Software Goals

Bic design scenarios

Information to be input :

SoftwareEngineering

WebApplications

Front end, Back end, and Full Stack Development

WebApplication Frameworks

FrameworkApplications

Basic Arithmetic and Logic

Logic and Syntax

Expressions

Equations

Statements

Operations

The Equals sign

Order of Operations

Order of Execution

Basic Programming

Computer Software

Flowcharts

Execut

ablesversusScripts

Syntax

JavaScript

DataS

tructures

Tokens

Statements

CodeandData

Objects

Data Types

Variables

Operators Unary Operators

Assignment Operators

Comparison operators Logical Operators

&&

|| !

Ternary operator

Null versus undefined

Expressions

Keywords

Truthy and Falsy Comments

Strings

Blocks

Scope

Hoisting

Keywords

Prototypes

Classes Objects

Constructors

Object Literals

Properties

Nested Objects

Destructuring

Control Flow Conditionals

If statement

If / Else statement

Else / If statement Switch statement

Functions and Methods

Function Namespace and access

Function Parameters andArguments

Functions as Data

Functions as Parameters Callbacks

Arrow Function Expressions

Methods

Invoking methods

The this keyword

The this keyword andArrow Functions Privacy

Callbacks Arrays

Type Coercion

Array Methods

Iterators

Loops

For loops

Nested Loops While loops

JavaScript Programming Tools

Packages

Libraries

API Frameworks

Modules

Asynchronous operations

Event Loops

Async /Await

Promise objects Handling asynchronous errors

Network Fundamentals

AdobeIllustratorExtendScript

Using ExtendScript to edit an Illustrator Document’s content

Selecting items within groups by index number

Selecting Items Within Groups By Group Name

Selecting text items by name

Selecting paths by name

Selecting items by Group

Selecting groups and sub-groups by name

Script UI

Windows

Containers

Executables versus Scripts

Syntax

# Software Goals

The application would have two parts:

* ***The*** order input section – This would be where all the order’s information is entered.
* ***The*** order status section – This would be a list of all the orders that are currently being worked on, and where they are at in the production process. This section would grab information from each order that is entered in the order input section.

Create a user interface, probably written in HTML, CSS, and JavaScript, where each graphic designer can track the orders that they are working on, and give notifications on when each order needs to be put on a gabarit.

The code compares the order’s ship date to the current date, and if it is within 48 hours, it changes the color of the order or gives some other notification that the order is coming due.

A search field would be available to be able to search for a particular order.

There can also be a way to show the status of each order:

* ***pending*** – The art is currently being worked on.
* ***proof*** created – The proof has been created, but not sent.
* ***proof*** sent – The proof has been sent.
* ***proof*** revision – The customer sent back changes to be made and a new proof sent.
* ***proof*** approved – The customer has approved the proof.
* ***screens*** created – Screens have been made from the art on the approved proof.
* ***gabarit*** created – The order has been put on a gabarit, but it hasn’t been sent out to production yet. The name of the gabarit is listed.
* ***gabarit*** sent – The gabarit that the order is on has been sent to production. The name of the gabarit is listed, as well as the date and time it was sent to production.
* ***hold*** – customer (art), customer (shipping), customer (payment), BIC (art), BIC (shipping), BIC (payment), BIC

(production)

## *Production Status*

* **In the lab**
* **In production**
* **In shipping**

## *UI interface tools*

* **drop-down menu**
* **radio buttons**
* **form fields with auto-fill**
* **checkboxes**
* **buttons**
* **listboxes**
* **tabbed panels**
* **progress bars**
* **adding and removing controls dynamically. This would be used for redrawing the window if there ismore than one item for the order. Each item would get its own inputs.**
* **drag and drop PDF files, display the file name, a preview image, and the email information it was sentattached to.**
* **drag and drop emails**

## *Gabarit processing*

This section would see how many screens each order has, and arrange them on a gabarit template to try to keep orders together. It would mark each order as having been put on a gabarit. It would then save the gabarit as a PDF and then email to production.

## Bic design scenarios

* **Multiple orders for the same company;**
* **with multiple items for each order;**
* **some of the items are sleeves;**
* **each item uses a different art file;**
* **some of the orders ship to different locations;**
* **some of the orders are rush, or have other special instructions;**

Information to be input :

### Customer information

* **Can the rest of the fields be autofilled when the business name is entered?**
* **Customer Business name.**
* **Customer Business address.**
* **Rep first name and last name.**
* **Rep email address.**
* **Rep phone number.**
* **Is the rep a VIP?**

### Order information

* **How many orders for this customer?**
* **Is each item to be shipped when it is ready, or is everything to be shipped together?**
* **Are there special instructions from production or customer service?**
* **Are there any special instructions from the customer?**
* **Is there an in-hands date?**

### For each order

* **How many items in this order? For orders with multiple items, multiple artboards can be created inIllustrator for each item in the order.Also, the name of each artboard can reflect its item number on theATS form.**
* **Is this order a rush?**
* **What is the shipping address for this order?**
* **Barcodes are able to be generated for orders.**

### For each item in the order

* **What are the names of the art files that were sent for this item, for sideAand side C?**
* **Is this item an exact repeat of one of our previous orders?**
* **Are the graphics to be sized to propotionally fill the printable area, or are there specific sizinginstructions?**
* **Are there any special shipping instructions for this item?**
* **Are there special production instructions for this item?**

### Standard process

* **What is the lighter body color?**
* **What are the ink colors?**
* **Is each ink color a standard color or a custom color? This can be indicated on the proof throughscripting.**
* **Is this item an exact repeat of a previous order?**
* **Is there a specified file to print from?**
* **Are any of the screens double-hit?**

### Test Scenario

3 orders with different JDE numbers and PO numbers. Each order has 4 items: 2 standard print items and 2 sleeves.

# Software Engineering

Software engineering is the creation, improvement, and maintainence of software that meets the requirements set out by a client. It encompasses many different areas including brainstorming, flow design, coding, debugging, hardware, and so on.

A software engineer designs and controls the software as a whole, and often collaborate with different specialists to satisfy the requirements of the application.

## Web Applications

A web application is software stored on a web server, and is accessed by the use of a web browser with an active internet connection. They are made using the client/server model, where the client is provided services through an off-site server hosted by a third party

## Front end, Back end, and Full Stack Development

Web applications are divided into two sections: the front end, which is what the user interacts with using the browser. It is referred to as client-side software. The back end, or server-side software, is the software that works behind the scenes and provides the functionality of the software.

A front end developer specializes in only the client-side software. Back end developers only work with the server-side software. Full-stack developers work with both.

## Web Application Frameworks

A web application framework is a generic template made to have specific code inserted to suit the designer’s needs.

## Framework Applications

A framework application is software designed to support the construction of web applications based on a single programming language.

Examples of framework applications include:

* **Angular.js**
* **EmberJS**
* **ReactJS**
* **Vue.js**

Web applications typically use three languages as a foundation:

* ***HTML*** – HTML is used to create the content of the page.
* ***CSS*** – CSS is used to format the content and control how it appears.
* ***JavaScript*** – JavaScript is used to make the content dynamic and interactive.

# Basic Arithmetic and Logic

Programming uses formal logic and mathematics, so the rules and methods governing these disciplines are required.

Logic and Syntax

Syntax is the set of symbols, rules, principles, and processes that govern the use of a language.

A programming language’s logic is the set of particular symbols, tokens, processes and principles used to achieve a specific goal in that language.

Expressions

An expression is a combination of symbols that represent a value. It is just a description of something.

|  |  |
| --- | --- |
| 1. | 2+3 |
| 2. | 5x-7+(3\*7) |

The above two examples are expressions.

Equations

An equation is a statement that asserts the equality of two expressions with the use of an equals sign.

|  |  |
| --- | --- |
| 1. | 2+3=5 |
| 2. | 5-6+(2\*4)=8 |

Statements

In computer programming, a statement is the smallest standalone element that can be executed by a computer language. In JavaScript, statements are separated with semicolons.

Operations

An operation is a function that performs a task upon an inputted value called an operand. An example of an operation that uses only one operand (A unary operation) is the increment operation. The increment operation takes the value of the operand and increases it by 1.

The most common type is the binary operation, in which the first operand has an operation performed upon it using the following operand.

The Equals sign

The equals sign is the symbol used to show that the expression on its left side is the same as the expression on its right side.

|  |  |
| --- | --- |
| 1. | 5 = x = y |

The statement above shows that 5 is the same as x, and that x is the same as y.

|  |  |
| --- | --- |
| 1. | 7 - 2 = 5 |

7 is the main operand, - is the subtraction operation, 2 is the second operand which is subtracted from the main operand. = means that the next value is the result of the operation.

The above statement is composed of four elements: The left-most value, operand 1, is the input that the operation is performed upon. The next symbol defines the type of operation, in this case subtraction. The subtraction operation requires two operands, the main (left) operand and the secondary (right) operand operand 2. The value of operand 2 is taken away from the value of operand 1. The fourth symbol indicates that the value to the right of it is the result of the operation

(5).

Order of Operations

Since operations can only be performed on two operands at a time, rules for which operations are evaluated first are required.

|  |  |
| --- | --- |
| 1. | 3 + 2 + 10 = n |

The default rule is to evaluate from left to right. In the above example, 2 is added to the 3 resulting in 5. 10 is then added to 5, resulting in 15.

|  |  |
| --- | --- |
| 1. | (3 + 2) + 10 = 15 |
| 2. | (5) + 10 = 15 |
| 3. | 15 = 15 |

Order of Execution

**Parentheses** - If there are parentheses within parentheses, evaluate them from the inner-most pair of parentheses outwards.

|  |  |
| --- | --- |
| 1. | 2 + (3 + (4 + 4) = n |
| 2. | 2 + (3 + 8) = n |
| 3. | 2 + 11 = n |
| 4. | 13 = n |

**Exponents** - Exponents are reduced after any operations within parentheses are evaluated.

Multiplication and Division - After operations within parentheses are evaluated, and then after the exponents are reduced, multiplication and division operations are done. When there are multiple operations, the order is to proceed from left to right.

|  |  |
| --- | --- |
| 1. | 3\*2/6 = n |
| 2. | 3\*2 = 6 |
| 3. | 6/6 = 1 |

**Addition and Subtraction** - When there are multiple operations, the order is to proceed from left to right.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | 60 - 30 + 5 | |
| 2. | | 60 - 30 = 30 | |
| 3. | | 30 + 5 = 35 | |
| 1. | | 22 x 4 + (3 + 1) = n | |
| 2. | | 22 x 4 + (4) = n | |
| 3. | | 22 x 4 + (4) = n | |
| 4. | | 4 x 4 + 4 = n | |
| 5. | | 16 + 4 = n | |
| 6. | | 20 = n | |

In the above example, even though addition and subtraction come last in the order of operations, the addition of 1 to 4 is done first because the operation is within parentheses, and any operation inside of parentheses comes first regardless of the type of operation it is.

Once all the operations within parentheses are evaluated, the next operation to be done is to reduce any exponents.

Once all the exponents are reduced, any multiplication and division operations are evaluated. When there is more than one multiplication or division operations in the statement, they are evaluated from left to right.

Once all the multiplication and division operations are done, then the addition and subtraction operations are performed, going from left to right.

Basic Programming

the selected programming language and outputs an executable file in machine code, which can then be run by the computer. An interpreter converts the A computer program is a list of instructions which are run one after another in a specific order to source code into machine code on the fly, as the script is run.

produce a desired result. There are two versions of any program: The source code, which is written by the programmers, and the machine code, a strictly numerical set of instructions that is run by the CPU.

Typically, a program’s source code must be converted into machine code by being compiled. The result of the source code’s compilation is a file called an executable. The executable file is the file that is launched by the user to perform specified tasks, as opposed to a data file, which is a file containing information that is to be used by an executable file or is produced by an executable file.

## Computer Software

Computer software is designed with a specific purpose. This purpose determines what programming language is chosen and the methods used to produce the desired result.

* ***Data*** - Data is the information that is manipulated by the code.
* ***Operation*** - An operation is a process used and the steps involved to manipulate the data.
* ***Output*** - Output is the result of the operations performed upon the data.

Flowcharts

A ***flowchart*** is a diagram that represents the sequence of possible steps performed by the code. They are used by the programmer to easily understand and organize the flow of the code operations.

There are flowchart programs available to aid in the creation and editing of flowcharts.

## Executables versus Scripts

The difference between an ***executable*** and a ***script*** is that executables are compiled into machine code before they can be run. Scripts are interpreted on the fly, meaning that they are converted into machine code as they are being executed.

A ***program*** is executed when the computer is given the command to begin performing the instructions in the code.

Syntax

The syntax of a computer language is the set of rules established by the language, which governs its functions and features. Each language has its own syntax, although different languages may share the same syntax overall.

Different syntaxes for different languages are stored in the computer’s memory. The syntax for a file is determined by the file extension the file is saved with. If a file has a file extension the computer doesn’t recognize, it will alert the user.

There are two parts of the programming process which involve the syntax of the language involved.

* **The *source code*, which is a text file written in a chosen programming language and identified using that language’s file extension in the file’s name.**
* **The *compiler* or *interpreter*. The compiler is a program which stores the rules and syntax of**

# JavaScript

Originally, JavaScript was developed to be one of the three computer languages to be used for web browsers and the internet. The first language, HTML, is for the content of a web page. The second language, CSS, is for how the content is displayed in the browser, its appearance, and the third, JavaScript is for user interactivity and dynamic behavior with web pages. This includes entering information, gathering information, and providing different results depending on the information entered by the user.

Despite the similarities in their names, JavaScript has nothing to do with the language Java. It was named JavaScript to take advantage of Java’s popularity at the time.

In addition to web page uses, JavaScript can be used to create script files, using the .jsx file format, which can be run from within Adobe Illustrator, to automate functions that cannot be done using Illustrator’s native Actions feature.

## Data Structures

A data structure is a particular way of organizing different data types and defining how they interact with each other.

|  |  |
| --- | --- |
| 1. | let alpha = [10, 20, 30, 40, 50]; |

In the code above, the variable **alpha** is a data structure called an array. Each individual piece of data is called an element. An array, in JavaScript, is enclosed in a pair of brackets, with each element separated with a comma. The data structure is the way the data is stored. In this case, the data structure is the opening and closing brackets and commas which are used to organize the data elements. Arrays, like other data types, have a specific set of rules that define how they are used.

## Tokens

A ***token*** is the most basic element of a programming language. A token can be a single letter, a word, or a symbol.

## Statements

In programming, a statement is an instruction to perform a specific action

Each statement begins at the left side of the line and ends with a semicolon. A statement may be long enough to be bumped down into the next line of the display, but statements only end with a semicolon.

Statements are read from left to right, and from top to bottom.

|  |  |
| --- | --- |
| 1. | 5x; |
| 2. | let alpha = 10; |
| 3. | bravo (); |

The above example shows three statements. Blocks of code are two or more statements that work sequentially as a single unit.

## Code and Data

Programming deals with two different kinds of information: Code and data.

The code defines the program’s behavior, and the data is the information the code uses to

produce a result.

Objects

Different operations can be performed only on certain kinds of data. Numbers can be subtracted from each other, but words can’t be.

A data type is a way to organize different kinds of data which can have the same set of operations performed upon them.

Since variables can be assigned different types of values, conflicts can occur when operations are performed upon variables with different data types.

Data Types

Different operations can be performed only on certain kinds of data. Numbers can be subtracted from each other, but words can’t be.

A data type is a way to organize different kinds of data which can have the same set of operations performed upon them.

Since variables can be assigned different types of values, conflicts can occur when operations are performed upon variables with different data types.

* ***Value*** - A value is a fundamental piece of data that represents itself, as opposed to a variable, which is an entity which is assigned a value. There are several different types of values in JavaScript.
* ***Number*** - Whole numbers and fractional numbers using decimals.
* ***String*** - A character or group of characters surrounded by either opening and closing single or double quotes.
* ***Boolean*** - A Boolean value can be only either true or false. They can be thought of the answer to a yes or no question.
* ***Null*** - The null type is used to represent a value that is unknown and is represented by the keyword

null.

* ***Undefined*** - The undefined type is represented by the keyword undefined and represents the absence of a value. It is like null but has different uses.
* ***Symbol*** - Symbols are unique identifiers.
* ***Object*** - An object type is an entity that has been given one or more properties and can have one or more methods. A property is a key : value pair, and describes some aspect of the variable. A method is something that the variable can do.

## Variables

***Variables*** are objects which use characters to represent values that are assigned to them. There are two actions involved with variables, which are frequently combined in a single statement.

JavaScript can be thought of as an assembly line. The workers perform actions on the data that comes to them on the assembly line. There are three things that are needed to get them the raw materials they need to produce a final product.

* **A box to hold the information;**
* **A name for each box, so that it can be sent to the correct worker and one that the worker canspecifically ask for;**
* **The material that is put in the box.**

Both creating the box and naming it are done in one step: declaring the variable. This is done by using one of the variable keywords of ***var***, ***let***, or ***const***.

Putting the material in the box is done by initializing the variable. This assigns a value to the variable by using an equals sign.

If a variable is declared (created and named), but is not assigned a value, it is like an empty box.

The value of the variable is returned as undefined, since the box is empty.

|  |  |
| --- | --- |
| 1. | var thing; |

In the example above, the variable ***thing*** is declared by using the var keyword.

|  |  |
| --- | --- |
| 1. | var thing; |
| 2. | thing = 5; |

In the example above, the word ***thing*** is being declared a variable by the keyword var on the first line, and on the second line the variable ***thing*** is initialized by assigning it the value of 5 using the equals sign.

This can be done in one step, however:

|  |  |
| --- | --- |
| 1. | var thing = 5; |

In the example above, the word ***thing*** is declared a variable by using the keyword ***var***, and is assigned the value of 5 by using the equals sign.

|  |  |
| --- | --- |
| 1. | var thing = alpha + beta; |

In the example above, the variable ***thing*** is being assigned the value of the variables ***alpha*** and ***beta*** added together.

This combining of the two different steps is a concept only within JavaScript, and can create issues. It is best to declare the variable first, then initialize it separately on the next line.

It is considered good practice to create variable with words that describe what they are doing, to help the programmer keep track of what is going on in the code.

## Operators

An ***operator*** is a symbol that represents an action performed upon data. There are three kinds of operators:

* ***Unary*** - unary operators are operations that require only a single operand to act upon. Examples include the Increment and Decrement operators (++) and (--).
* ***Binary*** - Operations that require two operands to act upon. Examples include 4 - 3 = 1. The subtraction operator subtracts the second operand from the first.
* ***Ternary*** - Also known as a conditional expression, it is the only JavaScript operator that requires three operands: The first is a condition, which compares two values. The second operand is the value which is assigned to the expression if the condition is true. The third operand is the value assigned to the expression is the condition is not true. For example,

|  |  |
| --- | --- |
| 1. | variableName = (condition) ? value1: value2; |
| 2. | voteable = (age < 18) ? “Too young.” : “Old enough.”; |

The condition is the variable age less than 18? If it is, then the expression is given the value “Too young.”. If age is not less than 18, then the expression is given the value “Old enough.”

Unary Operators

Unary operators need only one operand to function.

***++*** Postincrement .............***x++*** .... This increases the value of x by 1 AFTER using its value in the statement.

***++*** Preincrement ...............***++x*** .... This increases the value of x by 1 BEFORE using its value in the statement.

***--*** Postdecrement .............x-- ..... This decreases the value of x by 1 AFTER using its value in the statement.

***--*** Predecrement...............--***x***..... This decreases the value of x by 1 BEFORE using its value in the statement.

The difference between the ***POST*** and ***PRE*** operators is shown below.

|  |  |
| --- | --- |
| 1. | let x = 10 |
| 2. | y=x++ |

***POST*** operators increase the value of what they are operating on AFTER it is used everywhere else in the statement.

The code above uses the postincrement operator on x. It increases the value of x AFTER assigning the value of x to y. This yields y=10 and x=11.

|  |  |
| --- | --- |
| 1. | let x=10 |
| 2. | y=x— |

The code above yields y=10 and x=9.

***PRE*** operators increase the value of what they are operating on BEFORE being used anywhere.

|  |  |
| --- | --- |
| 1. | let x=10 |
| 2. | y=++x |

The code above yields y=11, x=11.

|  |  |
| --- | --- |
| 1. | let x=10 |
| 2. | y=--x |

The code above yields y=9, x=9.

## Assignment Operators

Assignment operators assign values to variables, or give the variables new values by taking the original value of the variable and changing it in some way, and then giving the variable the result as a new value.

***Operator ...... Example...... Same as***

**=** ................... x = y...............x = y

+=.................. x+=y................x=x+y.......This ***adds*** a value to variable x.

**-=** .................. x -= y...........x = x – y.....This ***subtracts*** a value from variable x.

**\*=** .................. x \*= y .......... x = x \* y .....This ***multiplies*** variable x by a value.

**/=** .................. x /= y........... x = x / y .....This ***divides*** variable x by a value.

**%=** ................ x %= y........ x = x % y ....This divides variable x by a value, and the ***remainder*** is assigned to x.

**\*\*=**................. x \*\*= y .........x = x \*\*y.....This raises the value of variable x to the ***power*** of value

## Comparison operators

A comparison operator is a symbol which compares the value to the left of the symbol with the value to the right of the symbol. The condition can only be true (yes) or false (no).

* ***A*** == B - ***Equal to.*** If A equals B, then the result is true. If it is not the same, then the result is false. 5 == ‘5’ returns a result of true, since both values are the same, regardless of type.
* ***A*** === B; - ***Equal to in both value and type.*** If A equals B in both value and type, then the result is true. If it is not the same, the result is false. 8===’8’ returns a result of false. The value of eight is equal, but one value is a number and the other is a string.
* ***A*** != B; - ***Not equal.*** 5 != 10 returns a value of true.
* ***A*** !== B - ***Not equal, in either value or type.*** 5 !== 10 returns a value of true. 5 !== ‘5’ returns a value of true. 5 !== 5 returns a value of false.
* ***A*** > B - ***Greater than.*** If A is greater than B, then the result is true. If it is not greater than B, then the result is false.
* ***A*** >= B - ***Greater than or equal to.*** If A is greater than or equal to B, then the result is true. If it is not greater than or equal to B, then the result is false.
* ***A*** < B; - ***Less than.*** If A is less than B, then the result is true. If it is not less than B, then the result is

false.

* ***A*** <= B - ***Less than or equal to.*** If A is less than or equal to B, then the result is true. If it is not less than or equal to B then the result is false.

## Logical Operators

In JavaScript, logical operators do not always return a Boolean value, but the value of one of the operands:

&&

The ***AND*** operator evaluates two expressions. If the first expression is converted to falsy, it returns the first expression, otherwise it returns the value of expression 2. When used with Boolean values, it returns a value of true when both operands are true. If one of them is false, it returns a value of false.

|  |  |
| --- | --- |
| 1. | var alpha = 2; |
| 2. | var bravo = 1; |
| 3. | alert (alpha && bravo); |
| 4. | The code above returns a value of 1 (expression 2), since expression 1 did not evaluate as falsy. |
| 5. 6. | var alpha = 0; |
| 7. | var bravo = 7; |
| 8. | alert(alpha && bravo); |

The code above returns a value of 0 (expression 1), since expression evaluates to falsy.

||

The ***Or*** operator evaluates two expressions. If expression 1 converts to truthy it returns the value of expression 1, otherwise it returns the value of expression 2. When used with Boolean values, it returns a value of false if both operands are falsy. If one or both expressions are truthy, it returns a value of true.

|  |  |
| --- | --- |
| 1. | alpha = 2; |
| 2. | bravo = 7; |
| 3. | alert(alpha || bravo); |
| 4. | The code above returns a value of 2 (expression 1), since expression 1 is truthy. If expression 1 was falsy, it would return the value of 7 (expression 2). |
| 5. 6. | alpha = false; |
| 7. | bravo = 7; |
| 8. | alert(alpha || bravo); |

The code above returns a value of 7 (expression 2), since expression 1 is falsy.

!

The ***NOT*** or ***BANG*** operator. The bang operator will reverse the value that is returned. A true value will be turned into false, and a false value will be turned into a true value.

|  |  |
| --- | --- |
| 1. | let a = true; |
| 2. | let b = false; |
| 3. | a &&!b; |

This will return the value of the first operand, a. The ***And*** operator requires both conditions to be true. Variable ***a*** is true. Variable ***b*** is false, but the exclamation point before it in the ***AND*** evaluation is the bang operator, which reverses the value of the variable it is placed before. B has a value of false, which is reversed by the bang operator. So, both conditions are true, and thus the ***AND*** evaluation is true.

## Ternary operator

The ternary operator provides a simpler way to provide the same functions as the if/else statement.

|  |  |
| --- | --- |
| 1. | condition ? expression 1 : expression 2; |

The condition that is evaluated to be either true or false is before the question mark. If the condition is true, then expression 1 is executed. If it is not true, then expression 2 is executed. The two expressions are separated by a colon.

|  |  |
| --- | --- |
| 1. | var alpha = (5==5) ? 10 : 200 |

In the example above, the variable ***alpha*** is declared. The condition within the parentheses, 5==5, is true, so the first value after the question mark, 10, is assigned to ***alpha***. If it isn’t true, than the value after the colon is assigned to ***alpha***.

## Null versus undefined

Null and undefined are keywords which represent the absence of a value. Other than this, they are completely different.

Null is a value that can be assigned to a variable that represents ‘no value’.

|  |  |
| --- | --- |
| 1. | var alpha = null; |

In the code above, the variable alpha has been declared and assigned a value that represents an empty value. It is not zero, since zero is an actual number. undefined means that a variable has not been defined.

|  |  |
| --- | --- |
| 1. | var alpha; |
| 2. | console.log(alpha); |

In the code above, the variable alpha has been declared, but not assigned a value. When the value is logged to the console, the response is a message that the variable hasn’t been assigned a value, or is undefined.

## Expressions

An expression is a statement that results in a value.

## Keywords

Keywords are words defined by the language’s syntax. They have special functions and cannot be used for anything else.

## Truthy and Falsy

In JavaScript, a truthy value is a value that translates to true when evaluated in a Boolean context. All values are truthy unless they are defined as falsy. There are very few falsy values, so it is often easier to remember the falsy ones:

* **false**
* **null**
* **undefined**
* **0**
* **NaN**
* **‘’, “”, `` (empty strings)**
* **document.all**
* **0n - BigInt**
* **-0**

## Comments

Comments are notes within the code meant to be read by programmers and are ignored when the code is executed. There are two kinds of comments: a single line comment, which is text preceded by two forward slashes (), and the other is a multi-line comment, which is text preceded by forward slash – asterisk (/\*) and ended with asterisk – forward slash (\*/).

|  |  |
| --- | --- |
| 1. | // This is a single line comment. |
| 2. 3. | /\* This is a multi-line comment. |
| 4. | It can go on and on until the |
| 5. | Ending token is reached. \*/ |

Strings

A string is the method used to store text. Text is a series of characters that has no function, meaning, or significance to the code. The words, numbers, and symbols stored as text play no part in the code’s execution. Text only has meaning to the user.

|  |  |
| --- | --- |
| 1. | var alpha = 2 + 2; |
| 2. | console.log(alpha); |

The above code will return the result:

***4***

|  |  |
| --- | --- |
| 1. | var alpha = “2 + 2”; |
| 2. | console.log(alpha); |

The above code will return the result:

***2 + 2***

When a series of characters is treated as text, it is no longer code to be executed.

### Template Literal

A template literal is new functionality that has been added to JavaScript ES6 which makes working with strings more readable, and adds new capabilities.

A character or group of characters contained with either opening and closing single or double quotes is a *string*. A template literal is a character or group of characters contained with ***backticks*** (also called a grave).

* ***String*** – ‘This is a string.’
* ***String*** – “This is a string.”
* ***Template*** Literal - `This is a template literal.`

The token **${}** can be used within the template literal as a placeholder for other values, variables, or expressions.

|  |  |
| --- | --- |
| 1. | const alpha = 5; |
| 2. | const bravo = 7; |
| 3. | const charlie = alpha + bravo; |
| 4. | console.log(“here is the result” + charlie); |

The code above is written using the older syntax for working with strings.

|  |  |
| --- | --- |
| 1. | const alpha = 5; |
| 2. | consta bravo = 7; |
| 3. | console.log ( `here is the result ${alpha + bravo} !`); |

The above code uses the new string literal to combine a string with the values for alpha and bravo in with the string. It logs the string ***Here is the result 12***! to the console.

The backticks start and end the template literal, instead of using single or double quotes.

### Interpolation

The curly braces with the dollar sign in front provide a way to insert a variable or expression into a string.

|  |  |
| --- | --- |
| 1. | const alpha = “test”; |
| 2. | const bravo = `something ${alpha}`; |

The code above produces the string ***Something Test***

|  |  |
| --- | --- |
| 1. | const alpha = `something ${1+2+3}` |

The above code returns the string

***Something 6***

|  |  |
| --- | --- |
| 1. | const alph = `something ${dosomethingfunction () ? “x”: “y”}` |

The code above returns the string

***Something xxxxx***

Where ***xxxxx*** is the result of the function ***Dosomethingfunction***, which is added to the string something.

Template literals essentially replace using + when inserting a variable’s value into a string.

|  |  |
| --- | --- |
| 1. | let x = 5; |
| 2. | console.log(“hello world “ + x + “times.”); |
| 3. | console.log(`hello world ${x} times.`); |

In the code above, x is given a value of 5. The string logged to the console is:

***Hello world 5 times.***

The second line is uses the older syntax with concantenation. A single pair of backticks contain all the strings. There is a ***${}*** set for each variable.

|  |  |
| --- | --- |
| 1. | let x = 5; |
| 2. | let y = 202; |
| 3. | console.log(`hello, world ${x} ${y} times.`); |

The above code logs the string: ***Hello, world 5 202 times.***

|  |  |
| --- | --- |
| 1. | let x = 5; |
| 2. | let y =10; |
| 3. | console.log(`hello, world ${x + y} times.`); |

The above code returns the string:

***Hello, world 15 times.***

The expression within the curly braces is executed, and the result is converted to a string and inserted into the final string.

String methods

Finding the number of characters in a string

.length

Extracting parts of a string

.slice(start, end)

.substring(start, end)

.substr(start, length)

Replacing the contents of a string

.replace(“old word”, “new word”)

Converting to upper or lower case

.toUpperCase()

.toLowerCase()

To join two or more strings

.concat()

+ operator

Remove whitespace from both sides of a string

.trim()

Extracting string characters

.charAt(index)

.charCodeAt(index)

Splitting a string of words into an array

.split(“ character for the location of each split “)

string.split(“ “) // The character between the quotes is a white space. The whitespace is where each element ends and a new one begins.

.string.split(“-“) // The character between the quotes is a dash. The dash is where each element ends and a new one begins.

Capitalizing the first letter of every word in a string

## Blocks

A block is the code found within a set of curly braces. Blocks help to organize the code, and they serve as an important structural marker.

|  |  |
| --- | --- |
| 1. | { |
| 2. | var a = “This is a string.”; |
| 3. | var b = “This is another string.”; |
| 4. | var c = a + b; |
| 5. | console.log (c); |
| 6. | } |

## Scope

Scope defines where a variable can be accessed from in relation to the code that is accessing it.

* ***Global*** scope – When a variable is declared outside of any block, it is a global variable. Global variables can be accessed from anywhere in the code, including inside any code block.
* ***Block*** scope – When a variable is declared within a block, it can only be accessed by other code within that block. Any code outside of the block that attempts to access it throw a Reference Error.

Variables should be block scope whenever possible. That is, declared within the function that uses them. Globally scoped variables must be held in memory for the duration of the code execution, while block-scoped variables are loaded into memory only when the function they are contained in is run. Globally-scoped variables can also have the same name as block-scoped variables, which can cause conflicts.

## Hoisting

When JavaScript is executed, the interpreter goes through the code twice. The first time it checks the syntax of the code, and then optimizes it. The second time is when it actually executes it.

During the first run, the interpreter takes all of the variable declarations and function declarations and puts them at the top of their scope orders. This is ***hoisting***.

Problems arise whenever the declaration and initialization of a variable is combined as a shortcut. Since hoisting moves the declaration but not the initialization of the variable.

|  |  |
| --- | --- |
| 1. | var a = 5; |

When the code is executed, this statement will be broken apart.

|  |  |
| --- | --- |
| 1. | var a; |
| 2. | a=5; |

This can cause problems because the order of the statements is not correct.

|  |  |
| --- | --- |
| 1. | console.log (bar); |
| 2. | var bar = ‘bar’; |
| 3. | console.log (bar); |

When the code is executed, the interpreter rearranges the statements into this:

|  |  |
| --- | --- |
| 1. | var bar; |
| 2. | console.log (bar); |
| 3. | bar = ‘bar’; |
| 4. | console.log (bar); |

In the above code, the variable *bar* is declared, but is not assigned a value. When the interpreter gets to the next line, it will give an undefined error, since bar has been assigned a value of undefined by default.

|  |  |
| --- | --- |
| 1. | { |
| 2. | var a = “This is a string.”; |
| 3. | var b = “This is another string.”; |
| 4. | var c = a + b; |
| 5. | console.log (c); |
| 6. | } |
| 7. 8. | { |
| 9. | var a = “this is a string.”; |
| 10. | var b = “this is another string.”; |
| 11. | var c = a + b; |
| 12. | console.log (c); |
| 13. | } |

In the above example of some previously used code, the code has been rewritten in a way which is more like the way it would be executed by the interpreter. Since the variables are declared within a block, they will be hoisted to the top of the block.

## Keywords

***Keywords*** are words which are reserved by JavaScript to mean certain things, and they cannot be used for any other purpose, and include words like let, var, if, and others.

## Prototypes

Prototypes are the mechanisms by which a JavaScript object inherits features from its precursor. Some properties of objects are custom made for the object, but objects also contain a number of preset properties. The source of these properties is the prototype object.

|  |  |
| --- | --- |
| 1. | let person1 = new person (); |
| 2. | person1.valueof(); |

In the code above, the variable ***person1*** is declared as a ***new Person()*** object. The object ***person1*** inherits all the properties of the object ***Person()***, which inherits all its properties from the ***Object()***.

These properties are not copied from one hierarchy to another, but rather the code goes up the hierarchy to the prototype master object.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | function alpha() { | |
| 2. | | this.name = “john”; | |
| 3. | | this.gender = “male”; | |
| 4. | | } | |
| 5. 6. | | var bravo = new alpha(); | |
| 7. | | bravo.age = 15; | |
| 8. | | alert(bravo.age); | |
| 9. 10. | | var charlie = new alpha(); | |
| 11. | | alert(charlie.age); | |

In the code above, the function ***alpha*** creates a prototype object.

A variable named ***bravo*** is created, which is an instance of the object ***alpha***. A property is added to ***bravo*** called **age**, and is assigned a value. When the variable ***bravo*** is called as an alert, it returns the value the age property was given, 15.

Another instance of the ***alpha*** object is created and named ***charlie***. When the age property that was assigned to ***bravo*** is called, it returns a result of undefined. This is because the age property was defined only on the ***bravo*** instance, and is not inherited to any other instance of the ***alpha*** object that may be created.

To create a property that is shared by all instances of the object, the **Prototype** object is used.

|  |  |
| --- | --- |
| 1. | function alpha() { |
| 2. | this.name = “john”; |
| 3. | this.gender = “male”; |
| 4. | } |
| 5. 6. | alpha.prototype.age = 15; |
| 7. 8. | var bravo = new alpha(); |
| 9. | alert(bravo.age); |
| 10. 11. | var charlie = new alpha(); |
| 12. | alert(charlie.age); |

In the code above, the object ***alpha*** is created with two properties. Outside of alpha’s code block the property ***age*** is added using the prototype keyword. Now every instance created of the ***alpha*** object, the ***age*** property is available without having to create it for every instance of the object.

## Classes

A class is a template for an object, grouping both data and behaviors into a single unit. It includes generic properties and functions common to all its instances.

|  |  |
| --- | --- |
| 1. | class car { |
| 2. | color: |
| 3. | manufacturer: |
| 4. | make: |
| 5. | model: |
| 6. | } |

The class ***car*** would have properties that all cars would have.

Classes are master blueprints for objects. Many copies of a class can be created and used for different, specific purposes. A copy of a class to be used is called an instance, where the properties and functions are populated with data.

### Constructors

A constructor is a method which creates and initializes an object. Keywords like ***var***, ***let***, and

***const*** are constructors. A function like ***function Alpha (a, b, c)*** is an object constructor function.

A function is a block of code that produces a result or performs an action. A method is a function which is associated with an object.

|  |  |
| --- | --- |
| 1. | class alpha { |
| 2. | constructor(name) { |
| 3. | this.name = name; |
| 4. | this.behavior = 0; |
| 5. | } |
| 6. | } |

In the code above, the name of the class is ***alpha***.

Every time a new instance of the class is created, the constructor() method is invoked.

The constructor method accepts one argument, in this case, ***name***.

Inside of the constructor method, the ***this*** keyword is used to create two properties of ***name*** and ***behavior***. The value of ***name*** passes the argument from the parameter ***name*** inside the method’s parentheses to the property ***this.name***.

The property ***this.behavior*** is given the value of ***0***.

|  |  |
| --- | --- |
| 1. | class bravo { |
| 2. | constructor (year, make, model) { |
| 3. | this.yearmade = year; |
| 4. | this.carmake = make; |
| 5. | this.carmodel = model; |
| 6. | } |
| 7. | } |

In the code above, the class ***bravo*** is created. The constructor method gives it three parameters, ***year***, ***make***, and ***model***. Within the method’s code block three properties are created using the ***this*** keyword, and assigned the parameters that were created in the parentheses of the constructor method.

Now, every time an instance of this class is created, it will have the three properties of year, make, and model.

### Instances of Classes

A class is the template or master design of an object. When a copy of the class is created to be used with specific values, the copy is referred to as an instance of the class. The class is the design of the object.

Classes specify the shared properties and methods that all the objects created from the class will have.

### Static Methods

A static method is a method that is called on the class itself, not on the individual objects instantiated from the class. They use the static keyword.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | class alpha { | |
| 2. | | constructor (name) { | |
| 3. | | this.\_name = name; | |
| 4. | | } | |
| 5. | | introduce() { | |
| 6. | | console.log(`this is ${this.\_name} !`); | |
| 7. | | } | |
| 8. | | static charlie() { | |
| 9. | | console.log(“woof!”); | |
| 10. | | } | |
| 11. | | const bravo = new alpha(“toaster”); | |
| 12. | | bravo.introduce(); // this calls the method introduce, and returns the string this is toaster!. | |
| 13. | | alpha.charlie(); //this calls the static method charlie and logs the string woof! to the console. | |

### The New keyword

An instance of a class is created using the ***new*** keyword.

|  |  |
| --- | --- |
| 1. | class alpha { |
| 2. | constructor (input1) { |
| 3. | this.color = input1; |
| 4. | }; |
| 5. | var bravo = new alpha (“blue”); |
| 6. | console.log(bravo); |

In the code above, the class alpha is created. The object bravo is made by instantiating the alpha class using the keyword new, and then any arguments specified in the class.

### The extends keyword

The ***extends*** keyword is used on an instance of a class to add properties and methods to that particular instance in addition to the properties and methods that already exist in the class.

|  |  |
| --- | --- |
| 1. | class alpha { |
| 2. | constructor (color, material) { |
| 3. | this.color = color; |
| 4. | this.material = material; |
| 5. | }; |
| 6. 7. | class bravo extends alpha { |
| 8. | constructor (color, material, size ) { |
| 9. | super(color, material) |
| 10. | this.size = size |
| 11. | }; |
| 12. | }; |
| 13. 14. | var charlie = new bravo(“blue”, “steel”, “large”); |

In the code above:

***class bravo extends alpha {*** //the keyword class creates a new class object named bravo. The

extends keyword allows access to the parent’s methods.

***constructor (color, material, size ) {*** //this constructor is called when a new bravo object is created, and accepts the parameters specified within the parentheses of color, material, and size).//

***super(color, material)*** //The super keyword calls the constructor of the parent class, and allows

access to the parent class’s parameters. The parentheses contain the parent class’s parameters. It is considered best practice to put the super() method call on the first line of subclass constructors to avoid reference errors.

|  |  |
| --- | --- |
| 1. | this.size = size //this is a new property unique to the bravo class object. |
| 2. | }; |
| 3. | }; |

The class ***bravo*** is then created, and using the ***extends*** keyword and the name of the class to be extended, adds properties to the ***alpha*** class to this new class.

In ***bravo***, the constructor method contains all of the parameters. The parent class’s parameters as well as the new ones that are being added.

### The Super() method

The super keyword calls the constructor of the parent class, making its parameters available to new, child class.

After the ***super*** method, the new properties and/or methods are added as they are listed within the ***bravo*** constructor’s parenetheses.

### Inheritance

Different classes can share several properties or methods. A way to reduce redundancy is through the use of inheritance.

|  |  |
| --- | --- |
| 1. | class animal { |
| 2. | constructor(name) { |
| 3. | this.\_name = name; |
| 4. | this.\_behavior = 0; |
| 5. | } |
| 6. 7. | get name() { |
| 8. | return this.\_name; |
| 9. | } |
| 10. 11. | get behavior() { |
| 12. | return this.\_behavior; |
| 13. | } |
| 14. 15. | incrementbehavior() { |
| 16. | this.\_behavior++; |
| 17. | } |
| 18. | } |

### Superclass and Subclass

The term superclass refers to the class being inherited from, also known as a parent class. The term subclass refers to a class that inherits properties and methods from another class. It is also known as a child class.

|  |  |
| --- | --- |
| 1. | class Ship { |
| 2. | constructor(type, mission, crew) { |
| 3. | this.type = type; |
| 4. | this.mission = mission; |
| 5. | this.crew = crew; |
| 6. | } |
| 7. | } |
| 8. | class Freighter extends Ship { |
| 9. | constructor (type, mission, crew, maxCargo) { |
| 10. | super(type, mission, crew); |
| 11. | this.maxCargo = maxCargo; |
| 12. | } |
| 13. | } |
| 14. | class warFreighter extends Freighter { |
| 15. | constructor (type, mission, crew, maxCargo, weapons) { |
| 16. | super (type, mission, crew, maxCargo); |
| 17. | this.weapons = weapons; |
| 18. | } |
| 19. | } |
| 20. | var Typhus = new warFreighter ("heavy", "long-range", "200", "40,000 tons", "full armed"); |
| 21. | alert(Typhus.mission); |

In the code above, three class objects are created, named ***Ship***, ***Freighter***, and ***warFreighter***. The class ***Freighter*** is a child class of the class ***Ship*** and uses the ***extends*** keyword to add properties to the ones it inherits from the ***Ship*** class. Relative to the class ***Freighter***, it is the subclass of ***Ship***, since it inherits ***Ship’s*** properties and methods. The class ***Freighter*** is the superclass of the class ***warFreighter***, since ***Freighter*** passes its properties and methods to the class ***warFreighter***.

### The this keyword

The ***this*** keyword refers to the object that it belongs to.

* ***This* used in a method refers to the owner object.**
* ***This* used alone refers to the global object.**
* ***This* used in a function refers to the global object, allowing access to objects outside of the function’s scope.**
* ***This* used in a function in strict mode returns undefined.**
* ***This* used in an event refers to the element that received the event.**
* ***This* used in methods like call(), or apply(), can refer to any object.**

Normally, only the objects that exist within a function can be called by objects within that function due to scope. To be able to use variables or other objects that exist outside of the function, the This keyword is used.

## Objects

In Javascript, an ***object*** is a character or group of characters that does something when the code is run. JavaScript is based on objects as the core of the language, so almost everything is an object. An object is an instance of a class, which contains multiple variables and their values. Where variables can be assigned only a single value, objects can be given many pairs of variables and their values.

Constructors

A constructor is a method which creates and initializes an object. Keywords like ***var***, ***let***, and ***const*** are constructors. A function like ***function Alpha (a, b, c)*** is an object constructor function.

Object Literals

Object literals is a list of name:value pairs separated with commas. The whole list is contained within a set of curly braces. The properties given to objects and their values are object literals.

Properties

***Properties*** are different variables which have been assigned values and are associated with an object. They can be pre-made by JavaScript, or they can be created by the programmer.

|  |  |
| --- | --- |
| 1. | var car = { |
| 2. | make: “ford”, |
| 3. | model: “mustang”, |
| 4. | year: “1970” |
| 5. | }; |

The value for the property *make* for the object *car* is ***ford***.

|  |  |
| --- | --- |
| 1. | car.make = “ford” |

The value for the property *model* for the object *car* is mustang.

|  |  |
| --- | --- |
| 1. | car.model = “mustang” |

The value for the property *year* for the object *car* is 1970.

|  |  |
| --- | --- |
| 1. | car.year = “1970” |

In the example above, the variable ***car*** is created and declared. The properties of the variable are the block that is within the curly braces. The name of the variable, or ***property*** as it is called here, is the word before the colon. What comes after the color is the ***value*** of the property. It is like the word make is a variable, but instead of declaring it with ***let***, ***var***, or ***const***, it is declared by being inside the curly braces and it comes before a colon. Instead of the value being assigned to the variable with an equals sign, it is assigned using the colon and ending with a comma.

These properties are referred to by using a period, called dot notation, to specify the property of

the object.

|  |  |
| --- | --- |
| 1. | car.make; this would return a value of “ford” |

An object can function as an array of properties, which are accessed by using dot notation.

|  |  |
| --- | --- |
| 1. | var car = { |
| 2. | year:”2020”, |
| 3. | make:”Ford”, |
| 4. | model:”Fusion”, |
| 5. | } |

Nested Objects

Nested objects are objects that have more than one subset of properties.

|  |  |
| --- | --- |
| 1. | var spaceship = { |
| 2. | classtype: “frigate”, |
| 3. | displacement: “4000 tons”, |
| 4. | armor: light, |
| 5. | crew: { |
| 6. | captain: { |
| 7. | firstName: “Tracy”, |
| 8. | lastName: “McAnna”, |
| 9. | age: 31, |
| 10. | }; |
| 11. 12. | firstOfficer: { |
| 13. | firstName: “Daniel”, |
| 14. | lastName: “McKloy”, |
| 15. | age: 22, |
| 16. | } |
| 17. | } |
| 18. | }; |

In the code above, the object ***spaceship*** has four properties:

* **classType**
* **displacement**
* **armor**
* **crew**

However, the property ***crew*** has two properties of its own, ***captain*** and ***firstOfficer***. Each of these sub-properties also has each its own properties: ***firstName***, ***lastName***, and ***age***.

To reference each of these properties, the dot notation is used. ***spaceship*** references the whole object.

***spaceship.classType*** returns the value ***“frigate”***. ***spacehip.armor r***eturns the value ***“light”***. ***spaceship.crew*** returns the objects ***captain*** and ***firstOfficer***. ***spaceship.crew.captain*** returns the objects ***firstName***, ***lastName***, and ***age***. ***spaceship.crew.captain.age*** returns the value of ***31***.

### Pass By Value / Pass By Reference

In JavaScript, everything is an object except for the primitive data types, which includes, numerals, strings, Booleans, null, and undefined. These primitive data types are immutable, which means that once they are created, they cannot be modified.

The difference between primitive values and objects is that primitive values are assigned to variables using the = symbol as pass by value, where objects are assigned to variables using the = symbol as pass by reference.

|  |  |
| --- | --- |
| 1. | var name = “joe”; |
| 2. | var firstname = name; |
| 3. | name = “bob”; |
| 4. | console.log(name); // “bob” is returned. |
| 5. | console.log(firstname); // “joe” is returned. |

The code above demonstrates pass by value, since strings are primitive values. Pass by value means that a variable’s value ***can’t*** be changed by another variable.

In the code above, the primitive value of “Joe” is assigned to the variable ***name***. The second line then creates a new variable ***firstName***, and assigns it a copy of ***name’s*** value (“Joe”). This means that the value of the new variable ***firstName*** is independent of ***name***, and anything that happens to ***name’s*** value doesn’t affect ***firstName*** at all.

The third line of the code changes the value of the variable ***name*** to “Bob”.

|  |  |
| --- | --- |
| 1. | var name = { |
| 2. | firstname: “joe” |
| 3. | }; |
| 4. 5. | var identity = name; |
| 6. | name.firstname = “bob” |
| 7. 8. | console.log(name.firstname); // this returns “bob”. |
| 9. | console.log(identity.firstname); // this also returns “bob”. |

The code above demonstrates pass by reference.

A variable ***name*** is created and given the value of an object which has a property called ***firstName***, which has the value of “Joe”.

A variable ***identity*** is created and is assigned the value of ***name***. Because these variables are objects, they are passed by reference, which means that when the value of ***name*** is changed, it ripples down through any other objects that reference it.

In the code above, when the ***firstName*** property of the object ***name*** was changed, this also changed the value of the original variable name.

Destructuring

Destructuring is a new set of shortcuts for assigning properties to variables.

### Property Value Shorthand

***Property Value Shorthand*** is a way of reducing the need for keystrokes when the name of a property is exactly the same as the value of the property.

|  |  |
| --- | --- |
| 1. | *const alpha = (name, age) => {* |
| 2. | *return {* |
| 3. | *name: name,* |
| 4. | *age: age* |
| 5. | *}* |
| 6. | *};* |

In the code above, the returned object is given two properties, named ***name*** and ***age***. The values of these properties are also ***name*** and ***age***.

|  |  |
| --- | --- |
| *1.* | *const alpha = (name, age) => {* |
| *2.* | *return {* |
| *3.* | *name,* |
| *4.* | *age* |
| *5.* | *}* |
| *6.* | *};* |

The code above uses the destructuring technique ***Property Value Shorthand*** to reduce the number of keystrokes needed.

### Destructured Assignment

***Key:value*** pairs can be extracted from an object and saved as variables using a technique called Destructured Assignment.

|  |  |
| --- | --- |
| *1.* | *const alpha = {* |
| *2.* | *type: “scout”,* |
| *3.* | *model: “3a”,* |
| *4.* | *version: “4”,* |
| *5.* | *mission: {* |
| *6.* | *recon: “defensive”,* |
| *7.* | *assault: “offensive”* |
| *8.* | *}* |
| *9.* | *};* |
| *10. 11.* | *const version = alpha.version;* |
| *12.* | *console.log(version);* |
| *13. 14.* | *const {version} = alpha;* |
| *15.* | *console.log(version);* |

In the code above, the object ***alpha*** contains several properties. In the first two statements after the object is the standard way of assigning the version ***property*** of the ***alpha*** object to a new variable named ***version***.

The second set of statements shows the assignment of the ***version*** property of ***alpha*** to a new variable ***version*** using the destructured assignment technique.

### Looping through Objects

Loops are statements that repeat a code block until a specified condition is met.

### The for / in loop

The ***for / in*** loop repeats a code block for each property in an object.

|  |  |
| --- | --- |
| 1. | let spaceship = { |
| 2. | crew: { |
| 3. | captain: { |
| 4. | name: 'Lily', |
| 5. | degree: 'Computer Engineering', |
| 6. | cheerTeam() { console.log('You got this!') } |
| 7. | }, |
| 8. 9. | 'chief officer': { |
| 10. | name: 'Dan', |
| 11. | degree: 'Aerospace Engineering', |
| 12. | agree() { console.log('I agree, captain!') } |
| 13. | }, |
| 14. 15. | medic: { |
| 16. | name: 'Clementine', |
| 17. | degree: 'Physics', |
| 18. | announce() { console.log(`Jets on!`) } }, |
| 19. 20. | translator: { |
| 21. | name: 'Shauna', |
| 22. | degree: 'Conservation Science', |
| 23. | powerFuel() { console.log('The tank is full!') } |
| 24. | } |
| 25. | } |
| 26. | }; |

The above code is the object ***spaceship***, which contains nested objects and their properties.

|  |  |
| --- | --- |
| 1. | for (let variableName in outerObject.innerObject) { |
| 2. | console.log(`${outerObject.innerObject[variableName].propertyName}: |
| 3. | ${outerObject.innerObject[variableName].differentPropertyName}`) |
| 4. | }: |

The above code is the syntax used in the ***for / in*** loop.

## Control Flow

Control flow is the order in which the statements are executed in a program. The default control flow is for statements to be read and executed from left-to-right, top-to-bottom in a program file.

Control statements allow the flow to be redirected if specified conditions are met.

* ***If*** statement – The if statement checks if the condition in the parenetheses is true, its code block in the curly braces will be run.
* ***if*** / else statement – This is an if statement with an added code block to be run if the condition fails.
* ***else*** / if statement – The else / if statement is added after an initial if statement. Each else / if statement adds its own condition to be checked. If the condition is true, its code block will run.
* ***ternary*** operator – This is a compact version of an if / else statement.
* ***switch*** statement – The switch statement compares multiple values prefixed with the keyword case, to the value within the parentheses that comes after the keyword switch. Each case has its own code block, which begins with a colon. If the case matches the switch value, then that case’s code block will run.
* ***logical*** operators – the three logical operators compare two values, and will return the value of one of the values depending on each operator’s condition.
* ***&&*** - The ***and*** operator. If both operands on either side of the operator are true, it returns the value of the first operand. If both values are not true, it returns the value of the second operand.
* ***||*** - The or operator. If either one or both values on either side of the operator are true, it returns the value of the first operand. If both operands are false, it will return the value of the second operand.
* ***!*** – The not (or bang) operator. The bang operator either inverts a boolean value, or it inverts the truthiness of a non-boolean value.
* ***comparison*** operators – Comparison operators compare two operands and return a value of true or false depending on which operator is used.

## Conditionals

Conditional statements decide which code block is executed based on the outcome of a comparison between two values. This comparison is called a condition.

If statement

The ***if*** keyword is followed by parentheses which contain the condition to be tested. After the parentheses is an opening and closing set of curly braces. Curly braces are used to define a block of code. In this case, the block of code that is to be executed if the result of the comparison is true. If the result is not true, the code block is skipped.

|  |  |
| --- | --- |
| 1. | const a = 7; |
| 2. | const b = 9; |
| 3. | if (a < b) { |
| 4. | if a is less than b, then the code block will be run. |
| 5. 6. | console.log (“the result is true.”); |
| 7. | } |

If / Else statement

The ***if/else*** statement is an extended version of the If statement, adding a block of code to be run if the result of the comparison is false.

|  |  |
| --- | --- |
| 1. | const a = 7; |
| 2. | const b = 9; |
| 3. | if (a<7) { |
| 4. | if this is true, then the next code block will run. |
| 5. 6. | console.log (“the result is true.”); |
| 7. | } else { |
| 8. | if the result of the comparison is not true, then this code block will run. |
| 9. 10. | console.log (“the result is not true.”); |
| 11. | } |

Else / If statement

The ***else/if*** statement is an even more extended version of the if statement. If the condition being evaluated has more than two possible outcomes, then the additional else/if statements can be added between the if statement and the else statement. The else statement always goes at the end.

|  |  |
| --- | --- |
| 1. | const a = 7; |
| 2. | if (a === 6) { |
| 3. | if a is equal to 6, then this code block will run. |
| 4. | console.log (“a is 6.”) |
| 5. | } else if (a = 7) { |
| 6. | if a is equal to 7, then this code block will run. |
| 7. | console.log (“a is 7”) |
| 8. | } else { |
| 9. | if any of the above conditions are not true, then this code block will run. |
| 10. 11. | console.log (“a is not 6 or 7.”); |
| 12. | } |

Switch statement

The ***switch*** statement is similar in function to the else/if statement, but is easier to read and write, particularly when dealing with many different conditions.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | Switch (expression) { | |
| 2. 3. | | Case x: | |
| 4. | | //Code block | |
| 5. | | Break; | |
| 6. 7. | | Case y: | |
| 8. | | //Code block | |
| 9. | | Break; | |
| 10. 11. | | default: | |
| 12. | | //Code block | |
| 13. | | } | |

The ***switch*** keyword initiates the statement, and the parentheses contain the value that each case will be compared to. Inside the switch’s code block are multiple ***case*** statements. The ***case*** keyword checks if the switch’s condition matches the value of the case’s. If it does, then that case’s code block, which is marked as beginning with a colon, will run.

The ***break*** statement stops the execution of the code within the switch’s code block. Unlike the i***f/else statement***, every case statement would be executed.

|  |  |
| --- | --- |
| 1. | Var planet = ‘Mercury’; |
| 2. 3. | switch (planet) { |
| 4. 5. | case 'Mercury': |
| 6. | return earthWeight \* 0.378; |
| 7. | break; |
| 8. 9. | case 'Venus': |
| 10. | return earthWeight \* 0.907; |
| 11. | break; |
| 12. 13. | case 'Mars': |
| 14. | return earthWeight \* 0.377; |
| 15. | break; |
| 16. 17. | case 'Jupiter': |
| 18. | return earthWeight \* 2.36; |
| 19. | break; |
| 20. 21. | case 'Saturn': |
| 22. | return earthWeight \* 0.916; |
| 23. | break; |
| 24. 25. | default: |
| 26. | return "Invalid Planet Entry. Try: Mercury, Venus, Mars, Jupiter, or Saturn." |
| 27. | } |

In the code above, the variable planet is assigned the string ‘Mercury’. The switch keyword takes the value of the variable within the parentheses and compares it to each of the case statements. If it matches the case statement, that case statement’s code block is run.

## Functions and Methods

A function is a block of code that performs a common operation which can be used by multiple parts of the code. For example, one function might have statements that convert inches to millimeters. The code might have many instances where this is needed, but instead of rewriting this same operation over and over, it is written once as a function, and then the function is used to convert inches to millimeters where it is needed.

let alpha = 77;

alpha = 5/9 \* 77-32;

let bravo = 45;

bravo = 5/9 \* 45-32;

let charlie = 65;

charlie = 5/9 \* 65-32;

In the code above, each variable is given a temperature given in Fahrenheit which must be convert to Celsius. Instead of writing the same operation over and over, the operation to convert Fahrenheit to Celsius can be written as a function. In this way, it only needs to be written once, and then can be accessed by the different variables.

let celsius\_function = function (n) {

return 5/9 \* n – 32;

}

let alpha = 77;

let bravo = 45;

let charlie = 65;

celsius\_function(alpha);

celsius\_function(bravo);

celsius\_function(charlie);

In the code above, the operation is written as the function celsius\_function (the \_function is just part of the name). The different variables are declared, and then the function is executed with the name of each of the variables within the parentheses. The operation is then run using each variable’s value for the operation, and the answer is returned to the function.

The code above may appear to need more statements than the original, but if the common operation were to need many statements, it becomes clear that functions can eliminate redundant code.

Functions are created by using the function keyword followed by the () operator.

Functions have two parts: the function code itself, and the function call. The function is not executed until it is called, which is a line of code outside of the function which triggers its execution.

|  |  |
| --- | --- |
| 1. | let alpha = function () { |
| 2. | console.log(‘This is the function alpha!’); |
| 3. | } |
| 4. 5. | alpha(); |

In the code above are the two parts of function alpha. The first part is the function’s code itself. The line after it, ***alpha()***, is the function’s call.

When this code is executed, the function will be ignored until the code gets to the function call. The function call triggers the function’s execution.

Function Namespace and access

The code within a function’s code block cannot be accessed from outside of the function. This includes any variables that are created within the function’s code block.

|  |  |
| --- | --- |
| 1. | let alpha = function() |
| 2. | { |
| 3. | let bravo = 5; |
| 4. | let charlie = 7; |
| 5. | let delta = charlie + bravo; |
| 6. | } |
| 7. 8. | console.log(bravo); |

In the code above the function ***alpha*** is created and inside of its code block it creates three variables: ***bravo***, ***charlie***, and ***delta***.

The last line, which is outside of the function (it is outside of the curly braces), attempts to log the value of ***bravo*** to the console. However, it will return an error since ***bravo*** is inside the function and not accessible to it.

However, variables declared outside of any function, in the global namespace, are accessible anywhere.

|  |  |
| --- | --- |
| 1. | let alpha = 10; |
| 2. 3. | let bravo = function () { |
| 4. | console.log(alpha); |
| 5. | }; |
| 6. 7. | bravo (); |

In the code above, the variable ***alpha*** is declared and assigned a value of 10. The function ***bravo*** is then created, and within its code block it logs the value of ***alpha*** to the console. When the function is called, the value of ***alpha*** is received inside of the function and successfully logged to the console.

Technically, in JavaScript, a function is actually a first-class function object. Like other objects, they can have properties and methods. The difference between function objects and other objects is that they can be called.

Function Parameters and Arguments

Functions have a way of passing values from outside of the function to inside the function with the use of parameters.

A parameter is a variable that acts as an input for the function. The parameters are contained within the parentheses and separated from each other with commas.

These parameters are given real values when the function is called. The values are called ***arguments***, and are contained within the parentheses in the function call. The values (arguments) are assigned to the parameters (inputs) of the function according to their position from left to right within the parentheses. These variables are not declared using ***const***, ***let***, or ***var***.

|  |  |
| --- | --- |
| 1. | let alpha = function (x, y) { |
| 2. | console.log(x); |
| 3. | console.log(y); |
| 4. | } |
| 5. 6. | alpha(5, 222); |

In the code above, the function ***alpha*** is created with the parameters ***x*** and ***y***. The code block first takes the value that is passed into ***x*** and logs it to the console. Then it takes the value passed into ***y*** and logs that to the console.

In the function call, the first value within the parentheses is 5 and the second value is 222. They are separated with commas. The first argument, 5, is assigned to the first parameter, ***x***. The second argument, 222, is assigned to the second parameter, ***y***.

Parentheses gives the result of the calculation performed by the function. Accessing the function without the parentheses only refers to the function object itself.

### Function Chaining

Function chaining is a syntax where multiple functions are called on the same object one after another.

|  |  |
| --- | --- |
| 1. | const alpha = new ClassIWrote() |
| 2.  3.  4. | alpha.method1 (alpha.method2 (alpha.method3 ())); |

In the code above, the variable alpha is created and assigned a new class object. Alpha contains three methods. Method 3 is contained inside of method 2, and method 2 (which contains method 3) is contained within method 1. This is an example of nested methods. Nested methods are difficult to debug, since they are executed from the innermost function to the outermost function

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | class methodChaining { | |
| 2. 3. | | method1() { | |
| 4. | | console.log(‘This is method 1’); | |
| 5. | | return this; | |
| 6. | | } | |
| 7. 8. | | method2() { | |
| 9. | | console.log(‘This is method 2’); | |
| 10. | | return.this; | |
| 11. | | } | |
| 12. 13. | | method3() { | |
| 14. | | console.log(‘This is method 3.’); | |
| 15. | | return.this; | |
| 16. | | } | |
| 17.  18.  19. | | } | |

The above code creates a class called ***methodChaining***, which has three methods. Each method’s last line uses the this keyword (***return this***). This is the current object instance, and has access to all of the methods defined on the instance, as well as the prototype chain.

The return this line is important for function chaining because it passes back the final values and associates them with the current object.

|  |  |
| --- | --- |
| 1. 2. | const alpha = new methodChaining() |
| 3. 4. | alpha |
| 5. | .method1() |
| 6. | .method2() |
| 7. 8. | .method3(); |

The code above calls all three of the methods defined in the ***methodChaining*** class using method chaining. Instead of calling each method individually, like this:

|  |  |
| --- | --- |
| 1. | alpha.method1(); |
| 2. | alpha.method2(); |
| 3. | alpha.method3(); |

Function chaining allows the object to be called only once, and each method called. Without function chaining, each method is called as its own object.

### The Return keyword

The ***return*** keyword is used to pass back the values that are the result of the execution of a statement.

|  |  |
| --- | --- |
| 1. | var alpha = function () { |
| 2. | var a = 5; |
| 3. | var b = 2; |
| 4. | return a + b; |
| 5. | } |
| 6. | console.log(alpha()); |

In the code above, the function ***alpha*** is created. Within the code block, the variable ***a*** is declared and assigned a value of 5. The variable ***b*** is declared and assigned a value of 2. The calculation is a + b, and the return keyword brings the resulting value of 7 and assigns it to the variable ***alpha***.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | function alpha () { | |
| 2. | | var a = 5; | |
| 3. | | var b = 7; | |
| 4. | | return a + b; | |
| 5. | | } | |
| 6. 7. | | console.log (alpha ()); | |

In the code above, the function ***alpha*** is created. Within its code block the variable ***a*** is assigned a

value of 5, and the variable ***b*** is assigned the value of 7.

The return keyword returns the result of the operation a + b to the function alpha.

Functions as Data

Functions behave like any other data type in JavaScript, and can be assigned to variables.

|  |  |
| --- | --- |
| 1. | const thisFunctionHasAVeryLongName () { |
| 2. | code block |
| 3. | } |
| 4. | var alpha = thisFunctionHasAVeryLongName; |

In the example above, the function named ***thisFunctionHasAVeryLongName*** has been assigned to the variable ***alpha***. Notice that the parentheses are absent when the function is assigned to the variable.

The function is called by using the variable’s name.

|  |  |
| --- | --- |
| 1. | alpha(); |

Functions as Parameters

Functions can be used as the parameters for other functions. A function that uses another function as a parameter, or that returns a function instead of a value is called a ***higher-order function***. The function used as a parameter is called a ***callback*** function.

|  |  |
| --- | --- |
| 1. | function alpha (question, yes, no) { |
| 2. | if (confirm(question)) yes(); |
| 3. | else no (); |
| 4. | } |
| 5. | function showOk () { |
| 6. | alert ("You agreed.") ; |
| 7. | } |
| 8. | function showCancel () { |
| 9. | alert ("You cancelled the execution."); |
| 10. | } |
| 11. | } |
| 12. | alpha ("Do you agree?", showOk, showCancel); |

In the code above, there are three functions. The function ***alpha***, the function ***showOk***, and the function ***showCancel***.

The function ***alpha*** has three parameters: ***question***, ***yes***, and ***no***. The parameters are used in the code block. The parameter ***question*** is the element of the ***if*** statement. If the response to the confirm function is yes, then the value is set to ***yes***. If the response to the confirm function is not ***yes***, then it is set to ***no***.

The ***yes*** parameter is in the second position inside the parentheses, and the ***no*** parameter is in

the last position inside the parentheses.

The last line of the code is the function call. The function ***alpha*** is called and the arguments are inside the parentheses. The positions they appear inside the parentheses indicate which parameter they will be associated with. Therefore, the string ***“Do you agree?”*** Is passed to the argument for the first parameter, question. The function ***showOk*** is used as the argument for the second parameter ***yes***, and the function ***showCancel*** is passed to the third parameter ***no***.

Callbacks

A callback is when a function uses the returned value of a different function as an argument.

|  |  |
| --- | --- |
| 1. | let alpha = function () { |
| 2. | return 1 + 4; |
| 3. | } |
| 4. 5. | let bravo = function () { |
| 6. | return 2 + 10; |
| 7. | } |
| 8. 9. | function charlie (alpha, bravo) { |
| 10. | return alpha + bravo; |
| 11. | } |
| 12. 13. | charlie(alpha(), bravo()); |

In the code above, the functions ***alpha*** and ***bravo*** return values.

The function ***charlie*** has the function ***alpha*** and ***bravo*** as parameters, because the names of those functions are within charlie’s parentheses.

Within ***charlie’s*** code block, the function adds the value of ***alpha*** and ***bravo*** and returns the result to ***charlie***.

The last line is ***charlie’s*** function call. Inside the parentheses, the functions ***alpha*** and ***bravo*** are written. They include their parentheses so that the value of the functions are used, and not the functions’ code.

### Factory Functions

A factory function is a function that can create an object. It can be used to initialize an object, like a constructor. The ***return*** keyword is what creates the new object.

|  |  |
| --- | --- |
| 1. | function alpha (input1, input2) { |
| 2. | return { |
| 3. | bravo: input1, |
| 4. | charlie: input2, |
| 5. | getCombined() { |
| 6. | return input1 + input 2; |
| 7. | } |
| 8. | } |
| 9. | }; |

In the code above, the function ***alpha*** is created, which is given two parameters. Inside its code block is the ***return*** keyword, which creates the object.

The ***return*** keyword creates a new object with two properties and a method. The method returns a new value using the two properties as inputs.

|  |  |
| --- | --- |
| 1. | let person = { |
| 2. | firstName: “John”, |
| 3. | lastName: “Doe”, |
| 4. | getFullName() { |
| 5. | return this.firstName + “ “ + this.lastName; |
| 6. | } |
| 7. | }; |
| 8. | Console.log(person.getFullName()); |

In the code above, the object ***person*** is created and given two properties and a *get* method. The *get* method returns a new value of the first property with a space after it and then the second property. When ***person’s*** method is called and logged to the console, it returns the result ***John Doe***. Imagine that there are many people whose first names and last names need to be combined into full names. This object can be duplicated as many times as needed to create objects for each of those people, but that would result in a lot of redundant code.

A factory function can be used as master template. It can be written once, and only the arguments passed through parameters to return the needed result.

|  |  |
| --- | --- |
| 1. | function person (firstName, lastName) { |
| 2. | return { |
| 3. | firstName: firstName, |
| 4. | lastName: lastName, |
| 5. | getFullName () { |
| 6. | return firstName + “”+ lastName; |
| 7. | } |
| 8. | } |
| 9. | }; |

The above code is a factory function, as can be seen with its use of the ***return*** keyword in the functions outer code block.

|  |  |
| --- | --- |
| 1. | let john = person(“john”, “doe”), |
| 2. | let jane = person(“jane”, “doe”); |
| 3. | console.log(john.getFullName()); |
| 4. | console.log(jane.getFullName()); |

In the code above, the variable ***john*** is created with the argument one as “John” and argument two as “Doe”. It then uses those input values with the ***person*** function.

The result of the get method within the ***person*** function returns the value of ***getFullName***, which is logged to the console.

The variable ***Jane*** is then created, and then the arguments given to this variable are used to return the appropriate result when ***Jane*** is logged to the console.

## Arrow Function Expressions

Arrow function expressions are a way of writing function expressions in a simpler and more concise way.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | const alpha = function (parameter1, parameter2) { | | |
| 2. | return parameter1 + parameter2; | | |
| 3. | | | }; |

The code above is a function written in the standard format.

|  |  |
| --- | --- |
| 1. | const alpha = (parameter1, parameter2) => parameter1 + parameter2; |
| 2. | console.log (alpha (5,7) ); |

In the code above, the arrow function is used with the use of the fat arrow symbol, **=>**. When this symbol is used, the function keyword is not needed. The fat arrow separates the parameters within the parentheses from the body, or code block.

|  |  |
| --- | --- |
| 1. | const alpha = (parameter) => { |
| 2. | console.log (parameter + parameter) |
| 3. | } |
| 4. | alpha (5); |

In the code above, the function which is assigned to the variable ***alpha*** has only one parameter. With arrow functions, when there is only one parameter, the parentheses normally surrounding them are not needed.

|  |  |
| --- | --- |
| 1. | const alpha = function (parameter) { |
| 2. | console.log (parameter + parameter); |
| 3. | } |
| 4. | alpha (5); |

The code above shows the sample function written as a standard function rather than using an arrow function expression.

|  |  |
| --- | --- |
| 1. | const alpha = (a, b) => a \* b; |
| 2. | console.log (alpha (2, 50) ); |

In the code above, the function has only one expression in its code block. With arrow function expressions, a single expression is separated from the parameters using the fat arrow, and also does not need the use of the ***return*** keyword to return the value of the expression. const alpha = () => console.log(“This is a function with no parameters.”);

|  |  |
| --- | --- |
| 1. | alpha (); |
| 2. | The code above shows a function that has no parameters written using arrow syntax. Since it only has one expression, the curly braces are not needed. |
| 3. 4. | const alpha =()=> { |
| 5. | console.log(“This is a function without parameters.”); |
| 6. | alert(“This is the second expression.”); |
| 7. | } |
| 8. 9. | alpha (); |

The code above shows a function using arrow syntax. Since there is more than one expression in the code block, the curly braces are required.

### Anonymous Functions

Anonymous functions do not have a name property. They are defined using the ***function*** keyword or arrow function.

|  |  |
| --- | --- |
| 1. | function alpha () { |
| 2. | code block; |
| 3. | } |

The code above shows a ***named*** function.

|  |  |
| --- | --- |
| 1. | const alpha = function () { |
| 2. | code block; |
| 3. | } |

The code above shows an ***anonymous*** function. The variable is declared as ***alpha***, but the function itself doesn’t have a name. It is just declared with the ***function*** keyword.

Methods

A ***method*** is a function that is attached to a particular object.

|  |  |
| --- | --- |
| 1. | object.method (); |

In the code above, the ***object*** is the first word. The ***method*** is the second string ending with a parentheses pair. The object and the method are separated by a period. This is called dot notation.

|  |  |
| --- | --- |
| 1. | console.log (“message”); |

In the code above, the method ***log*** is attached to the object ***console***. The method has a parameter which is the string “message”. This will log the text “message” to the console object.

|  |  |
| --- | --- |
| 1. | let alpha = { |
| 2. | happy () { |
| 3. | console.log (“this is the happy function. these three statements will be run.”); |
| 4. | console.log (“this is the second statement of the happy function.”); |
| 5. | console.log (“this is the last statement of the happy function.”) |
| 6. | }, |
| 7. | golf () { |
| 8. | console.log (“this is the golf function.”) |
| 9. | }, |
| 10. | foxtrot () { |
| 11. | console.log (“this is the foxtrot function. all the statements inside of these curly braces will be executed.”) |
| 12. | } |
| 13. | }; |

***Notice that each method (except for the last one) ends with a comma after the closing curly brace. The method and its code block is a key:value pair, and key:value pairs are separated with commas.***

When a function is a method of an object, the object’s name is appended with the function’s name

|  |  |
| --- | --- |
| 1. | var alpha = { |

1. width: 3,
2. length; 10,
3. area (width, length) {
4. return width \* length;
5. };
6. **alpha.area();**

Invoking methods

An object’s method is invoked by appending the object’s name with the dot separator and adding the method name and parentheses.

|  |  |
| --- | --- |
| 1. | alpha.happy (); |

The above statement calls the ***happy*** function of the ***alpha*** object.

|  |  |
| --- | --- |
| 1. | alpha.golf (); |

The above statement calls the ***golf*** function of the ***alpha*** object.

|  |  |
| --- | --- |
| 1. | alpha.foxtrot (); |

The above statement calls the ***foxtrot*** function of the alpha object.

The *this* keyword

When a method inside of an object refers to another property of that object, the method will return a result of undefined and not the value of the property referred to. This is because the method does not automatically have access to other properties of the object outside of the function.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. const = alpha { 2. bravo: “blue”, 3. charlie () { 4. console.log(“this is a message.”); 5. }   6.   |  |  |  |  | | --- | --- | --- | --- | | 7. |  | delta () { |  | | 8. |  | console.log(bravo); | |  1. } 2. };   11.  12. alpha.delta(); |

|  |
| --- |
| using a period. |

In the code above, the variable ***alpha*** has been given one property and two methods. Inside the second method, delta, the code block logs the value of the property ***bravo*** to the console. However, because the property is outside of the method, it will return a result of undefined.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | const = alpha { | |
| 2. | | bravo: “blue”, | |
| 3. | | charlie () { | |
| 4. | | console.log(“this is a message.”); | |
| 5. | | } | |
| 6. | | delta () { | |
| 7. | | console.log(this.bravo); | |
| 8. | | } | |
| 9. | | }; | |
| 10. | | alpha.delta(); | |

The ***this*** keyword reference the calling object. This provides access to the calling object’s properties. It is put before the name of the property.

In the example above, the second method (***delta***) contains a reference to the property ***bravo***. To do this, the ***this*** keyword is added before ***bravo***, as indicated in the hightlighted line of code.

The *this* keyword and Arrow Functions

When used with the ***this*** keyword, arrow functions behave differently than with standard syntax. Arrow functions bind the ***this*** value to the function itself and not the calling object. The value of ***this*** is the global object, or an object that exists in the global scope. Since there is no object in the global scope with a property ***bravo***, the ***this*** keyword returns a value of undefined. In short, it is best to avoid using arrow functions with the ***this*** keyword.

Privacy

Privacy refers to whether or not a property’s value can be changed or not. Some properties are meant to be read-only, so JavaScript uses a particular naming convention to lock a property’s value. The use of an underscore before the property’s name signifies that the property is not intended to be directly manipulated.

|  |  |
| --- | --- |
| 1. | const alpha = { |
| 2. | \_Bravo: 1000 |
| 3. | } |

In the code above, the property ***bravo*** has been prepended with an underscore.

However, it is still possible to reassign ***\_bravo***.

|  |  |
| --- | --- |
| 1. | alpha.\_bravo = 44; |

### Getters methods

***Getters*** are methods that get and return internal properties of an object.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | const person = { | |
| 2. | | \_firstName: “John”, | |
| 3. | | \_lastName: “Doe”, | |
| 4. 5. | | get fullName () { | |
| 6. | | if (this.\_firstName && this.lastName) { | |
| 7. | | return `$(this.\_firstName} ${this.\_lastName}`; | |
| 8. | | } else { | |
| 9. | | return `Missing a first name or a last name.` ; | |
| 10. | | } | |
| 11. | | } | |
| 12. | | } | |
| 13. 14. | | person.fullName; | |

### Setters methods

Setters are methods that are used within object code blocks that reassign values of properties within the code block.The ***Set*** function has one argument, which is called when the property is set.

|  |  |
| --- | --- |
| 1. | const person = { |
| 2. | firstName: “John”, |
| 3. | lastName: “Doe”, |
| 4. | set firstName () { |
| 5. | this.firstName = “Robert” |
| 6. | } |
| 7. | }; |

## Callbacks

A callback is a function that is passed into another function as an argument to be used later.

|  |  |
| --- | --- |
| 1. | let alpha = [1, 2, 4, 7, 3, 5, 6]; |

The code above creates an array named ***alpha***.

|  |  |
| --- | --- |
| 1. | function isOddNumber(number) { |
| 2. | return number % 2; |
| 3. | } |
| 4. 5. | const alpha = numbers.filter(isOddNumber); |
| 6. | console.log(alpha); |

In the code above, a function named ***isOddNumber*** is created, which returns a value of true if a number is an odd number.

The next line creates a variable named ***alpha***, which is assigned to a .filter method (which is a function). The argument of the .filter method is the function ***isOddNumber***. The value of alpha is then logged to the console.

The function ***isOddNumber*** is a callback, since it is used as an argument in the ***.filter*** method.

## Arrays

An ***array*** is a numbered list of ***elements***. An element can be of any data type, such as numbers, strings, variables, and so on. Each element in an array is identified by its position in the list, from left to right, and the elements are delimited with commas.

|  |  |
| --- | --- |
| 1. | var alpha = [element1, element2, element3]; |

In the code above, the variable named ***alpha*** is declared and assigned the value of an array. The array elements are within a pair of square brackets, and are separated with commas. Each element in an array is identified with an ***index numbe***r. The numbering starts with the left-most element, and proceeds to the right until the closing square bracket is encountered. The first element in all arrays is given an index of ***0***, the second is given an index of 1, and so on.

|  |  |
| --- | --- |
| 1. | console.log (alpha [1]); |

In the code above, the element ***element2*** from the array named ***alpha*** is logged to the console. The name of the array is followed by the index number of the desired element within that array. In this case, it is the second element in the list, since the first element of any array is always given the index number of ***0***.

Type Coercion

Type coercion is the process of converting one data type to another, such as a string to a number, or an object to a boolean.

Array Methods

Arrays have several methods dedicated solely to their use.

***.unshift ()***

This adds elements to the beginning of an array.

|  |  |
| --- | --- |
| 1. | .unshift () - the unshift method adds new items to the beginning of an array. |
| 2. | var alpha = [element1, element2, element3]; |
| 3. | alpha.unshift (“newitem1”, “newitem2”); |

The above example yields the array:

|  |  |
| --- | --- |
| 1. | alpha [“newItem1”, “newItem2”, element1, element2, element3]; |

***.shift ()*** The shift method removes the first item of the array.

|  |  |
| --- | --- |
| 1. | var alpha = [element1, element2, element3]; |
| 2. | alpha.shift (); |

The above example yields the array:

|  |  |
| --- | --- |
| 1. | alpha [element2, element3]; |

***.push ()*** The push method adds items to the end of the array.

|  |  |
| --- | --- |
| 1. | var alpha = [element1, element2, element3]; |
| 2. | alpha.push (element4, element5); |

The above example yields the array:

|  |  |
| --- | --- |
| 1. | alpha [element1, element2, element3, element4, element5]; |

### .pop ()

The pop method removes the last item of the array. It doesn’t take any arguments like push can.

|  |  |
| --- | --- |
| 1. | var alpha = [element1, element2, element3]; |
| 2. | alpha.pop (); |

The above example yields the array:

|  |  |
| --- | --- |
| 1. | alpha [element1, element2]; |

.split ()

The split method takes each word in a single string and creates an array with each word as an element of the array.

var alpha = ‘This is something new’;

var bravo = alpha.split(“ “); // The character between the quotes within the parentheses is the character the method uses to determine where to separate the words. In this case, the character is a blank space.

console.log(bravo); // This returns [‘This’, ‘is’, ‘something’, ‘new’];

### .join ()

The join method takes the individual elements in an array and joins them together in a single array, with the previously individual elements separated with commas or some other separator and

|  |  |
| --- | --- |
| 1. | var alpha = [element1, element2, element3]; |
| 2. | alpha.join (); |

The above example yields the array:

|  |  |
| --- | --- |
| 1. | alpha [“element1,element2,element3”]; |

### .slice ()

The slice method extracts specified elements of an array and creates a new array with those elements. The first parameter is the start parameter, which specifies the index number of the first item in the new array. The second parameter is the end parameter which specifies where the extraction ends. The element of the end parameter is not included in the new array, it just specifies where the operation stops.

|  |  |
| --- | --- |
| 1. | var alpha = [element1, element2, element3, element4, element5]; |
| 2. | alpha.slice (1, 4); |

The above example yields the array:

|  |  |
| --- | --- |
| 1. | alpha [element2, element3, element4]; |

***Element2*** is the element in the first array with the index number of ***1***, which is specified as the parameter in the slice method. The second parameter specifies the element with the index of ***4*** to stop the extraction, so all of the elements from element 1 and the elements up to element 4 are included in the new array.

### .splice ()

The splice method can add or remove elements from an array using 3 parameters.

|  |  |
| --- | --- |
| 1. | var alpha = [element1, element2, element3]; |
| 2. | alpha.splice (index, howManyElementsRemoved, newItem1, newItem2); |
| 3. | alpha.splice (1, 2, “Element 33”, “Element 44”); |

At index 1, remove two elements and add the new items. The above example yields the array:

|  |  |
| --- | --- |
| 1. | alpha [element1, “Element 33”, “Element 44”];]; |

***.concat ()*** The concat method joins two or more arrays together.

|  |  |
| --- | --- |
| 1. | var alpha = [element1, element2, element3]; |
| 2. | var bravo = [item1, item2, item3]; |
| 3. | var charlie = alpha.concat (bravo); |

The above example shows the array ***charlie*** as the result of combining the array ***alpha*** with array ***bravo***.

### Object properties within arrays

Arrays can contain objects, which are contained within curly braces.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | let alpha = [ | |
| 2. 3. | | {name: “first”, | |
| 4. | | size: “large”, | |
| 5. | | color: “blue”}, | |
| 6. 7. | | {name: “second”, | |
| 8. | | size: “small”, | |
| 9. | | color: “red”}, | |
| 10. 11. | | {name: “third”, | |
| 12. | | size: “medium”, | |
| 13. | | color: “yellow”} | |
| 14. 15. | | ] | |

In the code above, the variable alpha is created and assigned to an array. The array has three items, each with a name property, a size property, and a color property. Each object and its set of properties are contained within a set of curly braces. Each object in the array uses a comma separator.

### forEach()

The forEach() method calls a function on each element in an array.

|  |  |
| --- | --- |
| 1. | var alpha = [12, 13, 14, 15]; |
| 2. | alpha.foreach(bravofunction); |
| 3. 4. | function bravofunction (charlie) { |
| 5. | console.log(charlie); |
| 6. | } |

In the code above, the array object a***lpha*** is created. The next line calls the function named ***bravoFunction*** on each element in the ***alpha*** array. The name of the array is followed by a period, then the ***forEach*** keyword, and in the parentheses is the name of the function.

The function below it named ***bravoFunction*** has a parameter named ***Charlie***. The function’s code block logs the value of ***Charlie*** to the console. When the function is called by the ***forEach()*** function, the value of each element in the ***alpha*** array becomes assigned to ***Charlie***, which is then logged to the console.

The parameter for the function can also be anonymous.

Iterators

An iterator, or iteration method, is an object or process that traverses an iterable object, such as an array.

Traversing a data structure is the process of taking each element of an iterable object as a value, performing an action with that value, and then returning a result.

|  |  |
| --- | --- |
| 1. | Var alpha = [22, 23, 24, 25]; |
| 2. | Alpha.forEach (bravo); |
| 3. 4. | Function bravo (Charlie) { |
| 5. | Var delta = charlie \* 2; |
| 6. | Console.log(delta); |
| 7. | }; |

In the code above, the array ***alpha*** is created and given four elements. The ***forEach()*** iterator statement begins with the name of the iterable object whose elements are going to be used, adds the ***forEach*** keyword separated with a dot, and within the method’s parentheses is the name of the function to be run on each of the elements.

In this case, the function ***bravo*** is called to be run. The function ***bravo*** creates a new variable, ***delta***, and assigns it the value of ***Charlie*** multiplied by two. It then logs the value of ***delta*** to the console.

## Loops

A loop is a block of code that repeats until a specified condition is met. The simplest loop has two parts: The block of code to be run, and a stopping condition. After each time the code block is run, the stopping condition is checked. If the stopping condition is true, then the loop ends and the next statement after the loop is executed.

A loop takes a variable, runs a code block on it, then change the variable’s value, runs the code block on this new value, then continues to change the variable’s value after running the code block. This creates a series of different values that are produced from the code block. This process is stopped when the specified condition no longer evaluates as true.

* ***for*** - loops through a block of code a number of times
* ***for/in*** - loops through the properties of an object
* ***for/of*** - loops through the values of an iterable object
* ***while*** - loops through a block of code while a specified condition is true
* ***do/while*** - loops through a block of code once, and then repeats the loop while a specified condition is true

For loops

The for loops is the basic loop. It is completely self-contained. It establishes the variable and its starting value, the condition to be checked, and the incrementing action.

For loops are useful for when the number of times the loop is to be run is already known.

|  |  |
| --- | --- |
| 1. | for (startingValue, testCondition, iterator) { |
| 2. | code block |
| 3. | } |

* ***StartingValue*** - sets a variable before the loop starts. The variable acts as a counter.
* ***testCondition*** - defines the condition in order for the loop to run.
* ***Iterator*** - the incrementing action which changes the value of the variable that is evaluated by the condition. This increases the value of the counter. The loop will run until statement 3, the iterator, generates a value for the variable that evaluates as false in the condition and the loop ends.

|  |  |
| --- | --- |
| 1. | for (x=0; x<10; x++) { |
| 2. | console.log(x); |
| 3. | } |

In the code above, a ***for*** loop is created with the ***for*** keyword.

The first statement sets the value of the variable x to 0.

The second statement sets the condition to ‘if x is less than 10’. If it is true, then the loop will run.

The third statement increases the value of x by 1. This third statement changes the value of x that is going to be used for the next iteration of the loop. If the value of x didn’t change, the loop would run forever.

### For/In loops

The For/In loop is used to loop through an object’s properties.

|  |  |
| --- | --- |
| 1. | const car = { |
| 2. | year: “2020”, |
| 3. | make: “subaru”, |
| 4. | model: “wrx”, |
| 5. | color: “blue” |
| 6. | } |

The code above shows the properties for the object car.

|  |  |
| --- | --- |
| 1. | for (let x in car) { |
| 2. | console.log(x) |
| 3. | } |

In the code above, the for keyword initializes the for loop. In the parentheses, a variable is created. In this case the variable x is created using the let keyword. Then the keyword in is followed by the name of the object whose properties are to be looped through.

Each property’s value is assigned to the variable, and then the code block is run. In this case, each property is logged to the console.

### For/Of loops

The For/Of loop is used to loop through iterable objects, like arrays.

|  |  |
| --- | --- |
| 1. | let alpha = [1, 2, 3, 4, 5]; |

The code above creates the variable alpha, and assigns it an array.

|  |  |
| --- | --- |
| 1. | for (const x of alpha) { |
| 2. | console.log(x); |
| 3. | } |

The code above creates a For/Of loop using the For and Of keywords. In the parentheses a variable is created, then the keyword of is used, and then the name of the variable whose elements are to be looped through.

### While loops

The while loop executes a block of code as long as a specified condition is true.

|  |  |
| --- | --- |
| 1. | var x = 0; |
| 2. | while (x<5) { |
| 3. | console.log(“The number is “ + x); |

|  |  |  |
| --- | --- | --- |
| 1. x++; 2. } |  | 1. console.log(“the number is “ + i); 2. } |

In the code above, the variable x is set to 0.

The while loop is created with the while keyword, and the condition inside the parentheses is x is less than 5. If this condition is true, the code block will run.

The variable is then incremented to prevent an infinite loop.

### Do / While loop

The Do/While loop is a variation of the While loop. The While loop checks to see if the condition is true before the loop is run.

The Do/While loop runs the loop once, and then checks to see if the condition is true.

|  |  |
| --- | --- |
| 1. | do { |
| 2. | console.log(“the number is “ + x); |
| 3. | x++; |
| 4. | } |
| 5. | while (x < 10); |

In the code above, the Do/While loop is created. The code block within the curly braces is run, and then the condition for the loop is created using the while keyword. The condition is then put inside the parenetheses. The code block is run a first time, and then the condition is evaluated. If it is true, the code block is run again.

Note that an iterator is within the code block to change the value of the variable, otherwise an infinite loop would be created.

### The Break and Continue keywords

Both the ***break*** and ***continue*** keywords allow for the execution of the loop to be changed if specified conditions are met.

The ***break*** keyword stops the execution of a loop. The next statement outside of the loop is then run.

|  |  |
| --- | --- |
| 1. | for (i=0; i<10; i++) { |
| 2. | if ( i ===3) { break; } |
| 3. | console.log(“the number is “ + i); |
| 4. | } |

In the code above, a ***For*** loop is created and the variable i is assigned an initial value of 0. The code block will run if i is less than 10, and the value of i is increased by 1 each time the loop is run.

The next statement says that if the value of i is 3, then the loop ends and the code continues after the loop.

The statement after the break statement says to log the string “The number is “ and whatever value the variable i has for that iteration.

The ***continue*** keyword stops the execution of the loop, but continues with the next iteration of the loop.

|  |  |
| --- | --- |
| 1. | for (i=0; i<10; i++) { |
| 2. | if (i === 3) { continue; } |

In the code above, a ***For*** loop is created and the variable ***i*** is assigned an initial value of 0. The code block will run if i is less than 10, and the value of i is increased by 1 each time the loop is run. The next statement is the ***Continue*** statement. It says that if the value of ***i*** is 3, to stop that iteration of the loop and not run the code block, but to change the value of ***i*** as specified in the ***For*** loop, and begin the next iteration of the loop.

In this case, all of the values for the variable ***i*** are logged to the console along with the string ***The number is*** , except for when the value of i is 3

### Loops and arrays

A loop can be used to compare all the elements within an array to something else.

|  |  |
| --- | --- |
| 1. | var alpha = [1, 2, 3, 4, 5]; |
| 2. | for (var x = 0; x<alpha.length; x++) { |
| 3. | console.log (alpha[x]); |
| 4. | } |

In the code above, the array ***alpha*** is created. The ***for*** loop declares a variable named ***x*** and sets its value to 0. The stopping condition is ***if x is less than the total number of elements in the array (alpha.length),*** then the code will run. If not, the value of x is increased by one and the new value given to x is tested against the stopping condition again.

Nested Loops

|  |  |
| --- | --- |
| 1. | var alpha = ['blue', 'black', 'green']; |
| 2. | var bravo = ['black', 'Michelle', 'Heather']; |
| 3. | var charlie = []; |
| 4. | for (var n = 0; n<alpha.length; n++) { |
| 5. | for (var x = 0; x<bravo.length; x++) { |
| 6. | if (alpha [n] === bravo [x]) { |
| 7. | charlie.push(alpha [n]); |
| 8. | } |
| 9. | } |
| 10. | }; |
| 11. 12. | alert (charlie); |

The code above compares each element of the array ***bravo*** (the inner loop), with each element of the array ***alpha*** (the outer loop).

The first for loop sets the value of the ***alpha*** loop to the first element (index [0]), ‘***blue***’. The variable ***n*** represents the index number of each element in the array ***alpha***.

The second for loop sets the value of the ***bravo*** loop to the first element in its array, ‘***black***’. The variable ***x*** represents the index number of each element in the ***bravo*** array.

Next, the if statement compares the current value of the element in the ***bravo*** array to the currently held value in the ***alpha*** array. If they match, ***alpha’s*** value gets pushed into the empty array that was created, ***charlie***.

The inner loop then iterates, increases the value of x by 1, and then compares the value to the still-held value of the ***alpha*** array. If both match, then the ***alpha*** array’s value gets pushed into array ***charlie***.

When the inner loop (***bravo***) finally meets its stopping condition, the javaScript goes back to the outer array (***alpha***), iterates it by 1, and then goes to the ***bravo*** array. It then compares each element in the ***bravo*** array with this new value for the ***alpha*** array, and the process continues until both arrays have met their stopping conditions.

|  |  |
| --- | --- |
| 1. | var alpha = ['blue', 'black', 'green']; |
| 2. | var bravo = ['black', 'Michelle', 'Heather']; |
| 3. | var charlie = []; |
| 4. 5. | for (var n = 0; n<alpha.length; n++) { |

This creates the for loop, creates the variable ***n*** and sets its value to 0.

This loop will stop when ***n*** doesn't match the number of elements in the array alpha, and every time the loop restarts, it will increase the value of n by 1.

|  |  |
| --- | --- |
| 1. | for (var x = 0; x<bravo.length; x++) { |

This second for loop creates the variable ***x*** and sets its value to 0. This loop will stop when ***x*** doesn't match the number of elements in the array ***bravo***, and each time it restarts, the value of ***x***

increase by 1.

|  |  |
| --- | --- |
| 1. | if (alpha [n] === bravo [x]) { |

This ***if*** statement compares the value of the current element being held by the variable ***n*** in the ***alpha*** loop to the current element being held by the variable ***x*** in the ***bravo*** loop.

So, to follow the flow of this code:

When the first loop is executed, the value of ***n*** is 0, and the code block for the loop is started.

The next statement is the second for loop, which sets the value of ***x*** to 0, and the code block for this loop is started.

The ***if*** statement compares the value of the element number ***n*** of the ***alpha*** loop with the value of the element number ***x*** in the ***bravo*** loop.

|  |  |
| --- | --- |
| 1. | charlie.push(alpha [n]); |

If the element with the index number of ***x*** in the ***bravo*** loop matches the value of the element ***n*** in the ***alpha*** loop, then the value of ***alpha [n]*** is pushed into the empty array ***charlie*** that was created.

This completes the innermost loop, which is for the ***if*** statement. This then completes one iteration of the inner for loop (for ***bravo***), so the value of ***bravo's x*** variable increases by one. So now, the value of the ***n*** variable for the ***alpha*** loops remains at 0, while the ***x*** variable for the ***bravo*** loop increases to 1.

Now the ***if*** statement compares the value of ***alpha [0]*** with the new value of ***bravo [1***], and if the values match, this value for alpha will be pushed into the array ***charlie***.

The inner loop repeats, and each iteration compares the new value of ***bravo [x]*** with the held value of ***alpha [n]*** until the stopping condition for the for loop ***bravo*** is met, which means the inner loop is completed and the outer loop then iterates, increases the value of ***n*** by one, and then goes through the process again.

Essentially, the value for the outer loop's variable is compared to every iteration of the inner loop. When the inner loop is finished, then the outer loop increase its value by one, and compares this new value to all of the inner loop's values, and so on.

For loops are ideal when the number of times the loop must run is known

While loops

A ***While*** loop is a loop that executes its code block as long as a test condition is true.

|  |  |
| --- | --- |
| 1. | var alpha = 1; |
| 2. | while (alpha < 4) { |
| 3. | code block; |
| 4. | alpha++; |
| 5. | } |

The ***while*** keyword starts the loop, and within the parentheses is the test condition. As long as the test condition is true, the loop will run the code block. When the test condition doesn’t evaluate as true, the loop will stop.

In the code above, the variable ***alpha*** is declared and assigned a value of 1. The test condition for the ***while*** loop is the question ‘Is the value of ***alpha*** less than 4?’. If it is, the code block is executed. If not, the loop stops.

It is important to note that the iterator for the variable is inside the code block. If it wasn’t there, the value for ***alpha*** would always be 1, and the test condition would always evaluate to true, resulting in an infinite loop. ***Infinite loops must always be avoided.***

While loops are needed when the number of times the loop must run is unknown.

### forEach() method

The .forEach() method executes the same code on each element of an array.

|  |  |
| --- | --- |
| 1. | const alpha = [“square”, “circle”, “triangle”, “polygon”]; |

The code above is an array assigned to the variable ***filter***.

|  |  |
| --- | --- |
| 1. | const alpha = [“square”, “circle”, “triangle”, “polygon”]; |
| 2. 3. | alpha.foreach (function (hotel) { |
| 4. | console.log (‘I want a‘ + hotel); |
| 5. | }); |

In the code above, ***alpha*** is the identifier, which is the name of the variable the array is assigned to.

***Alpha.forEach()*** calls the *forEach* method on the ***alpha*** array.

In this example, the ***forEach*** method uses a callback function as an argument. The callback function has a parameter named ***hotel***.

The ***forEach()*** method loops through the array and executes the callback function on each element. During each execution, the current element is passed as an argument to the callback function.

In this example, the string “I want a “ is logged to the console with each element of the array ***alpha***.

The return value for ***.forEach()*** is always undefined.

|  |  |
| --- | --- |
| 1. | alpha.foreach(hotel => console.log(“I want a “ + hotel)); |

The code above shows the same ***forEach*** method written using arrow function syntax.

### .map() method

The .map() iterator executes a function’s code block on each element of an array, similar to the forEach iterator, except the .map() iterator returns a new array.

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

In the code above, the variable ***numbers*** is assigned an array.

The variable ***bigNumbers*** is assigned returned value of the ***numbers.map*** iterator. The iterator’s function takes the argument ***alpha***, which passes the value of each element to inside the code block and multiplies it by 10 and returns the result to the new array ***bigNumbers***.

### .filter()method

The .filter() method either pushes each element of the array into a new array if the element returns a value of ***true*** when the callback function is run on the element.

|  |  |
| --- | --- |
| 1. | const numbers = [1, 2, 3, 4, 5, 11, 12, 13, 14]; |
| 2. | const alpha = numbers.filter(n => { |
| 3. | return n < 10 |
| 4. | } |
| 5. | console.log(alpha); |

In the code above, the variable ***numbers*** is declared and assigned an array.

The variable ***alpha*** is declared, and uses the ***.filter*** method on the ***numbers*** array. The callback function of the ***.filter*** method is the question *‘Is the element less than 10?’* If the element of the ***numbers*** array is less than 10 (or returns a value of true from the callback function), then that element is kept in the new array. If the element of the first array is not less than 10 (or returns a value of false), then the element is not put into the new array.

The console shows [1, 2, 3, 4, 5].

### .findIndex() method

The .findIndex method returns the location of any element in an array which returns a value of true when the callback function is run.

1.

const alpha = [“hippo”, “tiger”, “goose”, “lion”, “bear”];

2.

3.

const bravo = alpha.findindex

(

charlie

=> {

4.

return charlie.charat(0) === “s”;

5.

}

6.

)

7.

;

8.

9.

console.log(charlie);

In the code above, the array ***alpha*** is created. The array ***bravo*** is created, and searches the ***alpha*** array for an element which starts with the letter s, using the ***.charAt(0)*** method.

Notice in the highlighted area that the parentheses contain the function’s argument, the fat arrow, and the code block.

### .reduce() method

The .reduce method uses two parameters and a function to ultimately return a single value. The first parameter, from left to right, is the ***accumulator***. It is either the initial value if the method is just starting, or the value returned from the function. The next value to the right of the ***accumulator*** is the ***current value parameter***.

|  |  |
| --- | --- |
| 1. | const alpha = [5, 10, 15]; |
| 2. | const bravo = alpha.reduce ((accumulator, currentvalue) => { |
| 3. | return accumulator + currentvalue |
| 4. | }); |

In the code above, the variable ***alpha*** is declared as an array with three elements.

The variable ***bravo*** is declared which is assigned the result of the .reduce method that is performed on the ***alpha*** array.

The parameters for the the ***.reduce*** method’s function are ***accumulator*** and ***currentValue***. The code block returns the result of the ***accumulator***, which is 5, to the next element in the ***alpha*** array, which is 10. The returned value of the first iteration is 15. This value now becomes the ***accumulator*** value, which is added to the next value in the ***alpha*** array, so 15 (***accumulator*** value) + 15 = 30. The ***.reduce*** method will keep repeating the function until there is only one value returned.

A second parameter, which is optional, can be added to set the initial value for ***accumulator***.

|  |  |
| --- | --- |
| 1. | const bravo = alpha.reduce ((accumulator, currentvalue) => { |
| 2. | return charlie.charat(0) === “s”; |
| 3. | }, 999); |

The code above is that same as the previous example, except that the ***comma 999*** was added inside the outer parenetheses set. This value is the second, optional parameter which sets the initial value of ***accumulator***.

## JavaScript Programming Tools

Packages

A package is one or more modules with a special file called the ***Manifest***. This file keeps track of how all the modules interact with each other (these interactions are called dependencies), along with other metadata about the project. In JavaScript, the file is named ***package.json***

|  |  |
| --- | --- |
| 1. | .Shapes |
| 2. | Package.json |
| 3. | Circle.js |
| 4. | Square.js |
| 5. | Triangle.js |

In the example above, the package is named ***shapes***, which is like a folder that contains the manifest file named package.json, and three modules; circle.js, square.js, and triangle.js.

Libraries

A library is a collection of packages which contain custom functions that can be attached to a web page to write common functions. Examples of JavaScript libraries are jQuery, React, and Mootools.

API

The API (Application Programming Interface) is the toolbox which allows the use of what is contained in a library. An API is like a woodworking workshop. It contains the space and tools.

Frameworks

A framework is an existing template for an application. It is a generic structure designed to used as A framework is a premade application template which is used to create APIs. They include basic functions already in place, and custom code is added for the application’s specific purpose. A framework can include support programs, libraries, or other software.

Examples of framework applications for web development include Electron, Express.js, Angular JS, EmberJS, ReactJS, and Vue.js.

## Modules

A module is a JavaScript file that contains functions and other elements of JavaScript code that are to be used in other JavaScript applications. In order to be imported into other applications, modules must be exported using the export and import keywords.

Modules help to keep the application organized by keeping blocks of similar code together in an external file. This keeps the application from becoming so large that it becomes difficult to edit or debug.

By keeping discrete, generic functions separate, they can be called only when needed and not duplicated.

Modules prevent the pollution of the global namespace and avoid two or more variables are given the same name, resulting in a naming collision.

Modules can refer to other modules. When a module requires information from another module it creates a dependency with that modu le.

### Module.exports

***Module.exports*** is an object and property combination used in Node to define and export modules.

|  |  |
| --- | --- |
| 1. | Let alpha = {}; |
| 2. | Module.exports = alpha; |

In the code above, the object ***alpha*** is declared, and then specified to be exported by assigning it to the module.exports statement.

### Default Export

Default Export is a newer version of the module.exports object.

|  |  |
| --- | --- |
| 1. | let alpha = {}; |
| 2. | export default alpha; |

In the code above, the object ***alpha*** is declared and then specified to be exported by the new, ES6 supported export default statement with the name of the object to be exported.

### Named Export

Named export allows data to be exported through the use of variables.

|  |  |
| --- | --- |
| 1. | let alpha = ‘’; |
| 2. | function bravo () { |
| 3. | }; |
| 4. | function charlie () { |
| 5. | }; |
| 6. 7. | export { alpha, bravo }; |

In the code above, there are three objects. ***Alpha*** is a string object. ***Bravo*** and ***charlie*** are functions. The ***export*** keyword sets up the export, and the curly braces contain the names of the objects separated with commas.

### Immediate Named Export

Variables can also be exported as soon as they are declared.

|  |  |
| --- | --- |
| 1. | let alpha = 5; |

The code above declares the variable ***alpha*** and sets its value to 5.

|  |  |
| --- | --- |
| 1. | export let alpha = 5; |

To export the variable, the keyword ***export*** is added before the variable declaration keyword.

### Export as

Named Exports also allow a way to change the name of items when they are exported using the as keyword.

|  |  |
| --- | --- |
| 1. | let alpha = 5; |
| 2. | let bravoFunction () { |
| 3. | }; |

export { alpha as charlie, bravoFunction as delta };

In the code above there is a variable alpha and a function bravoFunction. The export keyword is used to export them, but the as keyword is used to assign them new names.

### Import

Objects from external JavaScript files can be imported using the import keyword.

|  |  |
| --- | --- |
| 1. | Import alpha from ‘./bravo’; |

In the code above, the keyword ***import*** begins the statement. The next word specifies the name of the variable to store the default export in. ***‘./bravo’*** is the name of the module to load without the file extension, and also indicates the file path.

### Named Imports

Named import allows the use of objects stored in a variable in external JavaScript files. The keyword ***import*** is used, and the variables are enclosed in a set of curly braces.

|  |  |
| --- | --- |
| 1. | import {alphaObject, bravoFunction} from ‘./charlie’; |

In the code above, the ***import*** keyword is used, then the items to be imported are separated with commas inside the curly braces. These items can be variables, objects, functions, etc.

### Importing all the exported items from an external file

the use of the \* selector imports all the items that are marked for export in an external file.

|  |  |
| --- | --- |
| 1. | import \* from ‘module\_name’; |

### importing single functions

|  |  |
| --- | --- |
| 1. | import {funcA} as name from ‘module\_name’; |

In the code above, the ***import*** keyword is used. The curly braces contain the name of the exported function, and the name of the external file comes after as name from…

### Importing several functions

|  |  |
| --- | --- |
| 1. | import {funcA, funcB, funcC} as name from ‘module\_name’; |

In the code above, three exported functions are imported using the same syntax as importing a single function. Within the curly braces, the different functions to be imported are separated with commas.

|  |  |
| --- | --- |
| 1. | Import {alpha, bravo} from ‘./Menu’; |
| 2. | Console.log(alpha); |

In the code above, the variables ***alpha*** and ***bravo*** are imported from the external file ***Menu***, the file path is specifiec with ‘./Menu’ syntax. They then can be treated as any other variable. In this case, the variable ***alpha*** is logged to the console.

### Require()

To import a module which has been exported, the ***require()*** function is used. It uses a file path which points to the original module file.

|  |  |
| --- | --- |
| 1. | const bravo = require(‘./alpha.js’); |
| 2. | function charlie () { |
| 3. | console.log(‘the contents of the imported module alpha is ‘ + bravo); |
| 4. | } |
| 5. | charlie (); |

In the code above, the variable ***bravo*** is created and assigned the value of the ***require*** function which imports the module ***alpha***.

Within the parentheses are the single quotes which contain a period, a forward slash and then the name of the file with the extension.

The function ***Charlie*** is created. Its code block logs to the console a string with bravo’s value concantenated to it.

The last line calls the function ***Charlie***.

### Packages

A ***package*** is one or more modules with a special file called the ***manifest***. This file keeps track of how all the modules interact with each other (these interactions are called dependencies), along with other metadata about the project. In JavaScript, the file is named package.json.

**.Shapes**

**Package.json**

**Circle.js**

**Square.js**

**Triangle.js**

In the example above, the package is named shapes, which is like a folder that contains the manifest file named package.json, and three modules; circle.js, square.js, and triangle.js.

***Library*** A library is a collection of packages.

### API

The API (Application Programming Interface) is the toolbox which allows the use of what is contained in a library. An API is like a woodworking workshop. It contains the environment and tools that create the code.

### Frameworks

A framework is an existing template for an application. It is a generic structure designed to used as a starting point for the creation of software with a specific purpose.

## Asynchronous operations

JavaScript is a synchronous language, which means that it can only process one statement at a time. If a statement takes a long time to process, or requires some action by the user, nothing else can happen. Until the statement returns some result, the code is blocked and stuck.

Asynchronous code is a new development in software that can keep the code from being blocked. Since the code needs a response in order to continue, the response that is given with asynchronous operations is a response that the result of the operation will be sent as soon as possible. This allows for the code to continue to be executed.

In the analogy below, there are three types of code:

* ***producer*** code – This is an operation that takes a long time to complete.
* ***consumer*** code – This is code that needs the result of the producer code in order to work.
* ***promise*** – The promise is an object that is returned to the consumer code by the producer code. It tells the consumer code the actual result will be sent to it as soon as it is available.

Event Loops

JavaScript is non-blocking; instead of stopping the execution of code while it waits for a calculation to be completed, event loops are used to allow the execution of the code to continue while waiting for the calculation to finish.

Asynchronous coding allows the code to be executed out of sequence.

|  |  |
| --- | --- |
| 1. | console.log(‘Plant corn.’); |
| 2. | console.log(‘Water the plants.’); |
| 3. | console.log(‘Add fertilizer.’); |

In the code above, the strings will be logged to the console in the order they were written in. However, if the third line (‘Add fertilizer.’) was to be printed second, some asynchronous code would be needed.

|  |  |
| --- | --- |
| 1. | console.log(‘Plant corn.’); |
| 2. | setTimeout(function() { |
| 3. | console.log(‘Water the plants.’) |
| 4. | }, 3000); |
| 5. 6. | console.log(‘Add fertilizer.); |

The code above uses a ***setTimeout*** function on the code that prints ‘Water the plants.’. The code will execute the first line to print ‘Plant corn.’, then will come to the setTimeout function which will tell it to pass over it and proceed to the next line of code, which is the ‘Add fertilizer.’ line, and when the code is all done, return to the setTimeout function and execute the code block to print ‘Water the plants.’.

Async / Await

Callbacks were originally used to handle asynchronous operations, but they encourage complexity by nesting code blocks within other code blocks.

JavaScript was updated to version ES6 to add Promises, which allows the same functionality as callbacks but with code that is easier to read. JavaScript’s update to version ES8 now adds a new syntax to handle asynchronous actions as the syntactic sugar element of async / await.

Async / Await doesn’t add new functionality, but is a new way of making current functionality easier to read and scale. It is used with promises and generators.

### The Async keyword

The ***async*** keyword placed before a function means that the function will always return a promise.

An ***async function*** will return in one of three ways:

If there is nothing returned from the function, the async function will return a promise with a resolved value of undefined.

If there is a non-promise value returned from the function, the async function will return a promise resolved to that value.

If a promise is returned from the function, it will simply return that promise.

### The Await operator

The ***await*** keyword is an operator, which means that it returns the resolved value of a promise. It can only be used inside an async function.

Since the amount of time it takes for a promise to resolve is unknown, await pauses the execution of the async function until a given promise is resolved.

|  |  |
| --- | --- |
| 1. | const alpha = new Promise (resolve => |
| 2. | { |
| 3. | //\* This is asynchronous code that takes some time to finish being executed. \*// |
| 4. | }); |
| 5. | } |
| 6. 7. | async function bravo () |
| 8. | { |
| 9. | let charlie = await alpha(); |
| 10. | console.log(alpha); |
| 11. | } |

In the code above, a new promise called ***alpha*** is created. This function takes some time to run. Below that, an ***async*** function named ***bravo*** is created. Inside its code block, the variable ***charlie*** is assigned the value of ***alpha*** when it is eventually returned, and using the ***await*** keyword, pauses the execution of the rest of the code block until ***alpha’s*** value is returned.

### Dependent Promises

A dependent promise is an asynchronous action which uses the returned value of another promise as an input.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | function alpha() { | |
| 2. | | bravoPromise () | |
| 3. | | .then((firstValue) => { | |
| 4. | | console.log(firstValue); | |
| 5. | | return charliePromise(firstValue); | |
| 6. 7. | | }) | |
| 8. | | .then((secondValue) => { | |
| 9. | | console.log(secondValue); | |
| 10. | | }); | |
| 11. | | } | |

The code above uses native promise syntax.

A function named alpha was created. Inside its code block are two functions which return promises; bravoPromise() and charliePromise().

The first promise, bravoPromise() is invoked in the next line with the use of the .then() keyword. The .then() keyword also ensures that bravoPromise() is resolved.

The callback for the first .then() has two actions. The first is to log the argument firstValue to the console. The second is to return charliePromise(firstValue).

Then, outside of bravoPromise, another .then() is used to invoke and resolve the second promise, charliePromise(), and then log it to the console.

|  |  |
| --- | --- |
| 1. | async function alpha() { |
| 2. 3. | let firstValue = await bravoPromise(); |
| 4. | console.log(firstValue); |
| 5. 6. | let secondValue = await charliePromise(); |
| 7. | console.log(secondValue); |
| 8. | } |

The code above is the same as the previous code, but written using async function syntax.

The function named alpha is marked as an async function with the use of the async keyword. Inside of alpha’s code block the variable firstValue is created and assigned the resolved value of the bravoPromise() promise with the use of the await keyword.

A second variable named secondValue is then created and using the keyword await and is assigned the resolved value of the second promise charliePromise(). Finally, the value of secondValue is logged to the console.

Promise objects

A promise is an object that represents the outcome of an asynchronous operation. The outcome of the operation can be either success or failure. It can be in one of three states:

* ***Pending*** – This is the initial state. The operation hasn’t been finished yet.
* ***Fulfilled*** – The promise has been completed successfully and now has a resolved value.
* ***Rejected*** – The operation has failed and the promise has a reason for the failure, usually an error of some kind.

A promise is ***settled*** when it is no longer pending. It is either fulfilled or rejected. An example of this would be a dishwasher.

* ***Pending*** – The dishwasher is running but has not yet finished its cycle.
* ***Fulfilled*** – The dishwasher is finished running and is full of clean dishes.
* ***Rejected*** – The dishwasher encountered a problem, like it didn’t have detergent, and returns the unclean dishes.A promise object represents the outcome of an asynchronous operation and holds one of the three possible values:
* **The operation is pending and hasn’t been performed yet.**
* **The operation successfully met the condition and has been fulfilled.**
* **The operation did not meet the condition and has failed.**

|  |  |
| --- | --- |
| 1. | function successCallback (result) { |
| 2. | console.log (“The result of the operation is “ + result”); |
| 3. | } |
| 4. 5. | function failureCallback (error) { |
| 6. | console.error (“There was an error. The error was “ + error); |
| 7. | } |

### The Executor function

In order for the promise object to arrive at one of the three conditions, a built-in function is needed. This is the executor function, and it includes a condition that is evaluated. It is run automatically when the constructor is called.

If the condition is true, the promise is set to the Resolve() function. If the result of the evaluation is false, the promise is set to the Reject() function.

A Promise object is comprised of several parts:

* **The *new* keyword.**
* **The *Promise* constructor method.**
* **The *executor* function.**
* **The *resolve* function.**
* **The *reject* function.**

|  |  |
| --- | --- |
| 1. | const alpha = (resolve, reject) => { |
| 2. | if (someCondition) { |
| 3. | resolve (‘The promise is resolved.’); |
| 4. | } else { |
| 5. | reject {‘The promise is rejected.’); |
| 6. | } |
| 7. | } |
| 8. 9. | const bravo = new Promise(alpha); |

In the code above, the variable ***alpha*** is created as a function. This function sets up the condition that is asked and the response if the condition is met successfully or the response if the condition is not met.

It has the parameters resolve and reject. It then uses an ***IF*** conditional to return a value to either the resolve or reject parameter.

If the condition specified in the ***if*** conditional is true, then the ***resolve*** function is returned. If it isn’t

true, then the ***reject*** function is returned.

A promise object is created using the new Promise (note the capital P) keyword and assigned to the variable ***bravo***. The promise method uses the function that was created previously, which returns a value in answer to a condition, as an input. The promise then returns this value, either resolved or rejected, as its own value.

|  |  |
| --- | --- |
| 1. | function alpha (num) => { |
| 2. | return new Promise ( (resolve, reject) => { |
| 3. | if (num === 0) { |
| 4. | resolve (‘Zero’); |
| 5. | } else { |
| 6. | resolve (‘Not zero.’); |
| 7. | } |
| 8. | }) |
| 9. | } |

The above code shows how an ***if/else*** statement can be used to determine the value of only the ***resolve*** parameter, instead of both the ***resolve*** and the ***reject*** parameters.

### Consuming Promises

Promises control the flow of the script in the same way that if/else statements do. If a test condition is true, a particular code block is run. If the test condition fails, then some other code block is run.

### Node.JS setTimeOut() function

The ***setTimeOut()*** function is a Node API that delays the execution of code by a specified amount of time. It has two parameters:

* **The name of the function to be delayed.**
* **The amount of time (in milliseconds) of the delay.**

Essentially, the specified function is put at the bottom of the list of pending operations.

### .then() function

The ***.then()*** function is a method that comes with the Promise object. The .then() function allows the next steps that are to be run on the value when it is eventually returned.

* **It takes a callback function and returns another promise.**
* **It creates a response to either of the two values that are returned from the Promise It takestwo callback functions as arguments. These callback functions are called handlers.**

**The first handler is the success handler, sometimes referred to as onFulfilled. It sets up the code for when the Promise is resolved.**

The second handler is the ***failure*** handler, sometimes referred to as ***onRejected***. It contains the logic for when the Promise is rejected.

***Promise that you will go to the store.***

***.then (you’ll make dinner with the ingredients that were bought)***

***.then (I’ll do the washing up)***

|  |  |
| --- | --- |
| 1. | const alpha = new Promise ((resolve, reject) => { |
| 2. | let num = math.random(); |
| 3. | if(num <5) { |
| 4. | resolve(‘The number is less than 5.’); |
| 5. | } else { |
| 6. | reject(‘The number has been rejected because it is not less than 5.’); |
| 7. | } |
| 8. | }); |
| 9. 10. | const handleSuccess = (resolvedValue) => { |
| 11. | console.log(resolvedValue); |
| 12. | }; |
| 13. 14. | const handleFailure = (rejectionReason) => { |
| 15. | console.log(rejectionReason); |
| 16. | }; |
| 17. 18. | alpha.then(handleSuccess, handleFailure); |

In the code above, a promise object named ***alpha*** was created and given a callback function. The callback function is enclosed within the outer set of parentheses. Notice that in the first line the callback’s arguments start with two opening parentheses, but are closed with only one. The other, outer parentheses is closed at the end of the callback block several statements later. This outer set of parentheses contains the entire callback function.

The callback itself takes two arguments, ***resolve*** and ***reject***, which are contained within their own set of parentheses. After the callback’s arguments is the fat arrow symbol, then the curly braces that contain the conditional, the resolve value, and the reject value.

To break this down even further:

|  |  |
| --- | --- |
| 1. | const executorFunction = (resolve, reject) => {executor function code block}; |

The line of code above is the executor function used by the promise object as an argument:

|  |  |
| --- | --- |
| 1. | const fancyPromise = new Promise(executorFunction); |

Instead of writing the executor function separately from the promise object, they can be combined using arrow syntax:

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | const fancyPromise = new Promise(executorFunction (resolve, reject) // in arrow syntax, the name of the function is not necessary so it is eliminated: | |
| 2. 3. | | const fancyPromise = newPromise((resolve, reject) => | |
| 4. | | { | |
| 5. | | // executor function’s code block // | |
| 6. | | If conditional | |
| 7. | | resolve code | |
| 8. | | reject code | |
| 9. | | }); // This closing curly brace closes the executor function’s code block. This closing parentheses closes the very first opening parentheses, which starts the promise’s argument. | |

### The onFulfilled() function and the onRejected() function

The ***onFulfilled()*** function is a generic term for the code block that runs when the result of the condition’s evaluation is true. In the previous example, a function named handleSuccess was created with code that was meant to be executed if the condition tested to be true.

Like onFulfilled(), onRejected() is a general term for the block of code to be executed if the condition turns out to be false.

### Error Handling

There are three types of programming errors

* **Syntax errors**
* **Runtime errors**
* **Logic errors**

### Syntax errors

These occur when the code is interpreted, and involve grammatical mistakes with the language.

|  |  |
| --- | --- |
| 1. | var alpha = function (length, width) { |
| 2. | return length \* width; |

The code above generates a syntax error, because there is no closing curly brace.

### Runtime errors

Runtime errors, also known as exceptions, occur during the code’s execution after being interpreted or compiled. They are commonly referred to as bugs.

|  |  |
| --- | --- |
| 1. | var alpha = 5; |
| 2. | bravo (20,5); |

The code above generates a runtime error, because the function bravo is being called but the function itself doesn’t exist.

### Logical errors

Logic errors are errors that cause the wrong result to be returned. The syntax is correct, and the code compiles correctly so no errors are produced. They are errors because the programmer wrote code that is not useful for the goal.

An example of a logic error would be multiplying two values when they should be divided.

### Using the catch() function with Promises

The catch() function helps to create simpler code. It takes only one argument, ***onRejected***. In the case of a rejected promise, this failure handler will be invoked along with the reason for the rejection. Using .catch() does the same thing as .then(), but only uses one handler – the failure handler.

|  |  |
| --- | --- |
| 1. | alpha |
| 2. | .then((resolvedValue) => { |
| 3. | console.log(resolvedValue); |
| 4. | }) |
| 5. 6. | .catch((rejectionReason) => { |
| 7. | console.log(rejectionReason); |
| 8. | }); |

In the code above, the promise ***alpha*** randomly either resolves or rejects. The success handler is passed to the ***.then()*** function, and the failure handler is passed to the .***catch()*** function.

Handling asynchronous errors

When .catch() is used with a long chain of promises, it doesn’t indicate where in the chain the error was thrown.

### Try / Catch

The try/catch syntax combination narrows down where in the code any errors may occur.Try/Catch allows error handling for both synchronous and asynchronous code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. | | const paymentPromise = //\* This is a pre-written promise that contains confirmation that the customer has paid for the order with a yes or no. \*// | | |
| 2. 3. | | const orderPromise = //\* This is a pre-written promise that contains a customer’s order information. \*//. | | |
| 4. 5. | | async function customerOrder () { | | |
| 6. | | try { | | |
| 7. 8. | | const paymentConfirm = await paymentPromise(); | | |
| 9. | | if (paymentConfirm = true) { | | |
| 10. | | console.log(‘Payment has been made. Proceed with the order.’); | | |
| 11. | | } else { | | |
| 12. | | console.log(‘Payment has not been made. Place the order on hold until the issue is resolved.’); | | |
| 13. 14. | | const orderInfo = await orderPromise(); | | |
| 15. | | console.log(`${orderPromise} has been ordered.`); | | |
| 16. 17. | | } catch (error) { | | |
| 18. | | console.log(error); | | |
| 19. | |  | } | |
| 20. | | } |  | |

In the code above there are two prewritten promises: ***paymentPromise*** and ***orderPromise***.

The function ***customerOrder*** is created and made to be asynchronous with the use of the async keyword.

A ***try*** statement is then used. Within the try’s code block, the variable ***paymentConfirm*** is created and is given the resolved value of the ***paymentPromise*** promise. An ***If / Else*** conditional is set up to respond to either if payment has been made or not.

The ***try*** statement is also given the variable ***orderInfo***, which waits for the resolved value of the ***orderPromise*** promise, and then logs to the console the message that the order has been made.

### Handling independent promises

If multiple promises are being handled at the same time, but they do not rely on each other to run, they can be executed at the same time, or concurrently.

|  |  |
| --- | --- |
| 1. | async function wait() { |
| 2. | const alpha = await promiseOne(); |
| 3. | const bravo = await promiseTwo(); |
| 4. | console.log(firstValue, secondValue); |
| 5. | }a |

The code above creates the async function ***wait***, which contains two promises (***promiseOne*** and ***promiseTwo***) which are assigned to the variables ***alpha*** and ***bravo***. Because the ***await*** keyword is used for both, the first promise must be resolved before the second promise can be run. When both are successfully resolved, the values for both are logged to the console.

|  |  |
| --- | --- |
| 1. | async function concurrent () { |
| 2. | const alpha = promiseOne(); |
| 3. | const bravo = promiseTwo(); |
| 4. | console.log(await promiseOne, await promiseTwo) ; |
| 5. | } |

The code above is the same as the previous code, except that it is set up so that the promises are executed at the same time by putting the ***await*** keywords inside of the console.log method.

### Separation of Concerns

Separation of Concerns is a general programming principle which states that the code should be organized into coherent sections, each of which are designed to do specific things. This organization is useful to help navigate the code for debugging purposes.

### Dependent Promises

There are circumstances when the execution of a promise depends on whether or not a prior promise has been successful.

An example of this would be ordering pizza:

Toppings are chosen. This can take some time, and the pizza order may fail if it decided to order Chinese food instead.

The toppings are chosen and the order is placed. However, the order may fail if the restaurant doesn’t have the toppings that are wanted.

The pizza is finished. The order may fail if it is unable to be paid for.

|  |  |
| --- | --- |
| 1. | const pizza = (resolve, reject) => { |
| 2. | if (toppingsChosen) { |
| 3. | resolve(orderPizza( if (PizzaPaidFor){ |
| 4. | resolve (eat pizza.) |
| 5. | } else { |
| 6. | reject (Find money to pay for pizza.); |
| 7. | }); |
| 8. | } else { |
| 9. | reject(‘The restaurant didn’t have the toppings.’); |
| 10. | } |
| 11. | }; |

In the code above, if/else statements are nested inside of each other because each if/else statement depends on the result of the if/else statement before it. However, this can be difficult to debug.

### Chaining Multiple Promises

When one promise uses the result of another promise as an input, or when several promises need to be executed in a certain order, they are chained together rather than nesting them inside of each other. Chaining promises makes the code easier to debug and edit. The process of chaining promises together is called composition.

|  |  |
| --- | --- |
| 1. | checkInventory(order) |
| 2. | .then((resolvedValueArray) => { |
| 3. | processPayment(resolvedValueArray) |
| 4. | .then((resolvedValueArray) => { |
| 5. | shipOrder(resolvedValueArray) |
| 6. | .then((successMessage) => { |
| 7. | console.log(successMessage); |
| 8. | }); |
| 9. | }); |
| 10. | }); |

The code above has nested promises.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | checkInventory(order) | |
| 2. | | .then((resolvedValueArray) => { | |
| 3. | | return processPayment(resolvedValueArray); | |
| 4. | | }) | |
| 5. | | .then((resolvedValueArray) => { | |
| 6. | | return shipOrder(resolvedValueArray); | |
| 7. | | }) | |
| 8. | | .then((successMessage) => { | |
| 9. | | console.log(successMessage); | |
| 10. | | }); | |

The code above is the same code, but rewritten using chained promises.

### Promise.all()

The promise.all() function makes use of concurrency, which is when multiple asynchronous operations happen at the same time instead of one at a time.

If every promise in the array resolves, the promise.all() will resolve with an array containing the resolve value from each promise in the argument array.

***If any of the promises in the array fails, the promise.all() will fail.***

|  |  |
| --- | --- |
| 1. | let alpha = Promise.all ( [ returnsPromiseOne(), returnsPromiseTwo(), returnsPromiseThree() ] ); |

In the code above, the variable ***alpha*** has been assigned the ***Promise.all*** function, which has an array as a single parameter. The array contains three promises as elements: ***returnsPromiseOne()***, r***eturnsPromiseTwo()***, and ***returnsPromiseThree()***.

|  |  |
| --- | --- |
| 1. | alpha |
| 2. | .then ((arrayOfValues) => { |
| 3. | console.log(arrayOfValues); |
| 4. | }) |
| 5. 6. | .catch((rejectionReason) => { |
| 7. | console.log(rejectionReason); |
| 8. | }); |

The code above shows two methods of the ***alpha*** object. The ***.then()*** method is the success handler which will log the array of values.

The ***.catch()*** method is the failure handler which logs the first rejection method if any of the promises rejects.

|  |  |
| --- | --- |
| 1. | Await Promise.all() |
| 2. | The await keyword can be added to the Promise.all() function to have the code wait until all the promises are resolved. |
| 3. | let alpha = await Promise.all ( |
| 4. | [returnsPromiseOne(), returnsPromiseTwo(), returnsPromiseThree()] |
| 5. | ); |

The code above is the same as the previous code, but adds the await keyword before the

Promise.all function.

## Network Fundamentals

A computer network is a group of two or more computers that are connected by a common communications protocol with the purpose of sharing digital resources.

The simplest form of network would be two computers connected with a cable, whose operating systems acknowledge the connection and can see each other.

### Nodes

A node is either an end user of a network or a connected point between end users that redistributes the connection between end users. Nodes include the end users, and also any connected hardware that is capable of creating, receiving, or transmitting information over the connection. This includes hardware like modems, routers, switches, hubs, or firewalls.

### Communications protocol

A communications protocol defines all the elements of how two or more entities exchange information. These elements include the language, rules, syntax, and methods of synchronizing how the information is transmitted and then received by the individuals using the network.

### Hosts

A host is any device connected to a computer network. It is a node that participates in user applications, either as a server, as a client, or as both.

Hardware that is connected to a network but doesn’t participate in application-level functions, such as modems, ethernet hubs, or network switches are nodes, but not hosts.

### Clients

A client is computer hardware and/or software that requests services made available by a server.

#### Front End Programming

Also known as client-side programming, front end programming is what the user interacts with in their browser, and all the ways this interface is designed and written. This includes languages, typically HTML, CSS, and JavaScript.

### Servers

A server is computer hardware and/or software that provides services and functionality over a network to clients. These services include sharing resources or data, or performing computational functions and returning a result.

#### Back End Programming

Also known as server-side programming, back end programming is all the necessary processes and functions that are hidden from the user and run behind the scenes to actually make the application work. A traditional back end is a mix of server, databases, APIs and the computer’s operating system.

* ***Servers*** • ***Databases*** • ***Middleware*** -
* ***Programming*** languages and Frameworks -
* ***APIs*** -

### The Client / Server networking model

The client/server model describes the cooperative relationship between clients and servers.

Clients always initiate communication sessions by sending a request to the server.

### API

An API is server-side software that connects a client to a server and sends information back and forth between them. It defines the kinds of requests that can be made to the server.

An analogy of this would be a customer in a restaurant and a cook in the restaurant’s kitchen. The customer doesn’t tell the cook directly what they want to eat, and the cook doesn’t bring the food out to them. Rather, an intermediary is used, in the form of a waiter. The waiter takes the customer’s order ( a request) and brings it to the kitchen. When the order is finished, the waiter either takes the finished food from the kitchen and brings it to the customer as a resolved response, or takes an explanation why the order couldn’t be made, as an error. When the waiter told the cook what the customer wanted, the cook replied that they were out of that item, perhaps.

An API is also a set of protocols and rules. If the restaurant customer tries to order something in French, but the waiter doesn’t speak French, that would make the transaction unable to be completed. If the customer tried to order something that wasn’t on the menu, that would also make the transaction fail.

An API is also a set of premade blocks of code that do specific things. In the way that functions in JavaScript are blocks of code that are executed by calling the function anywhere in the code, APIs are implemented by function calls.

### API Endpoints

An endpoint is the software/hardware where communication with the outside world occurs. It’s the point of entry. lt is the location where the API sends the request to, and where the resource is located.

|  |  |
| --- | --- |
| 1. | https://forum.freecodecamp.org |

The code above is an example of an endpoint. When a user connects to this location, the server is going to return the options that are available for that website.

### AJAX

AJAX is an acronym for Asynchronous JavaScript and XML. It is describes a set of development techniques using languages like XML, HTML, CSS, and JavaScript to build websites and web applications.

* **HTML / XHTML is the main language.**
* **CSS is used for the presentation.**
* **The Document Object Model (DOM) is used to dynamically display and update data and its**

**interaction.**

* **XML is used for the interchange of data and XSLT for its manipulation. However, JSON isstarting to replace XML because JSON is closer in form to JavaScript.**
* **The XMLHttpRequest object is used for the asynchronous communication.**
* **JavaScript is used to bring all of these elements together.**

### Protocols

In computer networking, a protocol is a standardized set of rules for formatting and processing data. Protocols act as a shared third language, allowing a wide ranging of different hardware and software to be able to exchange data.

### Web Browsers

A web browser is an application designed to access information on the World Wide Web. Using HTTP, the browser sends a request to a server, which then delivers the requested information back to the client and displayed on the user’s device using the web browsers rendering engine.

### HTTP

HTTP stands for Hyper Text Transfer Protocol, which is the foundation of the World Wide Web. It functions as a request-response protocol in the client/server computing model, which means that the client sends a request to a server, which the server then performs and sends a response back to the client.

### URL

A Uniform Resource Locator, also known as a web address, is a line of code which specifies the exact location of a specified web resource and a mechanism for retrieving it.

|  |  |
| --- | --- |
| 1. | http://www.example.com/index.html |

The code above is an example of an URL. The first section indicates the protocol being used (http). ***Www.example.com*** is the hostname, which is separated from the requested file name with a forward slash.

### JSON

JSON stands for JavaScript Object Notation, and is a text-only data format suitable for transporting data to and from a server, and is a slightly stricter form of JavaScript.

JSON objects are enclosed with curly braces and may contain one or more property:value pairs.

Names in JSON require double quotes, not the single quotes which are acceptable in JavaScript.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | var jsonObject = { | |
| 2. | | “name” : “Joe”, | |
| 3. | | “gender” : “male”, | |
| 4. | | “age” : 55, | |
| 5. | | “charming” : true | |
| 6. | | } | |

|  |  |  |
| --- | --- | --- |
| are written in the HTML file and are added to HTML elements.   |  |  | | --- | --- | | 1. | <element event=”some JavaScript”> | |

The code above shows a JSON object named ***jsonObject***. The object code block is contained within a pair of curly braces, and like JavaScript, the key:value pairs are separated with commas.

The key:value elements are separated with colons. Strings are only enclosed with double quotes.

Single quotes, which are OK in JavaScript, are not allowed in JSON.

### HTTP Requests

An HTTP request is a command sent to a server on the web which tells the server to do something. There are many types of HTTP requests, but there are four that are most commonly used:

* ***Get*** – Get only retrieves data from a source.
* ***Post*** – Similar to a Get request, Post requests retrieve data from source. The difference is that Post can introduce new information to other sources in addition to requesting it.
* ***Put*** – This is most often used to update an existing resource.
* ***Delete*** – This is used to delete a resource specified by its URL.

HTTP requests are done by using the ***XMLHttpRequest()*** constructor, which is an object that is used to interact with servers. Data can be retrieved without needing to refresh the page.

To create an ***HTTP POST*** request with JavaScript, five things are required:

* **A request type**
* **A response type**
* **A request URL**
* **A request body**
* **A handler for the response data**

#### Events

HTML events are changes in the state of HTML elements that JavaScript can react to. An event can be something the browser does, or something that a user does. Some examples of HTML events are:

* **An HTML web page has finished loading.**
* **An HTML input field was changed.**
* **An HTML button was clicked.**

#### Event Handlers

JavaScript can execute code when events are detected through the use of event handlers, which

The code above, which is HTML, shows the JavaScript written with double quotation marks, although single quotation marks are allowed as well.

|  |  |
| --- | --- |
| 1. | <button onclick=”document.getElementById(‘demo’).innerHTML = Date()”> The time is? </button> |

The HTML code above shows an ***onclick*** attribute added to a button HTML element. Some common HTML events are:

* ***onchange*** – When an HTML element has been changed.
* ***onclick*** – When the user clicks an HTML element.
* ***onmouseover*** – When the user moves the mouse cursor over an HTML element.
* ***onmouseout*** – When the user moves the mouse cursor away from an HTML element.
* ***onkeydown*** – When the user pushes a keyboard key.
* ***onload*** – When the browser finishes loading the page.

Event handlers can be used to handle and verify user inputs, actions, and also actions the browser does, like:

* **Things that should be done every time a page loads.**
* **Things that should be done when the page is closed.**
* **Actions that should be performed when a user clicks a button.**
* **Verify content the the user has input.**

### Query Strings

A query string is a way to send additional information along with a request using a specific syntax. A key/value pair, joined with a = is added after the url, using a ? to separate the url and the key/ value pair.

|  |  |
| --- | --- |
| 1. | https://api.datamuse.com/words?key=value |

In the code above, the url comes first, and is then ended with a question mark. The key/value pair comes after the question mark.

### POST requests

### fetch() GET Requests

The fetch function:

Creates a request object that contains the relevant information that an API needs.

Sends that request object to the API endpoint.

Returns a promise that ultimately resolves to a response object, which contains the status of the promise with information that the API sent back.

|  |  |
| --- | --- |
| 1. | fetch(‘http://api-to-call.com/endpoint’).then (response) => { |

/\* The line above sends the request \*/

|  |  |
| --- | --- |
| 1. | if (response.ok) { |
| 2. | return response.json(); |
| 3. | } |

/\* The code block above converts the response object to JSON \*/

|  |  |
| --- | --- |
| 1. | throw new Error (‘Request failed!’); |
| 2. | }, networkError => console.log(networkError.message) |

/\* The code above handles errors. \*/

|  |  |
| --- | --- |
| 1. | ).then (jsonResponse => { |

// Code to execute with jsonResponse

|  |  |
| --- | --- |
| 1. | }); |

/\* The code above handles success. \*/

Adobe Illustrator ExtendScript

ExtendScript is a version of JavaScript used by Adobe applications for to create user interfaces. It differs from JavaScript in that it still uses some of the older syntax methods. JavaScript files written with ExtendScript are run from within Illustrator by going to Menu – File – Scripts.

The Illustrator Scripting Object Model

An object model is a description of how a system is organized. It includes the system’s different parts, the function of these parts, and how different parts interact with each other.

Object Hiearchy

Top-Level Objects

These objects are used to access global information about the Illustrator application or an individual document.

application – The properties of the application object give access to global values, such as preferences, text fonts, printers, Illustrator’s installation path, and any open documents. The current active document is the document that is displayed and accepting user input. It allows the manipulation on application-wide actions such as opening files.

document – The document object is accessed through the application object, and allows access to the document’s content, such as the current selection, all contained art objects (page items), layers (including the currently active layer), the path of the document, saving the document, printing the document, etc.

layer – The layer object, which is accessed through the document objcet, provides access to the contents of a specific layer. The layer object controls things such as whether the layer is visible or locked, its opacity, its z order position, etc.

Layers that are contained within other layers (sub-layers) are accessed by first writing the path from the top-level layer:

app.activeDocument.layers.getByName("Body Colors").layers.getByName("Light Blue");

In the code above, the top-level layer is accessed by name (“Body Colors”), and the particular layer contained within in (not a group), is then accessed using the layers keyword again.

A layers visiblity in Illustrator is controlled using the .visible property:

var alpha = app.activeDocument.layers.getByName(“layer name”)

alpha.visible = false;

The actual content of an illustrator document is called the Artwork Tree, it it contains the following objects:

page items -

compound path item –

graph item –

group item – Objects that are grouped together. .groupItems.getByName(‘name’); also .pageItems.getByName(‘name’); also

legacy text item –

mesh item –

non native item –

path item – Objects created with the pen tool. .pathItems.getByName(‘name’);

placed item –

plugin item –

raster item –

symbol item –

text frame – .textFrames.getByName(‘name’);

The visibility of illustrator art is controlled using the item’s .hidden property:

ExtendScript UI

statictext

Statictext elements are areas where text is shown to the user, but cannot be edited.

edittext

Edittext elements are areas where the user enters information in the interface.

var alpha = customerInfo\_outer.add("statictext", undefined, "Rep name");

var bravo = customerInfo\_outer.add("edittext", undefined, "");

In the code above, the variable alpha creates a statictext box with the text Rep name displayed.

The variable bravo creates an edittext box where the user types in information.

To use the information the user types into the edittext box, the .text property is accessed.

bravo.text // This represents the information the user enter in the edittext box.

var bravo = inkBox\_Master\_A1.add("edittext", undefined, ""); // This creates the edittext box and assigns it to the variable bravo. It is accessed using .text.

var alpha = app.activeDocument.textFrames.getByName(“Illustrator layer name”); // The variable alpha gets access to the text frame in the currently open illustrator document with the name “Illustrator layer name”. The contents of the text frame are referred to with the .contents property.

alpha.contents = bravo.text;

In the code above, the variable alpha is assigned to the name of the layer in Illustrator where the text is to go. The .contents property is used to put the text that was entered in the edittext box, which is accessed with the .text property.

console.log (bravo.text);

the .text property of the edittext element bravo is the content of the box that the user entered.

button

checkbox

radiobutton

listbox

dropdownlist

progressbar

## Script UI

Script UI is a JavaScript module in the Adobe CS/CC family of applications to create control elements in windows that pop up for the user to interact with.

Windows

It defines three types of windows:

* ***Dialog*** – This is a modal window, which means that it does not allow activity in any other window until it is dismissed. “dialog”
* ***Palette*** – This is a modeless window, which means that it allows activity in other windows. It is also called a palette. “palette”
* ***Main* Window** – This creates a window which can be used as the Adobe application’s main window. It is not commonly used by script developers.

|  |  |
| --- | --- |
| 1. | var alpha = new Window ("dialog", "title bar text"); |

The variable ***alpha*** is declared and ***‘new Window’*** creates a new window. Inside the parentheses, the first string is the type of window (***dialog***, ***palette***, or ***main window***.) The second string is the text that goes in the window’s title bar. The two strings are separated by a comma.

|  |  |
| --- | --- |
| 1. | alpha.show (); |

Initially, new windows are hidden. The show method makes them visible and responsive to user interaction.

Containers

***Containers*** are objects that are used to organize other elements are treat them as a single unit. There are three kinds of containers:

* ***Group*** – A group is the simplest container. “group”
* ***Panel*** – a panel is a group with a frame around it. “panel”
* ***TabbedPanel*** – a tabbed panel is a frame that contains Tab elements. A Tab element is a frame with a title in the selection tab.

|  |  |
| --- | --- |
| 1. | var alpha = new Window ("dialog", "Window Text"); |
| 2. | alpha.orientation = "row"; |
| 3. | var group1 = alpha.add ("panel"); |
| 4. | group1.orientation = "column"; |
| 5. | group1.add ("statictext", undefined, "Panel 1"); |
| 6. | group1.add ("statictext", undefined, "Panel 2"); |
| 7. | group1.add ("statictext", undefined, "Panel 3"); |
| 8. 9. | var group2 = alpha.add ("panel"); |
| 10. | group2.orientation = "column"; |
| 11. | group2.add ("statictext", undefined, "Group 2A"); |
| 12. | group2.add ("statictext", undefined, "Group 2B"); |
| 13. | group2.add ("statictext", undefined, "Group 2C"); |
| 14. 15. | var group3 = alpha.add ("panel"); |
| 16. | group3.orientation = "column"; |
| 17. | group3.add ("statictext", undefined, "Group 3A"); |
| 18. | group3.add ("statictext", undefined, "Group 3B"); |
| 19. | group3.add ("statictext", undefined, "Group 3C"); |
| 20. 21. | alpha.show (); |

In the example above, the first line declares the variable ***alpha*** and creates the window. Its type is dialog and it displays the text “Window Text”.

The second line sets the alignment of the elements of alpha to be in a row.

***Var group1*** declares the variable group1 and creates a panel-type container in the ***alpha*** window. Elements are added to the ***group1*** variable using the .add method. Within the parentheses are three elements separated by commas:

* **the type of element**
* **the location of the element**
* **the text string displayed by the element.**

In this case, the location is assigned the value ***undefined***. This will allow the placement of the element is done automatically.

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | var bic\_Window = new Window ("dialog", "Display text"); | |
| 2. | | bic\_Window.orientation = "column"; | |
| 3. | | var Name = bic\_Window.add ("statictext", undefined, "Display text"); | |
| 4. | | var Field = bic\_Window.add ("edittext"); | |
| 5. | | Field.characters = 40; | |
| 6. | | Field.active = true; | |
| 7. | | bic\_Window.show (); | |
| 8. 9. | | alert(Field.text); | |

The alert function is here just to demonstrate that the inputted information is accessible to the code.

Anything that uses the info the user inputs has to be put after the xxx.window.show () function in order to work.

The above collects the information entered in the “edittext” field and assigns it to the customer.name property.

## Using ExtendScript to edit an Illustrator Document’s content

There are many different Illustrator-specific objects and functions in ExtendScript.

|  |  |
| --- | --- |
| 1. | var purchaseOrder = app.activeDocument.layers[9].textFrames[2]; |
| 2. | purchaseOrder.contents = po; |

***app - activeDocument - layers[0] - textFrames[0] app*** is the application constant ***activeDocument*** refers to the currently active document object.

(You could access items at this scope if you'd like, but trust me, it's a better idea to get used to utilizing narrower scopes.)

***layers[0]*** is the first layer object in the active document.

(in javascript, counting starts at 0, so when you want the first layer, you reference it by 0, and when you want the 3rd layer, you reference it by 2, etc.)

***textFrames[0]*** as you might guess, refers to the first text frame in the first layer of the

document.(textFrame is the generic name for any kind of text object in illustrator. Regular point text, area text, text on a path, etc.)

Ok. So we've broken down the components of that variable declaration. So, at the end of that line, the ***purchaseOrder*** variable will hold the text object you created in your document. Now, in the next line, we're using the ***alert()*** method to display the ***"contents"*** of the ***purchaseOrder*** variable. The contents property of a textFrame is the text that is displayed in illustrator.

## Selecting items within groups by index number

In Illustrator, groups are selected by using the ***.groupItems[index]*** property for each group and subgroup.

The target layer in the image to the left is the highlighted layer labeled ***New contents***. It is in Layer 2, inside of the second group from the top down.

|  |  |
| --- | --- |
| 1. | var alpha = app.activeDocument.layers[0].groupItems[0].  groupItems[1].textFrames[0]; |
| 2. | alpha.contents = "New contents."; |

Here the variable ***alpha*** is declared and assigned the selection. ***App.activeDocument*** refers to whatever Illustrator file happens to be open, ***.layers[0]*** refers to the first layer in the document (from the top down), ***.groupItems[0]*** refers to the first group within that layer, the second ***.groupItems[1]*** refers to the second group within that first group, and ***.textFrames[0]*** refers to the first text frame that exists in that sub-group.

This item is selected, and ***alpha.contents*** assigns the string ***New contents*** to the contents of the variable ***alpha***.

## Selecting Items Within Groups By Group Name

|  |  |
| --- | --- |
| 1. | var alpha = app.activeDocument.groupitems.getbyname (‘sublayer name’).groupitems.getbyname (‘subsublayer name’); |

In the code above, the variable ***alpha*** is declared and assigned the selection.

***App.activeDocument*** refers to whatever Illustrator file happens to be open, ***.groupItems*** refers to a group, or sublayer of items, and the next***.getByName*** refers to the name of that group.

|  |  |
| --- | --- |
| 1. | var alpha = app.activeDocument.groupItems.getByName ('foxtrot').groupItems.getByName ('juliet'); |
| 2. | alpha.hidden = true; |

In this example, the goal is to hide the group of objects in Illustrator that are on the sublayer ***juliet***. One of the top layers is named dox, one of the sublayers is named foxtrot, and a sublayer in the foxtrot layer is named ***juliet***.

In the code above, the variable ***alpha*** is declared and assigned. .***App.activeDocument*** references the illustrator document which is currently open, ***.GroupItems.getByName (‘foxtrot’)*** specifies the first sublayer with the name ***foxtrot***, and the ***.groupItems.getByName (‘juliet’)*** specifies the group that is within the first group that was in the statement with the name ***juliet***.

If the group to be selected is within several nests of groups, the whole hiearchy of groups must be written in the code, like a file path.

If group A is the desired group of objects, and the group is part of another group, group medium, and group medium is itself a part of a larger group, group large, then the code would be this:

var alpha = app.activeDocument.groupItems.getByName(“large”).groupItems.getByName(“medium”).groupItems.getByName(“A”);

## Selecting text items by name

|  |  |
| --- | --- |
| 1. | var jobNumber = prompt ("Enter some text."); |
| 2. | var a = app.activeDocument.textFrames.getByName("job-number"); |
| 3. | if (a != undefined) { |
| 4. | a.contents = jobNumber; |
| 5. | } |

The code above prompts the user for some text, then in Illustrator selects the object with the layer name job-number, and inserts the entered text into the text box.

## Selecting paths by name

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | | var alpha = app.activeDocument.pathItems.getByName ("star"); | |
| 2. | | alpha.selected = true; | |
| 3. 4. | | var beta = app.activeDocument.pathItems.getByName("star"); | |
| 5. | | var col = new CMYKColor(); | |
| 6. | | col.cyan = 100; | |
| 7. | | col.magenta = 100; | |
| 8. | | col.yellow = 15; | |
| 9. | | col.black = 20; | |
| 10. 11. | | var swatch = col; | |
| 12. | | swatch.color = col; | |
| 13. | | swatch.name = "col"; | |
| 14. | | beta.filled = true; | |
| 15. | | beta.fillColor = swatch.color; | |

The above code selects the item with the layer name star in Illustrator, then creates a new fill color, makes a swatch from the new fill color, and then fills the selected shape with it.

## Selecting items by Group

|  |  |
| --- | --- |
| 1. | var vipColor = app.activeDocument.groupItems.getByName ("VIP Group"); |

The topmost layers in Illustrator and referred to as ***layers*** in JavaScript. Sublayers inside of the topmost layers, and sub-sublayers within them are referred to as ***groups*** in JavaScript.

## Selecting groups and sub-groups by name

|  |  |
| --- | --- |
| 1. | var vipBox = app.activeDocument.layers.getByName  ('CommonElements').groupItems.getByName ('Approval  Box').groupItems.getByName ('VIP Box'); |

In the above code, the topmost layer in the currently open Illustrator document named ***CommonElements*** is selected, then the ***groupItem*** named ***Approval Box,*** and then the group within that named ***VIP Box***.

Adobe Illustrator

This section deals with how to select and manipulate objects with javaScript that exist in an illustrator document, like text boxes or graphics.

There are two sections of the script:

UI section – The UI section just creates the user elements that bring in information to the actual javaScript code. It ends with the code that shows the window. The first object created in this section is the top-level container that contains all the other windows and sections. It is created using the new Window() keyword.

var alpha = new Window(«xxx”);

The UI section ends with the .show() method being called on the master window using the master window’s name.

alpha.show();

javaScript section – The javaScript section is where the code takes the information entered in the UI section and runs operations on it to produce the actual results that are needed.

Inserting text into a text frame

UI

var master = new Window(“title”); // This line creates the master window in Script UI.

var uiTextbox = master.add(“edittext”, undefined, “”); // This line creates the actual edit text box the user enters information into.

javaScipt

Using text entered in the UI has three parts:

Selecting the text box in Illustrator that the text is to go into.

Selecting the text that was entered in the UI.

Telling javaScript that the contents of the UI box is to go into the illustrator box.

var master = new Window(“xxx”);

var UIText = master.add(“edittext”, undefined, “ “);

master.show();

var alpha = app.activeDocument.textFrames.getByName(“text frame in illustrator”); // This line uses a way to refer to the box in illustrator using the box’s name given to it in the layers panel in illustrator and assigns it to the variable alpha.

alpha.contents = UIText.text; // This line puts the text that was entered in the UI using the .text property of the edittext box into the illustrator box using the .contents property of the illustrator box alpha. This is the part that grabs the content from the edittext box, selects the illustrator box, and puts the content into it.

HTML

Forms

CSS

CSS is a language used in conjunction with HTML that controls how HTML elements will appear. HTML is used to define the web page’s structure, but CSS governs how that structure looks.

External CSS files

A separate CSS file can be linked to from within the <head> section of an HTML file.

<link href=”style.css” type =”text/css” rel=”stylesheet”>

Selectors

Selectors are symbols used by CSS to define which HTML elements are to be affected.

Class selectors

.class {

}

ID selectors

#id {

}

Groups of elements

Groups of elements can be defined in a single CSS style using commas to separate the selectors.

.class, #id, h1, h2 {

}

Chaining selectors

Selectors can be chained together so that a CSS style is applied to the element that matches all the criteria.

h3.class {

}

The code above will apply the CSS style to the h3 element that also is the .class.