Contents

[***Professional Communication*** 2](#_Toc27480151)

[Time and Energy 3](#_Toc27480152)

[Speed vs Substance 3](#_Toc27480153)

[***Coding Principles in JS*** 4](#_Toc27480154)

[Syntax (V1.1.0) 5](#_Toc27480155)

[Variables (V1.1.0) 6](#_Toc27480156)

[Datatypes (V1.1.0) 10](#_Toc27480157)

[Arrays/Lists (V1.0.0) 14](#_Toc27480158)

[Conditional Statements (V1.0.0) 14](#_Toc27480159)

[Functions (V1.0.0) 14](#_Toc27480160)

[Loops (V1.0.0) 14](#_Toc27480161)

[Objects (V1.0.0) 14](#_Toc27480162)

[Scopes and Hierarchies (V1.0.0) 14](#_Toc27480163)

[Callbacks (V1.0.0) 14](#_Toc27480164)

[***Interactive/Online Media*** 15](#_Toc27480165)

[REST API (pt. 1) 16](#_Toc27480166)

[***Appendix A: Vocabulary*** 18](#_Toc27480167)

# Professional Communication

## Time and Energy

The only truly finite resources humans have are time and energy. Anything else can be found or created using time and energy. Indeed, having a job is simply selling your time and energy to create everything else humans need!

This is something you must always be cognizant of in both the workplace and your life. Not only is efficient use of your time and energy vital to your success but being aware of the cost your actions have on others’ time and energy will help your entire team succeed.

Your teammates can not accomplish work if they are spending all their time reading your long, pointless emails. If you are pessimistic, non-constructively critical, hold grudges, spread rumors, and are generally negative, you will find your team will start using their energy handling their work environment rather than working.

## Speed vs Substance

# Coding Principles in JS

## Syntax (V1.1.0)

JS is an interpreted language. All code is handles at runtime. This means JS executes each line one by one as it reads through the list of code.

##### Statement

A single instruction in Javascript.

##### Expression

An expression is any valid unit of code that evaluated and resolves to a value. They are most commonly simple Algebra equations, such as ‘(3 \* 2) + 7 ‘, which resolves to 13. Other times they are true/false conditions, such as ‘3 > 2’, which the computer resolves as ‘true’.

##### Assignment Operators

Symbols used to assign a value or expression on the right side of the operator to a variable or object on the left side of the operator. The most common operator is ‘=’, but you can combine it with **arithmetic operators** to manipulate a value already in a variable on the left side.

|  |  |  |  |
| --- | --- | --- | --- |
| = | Assign variable/object to this value. | | |
| += | Add this value to the current value | -= | Subtract this value by the current value |
| \*= | Multiply current value by this value | \*\*= | Multiply current value to this value’s power |
| /= | Divide current value by this value | %= | Remainder of the current value divided by this value |
| ++ | Add one to the current value (increment) | -- | Subtract one from the current value (decrement) |

##### Arithmetic Operators

Symbols used in **expressions** to perform mathematical operations. They function just like Algebra. The order of operations is even the same.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | Addition | - | Subtraction | \*\* | Exponents |
| \* | Multiplication | / | Division | % | Modulus (Remainder) |
| () | Parenthesis (expressions inside of parenthesis get evaluated and resolve first) | | | | |

##### Comparison Operators

Used in **expressions** and **conditions** to compare one value or **expression** to another.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| == | Equal to | <= | Less than or equal to | < | Less than |
| != | Not equal to | >= | Greater than or equal to | > | Greater than |

#### String Operators

Used to manipulation strings.

**+** , **+=**

##### Keywords

A predefined word in Javascript that has a specific function and cannot be used for a variable, function, object, or class name.

Some keywords include break, continue, debugger, do, while, for, function, if, else, return, switch, try, catch, var, and let.

## Variables (V1.1.0)

*An object that stores a single value in memory that can be changed.*

##### Explanation

Variables function just like variables in Algebra. They are symbols (or in this case, words) that represent either an unknown value, or a value that could change depending on the program’s state.

##### Syntax

First, use the ‘let’ keyword to declare a variable. This gets done once in the program. You cannot use a variable that has not been declared. When you declare a variable, you don’t have to give it a value right away. You do have to give it a good name. One that explains what the variable’s purpose is in as few letters as possible. You should also always use **camelCase**.

|  |
| --- |
| const and var You can also declare variables with var and const. const creates a variable whose value cannot be changed. |

After a variable has been declared, you can change its value by ‘assigning’ it a value. This is done with an **assignment operator**. The most basic and frequently used assignment operator is ‘=’. The statement looks like this:

keyword variableName assignment operator expression or value semicolon

Which looks like this in practice:

let applesInBasket = 13;

##### Substitution

As your program encounters variables, it will substitute the variable name with its value that is stored in memory at that moment. This is exactly like substituting the value in Algebra!

##### Manipulation

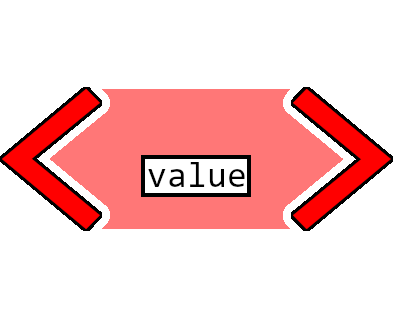
Changing a variable looks a lot like Algebra. Just like ‘x = (y \* 2) + 7’ is a valid equation in Algebra, so too is it valid in Javacript (albeit for slightly different reasons). In addition to **assignment operators**, you can create **expressions** with **arithmetic operators** to create new values for the variable to store. **Expressions** even follow the same order of operations as Algebra. PEMDAS!

Other **assignment operators** combined a **arithmetic operator** with the ‘=’ operator to provide fast and easy to read way of changing variable values. For example, while we could type out this statement:

applesInBasket = applesInBasket + 3; // Add three apples to the basket

It’s much faster to use the ‘+=’ **assignment operator**:

applesInBasket += 3; // Add three apples to the basket

Some variables can be ‘strings’ (see **data types**). Adding them together is called **concatenation**. This is done by using the **string operator** ‘+’ or the **assignment operator** ‘+=’.

#### Shapes

In this class, we will identify all variables (and indeed, object properties) with   
“ < > ”, as per our ***Shapes Definitions*** guide.

### Knowledge (V1.1.0)

Go through these code blocks and identify each variable by putting the “ < > ” shape markers around it. You don’t have to understand the program. Just identify the variables. For example, change:

let sayHelloWorld = sayHello + sayWorld ; // sayHelloWorld now equals ‘Hello World’

…into:

let < sayHelloWorld > = < sayHello > + < sayWorld >; // sayHelloWorld now equals ‘Hello World’

… to identify ‘sayHelloWorld’, ‘sayHello’, and ‘sayWorld’ as variables.

function sayHelloWorld ( iterant ) {  
  
 let phrase = ‘Hello World! ‘ ;

let loopCounter = 0 ;  
  
 while ( loopCounter < iterant ) {  
  
 console.log ( phrase ) ;  
  
 loopCounter += 1;  
  
 }  
}

function updateBar ( player , hpsp , min , max ) {

let calculated = ( ( min / max ) \* 100);

if ( calculated > 100 ) {

calculated = 100;

} else if (calculated < 0) {

calculated = 0;

}

}

for ( let i = 0 ; i < 10 ; i += 1 ; ) {  
   
 console.log ( ‘We have looped ‘ + i + ‘ times!’ ) ;  
   
}

### Comprehension (V1.1.0)

For each of these code blocks, replace the variables with the value they hold at the place in the code where they appear. You do not have to replace variables that are being declared or are on the left side of an **assignment operator**:

let applesPerBasket = 12;

let baskets = 3;

let totalApples;

console.log('You have ' + baskets + ' baskets');

\_\_\_\_\_\_\_\_\_\_\_

console.log('They have ' + applesPerBasket + ' each');

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

totalApples = baskets \* applesPerBasket;

\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

console.log('You have ' + totalApples + ' in all');

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

let roaches = 16;  
let maurauders = 12;  
if ( mauraders > roaches ) {

\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_

roaches -= ( mauraders / 2 ); // Two hits to kill a roach  
  
  
 \_\_\_\_\_\_\_\_\_

console.log( roaches + ' roaches remain!');  
  
  
 \_\_\_\_\_\_\_\_

} else {  
 rageQuit( 'terran' );  
}

let dingdong = 7;  
let wingding = 5.5;  
let herpderp = ((dingdong \* 3) + wingding) / (dingdong - wingding);

\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_  
console.log( herpderp );  
  
  
 \_\_\_\_\_\_\_\_

### Application (V1.1.0)

On the lines below, write a program that does the following:

* Declare a variable called *name* and assign the value ‘Bob’ to it.
* Declare another variable called *bobsPigeons* and assign the value 3 to it.
* Make *bobsPigeons* equal its current value multiplied by 3.
* Make *name* equal its current value plus the string ‘s Pigeons’ at the end.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Declare a variable called *soon*. Do not give it a value.
* Declare a variable called *thirdNum*. Do not give it a value.
* Declare a variable called *firstNum* and assign it the value 4.
* Declare a variable called *secondNum* and assign it the value 7.
* Change *thirdNum* to equal the expression ‘*firstNum* multiplied by 3, then subtract *secondNum’*
* Change the value of *soon* to ‘I will have my revenge’.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Datatypes (V1.1.0)

##### **Types** **of** **Values**

There are two types of values. Variable values (can change), and fixed values (never change). We already know variable values as simply **variables**. Fixed values are known as **literals**. Their names and their values are exactly the same, making them literally what they say they are. This means typing 10 in code is the number ten, and typing ‘Hello, World!’ in code is literally the human phrase ‘Hello, World!’ and not any variable named *Hello* or *World*.

##### **Data** **Types**

Every value has a data type. The five data types of Javascript are numbers, strings, booleans, arrays, and objects. A variable with no value has an undefined value. A value that doesn’t exist is null, but Javascript counts null as an object.

Javascript stores data in formats that cannot be directly compared. For example, human-readable words are not numbers, so you cannot multiply or divide a number by a letter or word. Sometimes, Javascript can resolve these mismatches on its own. For example, if you attempt to concatenate a number onto a string, it will convert the number to a string before concatenating.

Unlike other languages, variables do not need to stay the data type they were declared as. Assigning a new data type to them overwrites the old data type. You can check what data type a variable is by using the typeof keyword. It will return the data type like a function.

##### Numbers

The most obvious data type is a number. Unlike other programming languages, Javascript stores all real numbers as one type, whether positive, negative, with or without a decimal place.

##### Strings

A string is a list of human-readable letters that are not code. Strings are surrounded by **single quotes** ( ‘ ‘ ), **double-quotes** ( “ “ ), or **backticks** ( ` ` ). Strings can be **concatenated**, which adds one string onto the end of another. Concatenating another value onto a string will turn the result into a string, also.

##### Booleans

true or false. Also resolves as 1 and 0. It is more accurate to say that it is ‘not false’ and ‘false’, as any value other than false or 0 is counted as true.

##### Arrays

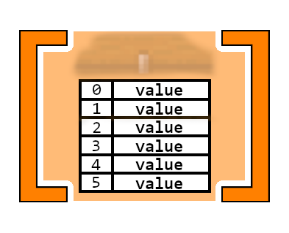
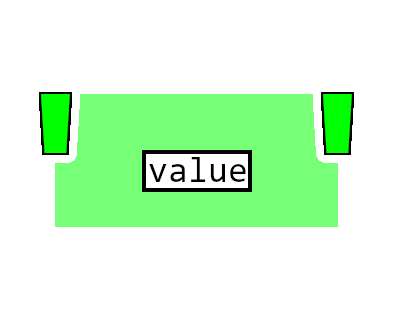
Arrays are a list of values, separated by commas, contained in square brackets ( [ ] ). Unlike other languages, each value in an array in Javascript can be a different datatype. Individual values can be accessed by their **index**. Think of it like a page number, or address.

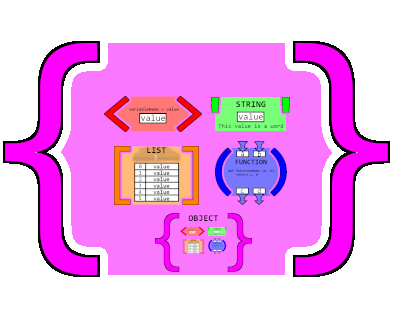
##### Objects

Technically, everything in Javascript is an object. Ultimately, an object is a variable whose value is more variables and functions (properties and methods). They can be treated as individual items, widgets, or indeed – objects in your code that have all kinds of features.

##### Shapes

We will need three more shapes from our **Shapes Definitions** guide, though we will only start with one extensively. We can mark **literal strings** by wrapping them in single quotes and underlining them “ ‘ \_\_ ' “. *We are only marking literal strings, because they are frequently confused with variables and* functions. Later we will use **arrays** (otherwise known as ‘List’), and **objects**.





### Knowledge (V1.1.0)

For this exercise, we will focus on **variables**, **literal strings**, and **literal numbers**.

* Wrap all **variables** in “ < > ”
* Wrap all **literal strings** in “ ‘ \_\_ ' “
* Circle all **literal numbers**.

let roaches = 16;  
  
let maurauders = 12;  
  
if ( mauraders > roaches ) {

roaches -= ( mauraders / 2 ); // Two hits to kill a roach

console.log( roaches + ' roaches remain!' );

} else {

rageQuit( 'terran' );

}

let applesPerBasket = 12;

let baskets = 3;

let totalApples;

console.log('You have ' + baskets + ' baskets');

console.log('They have ' + applesPerBasket + ' each');

totalApples = baskets \* applesPerBasket;

console.log('You have ' + totalApples + ' in all');

function sayHelloWorld ( iterant ) {  
  
 let phrase = ‘Hello World!‘ ;

let loopCounter = 0 ;  
  
 while ( loopCounter < iterant ) {  
  
 console.log ( phrase ) ;  
  
 loopCounter += 1 ;  
  
 }  
}

### Comprehension (V1.1.0)

At the end of these programs, write the console output:

let apples = 10;  
let urmumSize = ‘large’;  
let customerList = [‘Bob’, ‘Earl’, ‘Frank’];  
let sizeApples = urmumSize + apples;  
let bananas = ‘12’;  
console.log(typeof apples);  
console.log(typeof urmumSize);  
console.log(typeof customerList);  
console.log(typeof sizeApples);  
console.log(typeof customerList[1]);  
console.log(typeof bananas);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Application (V1.1.0)

On the lines below, write a program that does the following:

* Declare a variable that is assigned a literal string as the value
* Declare a variable that is assigned a literal number as the value
* Declare a variable that is assigned an **expression** that resolves as a number
* Declare a variable that is assigned an **expression** that resolves as a string

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Quick **Reference** (V1.0.0)

#### Code Color Guide

Keyword Object Function Number String Comments

// SYNTAX  
let varyVanilla; // Declare a variable named ‘varyVanilla’, with no value to start  
let varyBerry = 10; // Declare a variable named ‘varyBerry’ assign it a value of 10  
  
// MANIPULATION / SUBSTITUTION  
varyBerry = 12; // Change the value of ‘varyBerry’ to 12  
varyVanilla = 11; // Change the value of ‘varyVanilla’ to 11  
varyVanilla = (varyBerry \* 2) + 7; //’varyVanilla’ is now equal to 31 [ (12\*2)+7 ]  
  
let sayHello = ‘Hello ‘; // These variables have stored strings  
let sayWorld = ‘ World’;  
let sayHelloWorld = sayHello + sayWorld; // sayHelloWorld now equals ‘Hello World’  
  
console.log(sayHelloWorld); // Replaces ‘sayHelloWorld’ with its value, and logs it to console  
  
// Increments/Decrements  
varyBerry++; // ‘varyBerry’ now equals 13  
varyBerry--; // ‘varyBerry’ now equals 12 again

varyVanilla -= varyBerry; // ‘varyVanilla’ now equals 19  
sayHelloWorld += sayHelloWorld; // ‘sayHelloWorld’ now equals ‘Hello WorldHello World’

## Arrays/Lists (V1.0.0)

## Conditional Statements (V1.0.0)

## Functions (V1.0.0)

## Loops (V1.0.0)

## Objects (V1.0.0)

## Scopes and Hierarchies (V1.0.0)

## Callbacks (V1.0.0)

# Interactive/Online Media

## REST API (pt. 1)

An **API** is an *Application Programming Interface*. They are designed by a developer to allow other developers to interact easily with their application, without ever exposing the source code of the application.

A **REST API** is a *Representational State Transfer*, which is a set of rules a developer follows as they make their API. Once you’ve learned how to use a **REST API** for one application, it’s easy to learn how to use them for other applications, too!

##### HTTP Methods

**REST APIs** rely heavily on **HTTP** (hyper-text transfer protocol). Typically, the user will send a **request**, and the application will send a **response**. How the application handles the **request** depends on the **HTTP method**. There are many types of **methods**, but we will focus on **GET** and **POST**. When you type a URL in your browser, it always sends a **GET request**. If you click a ‘Submit’ button on the page, it typically sends a **POST request**. This is because **GET** is used to *get* a **resource** from the application, whereas as **POST** is used to *send* information to the application.

By convention, most developers follow the **CRUD methodology**. This means while developing your API, you should allow your users to ***C****reate,* ***R****ead,* ***U****pdate, and* ***D****elete* **resources**.

##### Endpoints

An **endpoint** is a URL that when navigated to, will activate a command in the application depending on the **method** used. For example, if you type this into a browser:

https://api.github.com/users/csmith1188

… it will return information about the user ‘csmith1188’ from GitHub in **JSON** format. A **root endpoint** it is the point where all other **endpoints** build from. In this case, the **root endpoint** would be this:

https://api.github.com/

The structure of the URL that makes up the **endpoint** is known as the **path**. In other words, what “**path**” through the **REST API** of the application do you have to take to access specific **resources**? A **path :variable** is a part of the URL that can be changed by the user to access a specific **resource**. The root-endpoint/path/:path-variable of the previous example is as follows:

https://api.github.com/users/csmith1188

‘users/’ is the **path**, letting the application know we want to select from a list of users. ‘csmith1188’ is the specific user we want to look up. The application then finds the relevant information and sends it back in the **response**.

##### Query Parameters

Another way we can request information from an application is through **query parameters**. **HTTP** allows you to put extra variables into a URL to be read by the application you are connecting to. For example

https://api.github.com/repos/csmith1188/JSFighter-class-/issues?state=open&label=bug

Not only are we using the **path** to tell the application we want to access the issues in repos, and using **path variables** to tell the application which user and repo we want to access, but we are using **query parameters** to tell the application that we only want to look at issues whose state is open and are labelled as bugs.

The ‘**?**’ tells the application **query parameters** are coming. The syntax looks like ‘**parameterName=value**’, and each parameter is separated by ‘**&**’.

##### JSON Responses

After the client **requests** information from the server, the server will send a **response**. The response will contain the requested resource, which can be a file, an error code, or some data. Data plain text typically sent in **JSON** (JavaScript Object Notation) format, which looks like this:

{“userName”: “Bob”, “icon”: “icons/bob.png”}

This data uses syntax rules that make it easy for a computer to parse and convert to code.

### Concept Check & Anticipation Guide

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_ | REST API  Resource  REST Endpoint  Root Endpoint  HTTP Method  REST Path  Path :variable  Query parameters  HTTP Request  HTTP Response  CRUD | \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_  \_\_\_\_\_\_ | | Before Class: In the *left column*, mark each word with a symbol from the list below.   * ➕ if you know enough to teach someone else about this * ⚪ if you know a little bit about this, but could learn more * ➖ if you do not know anything about this  After Class: In the *right column*, mark each word with a symbol and see what has changed! |

### Before Class:

In the *left column*, put a ✔️ if you think the sentence is **true**. Otherwise, leave it blank.

### After Class:

In the *right column*, put a ✔️ if you think the sentence is **true**. Otherwise, leave it blank.

|  |  |  |
| --- | --- | --- |
|  | Path :variables are another way of saying query parameters |  |
|  | REST APIs are vital for making chat bots |  |
|  | CRUD is a method of cleaning out excess data from a request |  |
|  | An endpoint is last line of code run before the server sends a response |  |
|  | A GET request is used to retrieve data from a server over HTTP. |  |
|  | A POST request is to send data to a server over HTTP. |  |
|  | HTTP is how computers send web data over the internet. |  |
|  | The root path of a REST API is the top level of hierarchal categorization of API endpoints. |  |
|  | A ‘client’ is a computer requesting information. A ‘server’ is a computer that provides it. |  |
|  | JSON format is Javascript code sent over the internet. |  |
|  | A ‘resource’ is a piece of data or file stored on a server, which clients request access to. |  |
|  | Your company project uses a REST API. |  |

# Appendix A: Vocabulary

Code Color Guide

Keyword Object Function Number String Comments

### Variable

*An object that stores a single value that can be changed.*

This is used to store data to be manipulated later. It works like a variable in algebra.

Use the ‘let’ keyword to declare a variable. This gets done once in the program. You cannot use a variable that has not been declared.

let value = 10; // Declare value  
console.log(value); // Log the variable’s value to console

### Constant

*A variable whose value cannot be changed during runtime.*

They are useful for tweaking configurations during development.

Use the ‘const’ keyword to declare a constant in the same way as a variable.

const MAXLIMIT = 3; // Declares MAXLIMIT as a constant  
let counter = 4; // Declares a counter variable  
if (counter > MAXLIMIT) {  
 // Checks to see if counter is greater than MAXLIMIT  
 console.log(‘counter is greater than’ + MAXLIMIT);  
}

### Array

*A single variable that stores multiple values.*

Arrays are declared like a variable except all values are enclosed in square brackets, separated by commas. Elements (values) of an array are accessed by their index number. Index numbers start at 0.

let classmates = [‘Josh’, ‘Noah’, ‘Harry’]; // Declare array as a variable  
console.log(classmates[0]); // Prints the first value in the array  
console.log(classmates[2]); //Prints the third value in the array

### Statement

*An executable line of code.*

It is represented by a rectangle in a flow chart.

### Conditional Statement

*A line of code that executes when a given condition is NOT FALSE.*

This is used for “Program Control Flow”. It is represented by a diamond in a flow chart.

### “If” Statement

*A conditional statement that forks the program flow.*

Called by the keyword ‘if’, followed by a condition in parenthesis **()**, followed by a list of statements in curly braces **{}**.

Can also contain an ‘else if’ and ‘else’ branch. ‘else if’ requires a different condition. ‘else’ executes if no prior conditions were met.

Formerly known as an “if/then” statement and is verbalized as “if/then/else if/then/else”.

let name = ‘Isiah’;  
if (name == ‘Noah’) { // If name is Noah, execute statements in these curly braces  
 console.log(‘Hello, Noah! My neighbor.’);  
} else if (name == ‘Harry’) { // If name is not Noah, but IS Harry…  
 // Then say hello to Harry instead  
 console.log(‘Hello, Harry, who sits three rows away’);  
} else { //Else if no other previous conditions were met  
 // Then execute these statements  
 console.log(‘I have no idea who you are’);  
}

### while loop

*A conditional statement that loops through a series of statements if the condition is NOT FALSE.*

It is good for an unknown number of loops.

Called by the keyword ‘while’, followed by a condition in parenthesis **()**, followed by a list of statements in curly braces **{}**.

let looper = true;  
while (looper) {  
 console.log(‘Condition was NOT FALSE, so I did the loop.’);  
 looper = false; // Make looper false, so we won’t repeat loop  
}

### do while loop

*A while statement that runs the loop* ***before*** *checking the condition to run again.*

It is good for an unknown number of loops, where loop must run at least once.

Called by the keyword ‘do’, followed by a list of statements in curly braces **{}**. It is then followed by the ‘while’, followed by the condition.

let looper = false;  
do {  
 console.log(‘This gets logged once, even though the condition is false’);  
}  
while (looper);

### Functions

*A repeatable group of statements.*

Functions can take multiple arguments (parameters). Functions can also return a value as though it were a variable.

Functions are declared with the keyword ‘function’, followed by the function name, and parameter names in parenthesis (). Parameters are separated by commas. A list of statements to be executed go inside of curly braces {}. The return statement is declared with the keyword ‘return’.

function newFunction(argA, argB) {  
 console.log(argA);  
 console.log(argB);  
 let c = argA + argB;  
 return c;  
}  
console.log(newFunction(1, 2));

### Objects

*An object is an identifiable “thing” in Javascript. Everything in Javascript is technically an object.*

Objects have properties and methods. Properties are variables that belong to the object, whereas methods are functions that belong to the object.

Objects are written in JSON format (Javascript Object Notation). Everything inside of curly braces **{}** is part of the object. Objects are made up of smaller objects called attributes (objects, properties and methods), which are defined by an attribute name followed by a colon (**:**), and separated by commas (**,**).

You can access an object’s attributes by putting the object’s name and a period in front of the attribute name. Objects refer to their own scope with the ‘*this*’ keyword.

let Person = {  
 // This is a property attribute  
 name: ‘Bob’,  
 // This is a method attribute  
 sayName: function() {  
 console.log(‘My name is ‘ + this.name);  
 }  
}  
  
// Log the value of the ‘name’ property  
console.log(Person.name)  
// Call the function of the ‘sayName’ attribute  
Person.sayName()