



Pragmatic JavaScript

A glance into core JavaScript fundamentals with a practical approach
demonstrated using Node.js

Table of Contents

- i. [Before Starting](#)
- ii. [Getting Started](#)
- 1. [Variables & Constants](#)
- 2. [Data Types](#)
- 3. [Operators](#)
 - 3.1. [Assignment](#)
 - 3.2. [Arithmetic & Expressions](#)
 - 3.3. [Comparison](#)
 - 3.4. [Logical](#)
 - 3.5. [Short Circuiting](#)
- 4. [Structural Programming](#)
 - 4.1. [Getting Input From The User](#)
 - 4.2. [If, Else If, Else Statements](#)
 - 4.3. [Strict vs Loose Equality](#)
 - 4.4. [Switch Statements](#)
 - 4.5. [For Loops](#)
 - 4.6. [While & Do While Loops](#)
 - 4.7. [Breaking & Continuing](#)
 - 4.8. [Try, Catch, Finally](#)
 - 4.9. [Scope](#)

i. Before Starting

Much of this text will be demonstrated using the console and not in a web browser (unless otherwise noted). The purpose of this text is to provide you with an understanding of practical JavaScript and how to use it programmatically more so than for styling purposes. Therefore, you can expect a pragmatic approach to learning JavaScript with functionality as the primary target over UI and UX. This text is written under the assumption that you may have a little bit of prior knowledge of working with JavaScript, however it is not entirely necessary.

With that being said, becoming familiar with either the [Developer Tools](#) in your web browser, or with [Node.js](#) will be of great importance. We will be using the latter to run and debug our code as it will be far easier and faster to demonstrate with. If you intend to follow along with the web browser please note that any reference to **require(...)** should be replaced with **include(...)** and **readline(...)** to **prompt(...)**. You will be required to make any other changes necessary to have your scripts run. I am not providing any guarantees that the examples provided throughout will work without error or without requiring some form of modification to run in a web browser.

All the examples provided herein will be done on a Windows based machine using the Command Prompt. If you are using a Mac or Linux based operating system, please use the command line interface (CLI) that is applicable to you (ex. Terminal).

Required software for this text:

- [Node.js](#)
- [Visual Studio Code](#) (Or any other preferred IDE)

After acquiring and installing the required software mentioned above, verify that you have successfully installed Node and that it is accessible by running the “**node -v**” command in your CLI.

```
C:\Users\Jesse>node -v  
v8.11.1
```

Displayed Node version after successful installation

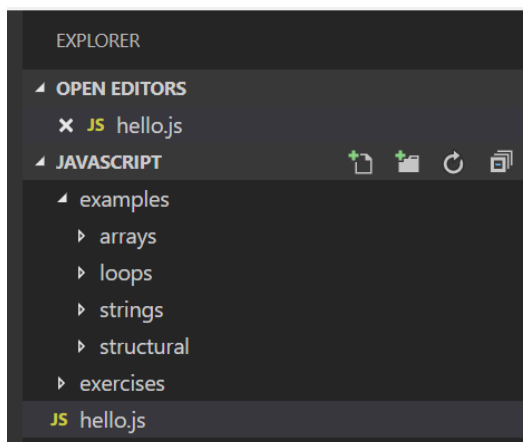
If you see the version of Node you installed, then you have successfully completed this step. Do not worry if your version is not the same as mine as it will have no affect on your ability to follow the samples.

ii. Getting Started

Before we begin jumping into code, we need to set up a directory that can be used to store all of your JavaScript files. Be sure to select a location that is easy to remember and navigate to, in my case I will be creating a directory in the root of my hard drive (C:\) aptly named “javascript”. It will be on you to determine how you would like to name, structure and organize your directory.

Once you have your directory created, open Visual Studio Code (VSCode) and:

1. Click on **File -> Open Folder**
2. Navigate to the location where you created your directory
3. Select the directory and click on **Select Folder**

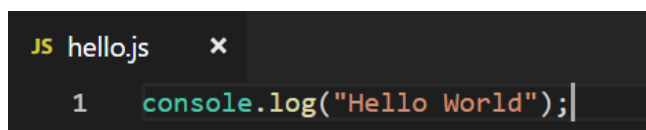


Example directory structure in VSCode

As a quick example and test to ensure everything is running correctly, let's create a new file named “hello.js” in the root directory with the contents:

```
console.log("Hello World");
```

Your files contents should look the same as the figure below:

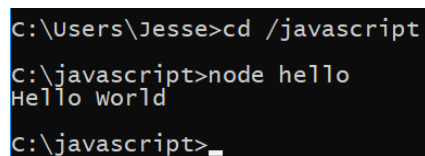


```
JS hello.js x
1 console.log("Hello World");
```

Sample JS file for our Hello World example

Save the file and navigate to the directory that contains it in your CLI. You will then want to run the file using Node. If all is successful we should see “Hello World” displayed on the screen.

```
cd /javascript
node hello
```



```
C:\Users\Jesse>cd /javascript
C:\javascript>node hello
Hello world
C:\javascript>_
```

Output of our hello.js file

If you were able to successfully run the file, then I would like to congratulate you on setting up your environment. You can now continue to the first chapter covering variables and constants.

1. Variables & Constants

Let's say that you're planning a trip to the movies with some of your friends. You call all of them to see who can attend. Initially, six of them answer and three of them say yes, another four are left a voicemail. Later, two of those four call back to confirm that they can also attend, while one of the previous friends that had initially said yes calls back to inform you they can no longer join you. The number of friends that can attend is an example of a variable. It is actively changing upon the retrieval of new information, in this case, a number that is either incrementing, or decrementing.

After the planning, you all meet at the movie theater and proceed to purchase your ticket; each of you pay \$14.99 and an additional 13% tax. The tax in this case is an example of a constant, a value that does not change or is not expected to change.

Going along with the previous scenario, we can determine that a variable is anything that is expected to change throughout the execution of a program at any given moment that it is assigned new information, whereas a constant is something that we can expect to remain consistent and never change.

By definition, a variable in programming is a storage location in memory associated with a symbolic name, or rather an identifier that holds some form of information referred to as a value. Using this identifier, we can access the value in other parts of our application, whether it be to read or update its contents. Constants are similar to variables in that they too, point to a storage location in memory and use an identifier, however the latter is false in that once a constant is defined it can not be altered throughout the rest of the script's execution and

remains read-only. Ideally, we would use a constant any time we have a bit of information that is likely to be repeated multiple times. This allows us to define the value once, and only have to change one line of code instead of searching for and modifying many.

In JavaScript there are two keywords that can be used to define a variable; **var** and **let**; and they both have their own uses. The keyword **var** creates a property on the global object, whereas **let** is only accessible within the scope in which it was created (more on this later). Constants are defined in the same way except they use the keyword **const** instead.

When it comes to naming your variables, try to be consistent and descriptive. JavaScript variables start with a lowercase character and follow camelCase for the remainder of its name. Constants typically consist of all uppercase characters. Both variables and constants can not start with or contain special characters except for an underscore, a dollar sign (advised against due to potential conflicts with other JavaScript libraries), or a hyphen (but can not be started with a hyphen). Names also can not start with a number but may contain a number elsewhere in the name. Note that variables and constants are CaSe SeNSiTiVE and can only be declared once.

```
var x = 5;  
let y = 3;  
const z = 6;
```

Fig 1.1: Defining variables “x” and “y” using var and let, and a constant “z” using const

With our newly defined variables and constant, let’s try to change them and see what happens. Assume we run the following code;


```
var x = 5;
let y = "something";
const z = 6;

console.log(x, y, z); //5 'something' 6

x = "else";
y = 3;
console.log(x, y); // 'else' 3
```

Fig 1.2: Declaration of variables and a constant and reassignment

We would expect the first several lines to execute, define, and initialize the variables `x` to 5, `y` to “something”, and our constant `z` to 6, and then using the names of the variables and constant, output them to the console using `console.log`.

The last few are a bit different though, we don’t have the keyword `var` or `let` in front of them. This is because they have already been defined and exist in memory, meaning we can assign new values to them using just their names. As expected, this runs fine, and we see the expected output.

```
C:\javascript>node hello
5 'something' 6
else 3
```

Fig 1.3: Output of Fig 1.2

Be sure to notice that we assigned different types of information to the variables `x` and `y`, with `x` becoming a string and `y` becoming a number. It is very important to note that JavaScript is a *loosely typed* or *dynamically typed* language, we can change the type of information that a variable holds at any point throughout execution. We will cover how to handle this in chapter two when we go over checking data types.

What about the constant “z” though? We didn’t try to assign a new value to it. Let’s modify our code to include an attempt to assign it a new value.

```
var x = 5;
let y = "something";
const z = 6;

console.log(x, y, z); //5 'something' 6

x = "else";
y = 3;
console.log(x, y); //‘else’ 3

z = "oops";
console.log(z); //???
```

Fig 1.4: Attempt to reassign a constant (File: variables-constants-1.js)

What do you think the output will be? If you guessed that an error would be raised, you would be correct. As stated earlier, constants can not be modified once initialized, and the engine running our JavaScript code is very happy to let us know we’ve made a mistake. On the bright side, it also informs us the line on which we’ve made the mistake following the colon beside the name of our file.

```
C:\javascript\hello.js:11
z = "oops";
  ^
TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\javascript\hello.js:11:3)
    at Module._compile (module.js:652:30)
    at Object.Module._extensions..js (module.js:663:10)
    at Module.load (module.js:565:32)
    at tryModuleLoad (module.js:505:12)
    at Function.Module._load (module.js:497:3)
    at Function.Module.runMain (module.js:693:10)
    at startup (bootstrap_node.js:188:16)
    at bootstrap_node.js:609:3
```

Fig 1.5: Output after attempting to reassign a constant

2. Data Types

Like many other programming languages, JavaScript comes with a list of six predefined primitive data types, and another more complex type referred to as an Object. In JavaScript, these primitive data types are immutable, meaning that they can not be modified. Any operation performed on these values return a new instance of the type being acted upon. These primitive data types are as follows:

- **Boolean** – a true or false value
- **Number** – whole and floating-point numbers
- **String** – textual data
- **Null** – a type of object that contains the value null used to represent no information
- **Undefined** – typically refers to the non-existence of a variable
- **Symbol** – used as a key to represent the uniqueness of an object

Symbols are new to ECMAScript 2015 and will not be covered throughout this text. If you would like to read further into their uses you may do so [here](#).

Objects are what we refer to as complex in that they can consist of one or many properties that have values of primitive data types or contain further, nested, complex objects. An object is typically something that we refer to as iterable, meaning we can access each value contained within them sequentially. Objects have a key as an index, indexed arrays have a number as an index, and associative arrays use strings to represent the index. More on this later.

There are multiple ways to determine what the type is of a piece of data using two built in functions **typeof** and **instanceof**, each having their own specific uses. Using **typeof** returns a textual representation of the data type, and **instanceof** determines if a piece of data matches an objects prototype. If you refer to the chapter on defining variables and constants you can see that we can assign any type of data to a variable or constant. Programmatically, this means that there is no guarantee that the type of one variable will remain the same by the time you use it elsewhere, it is on you as a programmer to ensure the data you are handling is what you expect.

```
console.log(typeof 3);           //"number"  
console.log(typeof "abc");       //"string"  
console.log(typeof true);        //"boolean"  
console.log(typeof null);        //"object"  
console.log(typeof undefined);   //"undefined"
```

*Fig 2.1: Outputs of **typeof** on the primitive types (File: typeof-1.js)*

Based on the outputs of Fig 2.1 you can see that it is possible to determine the type of a piece of primitive data. You may have also noticed that the output of “typeof null” returned “object”. This is the expected behaviour and can be a bit misleading. We can also use variables instead of hardcoded values to see the type of data a variable contains.

```
let x = "hello";  
console.log(typeof x);  //"string"  
x = 3;  
console.log(typeof x);  //"number"
```

Fig 2.2: Seeing the type of data a variable contains

Another method to check the type of something was mentioned, `instanceof`, and its use is different than `typeof` in that it is used to determine if a piece of data is an instance of a specified object. Even primitives can become an object and lose its initial `typeof` if associated with the **new** keyword and the object you'd like to create. Simply put, the keyword **new** calls an objects constructor and creates an instance of that object.

```
let x = "Hello";  
console.log(typeof x); // "string"  
x = new String("World")  
console.log(typeof x); // "object"  
console.log(x.constructor.name); // "String"  
console.log(x instanceof String); // "true"
```

Fig 2.3: Difference between `typeof` and `instanceof` on an Object (File: `instanceof-1.js`)

While it may be unlikely that you come across instantiated primitives it is important to note these differences as when you begin dealing with more complex objects it may be required to determine whether the data you're accessing is the data you require.

As Obi-Wan once said; "These aren't the droids you're looking for."

Star Wars: Episode IV - A New Hope

3. Operators

JavaScript comes packed with a long list of ways to manipulate your data. In fact, there are so many that we won't be able to cover them all here. We will only be covering the primary operators necessary for most basic to intermediate tasks. If you would like to read further into the long list of available operators you may do so on Mozilla's Developer Network (MDN) - [expressions and operators](#).

The operators we will be focusing on are:

- **Assignment** – storing information in a variable or constant
- **Arithmetic** – mathematical operations
- **Comparison** – conditional to determine sameness or likeness
- **Logical** – compound conditionals

3.1: Assignment Operators:

If you recall chapter one when we were creating our variables and constants we were using an equals sign (=) to give the variable a value. Assignment works in the sense that you'll have a left-hand operand, and a right-hand operand, with the latter being assigned to the prior.

```
let x = 3;
```

Fig 3.1: Simple assignment

There are also several shorthand arithmetic assignment operators that can be used to quickly perform an operation by adding an operation before the assignment. More will be mentioned on the available arithmetic operations in the next part of this chapter.

When using the shorthand assignment, the right-hand operand is applied to the left using the arithmetic sign preceding the equals. There is one exception with strings in that they can be used with the addition shorthand assignment to perform concatenation. Some sample shorthand arithmetic assignment operators include addition, subtraction, multiplication, division, modulus, and exponential, and are demonstrated below.

```
let x = 5;
x += 3; //addition: x = 8 (5 + 3) => (x = x + 3)
x -= 5; //subtraction: x = 3 (8 - 5) => (x = x - 5)
x *= 4; //multiplication: x = 12 (3 * 4) => (x = x * 4)
x /= 6; //division: x = 2 (12 / 6) => (x = x / 6)
x **= 3; //exponential: x = 8 (2 ** 3) => (x = x ** 3)
x %= 2; //modulus: x = 0 (2 % 2) => (x = x % 2)
x = "Hello "; //set x to "Hello "
x += "World"; //concatenation: x = "Hello World" ("Hello " + "World")
```

Fig 3.2: Shorthand arithmetic assignment (File: assignment-1.js)

3.2: Arithmetic Operators & Expressions:

We saw in Fig 3.2 how shorthand arithmetic assignment works, but you are also capable of performing a mathematical expression using just the operators as well, whether it be using number literals or a previously defined variable or constant. One core thing to note is that unlike many other programming languages, dividing by zero does not produce an error but returns Infinity instead.

You've already had a chance to see some of the primary arithmetic operators in action, so we'll take a moment to cover some of the more interesting ones that JavaScript has to offer which are the increment and decrement operators (`++` and `--`), and the unary negation and addition operators (`+` and `-`).

The increment and decrement operators both have two use cases, pre and post expression. In the case of pre-expression, the increase or decrease happens before the rest of the expression is executed, however in post-expression, the operation takes place after, using the initial value of your number. As for our unary operators, the minus (`-`) will negate any value provided returning the negative representation of it, and the unary addition will attempt to convert or typecast a piece of data to a number.

```
let x = 3;
3 + 4; //7
x + 5; //8
x++; //3, however the value of x is now 4 (post-increment)
++x; //5, and the value of x is now 5 (pre-increment)
x--; //5, and the value of x is now 4 (post-decrement)
--x; //3, and the value of x is now 3 (pre-decrement)
-x; //-3, the value of x remains the same (unary negation)
+true; //1, true is typecast to a number (unary addition)
+"5"; //5, the string "5" is typecast to a number (unary addition)
"JavaScript " + "Rules"; //"JavaScript Rules" (concatenation)
4 / 0; //Infinity
```

Fig 3.3: Concatenation, Addition, Increment, Decrement, and Unary operators

(File: arithmetic-1.js)

To continue with our arithmetic operators, we can perform more advanced expressions by chaining several of them together, just be aware of the order of operations in which they're performed. JavaScript follows your standard BEDMAS rules; *Brackets, Exponents, Division,*

Multiplication, Addition, and Subtraction. You can even combine the operators shown in Fig 3.3 in an expression as well.

```
let x = 3;
let y = 5;
let z = x + y; //8
z = z + x; //11
z += z + y / 5; //11 + (11 + 5 / 5) => 11 + (11 + 1) => 23
z = (z + 1) / x; //(23 + 1) / 3 => 24 / 3 => 8
z = --z + z++ - ++x * --y - -true; //7 + 7 - 4 * 4 - -1 => 15 - 16 => -1
z = "Number: " + z; //"Number: -1"
```

Fig 3.4 Mathematical expressions and concatenation (File: arithmetic-2.js)

3.3: Comparison Operators:

Comparison operators are used when comparing pieces of information to determine some form of condition. These typically come in the shape of *(left-hand operand) (comparator) (right-hand operand)* and return a logical value of true or false. With these operators we can determine if something is equal, not equal, greater than, or less than something else. A common mistake many new developers make is mistaking the assignment (=) operator with the comparator equals (==). It is of dire importance to note that JavaScript performs something known as typecasting (or type coercion) if the two data types you're comparing are not the same.

The following are usable comparison operators in JavaScript:

- **Equals (==)** – loose equality
- **Not-Equals (!=)** – loose inequality
- **Strict Equals (===)** – strong equality
- **Strict Not-Equals (!==)** – strong inequality

- Less Than (<)
- Less Than Or Equal (<=)
- Greater Than (>)
- Greater Than Or Equal (>=)

```
let x = 3;
let y = 5;
x == 3; //true
x == '3'; //true
x === '3'; //false
x === 3; //true
x == y; //false
x != y; //true
x != 3; //false
x != '3'; //true
y > x; //true
x <= 3; //true
x < 3; //false
```

Fig 3.5: Simple comparisons (File: comparisons-1.js)

3.4: Logical Operators:

JavaScript has three logical operators, logical AND (&&), logical OR (||), and logical NOT (!). While typically used in combining multiple comparisons and performing some form of boolean algebra it is important to note that the return value is one of the values being compared and not always a boolean true or false, except for in the case of the logical NOT.

There are certain values that will equate to a falsy expression without technically being false; these are:

- 0
- NaN
- An empty string ("")
- undefined
- null

```
"hello" && "world"; //"world"
true && true; //true
true && false; //false
true && undefined; //undefined
true && 0; //0
false && false; //false
true || false; //true
false || false; //false
"text" || false; //"text"
"text" || 0; //"text"
false || 0 || NaN; //NaN
!true; //false
!false; //true
!null; //true
!NaN; //true
false && !false; //false
true && !false; //true
```

Fig 3.6: Logical operators (File: logical-1.js)

3.5: Short Circuiting:

These types of expressions are evaluated from left to right, and depending on the operator further comparison stops when a certain condition is met. This is known as short-circuiting. For logical AND (&&), an expression is short-circuited upon the first falsy value that appears and returns said falsy value; or if the entire statement is truthy it returns the last truthy

value of the expression. For logical OR (`||`), an expression is short-circuited upon the first truthy value and returns said truthy value; or if the entire statement is false, returns the last falsy value in the expression.

In Fig 3.6 I listed several logical operator examples. Now if I were to chain many of them together you would be able to better see just how short-circuiting works.

```
console.log("hello" && "world" && "JavaScript"); //JavaScript
console.log(3 || 4 || 5 || 6); //3
```

Fig 3.7: Short-circuiting (File: logical-1.js)

In the first output, all 3 strings equate to true and are evaluated left to right, returning the last value. In the second output, all 4 numbers equate to true and the first instance of a true value is returned.

Depending on the scenario, you may come across a time where you need to run a complex function within your conditional, or possibly even many complex conditionals. With understanding short-circuiting you can see that for any combination of logical AND conditions, starting with the value that is most likely to be false could prevent excess calculations.

4. Structural Programming

Structural programming is where we, as a programmer, begin to decide and determine how we want our application to run through the use of control flow constructs. These control flow constructs include any type of decision making and iterative blocks of code such as our **if else** blocks, **switch** statements, **for** blocks, **do** blocks, **do while** blocks, a **recursive** function, and in some cases, **try catch finally** blocks. This will be our first introduction to how we can use some decision making to control what a user sees, perform some form of calculation, cease execution, or simply throw an error.

4.1: Getting Input From The User:

For various parts of this and later chapters, we will be using input that you, as a user, provide so we can actively see how our code reacts depending on our input, and provides you with a way to actively engage with the content. If you are not using Node.js please refer to [Getting Started](#).

For this you will require an npm package known as [readline-sync](#). It's very easy to install, all you'll need to do is navigate to the root directory where you're storing and running your JavaScript files from and run "**npm init -y**", and then "**npm i readline-sync**". After which we'll be able to use that package in our files here-on-out and you'll be able to interact with the scripts you write. If you cloned the directory from the GitHub repository you may be able to skip this step as it will have already been included.

```
C:\javascript>cd ..  
  
C:\>cd /javascript  
  
C:\javascript>npm i readline-sync  
npm WARN saveError ENOENT: no such file or directory, open 'C:\javascript\package.json'  
npm WARN enoent ENOENT: no such file or directory, open 'C:\javascript\package.json'  
npm WARN javascript No description  
npm WARN javascript No repository field.  
npm WARN javascript No README data  
npm WARN javascript No license field.  
  
+ readline-sync@1.4.9  
added 1 package in 1.392s
```

Fig 4.1.1: Installing the readline-sync module

Don't worry about the warnings if you receive any, I simply forgot to run `npm init -y` so I did not have the `package.json` and other files it was looking for. That's ok, Node will create what it needs to keep track of everything and we can still use the module as intended.

4.2: If, Else If, Else Statements:

Our first look into structural programming, and probably the most common control structure is our `if / else if / else` statements. These statements use some form of conditional to determine what to do next in the shape of **`if (condition) { actions }`**.

```
if (true) {  
  console.log("The statement was true.");  
}
```

Fig 4.2.1: Sample if statement

Look at the example in Fig 4.2.1, we check to see if a condition equates to true, if it does we perform the statements written within the curly braces (block). Since we hardcoded the value of true as our condition, our code is guaranteed to execute, and we see the statement logged to the console.

Similarly, we can do something if the condition equates to **false** using an **else**.

```
if (false == true) {  
    console.log("The statement was true.");  
} else {  
    console.log("The statement was false.");  
}
```

Fig 4.2.2: If Else statement

In Fig 4.2.2 we first test to see if false is equal to true, which we know it isn't, so the first block of code does not get executed and instead, since we have an **else**, execute that. Based on this we can conclude that the **else** portion of an if statement will be executed if and only if the preconditions are not met.

We don't need to just use hardcoded values, we can also supply variables as our conditions to make more complex decisions. Say we want to determine if a number is greater than another to apply some form of discount, we could achieve something of the sort with some type of conditional. In English that may look something like; if the customer spends more than or exactly \$500, apply a 15% discount to the purchase, otherwise apply only a 10% discount. This is a practical example of a real-world application and we can achieve something as such:

```

const MAX_DISCOUNT = .15;
const MIN_DISCOUNT = .10;
let amount = 570;

if (amount >= 500) {
  console.log("Your discount is: " + (MAX_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MAX_DISCOUNT)));
} else {
  console.log("Your discount is: " + (MIN_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MIN_DISCOUNT)));
}

```

Fig 4.2.3: Discount determiner

Using the module that we installed in Chapter 4.1, let's modify the above sample to accept input from the user.

```

const rl = require('readline-sync');
const MAX_DISCOUNT = .15;
const MIN_DISCOUNT = .10;

let amount = rl.question("Please enter how much was spent: $");

if (amount >= 500) {
  console.log("Your discount is: " + (MAX_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MAX_DISCOUNT)));
} else {
  console.log("Your discount is: " + (MIN_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MIN_DISCOUNT)));
}

```

Fig 4.2.4: Discount determiner with user input

Here we are using the require keyword to load our readline module into a constant rl that we can use elsewhere in our code. Then, by using the question method of the module, we

can create a prompt that will allow us to receive input. Run it a few times and supply some different numeric inputs. My output looked something like this:

```
C:\javascript>node discount
Please enter how much was spent: $400
Your discount is: 10%
Your total: $360

C:\javascript>node discount
Please enter how much was spent: $500
Your discount is: 15%
Your total: $425
```

Fig 4.2.5: Output of discount determiner

But what if you want to have more than two conditions? What if you have three possible discounts? I'm glad you asked! Luckily, we can chain our conditionals together with the use of **else if** statements. It's good practice to look at your conditionals from a top down perspective, starting with your max and ending with your min. In the previous example we start by determining if the total is greater than or equal to 500, then move on to our next. Doing this in such a manner allows us to avoid any unnecessary extra calculations.

```

const rl = require('readline-sync');
const MAX_DISCOUNT = .15;
const MID_DISCOUNT = .12;
const MIN_DISCOUNT = .10;

let amount = rl.question("Please enter how much was spent: $");

if (amount >= 500) {
  console.log("Your discount is: " + (MAX_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MAX_DISCOUNT)));
} else if (amount >= 250) {
  console.log("Your discount is: " + (MID_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MID_DISCOUNT)));
} else {
  console.log("Your discount is: " + (MIN_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MIN_DISCOUNT)));
}

```

Fig 4.2.6: If / Else If / Else (File: if-1.js)

This was a very simple example using only one condition. However, there is so much more you can do with if statements. From combining conditions to even nesting further if statements within another. Nesting is where we write another block statement within another. Take a look at Fig 4.2.7 to see it in action.

Once you feel comfortable with writing some basic conditionals, try something more creative, try comparing the input to different strings, show a message if their favourite fruit is a banana. We don't always have to use numbers as our conditions.

```

const rl = require('readline-sync');
const MAX_DISCOUNT = .15;
const MID_DISCOUNT = .12;
const MIN_DISCOUNT = .10;

let amount = rl.question("Please enter how much was spent: $");
let items = rl.question("Please enter how many items were purchased: ");

if (amount >= 500 && items >= 5) {
  //The user spent at least $500 and purchased more than 5 items.
  console.log("Your discount is: " + (MAX_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MAX_DISCOUNT)));
} else if (amount >= 250) {
  //If the user spent at least $250.
  if (items >= 10) {
    //If the user purchased at least 10 items.
    console.log("Your discount is: " + (MAX_DISCOUNT * 100) + "%");
    console.log("Your total: $" + (amount * (1 - MAX_DISCOUNT)));
  } else {
    //If the user purchased less than 10 items.
    console.log("Your discount is: " + (MID_DISCOUNT * 100) + "%");
    console.log("Your total: $" + (amount * (1 - MID_DISCOUNT)));
  }
} else {
  //The user spent less than $250.
  console.log("Your discount is: " + (MIN_DISCOUNT * 100) + "%");
  console.log("Your total: $" + (amount * (1 - MIN_DISCOUNT)));
}

```

Fig 4.2.7: Nested and multi-conditional if / else if / else (File: if-2.js)

There is one more method of writing a conditional that involves the use of a question mark and a colon that we refer to as the **ternary operator**. These are generally used when the output can only be one of two things however can be chained into longer expressions, although they do get messy if they become too long. One of the core uses of the **ternary** if is to quickly assign a value to a variable with as little code possible and are written in the form of **(condition) ? true statement : false statement;**

```
let x = 7;  
let y = x >= 5 ? "x is larger than 5." : "x is smaller than 5.";
```

Fig 4.2.8: Ternary statement (File: ternary-1.js)

4.3: Strict Equality vs Loose Equality:

Remember back to our [comparison operators](#), I had mentioned something about there being a difference between strict equality (`===`) and loose equality (`==`). In the previous examples we were asking the user for some input, a number, of how much a customer had spent and how many items they had purchased, but what if we were asking for a specific numeric value? Consider the following example:

```
const rl = require('readline-sync');  
  
let whatType = rl.question("What type of input is this: ");  
console.log(typeof whatType);
```

Fig 4.3.1: Determining the type of input

If I were to input a string, the type would be obvious, a string. However, what if I enter the number 3? One might assume that the type of the input would be a number, but they would be wrong. When receiving any kind of input from the user, we receive a string, and it is up to us, as a programmer, to determine if we want to leave the value as a string or parse it to another data type. This can be important when you're expecting one data type but receiving another.

```
const rl = require('readline-sync');

let whatType = rl.question("Input the number 3: ");
console.log(typeof whatType);

if (whatType === 3) {
    console.log("3 was entered and its type is number.");
}

if (whatType == 3) {
    console.log("3 was entered, but its type might not be a number.");
}
```

Fig 4.3.2: Strict vs Loose equality

If you were to run the above code you'd notice that the first if statement is not executed, but the second one is. That is because JavaScript performs type coercion on the input to have it match that of the right-hand operand. Let's try to do something with the input as it is and see what happens.

```
if (whatType == 3) {
    console.log("3 was entered, but it's type might not be number.");
    console.log(3 + whatType);
}
```

Fig 4.3.3: Performing calculations with user input

That's easy, the answer is 6! Well, mathematically speaking you would be correct; however you must remember that even though you entered a number, the input is still a string, and JavaScript will perform concatenation instead, leaving you with the following output.

```
C:\javascript\examples\structural\if>node if-3
Input the number 3: 3
string
3 was entered, but it's type might not be number.
33
```

Fig 4.3.4: Output of Fig 4.3.3

The easiest solution to this problem is if you know the data type you're expecting, you can try to cast the information to the type you need. We'll cover this more in depth later, however you may look at the following completed example for now to try to make sense of it. However, do note that inputting anything but a number will return **NaN**, or Not a Number for short, an early example of the use of the **isNaN** built in function demonstrates one of the ways to determine if the input genuinely was a number or some other type of data.

```
const rl = require('readline-sync');

let whatType = rl.question("Input the number 3: ");
console.log(typeof whatType);

whatType = parseInt(whatType);

if (whatType === 3) {
  console.log("3 was entered and its type is number.");
} else if (whatType == 3) {
  console.log("3 was entered, but its type might not be a number.");
} else if (isNaN(whatType)) {
  console.log("You didn't enter a number.");
}
console.log(3 + whatType);
```

Fig 4.3.5: Completed strict vs loose equality example (File: if-3.js)

```
C:\javascript\examples\structural\if>node if-3
Input the number 3: 3
string
3 was entered and its type is number.
6
```

Fig 4.3.6: Output of Fig 4.3.5

4.4: Switch Statements:

Another way that we can control the flow of our application is through something called a **switch statement**. These control constructs are very similar to if statements in that they are still conditional, but they take a more top down approach and are used when we have a specific idea of the information that we will be processing and come in the form of **switch(variable) { case 1: ... break;, default: }**.

To break it down a little, the variable we are taking in will typically contain a value that is known to us, allowing us to make specific decisions on the input. The case portion is a specific value that we are comparing the variable against, and the default will be executed should the variable not match any of our conditions. Consider the following example.

```
const rl = require('readline-sync');

let showSomething = rl.question("Please enter a number between 1-3 inclusive: ");
showSomething = parseInt(showSomething);

switch(showSomething) {
  case 1:
    console.log("You chose what's behind door #1!");
    break;
  case 2:
    console.log("You chose what's behind door #2!");
    break;
  case 3:
    console.log("You chose what's behind door #3!");
    break;
  default:
    console.log("You didn't follow the instructions.");
}
```

Fig 4.4.1: Sample switch statement (File: switch-1.js)

In Fig 4.4.1, we are prompting the user for a number between 1-3 inclusively, and then depending on what they've entered we display a corresponding message. A **break;** is used to cease any further execution within the block. If you're curious as to what would happen without the **break;** being present, try and remove one of them and then select that option. You will notice that the following **case** is also executed. Therefore, you can see that a **break;** is to prevent unintended code from being executed.

Another way to look at a switch statement is to consider it as an over glorified series of if, else if, else statements where we use specific values instead of a wide range of cases. We could have written the exact same code in the following manner.


```

const rl = require('readline-sync');

let showSomething = rl.question("Please enter a number between 1-3
inclusive: ");
showSomething = parseInt(showSomething);

if (showSomething === 1) {
  console.log("You chose what's behind door #1!");
} else if (showSomething === 2) {
  console.log("You chose what's behind door #2!");
} else if (showSomething === 3) {
  console.log("You chose what's behind door #3!");
} else {
  console.log("You didn't follow the instructions.");
}

```

Fig 4.4.2: Switch statement rewritten as a series of if statements

4.5: For Loops:

For loops are probably the staple to structural programming in that it makes our job of repeating a task extremely simple if we know how many times we want something to be performed. We can place a series of statements to be executed within the block to have them performed **x** amount of times, where **x** is typically a number.

A for loop consists of three components, **initialization**, **condition**, **expression**, where our initialization is declaring where we want to begin, our condition is some form of logical expression, and our expression is an action we perform to the variable in the initialization to meet our end condition.

The most common form of for loop that you will come across will be written as **for (initialization; condition; expression) { ... }** however there may be cases where one, or all, of the statements may be omitted, such as **for (; condition; expression) { ... }**, **for (; ; expression) { ... }**, **for (; ;) { ... }**, or even **for (; condition;) { ... }** depending on your requirements. You may

look at the file *for-variations.js* to see these differences. Be careful when using any of the variants as you will require additional logic within your loops body to know when to break out, otherwise you could end up with an **infinite loop**.

```
for (let i = 0; i <= 10; i++) {
  console.log("i: " + i);
}
```

Fig 4.5.1: Simple structure of a for loop to count from 0 to 10 inclusively (File: for-1.js)

It's not just counting that we can do with a for loop, we can also perform some other calculations or some other logic within the body. Say we wanted to calculate a given factorial in the form of $n_1 * n_2 * n_3 * \dots n_m$ where $n \geq 0$. We can begin to combine our knowledge of if statements and for loops to create a solution.

```
const rl = require('readline-sync');
let factorial = rl.question("Enter a number > 0: ");
factorial = parseInt(factorial); //Attempt to parse our input to a number

if (isNaN(factorial)) {
  console.log("You did not enter a number.");
} else if (factorial < 0) {
  console.log("You entered a negative number.");
} else if (factorial === 0) {
  console.log("Factorial of 0: 1");
} else {
  let sum = 1;
  for (let i = 1; i <= factorial; i++) {
    //Multiply our sum by current value of i
    sum *= i;
  }
  console.log("Factorial of " + factorial + ": " + sum);
}
```

Fig 4.5.2: Factorial with a for loop and if statements (File: for-2.js)

There are other variations of the for loop for arrays and objects such as the **for in** and **for of** that perform an action a little differently than your typical for loop and will be touched on when we reach arrays.

4.6: While & Do While:

There are two other common methods of repetition in JavaScript, the **while** and **do while** loops, and they both operate in a very similar manner with the core exception that a **do while** loop is guaranteed to execute at least once. A while loop will appear in the format of **while (condition) { ... }** whereas your do while loop will appear in the format of **do { ... } while (condition);**. They will both continue to repeat so long as their condition remains true.

```
while(false) {  
    console.log("Inside of while.");  
}  
//No output since condition is checked first.  
  
do {  
    console.log("Inside of do-while.");  
} while(false);  
//Output since condition is checked last.
```

Fig 4.6.1: While vs Do While (File: do-vs-dowhile.js)

The example that is used in many other programming languages to demonstrate an applicable use-case scenario is prompting the user for a number repeatedly until they provide valid input. Let's take a look at how we can achieve this using our **rl.question** and our while and do while loops.

```
const rl = require('readline-sync');
let userInput = +rl.question("Enter a number between 1 & 10 inclusive: ");

while(userInput < 1 || userInput > 10 || isNaN(userInput)) {
    userInput = +rl.question("Enter a number between 1 & 10 inclusive: ");
    //Using unary addition for quick typecasting
}
console.log("The user entered: " + userInput);
```

Fig 4.6.2: While loop checking user input for a number between 1 – 10 (File: while-1.js)

```
const rl = require('readline-sync');
let userInput = 0;

do {
    userInput = +rl.question("Enter a number between 1 & 10 inclusive: ");
    //Using unary addition for quick typecasting
} while(userInput < 1 || userInput > 10 || isNaN(userInput));
console.log("The user entered: " + userInput);
```

Fig 4.6.3: Do-While loop checking user input for a number between 1 – 10 (File: dowhile-1.js)

You may say there isn't much of a difference except for one, in the while loop I prompt both outside of the loop, and inside of the loop, whereas the do-while loop only has one point of prompting. There is no wrong, right, or is one better than the other between the two, it all depends on the problem you're trying to solve.

4.7: Breaking & Continuing:

There may be certain scenarios where you are required to either break out of a loop prematurely or continue to the next iteration. The two keywords to accomplish this are **break** and **continue**.

```

const rl = require('readline-sync');
let userInput = 0;

do {
  console.log("\nSelect an option:");
  console.log("1. Display a message and exit.");
  console.log("2. Skip to the next iteration.");
  console.log("0. Exit.");
  userInput = +rl.question("Option: ");

  if (isNaN(userInput) || userInput < 0 || userInput > 2) {
    console.log("You did not enter a valid option.");
    continue;
  } else if (userInput === 1) {
    console.log("You chose option 1.");
    break;
  } else if (userInput === 2) {
    continue;
  }
  console.log("Thanks for playing!");
  break;
} while(true);

```

Fig 4.7.1: Simple menu style prompt with break and continue (File: breakandcontinue-1.js)

To further emphasise how **continue** works try selecting option 2 or inputting an invalid option. What happens to the code outside of the if statement? Since it isn't contained within the body of any of the conditions it should technically be executed correct? Wrong. Continue will skip any remaining code within the body of the loop and return to the beginning. **Break** is probably a bit easier to understand in that it simply terminates the loop, preventing any further execution.

```
for (let x = 0; ; x++) {
  console.log('For Loop x: ' + x);
  if (x === 5)
    break;
}
console.log('Outside of For Loop\n');
```

Fig 4.7.2: Example of break statement (File: break-1.js)

```
let x = 0;
while (x <= 5) {
  x++;
  if (!(x % 2)) //If x % 2 === 0
    continue;
  console.log('While Loop x: ' + x);
}
console.log('Outside of While Loop\n');
```

Fig 4.7.3: Example of continue statement to show odd numbers (File: continue-1.js)

4.8: Try, Catch, Finally:

Our final structural block statement comes in the form of error handling; the **try**, **catch**, **finally** block statements. These blocks of code are used to wrap any statements that have the potential of throwing an error and allow you to handle them programmatically. A try block must always have at least either a catch block, a finally block, or both a catch and finally block following it. To break down what all three blocks perform:

- **Try** – a block that contains logic or statements that have the potential of producing an error
- **Catch** – a block that receives any error from the try block, and is only executed should an exception arise
- **Finally** – a block that executes at the end of a try block or catch block and is executed regardless if an error was raised or not

If you recall way back in [Chapter 1](#) Fig 1.4, we attempted to assign a new value to a constant that produced an error, if we were to handle that error properly we would be able to continue with the rest of our program and perhaps display a helpful message to the user.

```
const MY_CONST = "try";
try {
  MY_CONST = "catch";
} catch(err) {
  console.error("ERROR: " + err.name);
  console.error("MESSAGE: " + err.message);
  console.error("----- STACK TRACE -----");
  console.error(err.stack);
} finally {
  console.log("\nIt's ok though, we handled it.");
}
```

Fig 4.8.1: Try, Catch, Finally when attempting to assign a new value to a constant (File: try-1.js)

4.9: Scope:

You may have noticed me extensively using the keyword **let** and maybe even **var** throughout a lot of these examples, however I never really explained what the significance is between the two. While JavaScript is a lot more forgiving when it comes to variable access compared to more robust languages it still provides us with some ways to manage our variables in a local and global context.

- **Let** – local block scope variable that exist only within the block and inner blocks in which they are declared, typically encased in { ... }
- **Var** – globally scoped variable that creates a property on the global object regardless of where it was defined

```

let x = "hello";
//Scenario 1:
if (x) {
    let x = "world"; //define local x with "world"
    console.log("Redeclared x: " + x);
}
console.log("Outside of redeclared x: " + x); //remains "hello"

if (true) {
    let y = "They say I don't exist";
}

try {
    console.log('\n');
    console.log(y); //does not exist outside block it was defined
} catch(ex) {
    console.log(ex);
}

```

Fig 4.9.1: Scope of variable declared with let (File: let.js)

```

var x = 123;
//Scenario 1:
if (x) {
    var x = "hello"; //declare inner x with value of "hello"
    console.log(x); //prints "hello"
}
console.log(x); //prints "hello"

//Scenario 2:
if (true) {
    var y = "Declared inside";
}
//prints the variable declared inside the if statement
console.log("Displayed outside: " + y);
y = "New Value"; //gives y a new value
//prints the new value of y
console.log("New value of y: " + y);

```

Fig 4.9.2: Scope of variable declared with var (File: var.js)

Depending on the scenario you may find an instance where using **let** over **var** is a more suitable option when dealing with data that may have core application information or user information. However, there may also be a time when you need to define a variable conditionally for use elsewhere, in this instance **let** would not be a suitable candidate as it would not exist outside of the context in which it was declared.

There is no way to say that one is better than the other; it all depends on your use cases and how you intend to use the data that you're storing. I would argue that as long as you don't need the data outside of where you're declaring it **let** would be the optimal solution.