Essential JavaScript

Overview

JavaScript is a wide and diverse scripting language and it can be difficult to know where to begin. This document is here to help! We will be looking at the essential concepts from the JavaScript language you will need to understand to continue your progress in development. This is only the tip of the JavaScript iceberg, but you will find these concepts reappearing in the C# language.

JavaScript Introduction

What is JavaScript: JavaScript is a lightweight interpreted programming language.

Okay, but what *IS* JavaScript: JavaScript is the language used to interact with a webpage (the DOM), the user, and send information to the server. JavaScript is what developers use to bring a static webpage to life. It can modify the HTML that makes up the webpage, add/remove content, gather information from the user, perform calculations, and even pass information to a remote server. This document covers concepts that are found in JavaScript and in other programming languages. Some of the words may change, but the concepts will stay the same.

Statements

In programming a statement is a single instruction, whether that instruction is to the browser, server, or database. There is no specific length for a statement; they can be simple:

```
let x;
```

Or long enough to cover multiple lines:

```
document.getElementById("paragraph2").innerHTML =
  document.getElementById("paragraph1").innerText;
```

Semicolons ;

You may have noticed above that each statement ends in a semicolon (;). The semicolon is how developers end every statement. Learning to look for the semicolon will help as you are exposed to more complicated code. JavaScript does not currently enforce the semicolon end of statements, but C# requires them. Go ahead and build the good habit of ending your statements with a semicolon.

White Space

White space is the term we use to describe spaces/line breaks added to make code easier to read. JavaScript ignores white space and it is good practice to add white space around operators (= + - * / && ||). Because white space is ignored the following three lines of code are treated the same:

```
let person = "New Coder";
let person="New Coder";
let person =
  "New Coder";
```

You can use this to make long lines of code easier to read by adding line breaks. If you find a line of code extending past the edge of your screen you should consider adding white space.

Code Blocks

Statements can be organized into groups using curly brackets { }. These groups of statements are known as code blocks and will be run together. The most common place you will find code blocks is in JavaScript functions. Code blocks define the body of a function:

```
function sayHello() {
  let printArea = document.getElementById("welcomeParagraph");
  printArea.innerText = "Hello World!;
  alert('JavaScript is saying hello!');
}
```

Keywords

Keywords in programming are words and phrases that have a specific meaning in that language. As a developer you have a lot of freedom in what names you use in your code, but you can only use keywords for their intended purpose. Some commonly used keywords are below:

Keyword	Use
let/var	declares a variable - both work, but let is preferred
function	declares a function
for	creates a code block that runs while a condition is true
break	ends a switch or loop
return	exits a function
if ... else if ... else	creates one or more code blocks that runs based on a set condition
switch	create code blocks that run based on different cases

Keywords can change as programming languages evolve over time. **let** and **var** are examples of this. Both keywords are used to declare a variable, but they differ in scope. **var** came first and is more widely scoped, **let** was introduced later and is the keyword that will be used in future versions. It is unlikely that support for **var** will be removed in the short term developers are encouraged to use **let** in all new code they write.

Syntax

The syntax of any programming language is the set of rules that describe how that language is constructed. In the section above you were introduced to some of the syntax for the language; code blocks, semicolons, and keywords. Now we will be more specific.

Value

JavaScript has two types of values, **literals** and **variables**:

- **Literals** - these are unchanging, fixed values
    - Number literals can be written with or without decimals: 1, 23, 2.05, 100000.01
    - String literals (text) are written with single or double quotes: "A string", "y", 'Another string', '10.00'
- **Variables** - variables store data values
    - Variables do not require a value - they can be declared
        - Variables are declared using **let** or **var**
        - **Let** is the preferred option, but older code will use **var**
    - Variables are **assigned** a value using the = sign

- Variables can have their value changed after assignment
- The value of a variable can be set using other variables

NOTE: Numeric values can be stored as numbers or as strings. If you need to do calculations with the value use a number (math functions, monetary transactions, data processing), otherwise use a string (street address, phone number)

Operators

JavaScript has two types of operators: **assignment** and **arithmetic**

Assignment: the only assignment operator is "=" which assigns a value to a variable

Arithmetic: operators used to perform calculations

- "+" - addition
- "-" - subtraction
- "*" - multiplication
- "/" - division

Expressions

Expressions are lines of code that compute some value. An expression asks the computer to perform some sort of work, this is different from a declaration that creates a variable and an assignment which stores a value in a variable. You can combine these three to declare a variable and assign the result of an expression:

```
5 + 5; //Expression with numbers
"Bobby" + " " + "Davis"; //Expression with strings
let x; //Declaration
x = 10; //Assignment
let x = 5 + 5; //Declaration, assignment, expression
```

Keywords

As shown above keywords are specific words that instruct the program to do certain things. The **let** and **var** keywords tell the program to perform variable declaration.

Comments

Comments are lines of code that are not executed. JavaScript uses two ways to denote a comment:

- **//** - comments out one line of code after the //

- **/* */** - comments out multiple lines of code between /* and */

```
let x = "I run" //This comment does not run;
//let y = "I do not run"; Nothing on this line will run
/*let z = 10;
let a = 15;
let b = z * a; these three lines of code will not run */
```

Comments have two primary purposes; to provide information to developers who read your code in the future or to temporarily remove code, typically for troubleshooting or because of specific conditions (development vs production).

Well commented code will not explain every line, it will explain the intent behind the code and note an specifics of implementation. Leaving comments in your own code is a good way to remind yourself what you were thinking when you wrote the code. Commenting is a soft skill that you will develop as your work as a developer. When working on a team there may be rules about what needs comments and how to make them.

Identifiers

Identifiers are the names of things in JavaScript. Variables, keywords, functions, and labels are all identifiers. In JavaScript there are restrictions on identifiers: the first character must be a letter, underscore (_), or dollar sign ($). Numbers can be included, but not as the first character.

Case Sensitivity

JavaScript is a case sensitive language. The keyword **let** is different than Let or LET, only the lowercase version is used to declare a variable.

```
let firstName = "Bobby";
let firstname = "Sally";
//This is valid code because the variables firstName and firstname
//are considered different variables
```

Combining words: In the example above note the first variable "firstName"; this variable is written in **Camel case**. Camel case is when you want to combine multiple words into an identifier. The first letter of the first word is lower case, the first letter of every word after is capital: firstName, lastName, myVerySpecialFunction. This is not enforced by the language, but is the accepted style of the language.

C# Note: C# uses both **Camel case** notation and **Pascal case**. Pascal case is similar to Camel case, the only difference is that all words start with a capital letter: FirstName, LastName, MyVerySpecialMethod. Pascal case is used for property, class, and method names, Camel case is used for variables.

Character Set

JavaScript uses the Unicode character set. This is mostly for your information and not something that will come up often.

Variables

Variables are how developers store information. In JavaScript a variable can store any type of data; strings, numbers, arrays, or JSON objects.

Declaration

JavaScript has three ways to declare variables:

- **let** - declares a variable that can be assigned and reassigned a value
- **const** - (constant) declares a variable that can be assigned a value, but cannot reassign a value
- **var** - similar to **let**, declares a variable that can be assigned and reassigned. **var** is older code, you should use **let** in your code, but older code will use **var**

Declaring a variable allows you to use it in the following code, but does not assign a value to the variable. As mentioned above there are some rules you must follow when declaring a variable (no numbers to start) and some conventions that you should follow (Camel case). Variable should be descriptive, this will help you remember what values you have stored when writing more complex code. Descriptive variable names also make it easier for another developer to maintain your code.

NOTE: A variable that is declared, but not assigned a value is **undefined**. **Undefined** variables must have a value assigned before they can be used in your code.

Assignment

In JavaScript (and C#) assignment is done using the "=" sign. The variable is placed to the left of the "=" sign and the value or expression is placed to the right. Assignment can be done when the variable is declared or any time afterwards, but you must assign a value to a variable before you can use it. Variables can even be assigned the result of a function!

```
let undefinedVar; //Declared variable, undefined value
undefinedVar = "I now have a value"; //Assigned variable, data type string
undefinedVar = "My identifier is poorly named now"; //Reassigned variable
let newVar = 10; //Declared and assigned value, data type number
const noChanging = "I cannot be reassigned";
      //Constant declaration and assignment
```

```
let complexVar = myFunction();
      //Declared and assigned to the result of myFunction
```

Note: If you want to check if two variables hold the same value you use "==" to test. This is operator is the "equal to" operator. This is often used when your code needs to respond based on multiple conditions.

Data Types

Variables can hold many data types and each type will have its own behaviors. Numbers behave much like you would expect, when you perform math operations the result does not change just because we are writing code. Strings behave differently, they can only use the "+" operator and that will cause concatenation, the strings will be combined into one new, longer string.

```
let x = 1;
let y = 2;
let z = x + y; //z has a value of 3
let a = "a";
let b = "b";
let c = a + b; //c has a value of "ab"
let d = c + " " + a + b; //d has a value of "c ab"
```

Operators

We have seen some basic arithmetic operator and the basic assignment operator, but there are many more operator available when writing JavaScript.

Arithmetic Operators

- **+** : addition (numbers) / concatenation (strings)
- **-** : subtraction
- ***** : multiplication
- **/** : division
- **%** : division remainder (modulus operator)
- ****** : exponentiation (raising one number to the power of another)
- **++** : increment (increasing the value by 1)
- **--** : decrement (reduce the value by 1)

```
let x = 10 % 3; //x will have the value of 1
```

Assignment Operators

The "=" sign can be combined with arithmetic operators. When used this way the value of a variable is as the first part of the expression.

- **= :** x = y
- += : **x += y** is **x = x + y** (concatenation with strings)
- -= : **x -= y** is **x = x - y**
- *= : **x*= y** is **x = x * y**
- /= : **x /= y** is **x = x / y**
- %= : **x %= y** is **x = x % y**
- **= : x **= y** is **x = x ** y**

Comparison Operators

Comparison operators allow you to compare two variables and the result will be **true** or **false.** Each operator is listed with the condition that will result in **true** and are evaluated left to right**:**

- == : Is equal to
- === : Is equal value and type
- != : Is not equal to
- !== : Is not equal value or not equal type
- > : Is greater than
- < : Is less than
- >= : Is greater than or equal to
- <= : Is less than or equal to

"==" and "===" are very similar, but serve different purposes:

```
let x = 5; //x is the number 5
let y = "5"; //y is 5 as a character
let isTrue = x == y; //JavaScript translates
                     //the string to a number for comparison
let isFalse = x === y; //x is a number and y is a string so this is false
let alsoTrue = x !== y;
```

Logical Operators

Logical operators allow flexibility to developers. They allow for more complex evaluations when determining what code to run.

- **&&** : AND operator - both conditions must be true for the expression to be true
- || : OR operator - if either condition is true the expression is true
- **!** : NOT operator - inverts the statement

```
let x = 5;
let y = 5;
let z = 10;
let isTrue = (x == y) && (x != z) //x is equal to y AND x is not
                                  //equal to z - result true
let alsoTrue = (x == y) || (x == z) //x is equal to y OR x is
                                    //equal to z - result true
let isFalse = (x == y) && (x == z) //because x does not equal z result false
let invertTrue = !((x == y) && (x == z)) //the inner evaluation is false,
                                         //then inverted
```

Functions

Functions are blocks of code that run together. Functions run when they are **called/invoked** (both words are used to describe the same thing). Functions can be called automatically when the page loads, when the user interacts with the page, or when some other condition is met. As the developer you decide when a function is called and from where.

Syntax

When creating/declaring a function there are four components: **function** keyword, the **name** of the function, the **parameters/arguments** denoted with ( ), and the **body** of the function denoted with { }

- **function** keyword - similar to let, declares that the code that follows will define a function
- **name** - the identifier for the function, used to call the function in the rest of your code - Camel case
- **parameters/arguments** - data that is given to the function to manipulate, can be empty
    - **parameter** - in the declaration
    - **argument** - the value passed in as a parameter
        - often used interchangeably

- **body** - one or more lines of code that will run when the function is called, cannot be empty
    - the body of the function is where the parameters are used
    - functions may return a value, but this is not required

```
function myFunction(parameter1, argument2)
{
  let firstVar = parameter1;
  let secondVar = argument2;
  let thirdVar = firstVar + secondVar;
  document.getElementById("anId").innerText = thirdVar;
};
```

This function takes in two arguments, adds the values together, then updates a section of the webpage with that result.

Invocation

Invoking/calling a function is how we tell a function to run its code. Functions are invoke by using the **name** of the function followed by parenthesis ( ):

```
myFunction("Hello ", "World!");
```

There are three main ways a function will be invoked:

- When an event occurs (clicking a button, choosing from a menu, even pressing a key)
- When it is called by another function
- Automatically (self invoked)

In all three cases the syntax is the same: **name()**

Note: Variables can have their value set by a function, the function is invoked after the assignment operator (=)

```
let x = aMathFunction();
```

Return

Functions can be broken down into two large categories; those that return a value and those that don't. Functions that do not return a value run until the last line of code and then stop. Functions that return a

value run until they reach the **return** keyword. The return keyword is paired with a value to return and this value is returned (given) to whatever invoked (called) the function.

```
function doMath(number1, number2)
{
  let x = number1;
  let y = number2;
  return x + y;
}

let newVar = doMath(2,3); //newVar will have a value of 5
```

Functions as Variables

Once you have assigned the result of a function to a variable you can use that variable in the rest of your code.

```
function doMath(number1, number2)
{
  let x = number1;
  let y = number2;
  return x + y;
}

let newVar = doMath(2,3); //newVar will have a value of 5
let myString = "The doMath function returns " + newVar +
                                          " when given 2 and 3";
```

Local Variables/Scope

Variables declared within a function are only available to those function. This is called function or local scoping. The scope of a variable refers to what other code can access that variable.

It is best practice to scope your variables as narrowly as possible. We will look more at scoping later in the document.

Strings

String is a data type that represents zero or more characters between quotes (single or double). If you need to include quotes inside a string they must be different than the quotes you used to declare the string.

```
let string1 = "String with double quotes";
let string2 = 'String with single quotes';
let string3 = "Nesting 'quotes' inside a string";
let string4 = 'Alternate "nesting" option';
```

String Length

The length property of a string represents the total number of characters in the string, this includes any white space. A string with zero characters is called an "empty string". To access the length property of a string store the string in a variable, then add .length to that variable

```
let alphabet = 'abcdefghijklmnopqrstuvwxyz';
let numberOfLetters = alphabet.length; //numberOfLetters = 26
```

Escape Character

Occasionally you will encounter a situation where you need a string to store a single quote, double quote, or backslash, but you have already alternated your quote. We solve this by using the escape character \. When placed in a string the backslash instructs the code to treat the next character as part of the string and not as part of the code

```
let thisQuote =
  'A wise man once said, "I can/'t use contractions in my code!",' +
     'but he was wrong';
```

Arrays

In JavaScript arrays are a data type used to store multiple values in a single variable. Arrays contain zero or more elements and can store different data types in the same variable.

Like strings arrays have a length property, it counts the total number of values stored in the array. You can call the length property in the same manner as with a string.

Creating an Array

We create a new array in JavaScript using square brackets [] after the assignment operator. The values that the array will hold are put between the square brackets and separated by commas.

```
let newArray =
    ["First Value", "Second Value", 42, "Multiple Data Types Are Fun!"];
```

Accessing the Array

Once we have created an array we will want to access the values in that array. We do this using the **index number** of the value (you may also see this simply as **index**). Array indexes begin at **0** and end at one less than the length of the array. To access the value at a specific index you use the variable representing the array, square brackets, and the number of the index you want to access:

```
let newArray =
    ["First Value", "Second Value", 42, "Multiple Data Types Are Fun!"];
let indexZero = newArray[0];
let firstValue = newArray[0];
let numberValue = newArray[2];
                  //remember our index starts at 0 so the third value is
                  //index number 2
let arrayLength = newArray.length;
let lastValue = newArray[arrayLength - 1];
                  //Length in total count so we subtract 1
```

If you need to access the entire array you call the variable storing the array with no square brackets

```
let newArray =
    ["First Value", "Second Value", 42, "Multiple Data Types Are Fun!"];
function printArray()
{
  let printArea = document.getElementById('printHere');
  printArea.innerHTML = newArray;
}
//This function will print the entire array inside
//the element with the Id of printHere
```

Changing the Array

Each value in the array can be changed with the assignment operator just like a variable:

```
let newArray =
    ["First Value", "Second Value", 42, "Multiple Data Types Are Fun!"];
let originalValue = newArray[0]; //originalValue = "First Value"
newArray[0] = "Index Zero";
let updatedValue = newArray[0]; //updatedValue = "Index Zero"
```

First and Last Element

Properly accessing the elements in an array is a concept that is going to continue through JavaScript and into C#. Proper indexing is so essential that there is an error/exception specifically devoted to it: Index Out of Range. So remember arrays start at 0 and end at length - 1.

```
let newArray =
    ["First Value", "Second Value", 42, "Multiple Data Types Are Fun!"];
let indexZero = newArray[0];
let firstValue = newArray[0];
let arrayLength = newArray.length;
let lastValue = newArray[arrayLength - 1];
              //Length in total count so we subtract 1
let alternateLast = newArray[newArray.length - 1]
              //Directly calling the length property
```

Looping and Arrays

Using the index of a value will allow you to access or modify that single element, but what if we want to affect the entire array? In JavaScript we have two good ways the **.forEach()** function and the **for loop**. The **.forEach()** function will be discussed in the next section and the **for loop** has it's own dedicated section later. Whether you use the function or the loop is determined by what your code needs to accomplish.

Adding to/Removing from the Array

You've seen how to create an array with elements inside and how to change the value of an element in the array, but what about adding a new element to the array or removing an element from the array? There are four methods specific to arrays that you can use, .**push()/.pop()** and .**shift()/.unshift()**:

- .push() - adds the value inside the ( ) to the end of the array
- .pop() - removes the last element from the array
- .unshift() - adds the value inside the ( ) to the beginning of the array
- .shift() - removes the first value from the array

.push()/.pop() are the most commonly used functions, but be aware that .unshift()/.shift() exist

Array Iteration

Iteration is the repetition of a process. In JavaScript Arrays have several functions that evaluate every element in the array. When we look at these functions remember that one function can be passed to another

function as an argument. The function passed as an argument is known as a callback function.

forEach()

The .forEach() function is used when you want to use every value in an array as the argument of a second function (callback function). Use this method when you need to use every value in an array once with another function.

```
let newCustomers = "";
function todaysCustomers(name)
{
  newCustomers = newCustomers + name + "<br />";
}

let enteredNames = ["Johnny", "Larry", "Abigal", "Evander"];
enteredNames.forEach(todaysCustomers);
```

filter()

The .filter() function passed every value in the array through another function that tests that value. Every value that passes the test is placed into a new array.

```
let familyAges = [56, 54, 32, 26, 18, 10, 8];
function canVote(age)
{
  return age >= 18;
};
function canDrink(age)
{
  return age >= 21;
};

let voters = familyAges.filter(canVote); //voters = [56, 54, 32, 26, 18];
let drinkers = familyAges.filter(canDrink); //drinkers = [56, 54, 32, 26];
```

every()

The .every() function passes each value in the array as an argument to another function where it is tested. If **all** of the values pass the test the .every() returns **true**, if **even one fails** the test the return is **false**.

```
let application = ["Jane", "Doe", "123 Street Dr.", "Haberdasher"]
function hasLetters(word)
```

```
{
  return word.length > 1; //Checks for 2 or more letters
};

let goodApplication = application.every(hasLetters); //goodApplication = true
```

some()

The .some() function is the inverse of the .every() function, .some() will return **true** if **any** value in the array passes the test only returning **false** if **every** values fails the test

```
let partyReservation = [20, 17, 17, 16, 15];
function mustHaveAdult(age)
{
  return age >= 18;
};

let canReserve = partyReservation.some(mustHaveAdult); //canReserve = true
```

indexOf() / lastIndexOf()

Both the .indexOf() and .lastIndexOf() functions are given a value to search the array for. .indexOf() returns the position of the first element that matches the value given. .lastIndexOf() last occurrence of the value.

If the value is not found in the array both functions will return -1. By storing this value in a variable you can easily test for the presence of a value in the array.

```
let someNumbers = [1, 2, 3, 345, 32, 7, 453, 3, 8];
let indexOfTwo = someNumbers.indexOf(2);
                //indexOfTwo = 1 - index starts at 0
let firstIndexOfThree = someNumbers.indexOf(3); //firstIndexOfThree = 2
let lastIndexOfThree = someNumbers.lastIndexOf(3); //lastIndexOfThree = 7
let indexOfNine = someNumbers.indexOf(9);
                //indexOfNine = -1 - value not found
let lastIndexOfFour = someNumbers.lastIndexOf(4); //lastIndexOfFour = -1
```

find() / findIndex()

These two methods pass each element in the array to a second function that performs a test returning the first value that passes the test (.find()) or the position in the array (.findIndex())

If the value is not found these functions have different returns:

- **.find()** - returns **undefined**
- **.findIndex()** - returns **-1**

Because .findIndex() returns -1 if no element in the array passes the test it can easily be combined with an if/else statement. While this does require more code to be written the end result is safer code with less need for debugging.

```
let groupNames = ["Larry", "Mary", "Jerry", "Sari"];
function shortName(name)
{
  return name.length < 5;
}
function namedJohn(name)
{
  return name == "John";
}

let short1 = groupNames.find(shortName); //short1 = "Mary"
let short2 = groupNames.findIndex(shortName); //short2 = 1 - index starts at 0
let short3 = groupNames.find(namedJohn); //short3 = undefined
let short4 = groupNames.findIndex(namedJohn); //short4 = -1
```

Booleans

The Boolean data type always has one of two values. What that value is depends on the language used, but in JavaScript and C# they store **true** and **false**. All decision making in programming involves Boolean values at some level.

The keywords **true** and **false** are written in lowercase with no quotes. You can test against **true** and **false** when making decisions in your code. You can make a variable a Boolean by assigning a value of **true** or **false**.

```
let trueBool = true;
let falseBool = false;
```

Comparisons and If/Else

The following two sections are based on Boolean logic and values. Comparisons perform a calculation and return **true**/**false** while If/Else statement determine which code runs based on a condition; **if** the condition is **true** do 1, **else** do 2.

Comparisons

Comparisons involve testing two things to see if they are the same. You can compare simple data types, the results of functions, or complex JavaScript objects. As a developer you have several tools you can use to make these comparison.

Logical Operators allow the developer to combine the results of multiple comparisons when making decisions in code.

Ternary Operators assign one of two values based on the result of a comparison.

Comparison Operators

There are three sets of comparison operators, equal to, not equal to, and greater/less than. The version used is determined by which outcome should produce true. Equal/not equal can test for value or value and type. The value only comparison is the most commonly used.

- == - is equal value
    - string "5" and number 5 are equal value
- === - is equal value and type
    - string "5" and number 5 are equal value but not equal type
- != - is not equal value
    - string "Bob" and string "David" are not equal value
- !== - is not equal value or not equal type
    - string "Bob" and string "David" are not equal value but are equal type
    - string "5" and number 5 are equal value but not equal type
    - string "Bob" and number 4 are not equal value and not equal type
- > / >= - greater than / greater than or equal to - is the value on the left larger than the one on the right
    - for numerical comparison
- < / <= - less than / less than or equal to - is the value on the left smaller than the one on the right
    - for numerical comparison

Logical Operators

The three main logical operators you will see are **&&** (AND / logical and), || (OR / logical or), and **!** (NOT / logical not). **&&** and || are used to combine multiple **true**/**false** results before returning a final **true** or **false**. **!**

is used to reverse the value of a comparison. You can use parenthesis ( ) to surround logical operators, comparisons inside the ( ) will be run together before a final result is returned.

- && - all comparisons connected must return true
- || - if any comparison returns true
- ! - change true to false or false to true

```
let x = 5;
let y = "5";
let z = 7;

let true1 = x == y && x != z;
            //5 is equal to "5" (true) AND 5 is not equal to 7 (true)
let true2 = x == y || x == z;
            //5 is equal to "5" (true) OR 5 is equal to 7 (false)
let true3 = !(x == z);
            //5 is equal to 7 (false), value is changed (true)

let false1 = x == y && x == z;
           //5 is equal to "5" (true) AND 5 is equal to 7 (false)
let false2 = x != y || x == z;
           //5 is not equal to "5" (false) OR 5 is equal to 7 (false)
let false3 = !(x == y)
           //5 is equal to "5" (true), value is changed (false)
```

Ternary Operator

Ternary operators allow you to assign the value of a variable based on the result of a comparison in one line of code. Ternary operators can quickly become confusing, try to limit their use to simple value assignments until you have more experience as a programmer.

The ternary operator consists of three portions: the comparison followed by a ?, the true case followed by a :, and the false case.

```
let sentence1 = (5 < 7) ? "I'm true!" : "I'm false!";
              //5 is less than 7 = "I'm true!"
let sentence2 = (5 > 7) ? "I'm true!" : "I'm false!";
              //5 is not greater than 7 = "I'm false!"
let sentence3 = (sentence1 == sentence2) ? "Same words" : "Different words";
              //sentence1 and sentence2 are not the same = "Different words"
```

If/Else If/Else

Conditional Statements

Conditional statements allow a developer to specify when certain pieces of code run. This decision can be based on user input or the result of other code. **if**/**else** statements evaluate a condition, then execute a specified block of code if the result is **true**/**false**. These statements can by changed with the **else if** keyword. **switch** statements are a more concise manner of writing **if**/**else** statements when testing one value against know outcomes. Anything that can be written with a **switch** statement can also be done with **if**/**else if**/**else** statements.

If

An **if** statement is composed of three parts: the **if** keyword, a set of parenthesis ( ) around one or more evaluations, and a set of curly brackets around one or more lines of code to execute. Logical operators can be used to combine or invert evaluations.

```
let output = "";
let x = 5;
let y = "5";
let z = 7;

if (x == y) //this is true so the code between { } will run
{
  output += "x is equal to y, ";
}
if (x == z) //this is false so the code between { } will not run
{
  output += "x is equal to z, ";
}
if (x == y && !(x == z)) //this is true so the code between { } will run
{
  output += "x is equal to y, but not equal to z."
}

//output = "x is equal to y, x is equal to y, but not equal to z."
```

Else

The **else** keyword must be paired with an **if** keyword and uses a set of curly brackets { } to denote code that will run when the **if** statement is false.

```
let output = "";
let x = 5;
let y = "7";

if (x == y) //this is false so the code between { } will not run
{
  output = "x is equal to y";
}
else //the if statement was false so the code between { } will run
{
  output = "x is not equal to y";
}
```

Else If

The **else if** keyword is used to introduce additional evaluations when the first **if** statement is false. The **else if** keyword must follow an **if** or an **else if** statement and will only run the evaluation if all **if** statements before it were false. You may also include a final **else** statement.

Because only one **if** statement will execute remember to put your most specific conditions first.

```
let output = "";
let x = 5;
let y = "5";
let z = 7;

if (x != y) //this is false, code moves to next statement
{
  output = "x does not equal y";
}
else if (x != z) //this is true, code between { } will run and code ends
{
  output = "x does not equal z";
}
else if (x == y && x != z) //this is true, but the code doesn't reach here
{
  output = "x is equal to y, but not to z";
}
else //an if statement was true so this will not run
{
  output = "All variables are equal";
};
```

For Loops

Looping

The basic concept of a loop is "Code that will run ***x*** number of times". ***x*** can be any positive integer or 0. There are multiple types of loops, but you can succeed in this course by understand the concept of a for loop. Once you grasp the concept of the for loop the other looping structures will be easy to pick up.

For Loop

A **for** loop is a section of code that will run a certain number of times based on a condition. It is made of five parts: the **for** keyword followed by a set of ( ) with three statements, statement 1 which runs first, statement 2 the condition for the loop, statement 3 which runs after each loop, and a section of code inside curly brackets { } that is executed during each loop.

The three statements are separated by semicolons ;

```
for (statement 1; statement 2; statement 3)
{
  //some code to run multiple times
};
```

First Statement

The first statement in a **for** loop declares and assigns the variable to be tested during the second statement. This also sets the start of the loop. You will commonly see the letter "i" used for this variable, but that is not required. This variable can be used by code inside the curly brackets { }, but not by any other code.

```
for (i = 0; //start at 0
```

Second Statement

The second statement in a **for** loop sets the condition for the loop to run. If the condition is true the loop will execute one time, then check if the condition is still true. If the condition set is always false the loop will not run. If the condition can never be false the loop will run until the browser runs out of memory and crashes, this is bad.

```
for (i = 0; i < 5; //start at zero and run until i is not less than 5
```

```
for (i = 7; i < 5; //this will not run 7 > 5 from the start
```

Third Statement

The third statement in a **for** loop runs after each completion of the loop and modifies the variable set in the first statement. Commonly you will increment (i++) or decrement (i–) the variable, but other values can be used.

```
for (i = 0; i < 5; i++)
            //start at zero, run until i >= 5, increment i after each loop
            //runs 5 loops: 0, 1, 2, 3, 4
for (i = 7; i >= 5; i--)
            //start at seven, run until i < 5, decrement i after each loop
            //runs 3 loops: 7, 6, 5
for (i = 5; i < 4; i++)
            //start at 5, run until i < 4, increment i after each loop
            //this loop cannot end and will run until the browser crashes

let counter = 0;
for (i = 0; i <= 10; i++) //start at zero, run until i > 10, increment i after
{
  counter += i; //add the number of the loop to our variable
}
//counter = 55 after the loop finishes
```

Objects

What is an Object?

In JavaScript, and in all object oriented programming, **objects** are a collection of information related to a single concept. You can think of **objects** as a collection of variables. Objects can describe physical objects (a car, a can of soda, a person) or concepts (the data to create a chart, a song, the population of all cities in a US state). Each piece of information an object stores is called a property and as the developer you create the names of the properties in the same way you were able to create variable names.

Non-code Example

Before we look at how to create objects let's see an example without using code. If you were to create an object to describe a person's full name it would have a FirstName property, a MiddleName property, and a LastName property.

If we use this object to describe Jonathon Quincy Public the FirstName property has the value of Jonathon, the MiddleName property has the value of Quincy, and the LastName property has the value of Public. We can use this same object to describe Janice Eugenia Doe: FirstName has the value of Janice, MiddleName has the value of Eugenia, and LastName has the value of Doe.

More Complex Objects

In the example above we had an object with three properties each of which stored a string, but that is not the extent of what objects can handle. The properties of an object can store any kind of data, including other objects. Above we looked at using an object to store a person's full name in three properties. Using the concept of objects we can describe a person by combining objects. The object the describes Janice Doe could have a Name property that stores the FullName object from above, a PhysicalDescription property that stores an object containing her height, weight, hair color, eye color, etc., and a WorkInformation property that stores an object containing her place of employment, salary, work phone number, etc.

Objects can store both complex and simple information at the same time. The Person object above can store the person's age as a property along with the three objects discussed above.

Why Objects

Objects allow us to pass large amounts of data around our code using a single variable. Objects are also repeatable and reusable. If I create a Person object, I can use that object to describe any person and all of my Person objects will have the same properties. The code I write based on this object will work for all versions because each has the same properties.

Creating an Object

Much like arrays objects can be created empty or containing information. Objects are declared using curly brackets { }:

```
let myObject = {}; //myObject is an object, but has no properties
```

When adding properties to an object there are two formats based on the syntax you are using. The rule of thumb is that is you are writing code between the curly brackets { } you will use a colon for assignment (:) outside of the curly brackets you will use the equal sign (=) as with variables.

The properties of an object also have two syntaxes for access, [] or .

```
let myObject = {}; //empty object declared
myObject.FirstName = "Jonathon"; //using "dot notation" to create
                                 //and assign the value of the property
myObject["LastName"] = "Public"; //using array notation to create
                                 //and assign the value of a property
```

```
let firstName = myObject.FirstName; //using "dot notation" to read the value
let lastName = myObject["LastName"]; //using array notation

let anotherObject = {     //declaring the object and setting properties
  FirstName: "Janice",
  LastName: "Doe"
};

let firstName2 = anotherObject.FirstName;
let lastName2 = anotherObject["LastName"];
```

Accessing Complex Objects

To access the information in a complex object we can treat the object like a folder structure. When working with VS Code you created a folder on your Desktop to hold each project. To access your FizzBuzz code first you need to open your Code folder. Objects work on the same logic.

```
let nameObj = {
  FirstName: "Jonathon",
  MiddleName: "Quincy",
  LastName: "Public"
};

let physicalObject = {
  Height: 1.8,
  Weight: 120,
  HairColor: "Brown",
  EyeColor: "Green"
};

let personObj = {
  Name: nameObj,
  Physical: physicalObject
};

let firstName = personObj.Name.FirstName; //access the Name property of the
                                          //personObject, then access the Firs
                                          //property of the nameObj that we st

let height = personObj.Physical.Height;
```

Scope

Function Scope

The **scope** of a variable describes what code can access its value. JavaScript uses the concept of **function scope** to separate **local** and **global** scope for variables. **Function scope** means that every function has its own **local** scope that only it has access to. Outside of the **function** declarations is the **global** scope.

It is better design to use **local** scope over **global** scope. Because functions can invoke other functions you can create **local** scopes inside **local** scopes. **Local scoping** only requires memory allocation while the function is executing.

Local Scope

Every **function** declaration creates a **local** scope for that function. Variables declared inside the body of a function are available to all other code inside that function. Those variables are not available to any code outside of the function.

With this limitation in mind, if you find that you need the same information for two (or more) functions write a third function that declares the variables and invokes the functions.

```
//this code does not have access to the localScope variable

function scopingFunction()
{
  let localScope = "This";
  //code between the { } has access to the localScope variable
  localScope += " is a locally scoped variable";
};

function secondScoping()
{
  //this code does not have access to the localScope variable
};
```

```
function doMath()
{
  let x = 5;
  let y = 7;
  //x and y are scoped to the doMath() function
```

```
  //the doMath() function can pass those values to other functions
  let addNum = addTwo(x, y);
  let subtractNum = subtractTwo(x, y);
  let multipleNum = multiplyTwo(x, y);
};

function addTwo(num1, num2)
{
  return num1 + num2;
};

function subtractTwo(num1, num2)
{
  return num1 - num2;
}

function multiplyTwo(num1, num2)
{
  return num1 * num2;
}
```

Global Scope

Variables declared outside of a **function** are known as global variables. **Global** variables are created when the webpage first runs and last until the next load or reload. You should not write **global** variables, but be aware of how they function in case you see them in other code.

```
var globalString1 = "I am a globally scoped string";
var globalString2 = "I am another global string";
var globalNumber1 = 31;
var globalNumber2 = 23;
//The variables above are globally scoped
//They can be called by any code in the script

let finalString = globalStringConcat(globalString1, globalString2);
let finalNumber = globalNumberAdd(globalNumber1, globalNumber2);

function globalStringConcat(word1, word2)
{
  return word1 + " " + word2;
};
```

```
function globalNumberAdd(num1, num2)
{
  return num1 + num2;
};
```

Debugging

Debugging is the process of finding and removing errors from your code. Errors can come from mistakes in your syntax or in the logic of your code. Errors in code manifest in three broad ways:

- Errors that produce Error Messages - the easiest to find, the browser will indicate which line of code caused the error
- Errors that cause the code not to run - nothing happens when something should have happened. Set breakpoints and use the watch window to monitor your code it might be a typo or a bad index value
- Errors that cause unexpected results - your code ran, but didn't output what you expected. Use breakpoints and the watch window to find where in the code things went wrong.

Breakpoints

Breakpoints are stop signs in your code. A breakpoint instructs the code to stop execution right before the breakpoint. Breakpoints can be set in VS Code by using the Debugger Extension or in the debugging window of the browser. The Debugger extension for VS Code provides a much larger work area for debugging your code.

Once the code execution has reached the breakpoint it will stop and you will have the opportunity to check the values of any declared variables. To move the code forward you have several options, but **Step Over**, **Step Into**, and **Continue** are the most useful early on.

- **Step Over** - the code will execute the next line of code and then pause
    - allows you to see your code run one line at a time
- **Step Into** - used when one function invokes another function
    - pauses execution of the invoked function
    - allows you to move line by line through the invoked function
    - when the invoked function ends you will return to the main function
- **Continue** - code resumes execution and will continue until it:
    - encounters another breakpoint
    - causes an error

- finishes execution

Watch Window

The watch window will vary in location and functionality based on whether you are using VS Code or browser debugging, but some things will be constant. The primary function of the watch window is to track the current value of some or all variables as the code executes.

Additionally by placing your mouse cursor over a variable a tool tip will appear with the value of that variable.

Conclusion

This document only begins to scratch the surface of the JavaScript language. There are multiple ways to solve most problems in coding, but the concepts that are presented above will help you write functional code. Sophisticated code comes from functional, foundational code.

Download this file as a PDF