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.

**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 *and* 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.

*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.

**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.