Good for representing records and trees
* Every object is a little database

Object Literals

* Object literals are wrapped in { }
* Names can be names or strings
* Values can be expressions
* : separates names and values
* , separates pairs
* Object literals can be used anywhere a value can appear

var myObject = {name: “Jack B. Nimble”, ‘goto’: ‘Jail’, grade: ‘A’, level: 3};

“name” “Jack B. Nimble”

“goto” “Jail”

“grade” “A”

“level” 3

var theName = myObject.name;

var destination = myObject[‘goto’];

\*\*goto is a reserved word. Look up reserved word usage. Reserved words as strings.

Maker Function

*function maker (name, where, grade, level) {*

*var it = { }; //make a new empty object*

*it.name = name;*

*it[‘goto’] = where;*

*it.grade = grade;*

*it.level = level;*

*return it;*

*}*

*myObject = maker(“Jack B. Nimble”, ‘Jail’, ‘A’, 3);*

Object Literals

*var myObject = {*

*name: “Jack B. Nimble”,*

*‘goto’: ‘Jail’,*

*grade: ‘A’,*

*format: {*

*type: ‘rect’,*

*width: 1920,*

*height: 1080,*

*interlace: false,*

*framerate: 24*

*}*

*};*

\*\*can have nested objects!!

Object Augmentation

* New members can be added to any object by simple assignment
* There is no need to define a new class

myObject.format.colorModel = ‘YCgCb’;

myObject[name] = value;

\*\*you can add to an existing object using a key (name) and assigning it a value.

Linkage

* Objects can be created with a secret link to another object.
* If an attempt to access a name fails, the secret linked object will be used.
* The secret link is not used when storing. New members are only added to the primary object.
* the *object(o)* function makes a new empty object with a link to object o.

\*\*This is how inheritance is achieved.

*var myNewObject = object(myOldObject);*

myNewObject

myOldObject

“name” “Jack B. Nimble

“goto” “Jail”

“grade” “A”

“level” 3

*myNewObject.name =*

*“Tom Piperson”;*

*myNewObject.level += 1;*

*myNewObject.crime =*

*‘pignapping’;*

\*\*because myNewObject doesn’t have a *level* variable to start, it inherits the variable and value from myOldObject and it’s inserted into myNewObject with the updated value.

Inheritance

* Linkage provides simple inheritance.
* An object can inherit from an older object.

Prototypal Inheritance

* Some languages have classes, methods, constructors, and modules. JavaScript’s functions do the work of all of those.
* Instead of Classical Inheritance, JavaScript has Prototypal Inheritance.
* It accomplishes the same things, but differently.
* It offers greater expressive power.
* …but it’s different.
* Instead of organizing objects into rigid classes, new objects can be made that are similar to existing objects, and then customized.
* Object customization is a lot less work than making a class, and less overhead, too.
* One of the keys is the *object(o)* function.
* The other key is functions.

Object Methods

* All objects are linked directly or indirectly to *Object.prototype*
* All objects inherit some basic methods.
* None of them are very useful.
* *hasOwnProperty(name)*
  + Is the name a true member of this object?
* No *copy* method
* No *equals* method

Object Construction

* Make a new object
* All three of these expressions have exactly the same result:
  + *new Object ( )*
  + *{ }*
  + *object(Object.prototype)*
* *{ }* is the preferred form

Reference

* Objects can be passed as arguments to functions, and can be returned by functions
  + Objects are passed by reference
  + Objects are not passed by value
* The === operator compares object references, not values
  + *true* only if both operands are the same object

Delete

* Members can be removed from an object with the *delete* operator

*delete myObject[name];*

Arrays

* *Array* inherits from *Object.*
* Indexes are converted to strings and used as names for retrieving values.
* Very efficient for sparse arrays.
* Not very efficient in most other cases.
* One advantage: No need to provide a length or type when creating an array.

*length*

* Arrays, unlike objects, have a special *length* member.
* It is always 1 larger than the highest integer subscript.
* It allows use of the traditional *for* statement.

*for ( i = 0; i < a.length; i += 1) {*

*…*

*}*

* Do not use *for…in* with arrays

Array Literals

* An array literal uses [ ]
* It can contain any number of expressions, separated by commas
  + *myList = [ ‘oats’, ‘peas’, ‘beans’];*
* New items can be appended
  + *myList [ myList.length ] = ‘barley’;*
* The dot notation should not be used with arrays.
* [ ] is preferred to *new Array( ).*

Array Methods

|  |  |  |  |
| --- | --- | --- | --- |
| * concat | * join | * pop | * push |
| * slice | * sort | * splice |  |

Deleting Elements

* Removes the element, but leaves a hole in the numbering.
  + *delete array [ number ]*
* Removes the element and renumbers all the following elements.
  + *array.splice( number, 1)*

*myArray = [ ‘a’, ‘b’, ‘c’, ‘d’ ];*

*delete myArray [ 1 ];*

*// [ ‘a’, undefined , ‘c’ , ‘d’ ]*

*myArray.splice ( 1, 1 );*

*// [ ‘a’ , ‘c’ , ‘d’ ]*

Arrays vs Objects

* Use objects when the names are arbitrary strings.
* Use arrays when the names are sequential integers.
* Don’t get confused by the term Associative Array.

Distinguishing Arrays

*value. constructor === Array*

*value instanceof Array*

Neither of these work when the value comes from a different frame.

\*\*Can not inherit from an array

Arrays and Inheritance

* Don’t use arrays as prototypes.
  + The object produced this way does not have array nature. It will inherit the array’s values and methods, but not its length.
* You can augment an individual array.
  + Assign a method to it.
  + This works because arrays are objects.
* You can augment all arrays.
  + Assign methods to *Array.prototype*

Functions

* Functions are first-class objects
* Functions can be passed, returned, and stored just like any other value.
* Functions inherit from *Object* and can store name/value pairs.

*lambda*

* What JavaScript calls *function,* other languages call *lambda.*
* It is a source of enormous expressive power.
* Unlike most power-constructs, it is secure.

Closure

* The scope that an inner function enjoys continues even after the parent functions have returned; this is called **closure**

example

*function fade ( id ) {*

*var dom = document.getElementById( id ),*

*level = 1;*

*function step ( ) {*

*var h = level.toString( 16 );*

*dom.style.backgroundColor = ‘#FFFF’ + h + h;*

*if ( level < 15 ) {*

*level += 1;*

*setTimeout(step, 100);*

*}*

*}*

*setTimeout(step, 100);*

*}*

Method

* Since functions are values, functions can be stored in objects.
* A function in an object is called a *method*

Invocation

* If a function is called with too many arguments, the extra arguments are ignored.
* If a function is called with too few arguments, the missing values will be *undefined*.
* There is no implicit type checking on the arguments.
* There are four ways to call a function:
  + Function form
    - *functionObject ( arguments )*
      * When a function is called in the function form, *this* is set to the global object.
        + That is not very useful. It makes it harder to write helper functions within a method because the helper function does not get access to the outer *this*.
        + *var that = this; //can implement a variable within the inner function to get around the problem*
  + Method form
    - *thisObject.methodName( arguments )*
      * *When a function is called in the method form, this* is set to *thisObject*, the object containing the function.
      * This allows methods to have a reference to the object of interest.
    - *thisObject [ “methodName” ] ( arguments )*
  + Constructor form
    - *new functionObject ( arguments )*
      * When a function is called with the *new* operator, a new object is created and assigned to *this.*
      * If there is not an explicit return value, then *this* will be returned.
  + Apply form
    - *functionObject.apply( thisObject , [ arguments ] )*

|  |  |
| --- | --- |
| **Invocation**  **Form** | **this** |
| Function | the global object |
| Method | the object |
| Constructor | the new object |

*this*

* *this* is an extra parameter. Its value depends on the calling form.
* *this* gives methods access to their objects.
* *this* is bound at invocation time.

*arguments*

* When a function is invoked, in addition to its parameters, it also gets a special parameter called arguments.
* It contains all of the arguments from the invocation.
* It is an array-like object.
* *arguments.length* is the number of arguments passed.

Example

*function sum ( ) {*

*var i,*

*n = arguments.length,*

*total = 0;*

*for ( i = 0 ; i < n ; i += 1 ) {*

*total += arguments[ i ];*

*}*

*return total;*

*}*

Augmenting Built-in Types

* Object.prototype
* Array.prototype
* Function.prototype
* Number.prototype
* String.prototype
* Boolean.prototype

Example

***trim***

*//does not exist, so we write our own*

*String.prototype.trim = function ( ) {*

*return this.replace (* / ^ \ s \* ( \ S \* ( \ s + \ S + ) \* ) \ s \* $ / , “ $ 1 “ );

*};*

\*\*This is very powerful because you can essentially make your own program. #IDoWhatIWant

* We can directly modify individual objects to give them just the characteristics we want.
* We can do this without having to create classes.
* We can then use our new object as the prototype for lots of new objects, each of which can also be augmented.

**eval**

*eval ( string )*

* The *eval* function compiles and executes a string and returns the result.
* It is what the browser uses to convert strings into actions.
* It is the most misused feature of the language. Waste of time. Security issues.
* ….don’t use it, except when using with JSON. Have zero doubts about your server.

Function function

*new Function ( parameters, body )*

* The *Function* constructor takes zero or more parameter name strings, and a body string, and uses the JavaScript compiler to produce a function object.
* It should only be used to compile fresh source from a server.
* It is closely related to *eval*.
* …also should not be used

Built-in Type Wrappers

* Java has *int*  and *Integer*, two incompatible types which can both carry the same value with differing levels of efficiency and convenience.
* JavaScript copied this pattern to no advantage. **Avoid it.**
* **Avoid** *new Boolean ( ) new String ( ) new Number ( )*

**Confession???**

*function object ( o ) {*

*function F ( ) { }*

*F.prototype = o;*

*return new F ( );*

*}*

Working with the Grain

* Classical patterns are less effective than prototypal patterns or parasitic patterns.
* Formal classes are not needed for reuse or extension.

( global ) Object

* The object that dares not speak its name.
* It is the container for all global variables and all built-in objects.
* Sometimes *this* points to it.
  + *var global = this;*
* On browsers, *window* is the global object.

Global variables are evil

* Functions within an application can clobber each other.
* Cooperating applications can clobber each other.
* Use of the global namespace must be minimized.

Implied Global

* Any var which is not properly declared is assumed to be global by default.
* This makes it easy for people who do not know or care about encapsulation to be productive, but it makes applications less reliable.
* JSLint is a tool which helps identify implied globals and other weaknesses.
  + <http://www.JSLint.com>

Namespace

* Every object is a separate namespace.
* Use an object to organize your variables and functions.
* The *Yahoo* Object.
  + *<head>*

*<script>*

*YAHOO = { } ; //use all uppercase names*

*</script>*

Encapsulate

* Function scope can create an encapsulation.
* Use an anonymous function to wrap your application.

Thinking about type

* Trading type-safety for dynamism.
* JavaScript has no cast operator.
* Reflection is really easy, and usually unnecessary.
* Why inheritance?
  + Automatic casting
  + Code reuse
* Trading brittleness for flexibility

Date

* The *Date* function is based on Java’s Date class.
* It was not Y2K ready.

RegExp

* Regular expression pattern matcher
* Patterns are enclosed in slashes. Can be problematic since the / is used for comments and division.
* Example: a pattern that matches regular expressions

/\/(\\[^\x00-\xlf]|\[)\\[^\x00-\xlf]|[^\x00-\xlf\\\/])\*\]|\x00-\xlf\\\/\[])+\/[gim]\*/

* Bizarre notation, difficult to read.

Threads

* The language definition is neutral on threads
* Some language processors ( like SpiderMonkey ) provide thread support
* Most application environments ( like browsers ) do not provide it
* Threads are evil

Platforms

Browsers WSH and Dashboard Yahoo!Widgets

DreamWeaver and Photoshop Embedded

ActionScript

A JavaScript-like language

* Empty strings are truthy
* keywords are case insensitive
* No Unicode support
* No *RegExp*
* No *try*
* No statement labels
* || and && return Booleans
* separate operators for strings and numbers

E4X

* Extensions to ECMAScript for XML
* Proposed by BEA
* Allows < XML > literals
* Not compatible with ECMAScript Third Edition
* Not widely accepted yet
* Not in IE7

Style

* Programming style isn’t about personal taste.
* It is about rigor in expression.
* It is about clearness in presentation.
* It is about product adaptability and longevity.
* Good rules help us to keep the quality of our programs high.
* <http://javascript.crockford.com/code.html>
* Use semicolons!!!
* Line Ending
  + Break a line after a punctuator:
    - , . ; : { } ( [ = < > === || &&
  + Do not break after a name, string, number, or ) ] ++ - -
  + Defense against copy/paste errors.
* Avoid tricky expressions using the comma operators.
* Do not use extra commas in array literals
  + Good: [ 1, 2, 3 ]
  + Bad: [ 1, 2, 3, ] //different browsers interpret this differently; one might say it has three values, and the other might say it has four due to the comma.
* Use blocks around structured statements to avoid confusion.
  + Good:

*if (a) {*

*b();*

*}*

* + Bad: //easily corrupted

*if (a) b();*

* Forbidden Blocks
  + Blocks do not have scope in JavaScript.
  + Blocks should only be used with structured statements like, *functions, if, switch, while, for, do , try*
* Define all variables at the beginning of the function.
* JavaScript does not have block scope, so there is no advantage in declaring variables at the place of their first use.
* Any expression can be used as a statement. That can mask errors!!
* Only assignment expressions and invocation expressions should be used as statements.
  + Good:
    - *foo( );*
  + Bad:
    - *foo && foo( );*
* Avoid using fallthrough with *switch* statements
* Each clause should explicitly *break* or *return* or *throw*
* Do not use assignment expressions in the condition parts of *if, while,* or *for.*
* It is more likely that

*if ( a = b ) {…}*

* was intended to be

*if ( a == b ) {…}*

* Avoid tricky expressions.
* Be aware that == and != do type coercion . Use strict equals instead.
* Only use labels on these statements: *do, for, switch, while*
* Never use *javascript:* as a label.
* <http://www.JSLint.com> use in Eclipse….it will hurt your feelings. lol

**Arrays**

*Create an Array*

One way we can create an array is to use an array literal. An array literal creates an array by wrapping items in square brackets [].  Arrays can store any data type — we can have an array that holds all the same data types or an array that holds different data types.

*Accessing Elements*

Each element in an array has a numbered position known as its index. We can access individual items using their index, which is similar to referencing an item in a list based on the item’s position.

*const places = [ ‘Germany’, ‘Alabama’, ‘Ukraine’];*

*console.log(places[0]);*

*output // Germany*

You can also access individual characters in a string using bracket notation and the index.

*const hello = “Hello World”;*

*console.log(hello[6]);*

*output // W*

\*\*if you try to access an item beyond the last index of an array, you will get *undefined* as the output.

*Update Elements*

To update an item in an array, access the index and set it equal to the new value.

*const seasons = [‘Winter’, ‘Spring’, ‘Summer’, ‘Fall’];*

*seasons[3]= ‘Autumn’;*

*//output: [‘Winter’, ‘Spring’, ‘Summer’, ‘Autumn’];*

*Arrays with* ***let*** *and* ***const***

Variables declared with the const keyword cannot be reassigned. However, elements in an array declared with const remain mutable. Meaning that we can change the contents of a const array, but cannot reassign a new array or a different value.

*Array Methods*

.length

Use the *.length* property to return the number of items in an array.

*let chores = [‘dishes’, ‘laundry’, ‘cat pans’];*

*console.log(chores.length);*

*// output: 3*

.push( )

Add items to the end of an array. You might also see .push() referred to as a destructive array method since it changes the initial array.

*const itemTracker = ['item 0', 'item 1', 'item 2'];  
itemTracker.push('item 3', 'item 4');  
console.log(itemTracker);   
// Output: ['item 0', 'item 1', 'item 2', 'item 3', 'item 4'];*

.pop( )

Removes the last item of an array. It does not take any arguments, it simply removes the last element. Mutating\*\*

const newItemTracker = ['item 0', 'item 1', 'item 2'];  
const removed = newItemTracker.pop();  
console.log(newItemTracker);   
// Output: [ 'item 0', 'item 1' ]  
console.log(removed);  
// Output: item 2

*More Array Methods*

.pop() and .push() mutate the array on which they’re called. However, there are times that we don’t want to mutate the original array and we can use non-mutating array methods.

Some arrays methods that are available to JavaScript developers include: .join(), .slice(), .splice(), .shift(), .unshift(), and .concat() amongst many others.

*Nested Arrays*

When an array contains another array it is known as a *nested array*. You still access each element using bracket notation and the index, but to access items that are nested you can *chain* more bracket notation with index values.

*const nestedArr = [ [ 1 ], [ 2, 3, ] ];*

*console.log(nestedArr[ 1 ][ 0 ]); // output: 2*

*Array Constructor vs Array Literal*

Array Literal uses bracket notation to initiate variables and uses bracket notation to access each item.

*let colors = [ ‘white’, ‘black’, ‘custom’];*

*colors[0] // output: white*

Array Constructors use parentheses and are initiated using the word *new*. Use parentheses to access the items within an array constructor.

*let colors = new Array (‘white’, ‘black’, ‘custom’);*

*colors.item(0); //output: white*

*Review Arrays*

* Arrays are lists that store data in JavaScript.
* Arrays are created with brackets [].
* Each item inside of an array is at a numbered position, or index, starting at 0.
* We can access one item in an array using its index, with syntax like: myArray[0].
* We can also change an item in an array using its index, with syntax like myArray[0] = 'new string';
* Arrays have a length property, which allows you to see how many items are in an array.
* Arrays have their own methods, including .push() and .pop(), which add and remove items from an array, respectively.
* Arrays have many methods that perform different tasks, such as .slice() and .shift(), you can find documentation at the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) website.
* Some built-in methods are mutating, meaning the method will change the array, while others are not mutating. You can always check the documentation.
* Variables that contain arrays can be declared with let or const. Even when declared with const, arrays are still mutable. However, a variable declared with const cannot be reassigned.
* Arrays mutated inside of a function will keep that change even outside the function.
* Arrays can be nested inside other arrays.
* To access elements in nested arrays chain indices using bracket notation.

**Review Loops**

* Loops perform repetitive actions so we don’t have to code that process manually every time.
* How to write for loops with an iterator variable that increments or decrements
* How to use a for loop to iterate through an array
  + for(let i = 0; i < condition; i++){…}
* A nested for loop is a loop inside another loop
* while loops allow for different types of stopping conditions
  + let i = 0;
  + while(condition){ … }
* Stopping conditions are crucial for avoiding infinite loops.
* do...while loops run code at least once— only checking the stopping condition after the first execution
  + do { something; } while ( condition );
* The break keyword allows programs to leave a loop during the execution of its block

**Objects**

*Creating Object Literals*

Objects can be assigned to variables just like any JavaScript type. We use curly braces, {}, to designate an *object literal*:

let spaceship = {}; // spaceship is an empty object

We fill an object with unordered data. This data is organized into *key-value pairs*. A key is like a variable name that points to a location in memory that holds a value.

A key’s value can be of any data type in the language including functions or other objects.

We make a key-value pair by writing the key’s name, or *identifier*, followed by a colon and then the value. We separate each key-value pair in an object literal with a comma (,). Keys are strings, but when we have a key that does not have any special characters in it, JavaScript allows us to omit the quotation marks:

// An object literal with two key-value pairs  
let spaceship = {  
  'Fuel Type': 'diesel',  
  color: 'silver'  
};

The spaceship object has two properties Fuel Type and color. 'Fuel Type' has quotation marks because it contains a space character.

*Accessing Properties*

With property dot notation, we write the object’s name, followed by the dot operator and then the property name (key):

let spaceship = {  
  homePlanet: 'Earth',  
  color: 'silver'  
};  
spaceship.homePlanet; // Returns 'Earth',  
spaceship.color; // Returns 'silver',

If we try to access a property that does not exist on that object, undefined will be returned.

spaceship.favoriteIcecream; // Returns undefined

The second way to access a key’s value is by using bracket notation, [ ].

You’ve used bracket notation when indexing an array:

['A', 'B', 'C'][0]; // Returns 'A'

To use bracket notation to access an object’s property, we pass in the property name (key) as a string.

We **must** use bracket notation when accessing keys that have numbers, spaces, or special characters in them. Without bracket notation in these situations, our code would throw an error.

let spaceship = {  
  'Fuel Type': 'Turbo Fuel',  
  'Active Duty': true,  
  homePlanet: 'Earth',  
  numCrew: 5  
};  
spaceship['Active Duty'];   // Returns true  
spaceship['Fuel Type'];   // Returns  'Turbo Fuel'  
spaceship['numCrew'];   // Returns 5  
spaceship['!!!!!!!!!!!!!!!'];   // Returns undefined

With bracket notation you can also use a variable inside the brackets to select the keys of an object. This can be especially helpful when working with functions:

let returnAnyProp = (objectName, propName) => objectName[propName];  
   
returnAnyProp(spaceship, 'homePlanet'); // Returns 'Earth'

If we tried to write our returnAnyProp() function with dot notation (objectName.propName) the computer would look for a key of 'propName' on our object and not the value of the propName parameter.

*Property Assignment*

Once we’ve defined an object, we’re not stuck with all the properties we wrote. Objects are *mutable* meaning we can update them after we create them!

We can use either dot notation, ., or bracket notation, [], and the assignment operator, = to add new key-value pairs to an object or change an existing property.

One of two things can happen with property assignment:

* If the property already exists on the object, whatever value it held before will be replaced with the newly assigned value.
* If there was no property with that name, a new property will be added to the object.

It’s important to know that although we can’t reassign an object declared with const, we can still mutate it, meaning we can add new properties and change the properties that are there.

const spaceship = {type: 'shuttle'};  
spaceship = {type: 'alien'}; // TypeError: Assignment to constant variable.  
spaceship.type = 'alien'; // Changes the value of the type property  
spaceship.speed = 'Mach 5'; // Creates a new key of 'speed' with a value of 'Mach 5'

You can delete a property from an object with the delete operator.

const spaceship = {  
  'Fuel Type': 'Turbo Fuel',  
  homePlanet: 'Earth',  
  mission: 'Explore the universe'   
};  
   
delete spaceship.mission;  // Removes the mission property

*Methods*

When the data stored on an object is a function we call that a *method*. A property is what an object has, while a method is what an object does.

We can include methods in our object literals by creating ordinary, comma-separated key-value pairs. The key serves as our method’s name, while the value is an anonymous function expression.

const alienShip = {  
  invade: function () {   
    console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')  
  }  
};

With the new method syntax introduced in ES6 we can omit the colon and the function keyword.

const alienShip = {  
  invade () {   
    console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')  
  }  
};

Object methods are invoked by appending the object’s name with the dot operator followed by the method name and parentheses:

alienShip.invade(); // Prints 'Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.'

*Nested Objects*

In application code, objects are often nested— an object might have another object as a property which in turn could have a property that’s an array of even more objects!

const spaceship = {  
     telescope: {  
        yearBuilt: 2018,  
        model: '91031-XLT',  
        focalLength: 2032   
     },  
    crew: {  
        captain: {   
            name: 'Sandra',   
            degree: 'Computer Engineering',   
            encourageTeam() { console.log('We got this!') }   
         }  
    },  
    engine: {  
        model: 'Nimbus2000'  
     },  
     nanoelectronics: {  
         computer: {  
            terabytes: 100,  
            monitors: 'HD'  
         },  
        'back-up': {  
           battery: 'Lithium',  
           terabytes: 50  
         }  
    }  
};

We can chain operators to access nested properties. We’ll have to pay attention to which operator makes sense to use in each layer. It can be helpful to pretend you are the computer and evaluate each expression from left to right so that each operation starts to feel a little more manageable.

spaceship.nanoelectronics['back-up'].battery; // Returns 'Lithium'

*Add Objects to An Array within an object*

spaceship.passengers = [

  {psngr1: 'Lily', age: 35},

  {psngr2: 'James', age: 36}

  ];

  let firstPassenger = spaceship.passengers[0];

*Pass By Reference*

Objects are *passed by reference*. This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a const variable).

const spaceship = {  
  homePlanet : 'Earth',  
  color : 'silver'  
};  
   
let paintIt = obj => {  
  obj.color = 'glorious gold'  
};  
   
paintIt(spaceship);  
   
spaceship.color // Returns 'glorious gold'

Our function paintIt() permanently changed the color of our spaceship object. However, reassignment of the spaceship variable wouldn’t work in the same way:

let spaceship = {  
  homePlanet : 'Earth',  
  color : 'red'  
};  
let tryReassignment = obj => {  
  obj = {  
    identified : false,   
    'transport type' : 'flying'  
  }  
  console.log(obj) // Prints {'identified': false, 'transport type': 'flying'}  
   
};  
tryReassignment(spaceship) // The attempt at reassignment does not work.  
spaceship // Still returns {homePlanet : 'Earth', color : 'red'};  
   
spaceship = {  
  identified : false,   
  'transport type': 'flying'  
}; // Regular reassignment still works.

Let’s look at what happened in the code example:

* We declared this spaceship object with let. This allowed us to reassign it to a new object with identified and 'transport type' properties with no problems.
* When we tried the same thing using a function designed to reassign the object passed into it, the reassignment didn’t stick (even though calling console.log() on the object produced the expected result).
* When we passed spaceship into that function, obj became a reference to the memory location of the spaceship object, but *not* to the spaceship variable. This is because the obj parameter of the tryReassignment() function is a variable in its own right. The body of tryReassignment() has no knowledge of the spaceship variable at all!
* When we did the reassignment in the body of tryReassignment(), the obj variable came to refer to the memory location of the object {'identified' : false, 'transport type' : 'flying'}, while the spaceship variable was completely unchanged from its earlier value.

*Looping Through Objects*

[JavaScript has given us alternative solution for iterating through objects with the for...in syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in).

for...in will execute a given block of code for each property in an object.

let spaceship = {  
  crew: {  
    captain: {   
      name: 'Lily',   
      degree: 'Computer Engineering',   
      cheerTeam() { console.log('You got this!') }   
    },  
    'chief officer': {   
      name: 'Dan',   
      degree: 'Aerospace Engineering',   
      agree() { console.log('I agree, captain!') }   
    },  
    medic: {   
      name: 'Clementine',   
      degree: 'Physics',   
      announce() { console.log(`Jets on!`) } },  
    translator: {  
      name: 'Shauna',   
      degree: 'Conservation Science',   
      powerFuel() { console.log('The tank is full!') }   
    }  
  }  
};   
   
// for...in  
for (let crewMember in spaceship.crew) {  
  console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`);  
}

Our for...in will iterate through each element of the spaceship.crew object. In each iteration, the variable crewMember is set to one of spaceship.crew‘s keys, enabling us to log a list of crew members’ role and name.

*Review*

Way to go! You’re well on your way to understanding the mechanics of objects in JavaScript. By building your own objects, you will have a better understanding of how JavaScript built-in objects work as well. You can also start imagining organizing your code into objects and modeling real world things in code.

Let’s review what we learned in this lesson:

* Objects store collections of *key-value* pairs.
* Each key-value pair is a property—when a property is a function it is known as a method.
* An object literal is composed of comma-separated key-value pairs surrounded by curly braces.
* You can access, add or edit a property within an object by using dot notation or bracket notation.
* We can add methods to our object literals using key-value syntax with anonymous function expressions as values or by using the new ES6 method syntax.
* We can navigate complex, nested objects by chaining operators.
* Objects are mutable—we can change their properties even when they’re declared with const.
* Objects are passed by reference— when we make changes to an object passed into a function, those changes are permanent.
* We can iterate through objects using the For...in syntax.

**Advanced Objects**

*The* ***this*** *Keyword*

When a method ( a function inside of an object ) is made, in order to reference a key inside the object but outside the scope of the method you must use the keyword *this*. That’s because inside the scope of a method, we don’t automatically have access to other properties of an object.

const goat = {  
  dietType: 'herbivore',  
  makeSound() {  
    console.log('baaa');  
  },  
  diet() {  
    console.log(this.dietType);  
  }  
};  
   
goat.diet();   
// Output: herbivore

The this keyword references the calling object which provides access to the calling object’s properties. In the example above, the calling object is goat and by using this we’re accessing the goat object itself, and then the dietType property of goat by using property dot notation.

*Arrow Functions and this*

const goat = {  
  dietType: 'herbivore',  
  makeSound() {  
    console.log('baaa');  
  },  
  diet: () => {  
    console.log(this.dietType);  
  }  
};  
   
goat.diet(); // Prints undefined

In the comment, you can see that goat.diet() would log undefined. So what happened? Notice that in the .diet() is defined using an arrow function.

Arrow functions inherently bind, or tie, an already defined this value to the function itself that is NOT the calling object. In the code snippet above, the value of this is the global object, or an object that exists in the global scope, which doesn’t have a dietType property and therefore returns undefined.

The key takeaway from the example above is to avoid using arrow functions when using this in a method!

*Privacy*

Accessing and updating properties is fundamental in working with objects. However, there are cases in which we don’t want other code simply accessing and updating an object’s properties. When discussing *privacy* in objects, we define it as the idea that only certain properties should be mutable or able to change in value.

Certain languages have privacy built-in for objects, but JavaScript does not have this feature. Rather, JavaScript developers follow naming conventions that signal to other developers how to interact with a property. One common convention is to place an underscore \_ before the name of a property to mean that the property should not be altered. Here’s an example of using \_ to prepend a property.

const bankAccount = {  
  \_amount: 1000  
}

In the example above, the \_amount is not intended to be directly manipulated.

*Getters*

*Getters* are methods that get and return the internal properties of an object. But they can do more than just retrieve the value of a property! Let’s take a look at a getter method:

const person = {  
  \_firstName: 'John',  
  \_lastName: 'Doe',  
  get fullName() {  
    if (this.\_firstName && this.\_lastName){  
      return `${this.\_firstName} ${this.\_lastName}`;  
    } else {  
      return 'Missing a first name or a last name.';  
    }  
  }  
}  
   
// To call the getter method:   
person.fullName; // 'John Doe'

* In the last line we call fullName on person. In general, getter methods do not need to be called with a set of parentheses. Syntactically, it looks like we’re accessing a property.

Now that we’ve gone over syntax, let’s discuss some notable advantages of using getter methods:

* Getters can perform an action on the data when getting a property.
* Getters can return different values using conditionals.
* In a getter, we can access the properties of the calling object using this.
* The functionality of our code is easier for other developers to understand.

Another thing to keep in mind when using getter (and setter) methods is that properties cannot share the same name as the getter/setter function. If we do so, then calling the method will result in an infinite call stack error. One workaround is to add an underscore before the property name like we did in the example above.

*Setters*

Along with getter methods, we can also create *setter* methods which reassign values of existing properties within an object. Let’s see an example of a setter method:

const person = {  
  \_age: 37,  
  set age(newAge){  
    if (typeof newAge === 'number'){  
      this.\_age = newAge;  
    } else {  
      console.log('You must assign a number to age');  
    }  
  }  
};

Then to use the setter method:

person.age = 40;  
console.log(person.\_age); // Logs: 40  
person.age = '40'; // Logs: You must assign a number to age

Setter methods like age do not need to be called with a set of parentheses. Syntactically, it looks like we’re reassigning the value of a property.

Like getter methods, there are similar advantages to using setter methods that include checking input, performing actions on properties, and displaying a clear intention for how the object is supposed to be used. Nonetheless, even with a setter method, it is still possible to directly reassign properties. For example, in the example above, we can still set .\_age directly:

person.\_age = 'forty-five'  
console.log(person.\_age); // Prints forty-five

*Factory Functions*

A factory function is a function that returns an object and can be reused to make multiple object instances. Factory functions can also have parameters allowing us to customize the object that gets returned.

Let’s say we wanted to create an object to represent monsters in JavaScript. There are many different types of monsters and we could go about making each monster individually but we can also use a factory function to make our lives easier. To achieve this diabolical plan of creating multiple monsters objects, we can use a factory function that has parameters:

const monsterFactory = (name, age, energySource, catchPhrase) => {  
  return {   
    name: name,  
    age: age,   
    energySource: energySource,  
    scare() {  
      console.log(catchPhrase);  
    }   
  }  
};

In the monsterFactory function above, it has four parameters and returns an object that has the properties: name, age, energySource, and scare(). To make an object that represents a specific monster like a ghost, we can call monsterFactory with the necessary arguments and assign the return value to a variable:

const ghost = monsterFactory('Ghouly', 251, 'ectoplasm', 'BOO!');  
ghost.scare(); // 'BOO!'

Now we have a ghost object as a result of calling monsterFactory() with the needed arguments. With monsterFactory in place, we don’t have to create an object literal every time we need a new monster. Instead, we can invoke the monsterFactory function with the necessary arguments to make a monster for us!

*Property Value Shorthand*

ES6 introduced some new shortcuts for assigning properties to variables known as *destructuring*.

To remind ourselves, here’s a truncated version of the factory function:

const monsterFactory = (name, age) => {  
  return {   
    name: name,  
    age: age  
  }  
};

 We can use a destructuring technique, called *property value shorthand*, to save ourselves some keystrokes. The example below works exactly like the example above:

const monsterFactory = (name, age) => {  
  return {   
    name,  
    age   
  }  
};

Notice that we don’t have to repeat ourselves for property assignments!

*Destructured Assignment*

We often want to extract key-value pairs from objects and save them as variables. Take for example the following object:

const vampire = {  
  name: 'Dracula',  
  residence: 'Transylvania',  
  preferences: {  
    day: 'stay inside',  
    night: 'satisfy appetite'  
  }  
};

If we wanted to extract the residence property as a variable, we could using the following code:

const residence = vampire.residence;   
console.log(residence); // Prints 'Transylvania'

However, we can also take advantage of a destructuring technique called *destructured assignment* to save ourselves some keystrokes. In destructured assignment we create a variable with the name of an object’s key that is wrapped in curly braces { } and assign to it the object. Take a look at the example below:

const { residence } = vampire;   
console.log(residence); // Prints 'Transylvania'

Look back at the vampire object’s properties in the first code example. Then, in the example above, we declare a new variable residence that extracts the value of the residence property of vampire. When we log the value of residence to the console, 'Transylvania' is printed.

We can even use destructured assignment to grab nested properties of an object:

const { day } = vampire.preferences;   
console.log(day); // Prints 'stay inside'

*Built-in Object Methods*

[**Object.keys()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys)

Returns an array containing the names of all of the given object's **own** enumerable string properties.

[**Object.prototype.hasOwnProperty()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty)

Returns a boolean indicating whether an object contains the specified property as a direct property of that object and not inherited through the prototype chain.

[**Object.assign()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)

Copies the values of all enumerable own properties from one or more source objects to a target object. It returns the target object.

[**Object.entries()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries)

Returns an array containing all of the [key, value] pairs of a given object's **own** enumerable string properties.

[**Object.prototype.valueOf()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/valueOf)

Returns the primitive value of the specified object.

const robot = {

  model: 'SAL-1000',

  mobile: true,

  sentient: false,

  armor: 'Steel-plated',

  energyLevel: 75

};

// What is missing in the following method call?

const robotKeys = Object.keys(robot);

console.log(robotKeys);

// Declare robotEntries below this line:

const robotEntries = Object.entries(robot);

console.log(robotEntries);

// Declare newRobot below this line:

const newRobot = Object.assign({laserBlaster: true, voiceRecognition: true},robot);

console.log(newRobot);

*Review*

Congratulations on finishing Advanced Objects!

Let’s review the concepts covered in this lesson:

* The object that a method belongs to is called the *calling object*.
* The this keyword refers the calling object and can be used to access properties of the calling object.
* Methods do not automatically have access to other internal properties of the calling object.
* The value of this depends on where the this is being accessed from.
* We cannot use arrow functions as methods if we want to access other internal properties.
* JavaScript objects do not have built-in privacy, rather there are conventions to follow to notify other developers about the intent of the code.
* The usage of an underscore before a property name means that the original developer did not intend for that property to be directly changed.
* Setters and getter methods allow for more detailed ways of accessing and assigning properties.
* Factory functions allow us to create object instances quickly and repeatedly.
* There are different ways to use object destructuring: one way is the property value shorthand and another is destructured assignment.
* As with any concept, it is a good skill to learn how to use the documentation with objects!

**Higher-Order Functions**

When we speak to other humans, we share a vocabulary that gives us quick ways to communicate complicated concepts. When we say “bake”, it calls to mind a familiar subroutine— preheating an oven, putting something into an oven for a set amount of time, and finally removing it. This allows us to *abstract* away a lot of the details and communicate key concepts more concisely. Instead of listing all those details, we can say, “We baked a cake,” and still impart all that meaning to you.

In programming, we can accomplish “abstraction” by writing functions. In addition to allowing us to reuse our code, functions help to make clear, readable programs.

*Higher-order functions* are functions that accept other functions as arguments and/or return functions as output. This enables us to build abstractions on other abstractions, just like “We hosted a birthday party” is an abstraction that may build on the abstraction “We made a cake.”

In summary, using more abstraction in our code allows us to write more modular code which is easier to read and debug.

*Functions as Data*

JavaScript functions behave like any other data type in the language; we can assign functions to variables, and we can reassign them to new variables.

Below, we have an annoyingly long function name that hurts the readability of any code in which it’s used. Let’s pretend this function does important work and needs to be called repeatedly!

const announceThatIAmDoingImportantWork = () => {  
    console.log("I’m doing very important work!");  
};

What if we wanted to rename this function without sacrificing the source code? We can re-assign the function to a variable with a suitably short name:

const busy = announceThatIAmDoingImportantWork;  
   
busy(); // This function call barely takes any space!

busy is a variable that holds a *reference* to our original function. If we could look up the address in memory of busy and the address in memory of announceThatIAmDoingImportantWork they would point to the same place. Our new busy() function can be invoked with parentheses as if that was the name we originally gave our function.

Notice how we assign announceThatIAmDoingImportantWork without parentheses as the value to the busy variable. We want to assign the value of the function itself, not the value it returns when invoked.

*Functions as Parameters*

A higher-order function is a function that either accepts functions as parameters, returns a function, or both! We call the functions that get passed in as parameters and invoked callback functions because they get called during the execution of the higher-order function.

When we pass a function in as an argument to another function, we don’t invoke it. Invoking the function would evaluate to the return value of that function call. With callbacks, we pass in the function itself by typing the function name *without* the parentheses (that would evaluate to the result of calling the function):

const timeFuncRuntime = funcParameter => {  
   let t1 = Date.now();  
   funcParameter();  
   let t2 = Date.now();  
   return t2 - t1;  
}  
   
const addOneToOne = () => 1 + 1;  
   
timeFuncRuntime(addOneToOne);

We wrote a higher-order function, timeFuncRuntime(). It takes in a function as an argument, saves a starting time, invokes the callback function, records the time after the function was called, and returns the time the function took to run by subtracting the starting time from the ending time.

We then invoked timeFuncRuntime() first with the addOneToOne() function - note how we passed in addOneToOne and did not invoke it.

timeFuncRuntime(() => {  
  for (let i = 10; i>0; i--){  
    console.log(i);  
  }  
});

In this example, we invoked timeFuncRuntime() with an anonymous function that counts backwards from 10. Anonymous functions can be arguments too!

*Review*

Let’s review what we learned in this lesson:

* Abstraction allows us to write complicated code in a way that’s easy to reuse, debug, and understand for human readers
* We can work with functions the same way we would any other type of data including reassigning them to new variables
* JavaScript functions are first-class objects, so they have properties and methods like any object
* Functions can be passed into other functions as parameters
* A higher-order function is a function that either accepts functions as parameters, returns a function, or both

**Iterators**

The built-in JavaScript array methods that help us iterate are called iteration methods, at times referred to as iterators. Iterators are methods called on arrays to manipulate elements and return values. Like:  
.forEach()

.map()

.filter()

const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];

artists.forEach(artist => {

  console.log(artist + ' is one of my favorite artists.');

});

const numbers = [1, 2, 3, 4, 5];

const squareNumbers = numbers.map(number => {

  return number \* number;

});

console.log(squareNumbers);

const things = ['desk', 'chair', 5, 'backpack', 3.14, 100];

const onlyNumbers = things.filter(thing => {

  return typeof thing === 'number';

});

console.log(onlyNumbers);

Output:

Picasso is one of my favorite artists.

Kahlo is one of my favorite artists.

Matisse is one of my favorite artists.

Utamaro is one of my favorite artists.

[ 1, 4, 9, 16, 25 ]

[ 5, 3.14, 100 ]

*The .forEach() Method*

The first iteration method that we’re going to learn is .forEach(). Aptly named, .forEach() will execute the same code for each element of an array.

const groceries = [‘brown sugar’, ‘salt’, ‘cranberries’, ‘walnuts’];

groceries.forEach(function(groceryItem) { console.log(‘ – ‘ + groceryItem); });

* iterator
* callback function

The code above will log a nicely formatted list of the groceries to the console. Let’s explore the syntax of invoking .forEach().

* groceries.forEach() calls the forEach method on the groceries array.
* .forEach() takes an argument of callback function. Remember, a callback function is a function passed as an argument into another function.
* .forEach() loops through the array and executes the callback function for each element. During each execution, the current element is passed as an argument to the callback function.
* The return value for .forEach() will always be undefined.

Another way to pass a callback for .forEach() is to use arrow function syntax.

groceries.forEach(groceryItem => console.log(groceryItem));

We can also define a function beforehand to be used as the callback function.

function printGrocery(element){  
  console.log(element);  
}  
   
groceries.forEach(printGrocery);

The above example uses a function declaration but you can also use a function expression or arrow function as well.

*The .map() Method*

The second iterator we’re going to cover is .map(). When .map() is called on an array, it takes an argument of a callback function and returns a new array! Take a look at an example of calling .map():

const numbers = [1, 2, 3, 4, 5];   
   
const bigNumbers = numbers.map(number => {  
  return number \* 10;  
});

.map() works in a similar manner to .forEach()— the major difference is that .map() returns a new array.

In the example above:

* numbers is an array of numbers.
* bigNumbers will store the return value of calling .map() on numbers.
* numbers.map will iterate through each element in the numbers array and pass the element into the callback function.
* return number \* 10 is the code we wish to execute upon each element in the array. This will save each value from the numbers array, multiplied by 10, to a new array.

If we take a look at numbers and bigNumbers:

console.log(numbers); // Output: [1, 2, 3, 4, 5]  
console.log(bigNumbers); // Output: [10, 20, 30, 40, 50]

Notice that the elements in numbers were not altered and bigNumbers is a new array.

*The .filter() Method*

Another useful iterator method is .filter(). Like .map(), .filter() returns a new array. However, .filter() returns an array of elements after filtering out certain elements from the original array. The callback function for the .filter() method should return true or false depending on the element that is passed to it. The elements that cause the callback function to return true are added to the new array. Take a look at the following example:

const words = ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];   
   
const shortWords = words.filter(word => {  
  return word.length < 6;  
});

* words is an array that contains string elements.
* const shortWords = declares a new variable that will store the returned array from invoking .filter().
* The callback function is an arrow function has a single parameter, word. Each element in the words array will be passed to this function as an argument.
* word.length < 6; is the condition in the callback function. Any word from the words array that has fewer than 6 characters will be added to the shortWords array.

Let’s also check the values of words and shortWords:

console.log(words); // Output: ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];   
console.log(shortWords); // Output: ['chair', 'music', 'brick', 'pen', 'door']

Observe how words was not mutated, i.e. changed, and shortWords is a new array.

*The .findIndex() Method*

We sometimes want to find the location of an element in an array. That’s where the .findIndex() method comes in! Calling .findIndex() on an array will return the index of the first element that evaluates to true in the callback function.

const jumbledNums = [123, 25, 78, 5, 9];   
   
const lessThanTen = jumbledNums.findIndex(num => {  
  return num < 10;  
});

* jumbledNums is an array that contains elements that are numbers.
* const lessThanTen = declares a new variable that stores the returned index number from invoking .findIndex().
* The callback function is an arrow function has a single parameter, num. Each element in the jumbledNums array will be passed to this function as an argument.
* num < 10; is the condition that elements are checked against. .findIndex() will return the index of the first element which evaluates to true for that condition.

Let’s take a look at what lessThanTen evaluates to:

console.log(lessThanTen); // Output: 3

If we check what element has index of 3:

console.log(jumbledNums[3]); // Output: 5

Great, the element in index 3 is the number 5. This makes sense since 5 is the first element that is less than 10.

If there isn’t a single element in the array that satisfies the condition in the callback, then .findIndex() will return -1.

const greaterThan1000 = jumbledNums.findIndex(num => {  
  return num > 1000;  
});  
   
console.log(greaterThan1000); // Output: -1

Another example below:

const animals = ['hippo', 'tiger', 'lion', 'seal', 'cheetah', 'monkey', 'salamander', 'elephant'];

const foundAnimal = animals.findIndex(animal => {return (animal === 'elephant');});

const startsWithS = animals.findIndex(animal => {return(animal[0] === 's');});

console.log(animals[startsWithS]);

*The .reduce() Method*

Another widely used iteration method is .reduce(). The .reduce() method returns a single value after iterating through the elements of an array, thereby *reducing* the array. Take a look at the example below:

const numbers = [1, 2, 4, 10];  
   
const summedNums = numbers.reduce((accumulator, currentValue) => {  
  return accumulator + currentValue  
})  
   
console.log(summedNums) // Output: 17

Here are the values of accumulator and currentValue as we iterate through the numbers array:

| **Iteration** | **accumulator** | **currentValue** | **return value** |
| --- | --- | --- | --- |
| First | 1 | 2 | 3 |
| Second | 3 | 4 | 7 |
| Third | 7 | 10 | 17 |

Now let’s go over the use of .reduce() from the example above:

* numbers is an array that contains numbers.
* summedNums is a variable that stores the returned value of invoking .reduce() on numbers.
* numbers.reduce() calls the .reduce() method on the numbers array and takes in a callback function as argument.
* The callback function has two parameters, accumulator and currentValue. The value of accumulator starts off as the value of the first element in the array and the currentValue starts as the second element. To see the value of accumulator and currentValue change, review the chart above.
* As .reduce() iterates through the array, the return value of the callback function becomes the accumulator value for the next iteration, currentValue takes on the value of the current element in the looping process.

The .reduce() method can also take an optional second parameter to set an initial value for accumulator (remember, the first argument is the callback function!). For instance:

const numbers = [1, 2, 4, 10];  
   
const summedNums = numbers.reduce((accumulator, currentValue) => {  
  return accumulator + currentValue  
}, 100)  // <- Second argument for .reduce()  
   
console.log(summedNums); // Output: 117

Here’s an updated chart that accounts for the second argument of 100:

| **Iteration #** | **accumulator** | **currentValue** | **return value** |
| --- | --- | --- | --- |
| First | 100 | 1 | 101 |
| Second | 101 | 2 | 103 |
| Third | 103 | 4 | 107 |
| Fourth | 107 | 10 | 117 |

*Iterator Documentation*

There are many additional built-in array methods, a complete list of which is on the [MDN’s Array iteration methods page](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Iteration_methods).

The documentation for each method contains several sections:

1. A short definition.
2. A block with the correct syntax for using the method.
3. A list of parameters the method accepts or requires.
4. The return value of the function.
5. An extended description.
6. Examples of the method’s use.
7. Other additional information.

The **some()** method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

const array = [1, 2, 3, 4, 5];

// checks whether an element is even

const even = (element) => element % 2 === 0;

console.log(array.some(even));

// expected output: true

The **every()** method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

const isBelowThreshold = (currentValue) => currentValue < 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isBelowThreshold));

// expected output: true

**DEBUGGING JAVASCRIPT CODE**

*JavaScript Error Types*

Now that you can identify the type of error from an error stack trace, you might be wondering what the different types of errors mean.

Here are three common error types:

* **SyntaxError**: This error will be thrown when a typo creates invalid code — code that cannot be interpreted by the compiler. When this error is thrown, scan your code to make sure you properly opened and closed all brackets, braces, and parentheses and that you didn’t include any invalid semicolons.
* **ReferenceError**: This error will be thrown if you try to use a variable that does not exist. When this error is thrown, make sure all variables are properly declared.
* **TypeError**: This error will be thrown if you attempt to perform an operation on a value of the wrong type. For example, if we tried to use a string method on a number, it would throw a TypeError.

There are seven types of built-in JavaScript errors in total. You can find more information about all of them at the [MDN JavaScript Error documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error). Whenever you are confronted with an error you can’t comprehend, consulting this documentation is a great place to start.

*Locating Silent Bugs*

Errors thrown by the computer are really useful because they identify the bug type and location for you right away. However, even if your code runs error-free, it is not necessarily bug-free.

You may find that your code is consistently returning incorrect values without throwing any errors. A lack of thrown errors does not mean your code logic is completely correct.

An incredibly powerful tool for locating bugs is a method you likely learned very early on in your JavaScript journey: console.log()!

**Code Challenges: Intermediate JavaScript**

*declineEverything() and acceptEverything()*

Write a function declineEverything() that takes in an array of strings and, using .forEach(), loops through each element in the array and calls politelyDecline() with each of them.

The .forEach() function should apply politelyDecline() directly; it should NOT merely receive an argument function that *uses* politelyDecline().

const veggies = ['broccoli', 'spinach', 'cauliflower', 'broccoflower'];

const politelyDecline = (veg) => {

      console.log('No ' + veg + ' please. I will have pizza with extra cheese.');

}

// Write your code here:

function declineEverything(arr){

  arr.forEach(politelyDecline);

}

declineEverything(veggies);

function acceptEverything(arr){

  arr.forEach(string => {

    console.log(`Ok, I guess I will eat some ${string}.`);

  });

}

acceptEverything(veggies);

*justCoolStuff()*

Write a function justCoolStuff() that takes in two arrays of strings, and, using [the built-in .filter() method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter), returns an array with the items that are present in both arrays.

let arr1 = ['this', 'not this', 'nor this'];  
let arr2 = ['thing 1', 'thing 2', 'this'];  
   
justCoolStuff(arr1, arr2); // Should return ['this']

Solution:

function justCoolStuff(arr1, arr2){

  let newArr = arr1.filter(item => {

    return arr2.indexOf(item) > -1;

  });

  return newArr;

}

// Feel free to uncomment the code below to test your function

 const coolStuff = ['gameboys', 'skateboards', 'backwards hats', 'fruit-by-the-foot', 'pogs', 'my room', 'temporary tattoos'];

 const myStuff = [ 'rules', 'fruit-by-the-foot', 'wedgies', 'sweaters', 'skateboards', 'family-night', 'my room', 'braces', 'the information superhighway'];

 console.log(justCoolStuff(myStuff, coolStuff))

// // Should print [ 'fruit-by-the-foot', 'skateboards', 'my room' ]