1.0 Introduction:

1.1 What is JavaScript

Last year, millions of learners from our community started with JavaScript. Why? JavaScript is primarily known as the language of most modern web browsers, and its early quirks gave it a bit of a bad reputation. However, the language has continued to evolve and improve. JavaScript is a powerful, flexible, and fast programming language now being used for increasingly complex web development and beyond!

Since JavaScript remains at the core of web development, it's often the first language learned by self-taught coders eager to learn and build. We're excited for what you'll be able to create with the JavaScript foundation you gain here. JavaScript powers the dynamic behaviour on most websites, including this one.

1.2 Console

The console is a panel that displays important messages, like errors, for developers. Much of the work the computer does with our code is invisible to us by default. If we want to see things appear on our screen, we can print, or *log*, to our console directly.

In JavaScript, the `console` keyword refers to an object, a collection of data and actions, that we can use in our code. Keywords are words that are built into the JavaScript language, so the computer will recognize them and treats them specially.

One action, or method, that is built into the `console` object is the `.log()` method. When we write `console.log()` what we put inside the parentheses will get printed, or logged, to the console.

It's going to be very useful for us to print values to the console, so we can see the work that we're doing.

```
console.log(5);
```

This example logs `5` to the console. The semicolon denotes the end of the line, or statement. Although in JavaScript your code will usually run as intended without a semicolon, we recommend learning the habit of ending each statement with a semicolon so you never leave one out in the few instances when they are required.

You'll see later on that we can use `console.log()` to print different kinds of data.

Solution:

```
console.log(5);
console.log(1);
```

1.3 Data Types

Data Types are the classifications we give to the different kinds of data that we use in programming. In JavaScript, there are seven fundamental data types:

- *Number*: Any number, including numbers with decimals: `4`, `8`, `1516`, `23.42`.
- *String*: Any grouping of characters on your keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: `' ... '` or double quotes `" ... "`. Though we prefer single quotes. Some people like to think of string as a fancy word for text.
- *Boolean*: This data type only has two possible values— either `true` or `false` (without quotes). It's helpful to think of booleans as on and off switches or as the answers to a "yes" or "no" question.
- *Null*: This data type represents the intentional absence of a value, and is represented by the keyword `null` (without quotes).
- *Undefined*: This data type is denoted by the keyword `undefined` (without quotes). It also represents the absence of a value though it has a different use than `null`.
- *Symbol*: A newer feature to the language, symbols are unique identifiers, useful in more complex coding. No need to worry about these for now.
- *Object*: Collections of related data.

The first 6 of those types are considered *primitive data types*. They are the most basic data types in the language. *Objects* are more complex, and you'll learn much more about them as you progress through JavaScript. At first, seven types may not seem like that many, but soon you'll observe the world opens with possibilities once you start leveraging each one. As you learn more about objects, you'll be able to create complex collections of data.

But before we do that, let's get comfortable with strings and numbers!

```
console.log('Location of Codecademy headquarters: 575 Broadway, New York City');
console.log(40);
```

In the example above, we first printed a string. Our string isn't just a single word; it includes both capital and lowercase letters, spaces, and punctuation.

Solution:

```
console.log("JavaScript");

console.log(2011)
console.log('Woohoo! I love to code! #codecademy')
console.log(20.49)
```

1.4 Arithmetic Operators

Basic arithmetic often comes in handy when programming.

An operator is a character that performs a task in our code. JavaScript has several built-in *arithmetic operators*, that allow us to perform mathematical calculations on numbers. These include the following operators and their corresponding symbols:

1. Add: `+`
2. Subtract: `-`
3. Multiply: `*`
4. Divide: `/`
5. Remainder: `%`

The first four work how you might guess:

```
console.log(3 + 4); // Prints 7
console.log(5 - 1); // Prints 4
console.log(4 * 2); // Prints 8
console.log(9 / 3); // Prints 3
```

Note that when we `console.log()` the computer will evaluate the expression inside the parentheses and print that result to the console. If we wanted to print the characters `3 + 4`, we would wrap them in quotes and print them as a string.

```
console.log(11 % 3); // Prints 2
console.log(12 % 3); // Prints 0
```

The remainder operator, sometimes called *modulo*, returns the number that remains after the right-hand number divides into the left-hand number as many times as it evenly can: `11 % 3` equals 2 because 3 fits into 11 three times, leaving 2 as the remainder.

Solution:

```
console.log(35+3.5);
console.log(2022-1969);
console.log(65/240);
console.log(0.2708*100);
```

1.5 String Concatenation

Operators aren't just for numbers! When a `+` operator is used on two strings, it appends the right string to the left string:

```
console.log('hi' + 'ya'); // Prints 'hiya'
console.log('wo' + 'ah'); // Prints 'woah'
console.log('I love to ' + 'code.')
// Prints 'I love to code.'
```

This process of appending one string to another is called *concatenation*. Notice in the third example we had to make sure to include a space at the end of the first string. The computer will join the strings exactly, so we needed to make sure to include the space we wanted between the two strings.

```
console.log('front ' + 'space');
// Prints 'front space'
console.log('back' + ' space');
// Prints 'back space'
console.log('no' + 'space');
// Prints 'nospace'
console.log('middle' + ' ' + 'space');
// Prints 'middle space'
```

Just like with regular math, we can combine, or chain, our operations to get a final result:

```
console.log('One' + ', ' + 'two' + ', ' + 'three!');
// Prints 'One, two, three!'
```

Solution:

```
console.log('Hello' + 'World');
console.log('Hello'+ ' ' +'World')
```

1.6 Properties

When you introduce a new piece of data into a JavaScript program, the browser saves it as an instance of the data type. Every string instance has a property called `length` that stores the number of characters in that string. You can retrieve property information by appending the string with a period and the property name:

```
console.log('Hello'.length); // Prints 5
```

The `.` is another operator! We call it the *dot operator*.

In the example above, the value saved to the `length` property is retrieved from the instance of the string, `'Hello'`. The program prints `5` to the console, because `Hello` has five characters in it.

```
'Teaching the world how to code'
```

Solution:

```
console.log("Teaching the world how to code".length);
```

1.7 Methods

Method is a function that belongs to an object. Remember that methods are actions we can perform. JavaScript provides a number of string methods.

We *call*, or use, these methods by appending an instance with:

- a period (the dot operator)
- the name of the method
- opening and closing parentheses

E.g. `'example string'.methodName()`.

Does that syntax look a little familiar? When we use `console.log()` we're calling the `.log()` method on the `console` object. Let's see `console.log()` and some real string methods in action!

```
console.log('hello'.toUpperCase()); // Prints 'HELLO'
console.log('Hey'.startsWith('H')); // Prints true
```

Let's look at each of the lines above:

- On the first line, the `.toUpperCase()` method is called on the string instance `'hello'`. The result is logged to the console. This method returns a string in all capital letters: `'HELLO'`.
- On the second line, the `.startsWith()` method is called on the string instance `'Hey'`. This method also accepts the character `'H'` as an input, or *argument*, between the parentheses. Since the string `'Hey'` does start with the letter `'H'`, the method returns the boolean `true`.

You can find a list of built-in string methods in the JavaScript documentation. Developers use documentation as a reference tool. It describes JavaScript's keywords, methods, and syntax.

Solution:

```
// Use .toUpperCase() to log 'Codecademy' in all uppercase letters
console.log('Codecademy'.toUpperCase());

// Use a string method to log the following string without whitespace at t
he beginning and end of it.
console.log('    Remove whitespace    '.trim());
```

1.8 Built-in Objects

In addition to `console`, there are other objects built into JavaScript. Down the line, you'll build your own objects, but for now these "built-in" objects are full of useful functionality.

For example, if you wanted to perform more complex mathematical operations than arithmetic, JavaScript has the built-in `Math` object.

The great thing about objects is that they have methods! Let's call the `.random()` method from the built-in `Math` object:

```
console.log(Math.random()); // Prints a random number between 0 and 1
```

In the example above, we called the `.random()` method by appending the object name with the dot operator, the name of the method, and opening and closing parentheses. This method returns a random number between 0 (inclusive) and 1 (exclusive).

To generate a random number between 0 and 50, we could multiply this result by 50, like so:

```
Math.random() * 50;
```

The example above will likely evaluate to a decimal. To ensure the answer is a whole number, we can take advantage of another useful `Math` method called `Math.floor()`.

`Math.floor()` takes a decimal number, and rounds down to the nearest whole number. You can use `Math.floor()` to round down a random number like this:

```
Math.floor(Math.random() * 50);
```

In this case:

1. `Math.random` generates a random number between 0 and 1.
2. We then multiply that number by `50`, so now we have a number between 0 and 50.
3. Then, `Math.floor()` rounds the number down to the nearest whole number.

If you wanted to see the number printed to the terminal, you would still need to use a `console.log()` statement:

```
console.log(Math.floor(Math.random() * 50)); // Prints a random whole
number between 0 and 50
```

To see all of the properties and methods on the `Math` object, take a look at the documentation here.

Solution:

```
// Prints a random number between 0 and 1
console.log(Math.random() * 100);
// Prints a random number between 0 - 1 and Math.floor rounds down a neare
st whole number.
console.log(Math.floor(Math.random() * 100));
// rounds up a decimal number to nearest whole number
console.log(Math.ceil(43.8));
// Number.isInteger checks if a number is an integer
console.log(Number.isInteger(2017));
```

Review: Introduction to JavaScript

Let's take one more glance at the concepts we just learned:

- Data is printed, or logged, to the console, a panel that displays messages, with `console.log()`.
- We can write single-line comments with `//` and multi-line comments between `/*` and `*/`.
- There are 7 fundamental data types in JavaScript: strings, numbers, booleans, null, undefined, symbol, and object.
- Numbers are any number without quotes: `23.8879`
- Strings are characters wrapped in single or double quotes: `'Sample String'`
- The built-in arithmetic operators include `+`, `-`, `*`, `/`, and `%`.
- Objects, including instances of data types, can have properties, stored information. The properties are denoted with a `.` after the name of the object, for example: `'Hello'.length`.
- Objects, including instances of data types, can have methods which perform actions. Methods are called by appending the object or instance with a period, the method name, and parentheses. For example: `'hello'.toUpperCase()`.
- We can access properties and methods by using the `.`, dot operator.
- Built-in objects, including `Math`, are collections of methods and properties that JavaScript provides.

2.0 Variables

2.1 Introduction to variables

In programming, a *variable* is a container for a value. You can think of variables as little containers for information that live in a computer's memory. Information stored in variables, such as a username, account number, or even personalized greeting can then be found in memory.

Variables also provide a way of labelling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves.

In short, variables label and store data in memory. There are only a few things you can do with variables:

1. Create a variable with a descriptive name.
2. Store or update information stored in a variable.
3. Reference or "get" information stored in a variable.

It is important to distinguish that variables are not values; they contain values and represent them with a name. Observe the diagram with the colored boxes. Each box represents variables; the values are represented by the content, and the name is represented with the label.

In this lesson, we will cover how to use the `var`, `let`, and `const` keywords to create variables.

2.2 Create a Variable: var

There were a lot of changes introduced in the ES6 version of JavaScript in 2015. One of the biggest changes was two new keywords, `let` and `const`, to create, or *declare*, variables. Prior to the ES6, programmers could only use the `var` keyword to declare variables.

```
var myName = 'Arya';
console.log(myName);
// Output: Arya
```

Let's consider the example above:

1. `var`, short for variable, is a JavaScript *keyword* that creates, or *declares*, a new variable.
2. `myName` is the variable's name. Capitalizing in this way is a standard convention in JavaScript called *camel casing*. In camel casing you group words into one, the first word is lowercase, then every word that follows will have its first letter uppercased. (e.g. camelCaseEverything).
3. `=` is the *assignment operator*. It assigns the value (`'Arya'`) to the variable (`myName`).
4. `'Arya'` is the *value* assigned (`=`) to the variable `myName`. You can also say that the `myName` variable is *initialized* with a value of `'Arya'`.
5. After the variable is declared, the string value `'Arya'` is printed to the console by referencing the variable name: `console.log(myName)`.

There are a few general rules for naming variables:

- Variable names cannot start with numbers.
- Variable names are case sensitive, so `myName` and `myname` would be different variables. It is bad practice to create two variables that have the same name using different cases.
- Variable names cannot be the same as *keywords*. For a comprehensive list of keywords check out MDN's keyword documentation.

Note: In the next exercises, we will learn why ES6's `let` and `const` are the preferred variable keywords by many programmers. Because there is still a ton of code written prior to ES6, it's helpful to be familiar with the pre-ES6 `var` keyword.

If you want to learn more about `var` and the quirks associated with it, check out the MDN var documentation.

Solution:

```
var favoriteFood = "pizza"
var numOfSlices = 8
console.log(favoriteFood)
console.log(numOfSlices)
```

2.3 Create a Variable: let

As mentioned in the previous exercise, the `let` keyword was introduced in ES6.
The `let` keyword signals that the variable can be reassigned a different value. Take a look at the example:

```
let meal = 'Enchiladas';
console.log(meal); // Output: Enchiladas
meal = 'Burrito';
console.log(meal); // Output: Burrito
```

Another concept that we should be aware of when using `let` (and even `var`) is that we can declare a variable without assigning the variable a value. In such a case, the variable will be automatically initialized with a value of `undefined`:

```
let price;
console.log(price); // Output: undefined
price = 350;
console.log(price); // Output: 350
```

Notice in the example above:

- If we don't assign a value to a variable declared using the `let` keyword, it automatically has a value of `undefined`.
- We can reassign the value of the variable.

Solution:

```
let changeMe = true
changeMe = false
console.log(changeMe)
```

2.4 Create a Variable: const

The `const` keyword was also introduced in ES6, and is short for the word constant. Just like with `var` and `let` you can store any value in a `const` variable. The way you declare a `const` variable and assign a value to it follows the same structure as `let` and `var`. Take a look at the following example:

```
const myName = 'Gilberto';
console.log(myName); // Output: Gilberto
```

However, a `const` variable cannot be reassigned because it is *constant*. If you try to reassign a `const` variable, you'll get a `TypeError`.

Constant variables *must* be assigned a value when declared. If you try to declare a `const` variable without a value, you'll get a `SyntaxError`.

If you're trying to decide between which keyword to use, `let` or `const`, think about whether you'll need to reassign the variable later on. If you do need to reassign the variable use `let`, otherwise, use `const`.

Solution:

```
const entree =  "Enchiladas"
console.log(entree)
entree = "Tacos"
```

2.5 Mathematical Assignment Operators

Let's consider how we can use variables and math operators to calculate new values and assign them to a variable. Check out the example below:

```
let w = 4;
w = w + 1;

console.log(w); // Output: 5
```

In the example above, we created the variable `w` with the number `4` assigned to it. The following line, `w = w + 1`, increases the value of `w` from `4` to `5`.

Another way we could have reassigned `w` after performing some mathematical operation on it is to use built-in *mathematical assignment operators*. We could re-write the code above to be:

```
let w = 4;
w += 1;

console.log(w); // Output: 5
```

In the second example, we used the `+=` assignment operator to reassign `w`. We're performing the mathematical operation of the first operator `+` using the number to the right, then reassigning `w` to the computed value.

We also have access to other mathematical assignment operators: `-=`, `*=`, and `/=` which work in a similar fashion.

```
let x = 20;
x -= 5; // Can be written as x = x - 5
console.log(x); // Output: 15

let y = 50;
y *= 2; // Can be written as y = y * 2
console.log(y); // Output: 100

let z = 8;
z /= 2; // Can be written as z = z / 2
console.log(z); // Output: 4
```

Solution:

```
let levelUp = 10;
let powerLevel = 9001;
let multiplyMe = 32;
let quarterMe = 1152;

// Use the mathematical assignments in the space below:
levelUp += 5;
powerLevel -= 100;
multiplyMe *= 11;
quarterMe /= 4;


// These console.log() statements below will help you check the values of
the variables.
// You do not need to edit these statements.
console.log('The value of levelUp:', levelUp);
console.log('The value of powerLevel:', powerLevel);
console.log('The value of multiplyMe:', multiplyMe);
console.log('The value of quarterMe:', quarterMe);
```

2.6 The Increment and Decrement Operator

Other mathematical assignment operators include the *increment operator* (`++`) and *decrement operator* (`--`).

The increment operator will increase the value of the variable by 1. The decrement operator will decrease the value of the variable by 1. For example:

```
let a = 10;
a++;
console.log(a); // Output: 11
let b = 20;
b--;
console.log(b); // Output: 19
```

Just like the previous mathematical assignment operators (`+=`, `-=`, `*=`, `/=`), the variable's value is updated *and* assigned as the new value of that variable.

Solution:

```
let gainedDollar = 3;
let lostDollar = 50;
gainedDollar++;
lostDollar--;
```

2.7 String Concatenation with Variables

In previous exercises, we assigned strings to variables. Now, let's go over how to connect, or concatenate, strings in variables.

The `+` operator can be used to combine two string values even if those values are being stored in variables:

```
let myPet = 'armadillo';
console.log('I own a pet ' + myPet + '.');
// Output: 'I own a pet armadillo.'
```

In the example above, we assigned the value `'armadillo'` to the `myPet` variable. On the second line, the `+` operator is used to combine three strings: `'I own a pet'`, the value saved to `myPet`, and `'.'`. We log the result of this concatenation to the console as:

```
I own a pet armadillo.
```

Solution:

```
let favoriteAnimal = "lion"
console.log("My favorite animal: " + favoriteAnimal)
```

2.8 String Interpolation

In the ES6 version of JavaScript, we can insert, or *interpolate*, variables into strings using *template literals*. Check out the following example where a template literal is used to log strings together:

```
const myPet = 'armadillo';
console.log(`I own a pet ${myPet}.`);
// Output: I own a pet armadillo.
```

Notice that:

- a template literal is wrapped by backticks `` ` `` (this key is usually located on the top of your keyboard, left of the `1` key).
- Inside the template literal, you'll see a placeholder, `${myPet}`. The value of `myPet` is inserted into the template literal.
- When we interpolate `` `I own a pet ${myPet}.` ``, the output we print is the string: `'I own a pet armadillo.'`

One of the biggest benefits to using template literals is the readability of the code. Using template literals, you can more easily tell what the new string will be. You also don't have to worry about escaping double quotes or single quotes.

Solution:

```
const myName = "Abdul"
const myCity = "Berlin"
console.log(`My name is ${myName}. My favorite city is ${myCity}`)
```

2.9 typeof operator

While writing code, it can be useful to keep track of the data types of the variables in your program. If you need to check the data type of a variable's value, you can use the `typeof` operator.

The `typeof` operator checks the value to its right and *returns*, or passes back, a string of the data type.

```
const unknown1 = 'foo';
console.log(typeof unknown1); // Output: string

const unknown2 = 10;
console.log(typeof unknown2); // Output: number

const unknown3 = true;
console.log(typeof unknown3); // Output: boolean
```

Let's break down the first example. Since the value `unknown1` is `'foo'`, a string, `typeof unknown1` will return `'string'`.

Solution:

```
let newVariable = 'Playing around with typeof.';
console.log(typeof newVariable)
newVariable = 1
console.log(typeof newVariable)
```

Kelvin Weather

Deep in his mountain-side meteorology lab, the mad scientist Kelvin has mastered weather prediction.

Recently, Kelvin began publishing his weather forecasts on his website. However there's a problem: All of his forecasts describe the temperature in Kelvin.

With our knowledge of JavaScript, let's convert Kelvin to Celsius, then to Fahrenheit.

For example, 283 K converts to 10 °C which converts to 50 °F.

Solution:

```
const kelvin = 0;
// The variable kelvin is equal to 293
const celsius = kelvin - 273;
// Store celsius as kelvin - 273
let fahrenheit = celsius * (9/5) + 32;
// Convert to celsius
fahrenheit = Math.floor(fahrenheit);
// Round down
console.log(`The temparature is ${fahrenheit} degrees Fahrenheit.`);

let newton = celsius * (33 / 100);
newton = Math.floor(newton);
console.log(`The temparature is ${newton} in Newton temparature`);
```

Dog Years

Dogs mature at a faster rate than human beings. We often say a dog's age can be calculated in "dog years" to account for their growth compared to a human of the same age. In some ways we could say, time moves quickly for dogs — 8 years in a human's life equates to 45 years in a dog's life. How old would you be if you were a dog?

Here's how you convert your age from "human years" to "dog years":

- The first two years of a dog's life count as 10.5 dog years each.
- Each year following equates to 4 dog years.

Before you start doing the math in your head, let a computer take care of it! With your knowledge of math operators and variables, use JavaScript to convert your human age into dog years.

Solution:

```
const myAge = 35;
// assign variable as a number
let earlyYears = 2;
// store variable as 2
earlyYears *= 10.5;
let laterYears = myAge - 2;
laterYears *= 4;
// multiply by 4
console.log(earlyYears, laterYears);
const myAgeInDogYears = earlyYears + laterYears;
```

```
// add the two age variable and store it to myAgeInDogYears
const myName = "Abdul".toLowerCase();
// lowercaase my name
console.log(`My name is ${myName}. I am ${myAge} years old in human years
which is ${
myAgeInDogYears} years old in dog years`);
```

Review: Variables

Nice work! This lesson introduced you to variables, a powerful concept you will use in all your future programming endeavors.

Let's review what we learned:

- Variables hold reusable data in a program and associate it with a name.
- Variables are stored in memory.
- The `var` keyword is used in pre-ES6 versions of JS.
- `let` is the preferred way to declare a variable when it can be reassigned, and `const` is the preferred way to declare a variable with a constant value.
- Variables that have not been initialized store the primitive data type `undefined`.
- Mathematical assignment operators make it easy to calculate a new value and assign it to the same variable.
- The `+` operator is used to concatenate strings including string values held in variables
- In ES6, template literals use backticks `` ` `` and `${}` to interpolate values into a string.
- The `typeof` keyword returns the data type (as a string) of a value.

3.0 Conditional Statements:

In life, we make decisions based on circumstances. Think of an everyday decision as mundane as falling asleep — if we are tired, we go to bed, otherwise, we wake up and start our day.

These if-else decisions can be modeled in code by creating conditional statements.
A conditional statement checks a specific condition(s) and performs a task based on the condition(s).

In this lesson, we will explore how programs make decisions by evaluating conditions and introduce logic into our code!

We'll be covering the following concepts:

- `if`, `else if`, and `else` statements
- comparison operators
- logical operators
- truthy vs falsy values
- ternary operators
- `switch` statement

3.1 If Statement

We often perform a task based on a condition. For example, if the weather is nice today, then we will go outside. If the alarm clock rings, then we'll shut it off. If we're tired, then we'll go to sleep.

In programming, we can also perform a task based on a condition using an `if` statement:

```
if (true) {
  console.log('This message will print!');
}
// Prints: This message will print!
```

Notice in the example above, we have an `if` statement. The `if` statement is composed of:

- The `if` keyword followed by a set of parentheses `()` which is followed by a *code block*, or *block statement*, indicated by a set of curly braces `{}`.
- Inside the parentheses `()`, a condition is provided that evaluates to `true` or `false`.
- If the condition evaluates to `true`, the code inside the curly braces `{}` runs, or *executes*.
- If the condition evaluates to `false`, the block won't execute.

Solution:

```
let sale = false;
if (sale){
  console.log("Time to buy!");
}
```

3.2 If...Else Statements

In the previous exercise, we used an `if` statement that checked a condition to decide whether or not to run a block of code. In many cases, we'll have code we want to run if our condition evaluates to `false`.

If we wanted to add some default behavior to the `if` statement, we can add an `else` statement to run a block of code when the condition evaluates to `false`. Take a look at the inclusion of an `else` statement:

```
if (false) {
  console.log('The code in this block will not run.');
} else {
  console.log('But the code in this block will!');
}

// Prints: But the code in this block will!
```

An `else` statement must be paired with an `if` statement, and together they are referred to as an `if...else` statement.

In the example above, the `else` statement:

- Uses the `else` keyword following the code block of an `if` statement.
- Has a code block that is wrapped by a set of curly braces `{}`.
- The code inside the `else` statement code block will execute when the `if` statement's condition evaluates to `false`.

`if...else` statements allow us to automate Solutions to yes-or-no questions, also known as *binary decisions*.

Solution:

```
let sale = true;
sale = false;

if(sale) {
  console.log('Time to buy!');
}else{
  console.log("Time to wait for a sale.")
}
```

3.3 Comparison Operators

When writing conditional statements, sometimes we need to use different types of operators to compare values. These operators are called *comparison operators*.

Here is a list of some handy comparison operators and their syntax:

- Less than: `<`
- Greater than: `>`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`
- Is equal to: `===`
- Is not equal to: `!==`

Comparison operators compare the value on the left with the value on the right. For instance:

```
10 < 12 // Evaluates to true
```

It can be helpful to think of comparison statements as questions. When the answer is "yes", the statement evaluates to `true`, and when the answer is "no", the statement evaluates to `false`. The code above would be asking: is 10 less than 12? Yes! So `10 < 12` evaluates to `true`.

We can also use comparison operators on different data types like strings:

```
'apples' === 'oranges' // false
```

In the example above, we're using the *identity operator* (`===`) to check if the string `'apples'` is the same as the string `'oranges'`. Since the two strings are not the same, the comparison statement evaluates to `false`.

All comparison statements evaluate to either `true` or `false` and are made up of:

- Two values that will be compared.
- An operator that separates the values and compares them accordingly (`>`, `<`, `<=`,`>=`,`===`, `!==`).

Solution:

```
let hungerLevel = 7;
if(hungerLevel > 7){
  console.log("Time to eat!")
}else{
  console.log("We can eat later!")
}
```

3.4 Logical Operators

Working with conditionals means that we will be using booleans, `true` or `false` values. In JavaScript, there are operators that work with boolean values known as *logical operators*. We can use logical operators to add more sophisticated logic to our conditionals.

There are three logical operators:

- the *and* operator (`&&`)
- the *or* operator (`||`)
- the *not* operator, otherwise known as the *bang* operator (`!`)

When we use the `&&` operator, we are checking that two things are `true`:

```
if (stopLight === 'green' && pedestrians === 0) {
  console.log('Go!');
} else {
  console.log('Stop');
}
```

When using the `&&` operator, both conditions *must* evaluate to `true` for the entire condition to evaluate to `true` and execute. Otherwise, if either condition is `false`, the `&&` condition will evaluate to `false` and the `else` block will execute.

If we only care about either condition being `true`, we can use the `||` operator:

```
if (day === 'Saturday' || day === 'Sunday') {
  console.log('Enjoy the weekend!');
} else {
  console.log('Do some work.');
}
```

When using the `||` operator, only one of the conditions must evaluate to `true` for the overall statement to evaluate to `true`. In the code example above, if either `day === 'Saturday'` or `day === 'Sunday'` evaluates to `true` the `if`'s condition will evaluate to `true` and its code block will execute. If the first condition in an `||` statement evaluates to `true`, the second condition won't even be checked. Only if `day === 'Saturday'` evaluates to `false` will `day === 'Sunday'` be evaluated. The code in the `else` statement above will execute only if both comparisons evaluate to `false`.

The `!` *not operator* reverses, or *negates*, the value of a boolean:

```
let excited = true;
console.log(!excited); // Prints false
```

```
let sleepy = false;
console.log(!sleepy); // Prints true
```

Essentially, the `!` operator will either take a `true` value and pass back `false`, or it will take a `false` value and pass back `true`.

Logical operators are often used in conditional statements to add another layer of logic to our code.

Solution:

```
let mood = 'sleepy';
let tirednessLevel = 6;
if (mood === "sleepy" && tirednessLevel > 8){
  console.log("time to sleep")
}else{
  console.log("not bed time yet")
}
```

3.5 Truthy and Falsy

Let's consider how non-boolean data types, like strings or numbers, are evaluated when checked inside a condition.

Sometimes, you'll want to check if a variable exists and you won't necessarily want it to equal a specific value — you'll only check to see if the variable has been assigned a value.

Here's an example:

```
let myVariable = 'I Exist!';

if (myVariable) {
   console.log(myVariable)
} else {
   console.log('The variable does not exist.')
}
```

The code block in the `if` statement will run because `myVariable` has a *truthy* value; even though the value of `myVariable` is not explicitly the value `true`, when used in a boolean or conditional context, it evaluates to `true` because it has been assigned a non-falsy value.

So which values are *falsy*— or evaluate to `false` when checked as a condition? The list of falsy values includes:

- `0`
- Empty strings like `""` or `''`
- `null` which represent when there is no value at all
- `undefined` which represent when a declared variable lacks a value
- `NaN`, or Not a Number

Here's an example with numbers:

```
let numberOfApples = 0;

if (numberOfApples){
   console.log('Let us eat apples!');
} else {
   console.log('No apples left!');
}

// Prints 'No apples left!'
```

The condition evaluates to `false` because the value of the `numberOfApples` is `0`. Since `0` is a falsy value, the code block in the `else` statement will run.

Solution:

```
let wordCount = 1;

if (wordCount) {
  console.log("Great! You've started your work!");
} else {
  console.log('Better get to work!');
}


let favoritePhrase = '';

if (favoritePhrase) {
  console.log("This string doesn't seem to be empty.");
} else {
  console.log('This string is definitely empty.');
}
```

3.6 Truthy and Falsy Assignment

Truthy and falsy evaluations open a world of short-hand possibilities!

Say you have a website and want to take a user's username to make a personalized greeting. Sometimes, the user does not have an account, making the `username` variable falsy. The code below checks if `username` is defined and assigns a default string if it is not:

```
let username = '';
let defaultName;

if (username) {
  defaultName = username;
} else {
  defaultName = 'Stranger';
}

console.log(defaultName); // Prints: Stranger
```

If you combine your knowledge of logical operators you can use a short-hand for the code above. In a boolean condition, JavaScript assigns the truthy value to a variable if you use the `||` operator in your assignment:

```
let username = '';
let defaultName = username || 'Stranger';

console.log(defaultName); // Prints: Stranger
```

Because `||` or statements check the left-hand condition first, the variable `defaultName` will be assigned the actual value of `username` if it is truthy, and it will be assigned the value of `'Stranger'` if `username` is falsy. This concept is also referred to as *short-circuit evaluation*.

Solution:

```
let tool = 'marker';
```

```
// Use short circuit evaluation to assign  writingUtensil variable below:
let writingUtensil = tool || "pen";

console.log(`The ${writingUtensil} is mightier than the sword.`);
```

3.7 Ternary Operator

In the spirit of using short-hand syntax, we can use a *ternary operator* to simplify an `if...else` statement.

Take a look at the `if...else` statement example:

```
let isNightTime = true;

if (isNightTime) {
  console.log('Turn on the lights!');
} else {
  console.log('Turn off the lights!');
}
```

We can use a *ternary operator* to perform the same functionality:

```
isNightTime ? console.log('Turn on the lights!') : console.log('Turn off the lights!');
```

In the example above:

- The condition, `isNightTime`, is provided before the `?`.
- Two expressions follow the `?` and are separated by a colon `:`.
- If the condition evaluates to `true`, the first expression executes.
- If the condition evaluates to `false`, the second expression executes.

Like `if...else` statements, ternary operators can be used for conditions which evaluate to `true` or `false`.

Solution:

```
let isLocked = false;

isLocked ? console.log('You will need a key to open the door.') : console.log('You will not need a key to open the door.');

let isCorrect = true;
```

```
isCorrect ? console.log('Correct!') : console.log('Incorrect!');

let favoritePhrase = 'Love That!';

favoritePhrase === 'Love That!' ? console.log('I love that!') : console.lo
g("I don't love that!");
```

3.8 Else If Statements

We can add more conditions to our `if...else` with an `else if` statement. The `else if` statement allows for more than two possible outcomes. You can add as many `else if` statements as you'd like, to make more complex conditionals! The `else if` statement always comes after the `if` statement and before the `else` statement. The `else if` statement also takes a condition. Let's take a look at the syntax:

```
let stopLight = 'yellow';

if (stopLight === 'red') {
  console.log('Stop!');
} else if (stopLight === 'yellow') {
  console.log('Slow down.');
} else if (stopLight === 'green') {
  console.log('Go!');
} else {
  console.log('Caution, unknown!');
}
```

The `else if` statements allow you to have multiple possible outcomes. `if`/`else if`/`else` statements are read from top to bottom, so the first condition that evaluates to `true` from the top to bottom is the block that gets executed.

In the example above, since `stopLight === 'red'` evaluates to `false` and `stopLight === 'yellow'` evaluates to `true`, the code inside the first `else if` statement is executed. The rest of the conditions are not evaluated. If none of the conditions evaluated to `true`, then the code in the `else` statement would have executed.

Solution:

```
let season = 'summer';

if (season === 'spring') {
  console.log('It\'s spring! The trees are budding!');
} else if (season === "winter"){
  console.log("It\'s winter! Everything is covered in snow.")
} else if (season === "fall"){
  console.log("It\'s fall! Leaves are falling!")
```

```
} else if (season === "summer"){
  console.log("It\'s sunny and warm because it\'s summer!")
} else {
  console.log('Invalid season.');
}
```

3.9 The switch keyword

`else if` statements are a great tool if we need to check multiple conditions. In programming, we often find ourselves needing to check multiple values and handling each of them differently. For example:

```
let groceryItem = 'papaya';

if (groceryItem === 'tomato') {
  console.log('Tomatoes are $0.49');
} else if (groceryItem === 'papaya'){
  console.log('Papayas are $1.29');
} else {
  console.log('Invalid item');
}
```

In the code above, we have a series of conditions checking for a value that matches a `groceryItem` variable. Our code works fine, but imagine if we needed to check 100 different values! Having to write that many `else if` statements sounds like a pain!

A `switch` statement provides an alternative syntax that is easier to read and write. A `switch` statement looks like this:

```
let groceryItem = 'papaya';

switch (groceryItem) {
  case 'tomato':
    console.log('Tomatoes are $0.49');
    break;
  case 'lime':
    console.log('Limes are $1.49');
    break;
  case 'papaya':
    console.log('Papayas are $1.29');
    break;
  default:
    console.log('Invalid item');
    break;
}

// Prints 'Papayas are $1.29'
```

- The `switch` keyword initiates the statement and is followed by `( ... )`, which contains the value that each `case` will compare. In the example, the value or expression of the `switch` statement is `groceryItem`.
- Inside the block, `{ ... }`, there are multiple `cases`. The `case` keyword checks if the expression matches the specified value that comes after it. The value following the first `case` is `'tomato'`. If the value of `groceryItem` equalled `'tomato'`, that `case`'s `console.log()` would run.
- The value of `groceryItem` is `'papaya'`, so the third `case` runs— `Papayas are $1.29` is logged to the console.
- The `break` keyword tells the computer to exit the block and not execute any more code or check any other cases inside the code block. Note: Without `break` keywords, the first matching case will run, but so will every subsequent case regardless of whether or not it matches—including the default. This behavior is different from `if/else` conditional statements that execute only one block of code.
- At the end of each `switch` statement, there is a `default` statement. If none of the `cases` are true, then the code in the `default` statement will run.

Solution:

```
let athleteFinalPosition = 'first place';

switch (athleteFinalPosition){
  case "first place":
  console.log("You get the gold medal!");
  break;
  case "second place":
  console.log("You get the silver medal!");
  break;
  case "third place":
  console.log("You get the bronze medal!");
  break;
  default:
  console.log("No medal awarded.")
  break;
}
```

Practice Project: Magic Eight Ball

You've learned a powerful tool in JavaScript: control flow! It's so powerful, in fact, that it can be used to tell someone's fortune. In this project we will build the Magic Eight Ball using control flow in JavaScript. The user will be able to input a question, then our program will output a random fortune.

Solution:

```
const userName = "";
userName === "Jane" ? console.log(`Hello, ${userName}`) : console.log("Hello!")
const userQuestion = "Can you answer my question?"
console.log(`What the user asked: ${userQuestion}`);
const randomNumber = Math.floor(Math.random() * 8);
switch (randomNumber){
  case 0:
  eightBall = "It is certain";
  break;
  case 1:
  eightBall = "It is decidedly so";
  break;
  case 2:
  eightBall = "It is decidedly so";
  break;
  case 3:
  eightBall = "Reply hazy try again";
  break;
  case 4:
  eightBall = "Cannot predict now";
  break;
  case 5:
  eightBall = "Do not count on it";
  break;
  case 6:
  eightBall = "My sources say no";
  break;
  case 7:
  eightBall = "Outlook not so good"
  break;
  case 8:
  eightBall = "Signs point to yes"
  break;
  default:
  break;
}
console.log(`The eight ball answered: ${eightBall}`);
```

Change switch statement to if else statement:

```
if (randomNumber === 0){
  eightBall = "It is certain, 0";
} else if(randomNumber === 1){
  eightBall = "It is decidedly, 1";
} else if (randomNumber === 2){
  eightBall = "Reply hazy try again, 2"
} else if(randomNumber === 3){
  eightBall = "Cannot predict now, 3"
} else if(randomNumber === 4){
  eightBall = "Do not count on it, 4"
} else if(randomNumber === 5){
  eightBall = "My sources say no, 5"
} else if(randomNumber === 6){
  eightBall = "Outlook not so good, 6"
} else if(randomNumber === 7){
  eightBall = "Signs point to yes, 7"
}
console.log(`Magic eight answers: ${eightBall} ${randomNumber}`)
```

Practice Project : Race Day

Codecademy's annual race is just around the corner! This year, we have a lot of participants. You have been hired to write a program that will register runners for the race and give them instructions on race day.

As a timeline, registration would look like this: Here's how our registration works. There are adult runners (over 18 years of age) and youth runners (under 18 years of age). They can register early or late. Runners are assigned a race number and start time based on their age and registration.

Race number:

- Early adults receive a race number at or above 1000.
- All others receive a number below 1000.

Start time:

- Adult registrants run at 9:30 am or 11:00 am.
    - Early adults run at 9:30 am.
    - Late adults run at 11:00 am.
- Youth registrants run at 12:30 pm (regardless of registration).

Solution:

```
let raceNumber = Math.floor(Math.random() * 1000);
const early = true;
const age = 18;

if (early && age > 18){
  raceNumber += 1000
}

if (early && age > 18){
  console.log(`Runners over 18 who registered early start the race at 9.30
 am, their race number is: ${raceNumber}`)
}else if (age > 18 && !early){
  console.log(`Runners over 18 who are late will race at 11:00 am and race
 number is: ${raceNumber}`)
}else if(age < 18){
  console.log(`Youth runners will race at 12.30 pm and the race number is:
 ${raceNumber}`);
}else{
  console.log("Runners exactly 18 years need to report to the registration
 desk");

}
```

Review: Conditional Statements

Way to go! Here are some of the major concepts for conditionals:

- An `if` statement checks a condition and will execute a task if that condition evaluates to `true`.
- `if...else` statements make binary decisions and execute different code blocks based on a provided condition.
- We can add more conditions using `else if` statements.
- Comparison operators, including `<`, `>`, `<=`, `>=`, `===`, and `!==` can compare two values.
- The logical and operator, `&&`, or "and", checks if both provided expressions are truthy.
- The logical operator `||`, or "or", checks if either provided expression is truthy.
- The bang operator, `!`, switches the truthiness and falsiness of a value.
- The ternary operator is shorthand to simplify concise `if...else` statements.
- A `switch` statement can be used to simplify the process of writing multiple `else if` statements. The `break` keyword stops the remaining `case`s from being checked and executed in a `switch` statement.

4.0 Functions

4.1 What are Functions?

When first learning how to calculate the area of a rectangle, there's a sequence of steps to calculate the correct answer:

1. Measure the width of the rectangle.
2. Measure the height of the rectangle.
3. Multiply the width and height of the rectangle.

With practice, you can calculate the area of the rectangle without being instructed with these three steps every time.

We can calculate the area of one rectangle with the following code:

```
const width = 10;
const height = 6;
const area =  width * height;
console.log(area); // Output: 60
```

Imagine being asked to calculate the area of three different rectangles:

```
// Area of the first rectangle
const width1 = 10;
const height1 = 6;
const area1 =  width1 * height1;

// Area of the second rectangle
const width2 = 4;
const height2 = 9;
const area2 =  width2 * height2;

// Area of the third rectangle
const width3 = 10;
const height3 = 10;
const area3 =  width3 * height3;
```

In programming, we often use code to perform a specific task multiple times. Instead of rewriting the same code, we can group a block of code together and associate it with one task, then we can reuse that block of code whenever we need to perform the task again. We achieve this by creating a *function*. A function is a reusable block of code that groups together a sequence of statements to perform a specific task.

4.2 Function Declarations

In JavaScript, there are many ways to create a function. One way to create a function is by using a *function declaration*. Just like how a variable declaration binds a value to a variable name, a function declaration binds a function to a name, or an *identifier*. Take a look at the anatomy of a function declaration below:

A function declaration consists of:

- The `function` keyword.
- The name of the function, or its identifier, followed by parentheses.
- A function body, or the block of statements required to perform a specific task, enclosed in the function's curly brackets, `{ }`.

A function declaration is a function that is bound to an identifier, or name. In the next exercise we'll go over how to run the code inside the function body.

We should also be aware of the *hoisting* feature in JavaScript which allows access to function declarations before they're defined.

Take a look at example of hoisting:

```
greetWorld(); // Output: Hello, World!

function greetWorld() {
  console.log('Hello, World!');
}
```

Notice how hoisting allowed `greetWorld()` to be called before the `greetWorld()` function was defined! Since hoisting isn't considered good practice, we simply want you to be aware of this feature.

If you want to read more about hoisting, check out MDN documentation on hoisting.

Solution:

```
function getReminder(){
  console.log("Water the plants.")
}

function greetInSpanish(){
  console.log("Buenas Tardes.")
}
```

4.3 Calling a Function

As we saw in previous exercises, a function declaration binds a function to an identifier.

However, a function declaration does not ask the code inside the function body to run, it just declares the existence of the function. The code inside a function body runs, or *executes*, only when the function is *called*.

To call a function in your code, you type the function name followed by parentheses.

Solution:

```
function sayThanks(){
  console.log("Thank you for your purchase! We appreciate your business.")
}
sayThanks()
sayThanks()
sayThanks()
```

4.4 Parameters and Arguments

So far, the functions we've created execute a task without an input. However, some functions can take inputs and use the inputs to perform a task. When declaring a function, we can specify its *parameters*. Parameters allow functions to accept input(s) and perform a task using the input(s). We use parameters as placeholders for information that will be passed to the function when it is called.

Solution:

```
function sayThanks(name) {
  console.log(`Thank you for your purchase ${name}! We appreciate your bus
iness.`);
}

sayThanks("Cole")
```

4.5 Default Parameters

One of the features added in ES6 is the ability to use *default parameters*. Default parameters allow parameters to have a predetermined value in case there is no argument passed into the function or if the argument is `undefined` when called.

Take a look at the code snippet below that uses a default parameter:

```
function greeting (name = 'stranger') {
  console.log(`Hello, ${name}!`)
}

greeting('Nick') // Output: Hello, Nick!
greeting() // Output: Hello, stranger!
```

- In the example above, we used the `=` operator to assign the parameter `name` a default value of `'stranger'`. This is useful to have in case we ever want to include a non-personalized default greeting!
- When the code calls `greeting('Nick')` the value of the argument is passed in and, `'Nick'`, will override the default parameter of `'stranger'` to log `'Hello, Nick!'` to the console.
- When there isn't an argument passed into `greeting()`, the default value of `'stranger'` is used, and `'Hello, stranger!'` is logged to the console.

By using a default parameter, we account for situations when an argument isn't passed into a function that is expecting an argument.

Solution:

```
function makeShoppingList(item1 = "milk", item2 = "tea", item3 = "eggs"){
  console.log(`Remember to buy ${item1}`);
  console.log(`Remember to buy ${item2}`);
  console.log(`Remember to buy ${item3}`);
}
```

4.6 Return

When a function is called, the computer will run through the function's code and evaluate the result of calling the function. By default that resulting value is `undefined`.

```
function rectangleArea(width, height) {
  let area = width * height;
}
console.log(rectangleArea(5, 7)) // Prints undefined
```

In the code example, we defined our function to calculate the `area` of a `width` and `height` parameter. Then `rectangleArea()` is invoked with the arguments `5` and `7`. But when we went to print the results we got `undefined`. Did we write our function wrong? No! In fact, the function worked fine, and the computer did calculate the area as `35`, but we didn't capture it. So how can we do that? With the keyword `return`!

To pass back information from the function call, we use a return statement. To create a return statement, we use the `return` keyword followed by the value that we wish to return. Like we saw above, if the value is omitted, `undefined` is returned instead.

When a `return` statement is used in a function body, the execution of the function is stopped and the code that follows it will not be executed. Look at the example below:

```
function rectangleArea(width, height) {
  if (width < 0 || height < 0) {
    return 'You need positive integers to calculate area!';
  }
  return width * height;
}
```

If an argument for `width` or `height` is less than `0`, then `rectangleArea()` will return `'You need positive integers to calculate area!'`. The second return statement `width * height` will not run.

The `return` keyword is powerful because it allows functions to produce an output. We can then save the output to a variable for later use.

Solution:

```
function monitorCount(rows, columns){
  return rows * columns
}
const numOfMonitors = monitorCount(5, 4);
console.log(numOfMonitors);
```

4.7 Helper Functions

We can also use the return value of a function inside another function. These functions being called within another function are often referred to as *helper functions*. Since each function is carrying out a specific task, it makes our code easier to read and debug if necessary.

If we wanted to define a function that converts the temperature from Celsius to Fahrenheit, we could write two functions like:

```
function multiplyByNineFifths(number) {
  return number * (9/5);
};

function getFahrenheit(celsius) {
  return multiplyByNineFifths(celsius) + 32;
};

getFahrenheit(15); // Returns 59
```

In the example above:

- `getFahrenheit()` is called and `15` is passed as an argument.
- The code block inside of `getFahrenheit()` calls `multiplyByNineFifths()` and passes `15` as an argument.
- `multiplyByNineFifths()` takes the argument of `15` for the `number` parameter.
- The code block inside of `multiplyByNineFifths()` function multiplies `15` by `(9/5)`, which evaluates to `27`.
- `27` is returned back to the function call in `getFahrenheit()`.
- `getFahrenheit()` continues to execute. It adds `32` to `27`, which evaluates to `59`.
- Finally, `59` is returned back to the function call `getFahrenheit(15)`.

We can use functions to section off small bits of logic or tasks, then use them when we need to. Writing helper functions can help take large and difficult tasks and break them into smaller and more manageable tasks.

Solution:

```
function monitorCount(rows, columns) {
  return rows * columns;
}
function costOfMonitors(rows, columns){
  return monitorCount (rows, columns) * 200
}
const totalCost = costOfMonitors(5, 4);

console.log(totalCost);
```

4.8 Function Expressions

Another way to define a function is to use a *function expression*. To define a function inside an expression, we can use the `function` keyword. In a function expression, the function name is usually omitted. A function with no name is called an *anonymous function*. A function expression is often stored in a variable in order to refer to it.

Consider the following function expression:

To declare a function expression:

1. Declare a variable to make the variable's name be the name, or identifier, of your function. Since the release of ES6, it is common practice to use `const` as the keyword to declare the variable.
2. Assign as that variable's value an anonymous function created by using the `function` keyword followed by a set of parentheses with possible parameters. Then a set of curly braces that contain the function body.

To invoke a function expression, write the name of the variable in which the function is stored followed by parentheses enclosing any arguments being passed into the function.

```
variableName(argument1, argument2)
```

Unlike function declarations, function expressions are not hoisted so they cannot be called before they are defined. Let's define a new function using a function expression.

Solution:

```
const plantNeedsWater = function(day){
const plantNeedsWater = function(day){
  if (day === "Wednesday"){
    return true;
  }else{
    return false
  }
}
plantNeedsWater("Tuesday")

console.log(plantNeedsWater("Tuesday"))

// or use this way
// const checkWater = plantNeedsWater("Tuesday");
// console.log(checkWater);
```

4.9 Arrow Functions

ES6 introduced *arrow function syntax*, a shorter way to write functions by using the special "fat arrow" `() =>` notation.

Arrow functions remove the need to type out the keyword `function` every time you need to create a function. Instead, you first include the parameters inside the `( )` and then add an arrow `=>` that points to the function body surrounded in `{ }` like this:

```
const rectangleArea = (width, height) => {
  let area = width * height;
  return area;
};
```

It's important to be familiar with the multiple ways of writing functions because you will come across each of these when reading other JavaScript code.

Solution:

```
// const plantNeedsWater = function(day) {
//   if (day === 'Wednesday') {
//     return true;
//   } else {
//     return false;
//   }
// };

// Turn the function expression above to arrow function
const plantNeedsWater = day => {
  if (day === 'Wednesday') {
    return true;
  } else {
    return false;
  }
};
```

4.10 Function declaration vs expression vs arrow functions

```
Const plantNeedswater = function(day){ // function expression
function plantNeedswater(day){     // function declaration
const plantNeedsWater = day => { // arrow function
```

4.11 Concise Body Arrow Functions

JavaScript also provides several ways to refactor arrow function syntax. The most condensed form of the function is known as *concise body*. We'll explore a few of these techniques below:

1. Functions that take only a single parameter do not need that parameter to be enclosed in parentheses. However, if a function takes zero or multiple parameters, parentheses are required.
2. A function body composed of a single-line block does not need curly braces. Without the curly braces, whatever that line evaluates will be automatically returned. The contents of the block should immediately follow the arrow `=>` and the `return` keyword can be removed. This is referred to as *implicit return*.

So if we have a function:

```
const squareNum = (num) => {
  return num * num;
};
```

We can refactor the function to:

```
const squareNum = num => num * num;
```

Notice the following changes:

- The parentheses around `num` have been removed, since it has a single parameter.
- The curly braces `{ }` have been removed since the function consists of a single-line block.
- The `return` keyword has been removed since the function consists of a single-line block.

Solution:

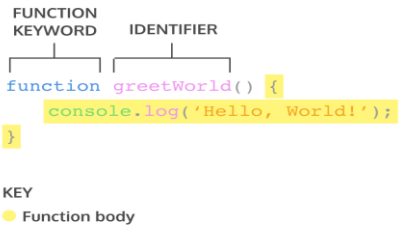
```
const plantNeedsWater = day => day === 'Wednesday' ? true : false;
```

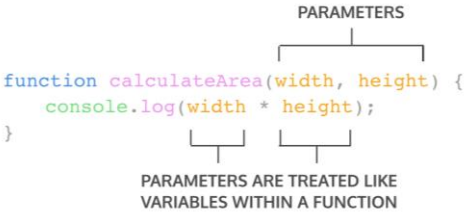
Review: Functions

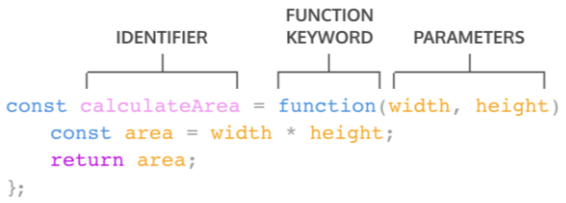
Give yourself a pat on the back, you just navigated through functions!

In this lesson, we covered some important concepts about functions:

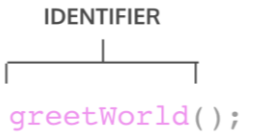
- A *function* is a reusable block of code that groups together a sequence of statements to perform a specific task.
- A *function declaration* :





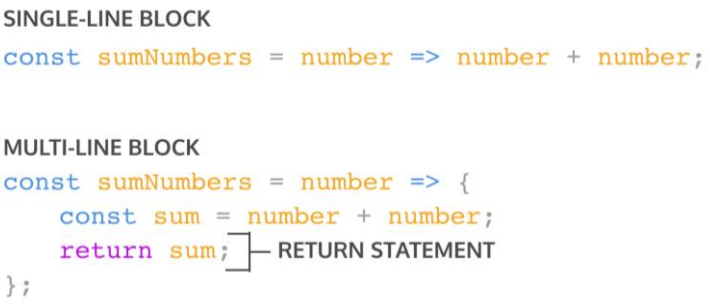


- A parameter is a named variable inside a function's block which will be assigned the value of the argument passed in when the function is invoked:
- To *call* a function in your code:
- ES6 introduces new ways of handling arbitrary parameters through *default parameters* which allow us assign a default value to a parameter in case no argument is passed into the function.

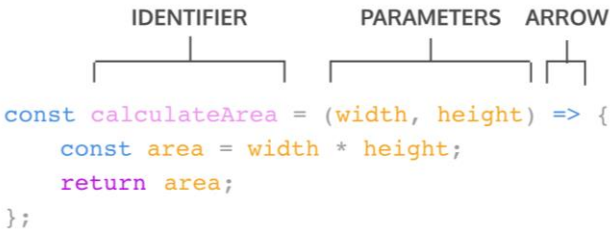


to

- To return a value from a function, we use a *return statement*.
- To define a function using *function expressions*:
- To define a function using *arrow function notation*:
- Function definition can be made concise using concise arrow notation



It's good to be aware of the differences between function expressions, arrow functions, and function declarations. As you program more in JavaScript, you'll see a wide variety of how these function types are used.



Practice project: Rock, Paper or Scissors

Rock paper scissors is a classic two player game. Each player chooses either rock, paper, or scissors. The items are compared, and whichever player chooses the more powerful item wins.

The possible outcomes are:

- Rock destroys scissors.
- Scissors cut paper.
- Paper covers rock.
- If there's a tie, then the game ends in a draw.

Our code will break the game into four parts:

1. Get the user's choice.
2. Get the computer's choice.
3. Compare the two choices and determine a winner.
4. Start the program and display the results.

Solution:

```
const getUserChoice =(userInput) => {
  userInput = userInput.toLowerCase();
if (userInput === "rock" || userInput === "paper" || userInput === "scisso
rs" || userInput === "bomb"){
  return userInput;
}else{
  console.log("Error, please choose: rock, paper or scissors.")
  }
}

function getComputerChoice(){
  const randomNumber = Math.floor(Math.random() * 3);
if (randomNumber === 0){
 return "rock";
}else if (randomNumber === 1){
  return "paper"
}else if (randomNumber === 2){
  return "scissors"
}
};

function determineWinner(userChoice, computerChoice){
```

```
  if (userChoice === computerChoice){
    return "The game was a tie!";
  }
  if (userChoice === "rock"){
    if (computerChoice === "paper"){
    return "The computer won!";
  }else{
    return "Congrats user won!";
    }
  }
  if (userChoice === "paper"){
    if (computerChoice === "scissors"){
    return "The computer won!";
  }else{
    return "Congrats user won!";
    }
}
   if (userChoice === "scissors"){
    if (computerChoice === "rock"){
    return "The computer won!";
  }else{
    return "Congrats user won!";
    }
   }
   if (userChoice === "bomb"){
     return "Congrats, bomb away! You won"
   }
};

function playGame() {
  const userChoice = getUserChoice("bomb");
  const computerChoice = getComputerChoice();
console.log(`You threw: ${userChoice}`)
console.log(`The computer threw:${computerChoice}`)
console.log(determineWinner(userChoice, computerChoice))
};
playGame()
```

Practice Project: Sleep Debt Calculator

In this project we'll calculate if you're getting enough sleep each week using a sleep debt calculator. The program will determine the actual and ideal hours of sleep for each night of the last week. Finally, it will calculate, in hours, how far you are from your weekly sleep goal.

Solution:

```
function getSleepHours(day){
    switch(day){
    case "Monday":
    return 15;
    break;
    case"Tuesday":
    return 8;
    break;
    case "Wednesday":
    return 8;
    break;
    case "Thursday":
    return 8;
    break;
    case "Friday":
    return 8;
    break;
    case "Saturday":
    return 7;
    break;
    case "Sunday":
    return 8;
    break;
    default:
    return "Error!"
    break;
  }
};

const getActualSleepHours = () =>
  getSleepHours("Monday")
  + getSleepHours("Tuesday")
  + getSleepHours("Wednesday")
```

```
  + getSleepHours("Thursday")
  + getSleepHours("Friday")
  + getSleepHours("Saturday")
  + getSleepHours("Sunday");

console.log(getSleepHours("Tuesday"));
console.log(getActualSleepHours())

const getIdealSleepHours = () =>{
  const idealHours = 8;
  return idealHours * 7;
};

const calculateSleepDebt = () => {
  const actualSleepHours = getActualSleepHours();
  const idealSleepHours = getIdealSleepHours();

if (actualSleepHours === idealSleepHours){
  console.log(`The user got the perfect amount of sleep`);
}else if(actualSleepHours > idealSleepHours){
  console.log(`The user got ${actualSleepHours- idealSleepHours} hours mor
e sleep this week.`);
}else if(actualSleepHours < idealSleepHours){
  console.log(`The user should get some rest, ${ idealSleepHours - actualS
leepHours} hours less sleep this week.`);
}else{
  console.log("Error!");
}
}
calculateSleepDebt()


// function getSleepHours(day){
//   if (day === "Monday"){
//     return 8;
//   }else if (day === "Tuesday"){
//     return 7;
//   }else if (day === "Wednesday"){
//     return 6;
```

```
//    }else if (day === "Thursday"){
//      return 5;
//    }else if (day === "Friday"){
//      return 8;
//    }else if (day === "Saturday"){
//      return 8;
//    }else if (day === "Sunday"){
//      return 7;
//    }
// }
```

5.0 Scope

An important idea in programming is *scope*. Scope defines where variables can be accessed or referenced. While some variables can be accessed from anywhere within a program, other variables may only be available in a specific context.

You can think of scope like the view of the night sky from your window. Everyone who lives on the planet Earth is in the global scope of the stars. The stars are accessible *globally*. Meanwhile, if you live in a city, you may see the city skyline or the river. The skyline and river are only accessible *locally* in your city, but you can still see the stars that are available globally.

5.1 Blocks and Scope

Before we talk more about scope, we first need to talk about *blocks*.

We've seen blocks used before in functions and `if` statements. A block is the code found inside a set of curly braces `{}`. Blocks help us group one or more statements together and serve as an important structural marker for our code.

A block of code could be a function, like this:

```
const logSkyColor = () => {
  let color = 'blue';
  console.log(color); // blue
}
```

Notice that the function body is actually a block of code. Observe the block in an `if` statement:

```
if (dusk) {
  let color = 'pink';
  console.log(color); // pink
}
```

Solution:

```
const city = "New York City";

const logCitySkyline = function(){
  let skyscraper = "Empire State Building";
  return `The stars over the ${skyscraper} in ${city}`;
};
console.log(logCitySkyline())
```

5.2 Global Scope

Scope is the context in which our variables are declared. We think about scope in relation to blocks because variables can exist either outside of or within these blocks.

In *global scope*, variables are declared outside of blocks. These variables are called *global variables*. Because global variables are not bound inside a block, they can be accessed by any code in the program, including code in blocks. Let's take a look at an example of global scope:

```
const color = 'blue';

const returnSkyColor = () => {
  return color; // blue
};

console.log(returnSkyColor()); // blue
```

- Even though the `color` variable is defined outside of the block, it can be accessed in the function block, giving it global scope.
- In turn, `color` can be accessed within the `returnSkyColor` function block.

Solution:

```
const satellite = "The Moon";
const galaxy = "The Milky Way";
const stars = "North Star";

const callMyNightSky = function(){
  return `Night Sky: ${satellite}, ${stars}, and ${galaxy}`;
}
console.log(callMyNightSky());
```

5.3 Block Scope

The next context we'll cover is *block scope*. When a variable is defined inside a block, it is only accessible to the code within the curly braces `{}`. We say that variable has *block scope* because it is *only* accessible to the lines of code within that block.

Variables that are declared with block scope are known as *local variables* because they are only available to the code that is part of the same block.

Block scope works like this:

```
const logSkyColor = () => {
  let color = 'blue';
  console.log(color); // blue
};

logSkyColor(); // blue
console.log(color); // ReferenceError
```

You'll notice:

- We define a function `logSkyColor()`.
- Within the function, the `color` variable is only available within the curly braces of the function.
- If we try to log the same variable outside the function, it throws a `ReferenceError`.

Solution:

```
const logVisibleLightWaves = function(){
  const lightWaves = "Moonlight";
  console.log(lightWaves);
}
logVisibleLightWaves();
console.log(lightWaves);
```

5.4 Scope Pollution

It may seem like a great idea to always make your variables accessible, but having too many global variables can cause problems in a program.

When you declare global variables, they go to the *global namespace*. The global namespace allows the variables to be accessible from anywhere in the program. These variables remain there until the program finishes which means our global namespace can fill up really quickly.

Scope pollution is when we have too many global variables that exist in the global namespace, or when we reuse variables across different scopes. Scope pollution makes it difficult to keep track of our different variables and sets us up for potential accidents.

For example, globally scoped variables can collide with other variables that are more locally scoped, causing unexpected behavior in our code.

Let's look at an example of scope pollution in practice so we know how to avoid it:

```
let num = 50;

const logNum = () => {
  num = 100; // Take note of this line of code
  console.log(num);
};

logNum(); // Prints 100
console.log(num); // Prints 100
```

You'll notice:

- We have a variable `num`.
- Inside the function body of `logNum()`, we want to declare a new variable but forgot to use the `let` keyword.
- When we call `logNum()`, `num` gets reassigned to `100`.
- The reassignment inside `logNum()` affects the global variable `num`.
- Even though the reassignment is allowed and we won't get an error, if we decided to use `num` later, we'll unknowingly use the new value of `num`.

While it's important to know what global scope is, it's best practice to not define variables in the global scope.

Solution:

```
const satellite = 'The Moon';
const galaxy = 'The Milky Way';
let stars = 'North Star';

const callMyNightSky = () => {
  stars = "Sirius"
  return 'Night Sky: ' + satellite + ', ' + stars + ', ' + galaxy;
};

console.log(callMyNightSky());
console.log(stars)
```

5.5 Practice Good Scoping

Given the challenges with global variables and scope pollution, we should follow best practices for scoping our variables as tightly as possible using block scope.

Tightly scoping your variables will greatly improve your code in several ways:

- It will make your code more legible since the blocks will organize your code into discrete sections.
- It makes your code more understandable since it clarifies which variables are associated with different parts of the program rather than having to keep track of them line after line!
- It's easier to maintain your code, since your code will be modular.
- It will save memory in your code because it will cease to exist after the block finishes running.

Here's another example of how to use block scope, as defined within an `if` block:

```
const logSkyColor = () => {
  const dusk = true;
  let color = 'blue';
  if (dusk) {
    let color = 'pink';
    console.log(color); // pink
  }
  console.log(color); // blue
};

console.log(color); // ReferenceError
```

Here, you'll notice:

- We create a variable `dusk` inside the `logSkyColor()` function.
- After the `if` statement, we define a new code block with the `{}` braces. Here we assign a new value to the variable `color` if the `if` statement is truthy.
- Within the `if` block, the `color` variable holds the value `'pink'`, though outside the `if` block, in the function body, the `color` variable holds the value `'blue'`.
- While we use block scope, we still pollute our namespace by reusing the same variable name twice. A better practice would be to rename the variable inside the block.

Block scope is a powerful tool in JavaScript, since it allows us to define variables with precision, and not pollute the global namespace. If a variable does not need to exist outside a block— it shouldn't!

Solution:

```
const logVisibleLightWaves = () => {
  let lightWaves = 'Moonlight';
  let region = 'The Arctic';
  // Add if statement here:
  if (region === "The Arctic"){
  let lightWaves = "Northern Lights";
  console.log(lightWaves)
  }
  console.log(lightWaves);
};

logVisibleLightWaves();
```

Scope: Review

In this lesson, you learned about scope and how it impacts the accessibility of different variables.

Let's review the following terms:

- **Scope** is the idea in programming that some variables are accessible/inaccessible from other parts of the program.
- **Blocks** are statements that exist within curly braces `{}`.
- **Global scope** refers to the context within which variables are accessible to every part of the program.
- **Global variables** are variables that exist within global scope.
- **Block scope** refers to the context within which variables are accessible only within the block they are defined.
- **Local variables** are variables that exist within block scope.
- **Global namespace** is the space in our code that contains globally scoped information.
- **Scope pollution** is when too many variables exist in a namespace or variable names are reused.

As you continue your coding journey, remember to use best practices when declaring your variables! Scoping your variables tightly will ensure that your code has clean, organized, and modular logic.

Practice Project: Training Days

As a seasoned athlete, one of your favorite activities is running marathons. You use a service called Training Days that sends you a message for the event you signed up for and the days you have left to train.

Since you also code, Training Days has asked you to help them solve a problem: The program currently uses the wrong scope for its variables. They know this can be troublesome as their service evolves. In this project you will make Training Days more maintainable and less error-prone by fixing variable scopes.

Solution:

```
// The scope of `random` is too loose

const getRandEvent = () => {
  const random = Math.floor(Math.random() * 3);
  if (random === 0) {
    return 'Marathon';
  } else if (random === 1) {
    return 'Triathlon';
  } else if (random === 2) {
    return 'Pentathlon';
  }
};

// The scope of `days` is too tight
const getTrainingDays = event => {
  let days;
  if (event === 'Marathon') {
     days = 50;
  } else if (event === 'Triathlon') {
     days = 100;
  } else if (event === 'Pentathlon') {
     days = 200;
  }

  return days;
};

const name = 'Nala';
// The scope of `name` is too tight
```

```
const logEvent = (name, event) => {
  console.log(`${name}'s event is: ${event}`);
};

const logTime = (name, days) => {
  console.log(`${name}'s time to train is: ${days} days`);
};

const event = getRandEvent();
const days = getTrainingDays(event);
// Define a `name` variable. Use it as an argument after updating logEvent
 and logTime


logEvent(name, event);
logTime(name, days);

const event2 = getRandEvent();
const days2 = getTrainingDays(event2);
const name2 = 'Warren';

logEvent(name2, event2);
logTime(name2, days2);
```

6.0 Challenge Project: Number Guesser

I have learned also about Math.abs(), which is function that returns absolute value of a number. No matter if it is positive or negative.

```
let humanScore = 0;
let computerScore = 0;
let currentRoundNumber = 1;

const generateTarget = function(secretNum){
  return Math.floor(Math.random() * 10);
}
const compareGuesses = function(human, computer, secretNum){

  const humanWin = Math.abs(human - secretNum);
  const computerWin = Math.abs(computer - secretNum);
  return humanWin < computerDifferences;
}

const updateScore = function(winner){
  if (winner === "human"){
    humanScore++;
  }else if(winner === "computer"){
    computerScore++;
  }
}

const advanceRound = function(){
  currentRoundNumber++;
}

updateScore(“human”);
console.log(humanScore); // To confirm that this value increased by 1
updateScore(“computer”);
console.log(computerScore); // To confirm that this value increased by 1
```