# Coding Principles in JavaScript

## Syntax (V1.2.0)

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

##### Comments

Anytime you see // , all text the comes after it on that line is a **comment**, meaning that text is not considered code by Javascript. This is great for leaving explanatory notes for yourself or other programmers. Comments can also span multiple lines by putting the text between /\* and \*/ .

##### Statement

A single instruction in Javascript. We typically end every statement with a **semicolon**  ; , though it’s not required.

##### 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’.

##### Literal Value

Any code that is literally the value you mean to represent. For example, the number 3 in code is literally the number three. Text inside quotes – 'three' – is a string literal.

|  |  |  |  |
| --- | --- | --- | --- |
| number literal: | 0 | string literal: | 'Hello, world' |
| array literal: | ['A', 'B', 'C'] | object literal: | { /\*properties\*/ } |
| function literal: | () => { /\*function code\*/ } | | |

##### Object Identifier

We can assign an identifier to a value or expression, so we can call it again later using *substitution*. This is done with the let keyword and an assignment operator. You could also use the const keyword instead to make the value un-editable, or constant.

let myName = ‘Bob’; //This will give the literal string ‘Bob’ an identifier called ‘myName’

console.log(‘My name is ‘ + myName); //This will replace ‘myName’ with the stored value ‘Bob’

##### 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

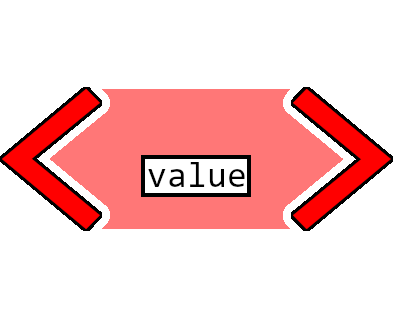
Changing a variable looks a lot like Algebra. Just like ‘x = (y \* 2) + 7’ is a valid equation in Algebra, so too is it valid in Javascript (albeit for slightly different reasons). In addition to **assignment operators**, you can create **expressions** with **arithmetic operators** to create new values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

Other **assignment operators** combine an **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.

### Variables: Knowledge (V1.1.2)

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!’ ) ;  
   
}

### Variables: Comprehension (V1.1.1)

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 );  
  
  
 \_\_\_\_\_\_\_\_

### Variables: Application (V1.1.1)

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 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

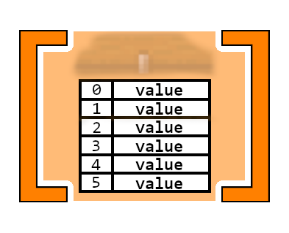
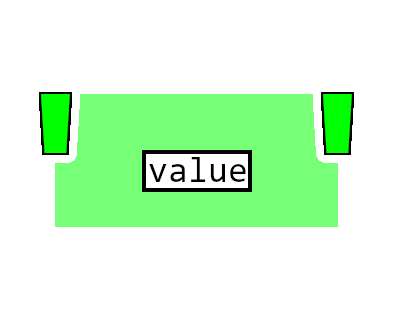
Arrays are a list of values, separated by commas, contained in square brackets ( [ ] ). Unlike other languages, each value in an array in Javascript can be a different datatype. Individual values can be accessed by their **index**. Think of it like a page number, or address.

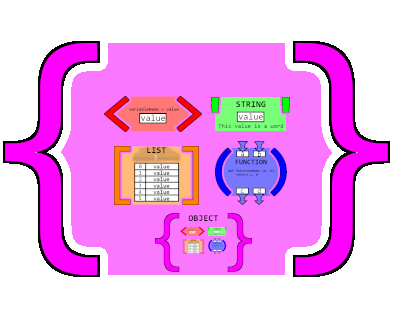
##### Objects

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**.





### Datatypes: Knowledge (V1.1.1)

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 ;  
  
 }  
}

### Datatypes: Comprehension (V1.1.1)

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);

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

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

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

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

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

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

### Datatypes: Application (V1.1.1)

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

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

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

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

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

## Conditional Statements (V1.0.0)

An important principle of programming is “flow control”. A program is a lot like a movie script that can listen to the audience and change the outcome of the movie accordingly. Conditional statements change the “flow” of the program according to changes in information, like user input.

##### If/Then/Else

The “classic” conditional statement is the **If**/**Then**/**Else** statement. The syntax is different in every language, but the function is always the same:

if this expression resolves as true, then do this; else do that.

The important parts of this are the **keywords** “**if**” and “**else**”, the **condition**, and code to be run if the **condition** is true or false. In Javascript, it looks like this:

if (this experession is true) {  
 then do this;  
} else {  
 do that;  
}

Using an “**if**/**else**” statement, we can “fork” our code into a “**true**” path and “**false**” path. We can create a “side quest” by only using “**if**” and no “**else**”.

do this;  
if (this experession is true) {  
 then also do this;  
}

do that;

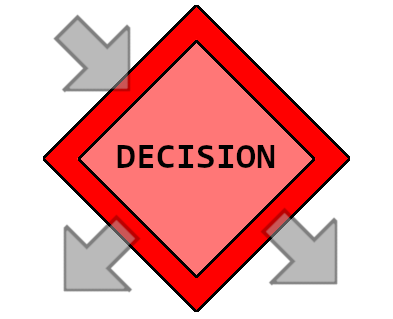
We can also create more than two paths using an “**else if**” keyword. If the first **condition** is not true, then it will check the second **condition**. If none of the provided conditions are true, it will run the **Else** statement. There can be as many “**else if**” statements as you want.

if (this experession is true) {  
 then do this;  
} else if (this other expression is true) {  
 then do this other thing;  
} else {  
 do that;  
}

##### Flow Control and Shapes

In **flow charts**, a **decision step** is represented by a diamond. Like all other types of flow chart steps, it can take as many inputs as you want. Unlike all other steps, however, it is the only step that that can have any number of outputs.

While **decision steps** are just a vague description of a general “decision”, you can always break decision steps down into smaller series of steps. You can choose to closely reflect your code in the flow chart, which is what online tools such as **code2flow** seek to do. This will create patterns of shapes that you will frequently see while writing flow charts, creating a visual link to the abstract thought of forks in code.



### Conditional Statements: Knowledge

For each of these code blocks:

* Circle each keyword.
* Wrap each **condition** in ‘ **< >** ‘.
* Underline any code that will run if the condition is **true** or **false**.

socks.owner = ‘Jimmy’;

socks.putOnUser(‘Fred’);  
  
if (socks.user != socks.owner) {

socks.userMood = ‘Not great’;

} else {

shoes.insert(socks);

}

if (temperature > 60 && weather != ‘rain’) {

vehicle = ‘Wrangler’;

} else if (weather == ‘snow’) {

vehicle = ‘Cherokee’;

} else {

vehicle = ‘tC’;

}

if (student.arrivalTime > class.startTime) {

student.tardy++;

if (student.tardy > 3) {

student.tardy = 0;

student.detention++;

student.detention();

} else {

student.tardyWarning();

}

}

### Conditional Statements: Comprehension

At the end of each of these programs, write the console output that would be displayed if the programs were run.

trainer.gymBadges = 7;

if (trainer.gymBadges < 8) {

console.log(‘Get out of here, scrub!’);

} else {

console.log(‘Welcome to the Indigo Platuea!’);

}

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

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

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

vegeta.scouter.horizontalFlip = true;

kakarot.powerLevel = 9001;

console.log(‘Vegeta, what does the Scouter say about his power level?’);  
  
if (kakarot.powerLevel > 9000) {

if (vegeta.scouter.horizontalFlip == true) {

console.log(‘It\’s 1006. Kick his butt!’);

} else {

console.log(‘It\’s over nine-thousaaand!’);  
 vegeta.destroy(this.scouter);

}

} else {

console.log(‘Hardly worth our time!’);

vegeta.destroy(kakarot);  
}

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

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

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

userNumber = 71;  
randomNumber = 54;  
  
if (randomNumber == userNumber) {

console.log(‘You guessed correctly!’);

} else if (randomNumber > userNumber) {

console.log(‘You guessed too low’);  
} else {

console.log(‘You guessed too high!’);  
}

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

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

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

### Conditional Statements: Application (V1.0.1)

For each of the following word problems, write a program below it that solves the problem. You will need to pick good variable names to describe variables in the word problems.

/\* If the user’s age is less than 21, then check to see if the the user’s age is less than 12. If it is, tell the user they’re too young for Slurp Juice. Otherwise, tell the user they’re too

young for Krunk Juice.

\*/

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

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

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

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

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

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

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

/\* If the user’s name if ‘Frank’, tell the user “Welcome back, Frank!”. Otherwise, if the user’s name is ‘Bill’, ask the user “Where’s my money, Bill?”. Otherwise, tell the user “I don’t talk to strangers.”.

\*/

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

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

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

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

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

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

/\* If the user’s wool color is blue, tell the user “Blue Wool”. Otherwise, if the user’s wool color is red, tell the user “Red Wool”. Otherwise, if the user’s wool color is green, tell the user “Green Wool”. Otherwise, tell they user “I can’t see that color”

\*/

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

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

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

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

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

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

## Functions (V1.0.0)

When you take a step, your brain sends commands to a list of muscles in your body in a precise order. Which muscles it sends commands to depends on the direction of stepping. We can tell our brain to “take a step forward”, and our brain will take care of all the commands needed to accomplish that task for you. We can describe this is code like this:

takeStep( ‘forward’ );

‘takeStep’ is the action we want to perform, and ‘forward’ is a modifier or input to that action. This are what we call the **function name** and **function arguments**.

**Functions** are repeatable lists of statements that a program can jump to. They are also known as subroutines (considering that you program is a ‘routine’). Code that contains the list of statements a **function** performs is called a **function declaration**. Code that tells the program to jump to a function is called a **function call**. **Functions** take inputs, called **arguments**. All **function declaration** and **calls** are immediately followed by “ **( )** “, which contain the **function arguments**, if any.

// Function Declaration Example

function functionName ( arguments, separated, by, commas ) {

console.log(‘The first argument\’s value was: ‘ + arguments);

return value;

}

**Functions** have four components to consider when **declaring** one.

|  |  |  |
| --- | --- | --- |
| Function Name There are many ways to give a function a name.  The classic way is to use the **function** keyword:  function takeStep() {};  Because variables and functions are also objects, we can create an object that is a function:  let takeStep = function() {};  In Javascript, we can also use **arrow functions**: (We will discuss **arrow** **functions** more later, as they are usually used to create **functions** without a **function** **name**, because they are actually function **literals**)  let takeStep = () => {}; Function Arguments Function arguments are temporary objects that are created when the **function** is **called** and given values in the **function** **call** itself.  function takeStep( direction ) {  console.log(‘Taking a step ‘ + direction);  } Function Statements Simply put, this is where your repeatable code goes. | |  | | --- | | Shapes Using our Shapes Definitions Guide, we can mark function declarations and calls using parenthesis “ **( )** “. | |
| |  | | --- | | Callback Functions Functions can be passed as arguments in Javascript. Many functions have an argument for **callback** functions, which are code to be execute once the rest of the function has finished. | |

##### Return Statement

When a function reaches a **return statement**, it ends the function. It then *returns* to the point in the code where the function was **called** and replaces the **function call** with the **return value**. The return value is an **expression** after the **return** keyword.

function sayHelloName ( name ) { // Function is named ‘sayHelloName’, and it takes a ‘name’ input

return ‘Hello, ‘ + name; // Sends the value of <‘Hello, ‘ + name> back to the function call

}

console.log(sayHelloName(‘Bob’)); // This is the function call for ‘sayHelloName()’

// When it returns, it will replace sayHelloName(‘Bob’)

// with the return value (‘Hello, Bob’)

### Functions: Knowledge

For each of the code blocks below:

* Put each **function** **name** and the **arguments** that follow inside of “ **( )** “
  + Remember, function names always have (arguments) after them
* Circle **function declarations**
* Underline **function calls**
  + Some things might be underlined multiple times

function sayHello ( name ) {

console.log ( ‘Hello, ‘ + name ) ;

}

sayHello ( ‘Bob’ ) ;

let addFunc = function ( a , b ) {

return a + b;

}

console.log ( ‘Five plus three is ‘ + addFunc(5, 3) ) ;

let firstNum = 5.5 ;

let secondNum = 3.2 ;

let newNumber = ( 1stn, 2ndn ) => {

return Math.round ( addFunc ( 1stn, 2ndn ) ) ;

}

console.log ( ‘Your rounded answer was ‘ + newNumber ( firstNum , secondNum ) ) ;

function sayThenDo ( callback ) {

console.log( “I’m going to do the callback function at the end of this function” );

callback();

}

sayThenDo ( () => { console.log ( “We declared this function as an argument!” ) );

### Functions: Comprehension

For each of the code blocks below, write the console output:

function sayHello (name) {

console.log (‘Hello, ‘ + name);

}

sayHello (‘Bob’) ;

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

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

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

let addFunc = function (a , b) {

return a + b;

}

console.log (‘Five plus three is ‘ + addFunc(5, 3)) ;

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

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

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

let firstNum = 5.5 ;

let secondNum = 3.2 ;

let newNumber = (1stn, 2ndn) => {

return Math.round(addFunc(1stn, 2ndn));

}

console.log (‘Your rounded answer was ‘ + newNumber(firstNum, secondNum));

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

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

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

function sayThenDo(callback) {

console.log(“I’m going to do the callback function at the end of this function”);

callback();

}

sayThenDo (() => {console.log(“We declared this function as an argument!”));

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

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

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

## Functions: Application

Write a program in the lines beneath each prompt:

/\* Write a function that takes one argument. Make the function log the argument to the console.

Then call the function and pass it an argument.

\*/

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

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

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

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

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

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

/\* Write a function that takes two numbers as arguments. If the first number is greater than the second number, return true. Otherwise, return false. Then call the function with two numbers as the argument for console.log()

\*/

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

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

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

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

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

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

/\* Write a function that logs “Returning true” to the console and returns true. Then call the function in the condition of an if statement. Inside of the if statement, log “Function returned true” in the console.

\*/

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

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

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

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

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

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

## Arrays/Lists pt.1 (V1.0.0)

Arrays are a type of **Data Structure** that can hold many different **values** (known as **elements**) in a list, using only one **identifier** (name).

According to WIkipedia:

An [*array*](https://en.wikipedia.org/wiki/Array_data_structure) is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type). Elements are accessed using an integer index to specify which element is required. Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.

Let’s break this down:

##### “An array is a number of elements in a specific order…”

An array is an **Object** that holds a series of **elements** in a list.

let newArray = [‘Bob’, ‘Frank’, ‘Jim’]; //Create new array with three elements  
let newArray = new Array(‘Bob’, ‘Frank’, ‘Jim’); //Or you can use an Array object constructor

“…typically all of the same type…”

Unlike most languages, JavaScript can store values of *different* types, meaning we can mix *strings* with *numbers* and *objects*.

//Change elements to be a string, a number, and a function

newArray = [ ‘Bob’, 73, function() { console.log(‘hello!’) } ];

##### “Elements are accessed using an integer index to specify which element is required.”

You can access the **value** of each **element** by using the element’s **index**. This is a lot like finding the contents of a page in this book by using its page number.

|  |  |
| --- | --- |
| // Syntax is: arrayName[<index number>]  // Indexes start counting at 0  console.log(newArray[0]); //Log ‘Bob’  let age = newArray[1]; //Save 73 to age  newArray[2](); //Do the function at index 2 | ***NOTE:*** *Though there are other places in JS that you can encounter square brackets* ***[ ]****, programmers have moved the convention towards only using them with arrays. If you see square brackets, you are almost certainly dealing with an array.* |

##### “Arrays may be fixed-length or resizable.”

Arrays in Javascript are *resizable*. You can add or remove elements from them. This is usually done by using the Array object’s methods, such as push().

//Add an element to the array

newArray.push(‘Billy’); //Add ‘Billy’ to the end of the array

console.log(newArray[3]); //Log the new element to the console

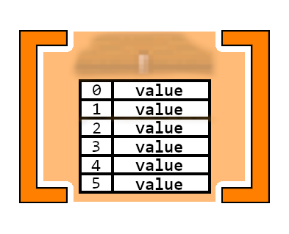
Again, arrays are **objects** in Javascript, and arrays can store **objects**. This means arrays can store other arrays. This is called a **multi-dimensional array**. These are not used as much anymore, as **objects** have robust formatting.

let multiArray = [ [‘Bob’, ‘Frank’] , [‘Jim’, ‘Bill’] ] //Create an array that contains two arrays

console.log(mutliArray[0][1]); //Log ‘Frank’

console.log(multiArray[1][0]); //Log ‘Jim’

#### Shapes

Because *square brackets* are almost always associated arrays, we can identify them in code with a square/rectangular shape. I like to think of arrays as MineCraft chests, as they are rectangular, and contain a grid of spots to store different types of items.

### Arrays/Lists (V1.0.0): Knowledge

In the following code:

* Put all *array objects* inside of “ **[ ]** “
* Underline all array indexes
* Circle all *methods* or *properties* that belong to an *array* *object*.

let shoppingCart = [ 'Apple', 'Banana', 'Coconut' ];

console.log( 'You have ' + shoppingCart.length + ' items in your cart.' );

// Loop through each element of the shoppingCart and log its value

for ( item of shoppingCart ) {

console.log( item );

}

let teamList = [ 'Bruno', 'Narancia', 'Guido', 'Mista', 'Leone', 'Pannacotta' ];

console.log( 'The first member of the team is ' + teamList[ 0 ] );

// Add a new name to the teamList

teamList.push( 'Giorno' );

console.log ( 'Welcome to the team, ' + teamList[ teamList.length - 1 ] );

let masterList = [ 'Tim', 'Billy', 'Leroy the VooDoo Wizard' ];

let newMemberList = [ 'Susan', 'Tim 2' ]

// This function takes an array you give it and adds each element to the master list

function addToMaster ( newList ) {

// Loop as many times as the number of elements in the provided array

for ( let i = 0; i < newList.length; i++ ) {

// Add the the element whose index is equal to i to the masterList

masterList.push( newList[ i ] );

}

}

// Call the function to add the newMemberList to the masterList

addToMaster( newMemberList );

### Arrays/Lists (V1.0.0): Comprehension

For each of these code blocks, write the value of each expression listed below it.

// Declare array in the classic style

let positionList = ['First', 'Second', 'Third'];

// Declare from an object constructor

let medalList = new Array('Gold', 'Silver', 'Bronze');

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| positionList[0] |  |  | medalList[2] |  |

let counter = 12;

let hexValue = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'];

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| counter |  |  | hexValue.length |  |
| hexValue[counter] |  |  | hexValue[counter - 2] |  |

// 15 random words from dictionary.com

// You can break arrays across lines, like this:

let wordBank = ['perplex', 'waylay', 'curtain', 'often', 'gonna',

'down', 'never', 'crucible', 'up', 'behold',

'parabola', 'you', 'let', 'give', 'expendience' ];

|  |  |
| --- | --- |
| wordBank[6] |  |
| wordBank[4] |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| wordBank[13] |  |  | wordBank[12] |  |
| wordBank[11] |  |  | wordBank[11] |  |
| wordBank[8] |  |  | wordBank[5] |  |

### Arrays/Lists (V1.0.0): Application

For each of these code blocks, write a program below it that accomplishes the task given.

/\* Write a program that:

- Makes a new array that includes, Bob, Jim, and Steve

- Logs the array element whose value is ‘Jim’

\*/

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

/\* Write a program that:

- Makes a new variable whose value is 0

- Makes a new array whose values are ‘A’, ‘B’, and ‘C’

- Logs the value of the new array element whose index is equal to the new variable

- Increases the value of the new variable by one

- Logs the value of the new array element whose index is equal to the new variable

- Increases the value of the new variable by one

- Logs the value of the new array element whose index is equal to the new variable

\*/

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

/\* Write a program that:

- Logs the last value of an array called ‘mysteryBox’

(Assume mysteryBox had already been declared somewhere else in the program,

But you don’t know where)

The previous two pages have clues! \*/

|  |
| --- |
|  |
|  |
|  |
|  |

/\* Write a program that:

- Makes a new array whose values ‘Bob’, ‘Jim’, and ‘Steve’

- Removes ‘Steve’ from the new array, because Steve sucks

You will need Google for this one! \*/

|  |
| --- |
|  |
|  |
|  |
|  |

## **Objects** (V1.0.1)

Technically, *everything in Javascript is an object*. JS takes a lot a shortcuts on behalf of the programmer, so you tend not to notice. Objects are just representations of data, and can be thought of a lot like an object in real life. It’s a “thing” that has “features” and can do “stuff”.

##### Object Literals

Though there are many places in syntax that use **curly braces**  { } , curly braces without a preceding **keyword**, such as *while*, *for*, or *if*, typically indicates an **object literal**. This means the code inside the curly braces in literally the value of the object.

let newObject = { objectName : ‘Bob’ } //This is the value of the object, and it has a property.

|  |  |
| --- | --- |
| Object Attributes (anything that belongs to an object) | |
| Object Properties Objects can have many **properties**, or stored *values* with *identifiers*. Basically, any variable that belongs to an object is an object **property**.  let person = {  name: 'Bob';  age: 912  }  console.log(person.name); //Log person’s name  console.log(person.age); //Log person’s age | Object Methods Objects can also have **methods**, or stored *functions* with *identifiers*. Basically, any function that belongs to an object in an object **method**.  let person = {  name: 'Bob';  sayPhrase: (phrase) => {  console.log(phrase);  }  }  //Make person use their 'sayPhrase' method  person.sayPhrase('Hello'); |

##### Object Literal Syntax

Object syntax doesn’t use normal *keywords*, *assignment operators* or *semi-colons* like normal Javascript code. Instead, attributes are assigned to *identifiers* using a colon : , and they are separated by commas , .

##### Accessing Attributes

You can access the values of attributes my using **dot notation** or **square brackets**. Square brackets are not often used by convention, as they are easily confused with **Arrays**.

let person = { name: 'Bob'; age: 912 }

console.log( person.name ); //Access person's 'name' by dot notation

console.log( person["name"] ); //Access person's 'name' by square bracket notation

|  |  |  |
| --- | --- | --- |
| In order to access a property this exists inside the object we are working within, we need to use the keyword this. let person = {  name: 'Bob';  sayName: () => {  console.log(this.name);  }  }  //Make person say their own name property  person.sayName(); | |  | | --- | | Shapes  Because all object literals require curly braces { }, we can use “ { } “ on our text to identify objects. | |

##### Hidden Attributes

As mentioned earlier, JS takes a lot of shortcuts on behalf of the user. For example, when you assign a literal to an identifier, you are actually *creating a new object from a constructor*. This means that variables that are numbers, strings, arrays, functions, and objects are actually premade objects that have premade attributes. For example, we can get the number of letters in a string by using its hidden length property.

let name = ‘Bob’;

console.log(name + ‘ is ‘ + name.length + ‘ letters long’);

For a list of hidden attributes, check out the Javascript documentation on Mozilla Developer’s Network.

### Objects (V1.0.0): Knowledge

For each of these code blocks:

* Wrap all *Object Identifiers* inside of “ { } “
* Wrap all *Number Object Identifiers* inside of “ < > “
* Wrap all *String Object Identifiers* inside of “ ‘ ‘ “
* Wrap all *Array Object Identifiers* inside of “ [ ] “
* Wrap all *Function Object Identifiers* inside of “ ( ) “
* Underline all *Object Property Identifiers*
* Circle all *Object Method Identifiers*

let catOwner = {

name: 'Debrah',

catsList: [ { name: 'Mr. Speckles', type: 'calico'},

{ name: 'Boots', type: 'black'} ],

logCat: function (catIndex) {

console.log('My cat\'s name is ' + this.catsList[catIndex].name);

console.log('It is a ' + this.catsList[catIndex].type + ' cat');

},

introduction: function() {

console.log('My name is ' + this.name);

console.log('I own ' + this.catsList.length + ' cats');

}

}

catOwner.logCat(0);

catOwner.logCat(1);

catOwner.introduction();

### Objects (V1.0.0): Comprehension

Each of these code blocks are connected. After each code block, write what the console output would be at that point in the program.

let catOwner = {

name: 'Debrah',

catsList: [ { name: 'Mr. Speckles', type: 'calico'},

{ name: 'Boots', type: 'black'} ],

logCat: function (catIndex) {

console.log('My cat\'s name is ' + this.catsList[catIndex].name);

console.log('It is a ' + this.catsList[catIndex].type + ' cat');

},

introduction: function() {

console.log('My name is ' + this.name);

console.log('I own ' + this.catsList.length + ' cats');

}

}

catOwner.logCat(0);

|  |
| --- |
|  |
|  |
|  |
|  |

catOwner.logCat(1);

|  |
| --- |
|  |
|  |
|  |
|  |

catOwner.introduction();

|  |
| --- |
|  |
|  |
|  |
|  |

catOwner.name = ‘Debbie’;

catOwner.introduction();

|  |
| --- |
|  |
|  |
|  |
|  |

catOwner.catsList.push( { name: ‘Mr. Snuggles’, type: ‘himalayan’ } );

catOwner.logCat(2);

|  |
| --- |
|  |
|  |
|  |
|  |

### Objects (V1.0.0): Application

In the lines below each code block, write a program that does what is asked in the code block.

/\* Write a program that:

- Creates a new object with the identifier “vehicle1”

- vehicle1 has a property which is a string object called “model”

whose value is “Cherokee”

- vehicle1 has a property which is an array object called “tirePressure”

whose values are 32,28, 16, and 8

- vehicle1 has a method called "checkPressure" that takes one argument

called "tireIndex"

- checkPressure() should check the value of tirePressure[] using

tireIndex as the index

- if the value is less then 26, log "Low pressure!" to the console

- otherwise, log "Pressure ok!" to the console

- Call vehicle1's checkPressure() method for each value in tirePressure

to check the pressure of every tire

\*/

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

# ***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. |  |

## NodeJS Web Server with Express.js Download from cloud [nodejs\_server]

This will help you through a basic NodeJS server. This will let you serve basic HTML web pages while using Express to handle your website’s addresses.

##### npm init

Running this command will collect information about your application and create the files needed to run it.

* The name of your application
* Starting version number
* Description of this application
* Entry point is the name of the js file that starts the application
* ‘test’ as the test command
* If you have a URL for a git repo, put it here
* keywords are tags that help others on NPM find your package (you can skip)
* Your name or company name as the author
* ‘MIT’ is the standard license

##### Create index.js and import Express.js

* Download and install Express with Node Package Manager:

npm install express

* Create a file called “index.js”
* Import Express.js:

**// import expressjs  
const express = require('express');**

* Create an express application object

// make an 'app' object from expressjs

const app = express();

##### Create a listen server

* Create a “PORT” constant. Remember, all **constants** names are all capital letters.

// network port to take connections on

const PORT = 3000;

* Use the app object’s listen() method to create a listen server (listens for connections to serve data back to). listen() takes two arguments. The first is the connection port the server will run on. The second is a callback function, which we will create right there in the arguments list.

//listen() 1st argument: network port taking web connections

//listen() 2nd argument: callback function to run after the server has started

app.listen(PORT, function() {

console.log("Listening on port " + PORT);

});

##### Create your first resource

In the **URL**, everything after your address and port is the **path**. By default, users will navigate to your **path root**, which is “**/**”. Let’s add code to handle a user using the **GET method** at the path “/”.

We do this using the app object’s **get()** method. get() takes two arguments. The first argument is the path the user would have to send their GET request to run this code. The second argument is a callback function to run once the connection is complete.

//get() 1st argument: path the user connects to

//get() 2nd argument: function to run after the connection is complete

app.get('/', function(req, res){

res.send("<h1>Hello World!</h1>");

});

Once the GET request is handled, it sends a **request** and **response** object to the callback function. We choose the names **req** and **res** for this. res has a method called send(). This will send() a string of HTML data back to the user and close the connection.

## Handling Query Parameters with ExpressJS Download from cloud [nodejs\_queryparams]

When making a GET request over HTTP, users can add **Query Paremeters** to the end of the URL in order to pass extra data to the server. Query parameters start after a single “**?**”, are separated by “**&**”, and use the “**parameterName**=value” format.

http://localhost:3000/home?firstName=Bob&lastName=Johnson

In the case above, the user sent a GET request to “/home” and sent the query parameters called firstName and lastName with the values “Bob” and “Johnson”. The get() method of our app object can parse these query parameters and save them as properties of the request object.

You can try it by adding this path handler to your NodeJS server, and using the URL above:

app.get('/home', function(req, res){

// Log the firstName and lastName query parameters to check their values

console.log(req.query.firstName);

console.log(req.query.lastName);

});

Notice that if you nagivate to “/home” without “?firstName=Bob&lastName=Johnson”, the console will log “null” instead. The user can choose whether to send any query parameter or not. This can be very problematic if your code expected specific query parameters and didn’t receive them. This is why we ***Never Trust The Client***.

Let’s use an **if statement** to check to make sure the client user is behaving correctly.

app.get('/home', function(req, res){

//If the firstName property has any valid value

if (req.query.firstName) {

//Then log it to the console so we can check it

console.log(req.query.firstName);

} else {

//If it's missing, log an error message

console.log('firstName query parameter was missing!');

}

//Repeat for lastName

if (req.query.lastName) {

console.log(req.query.lastName);

} else {

console.log('lastName query parameter was missing!');

}

});

You’ve probably noticed navigating to “/home” makes your browser appear to freeze up. This is because our code never sends a response to the user. Let’s remove our debugging log()s and build a response string instead:

app.get('/home', function(req, res){

// Make a response string of HTML to send back to the user

let responseString = '<h1>Hello';

// Check to see if firstName is not null

if (req.query.firstName) {

// Add a space and the value to the response string

responseString += ' ' + req.query.firstName;

}

// Repeat for lastName

if (req.query.lastName) {

responseString += ' ' + req.query.lastName;

}

//Finish responseString

responseString += '!</h1>'

// Send the response string

res.send(responseString);

});

Computer Science/Engineering

# 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()