

What is JavaScript?

JavaScript is a powerful, flexible, and fast programming language now being used for increasingly complex web development and beyond!

JavaScript powers the dynamic behavior on most websites.

Console

The console is a panel that displays important messages, like errors, for developers.

`console` is a object of the `.log()` method. When we write `console.log()` what we put inside the parentheses will get printed, or logged, to the console.

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

Comments

Comments can explain what the code is doing, leave instructions for developers using the code, or add any other useful annotations.

There are two types of code comments in JavaScript:

1. *A single line comment :*

```
// Prints 5 to the console  
console.log(5);
```

You can also use a single line comment to comment after a line of code:

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

2. *A multi-line comment :*

```
/*  
This is all commented  
console.log(10);  
None of this is going to run!  
console.log(99);  
*/
```

Use this middle of a line of code:

```
console.log(/*IGNORED!*/ 5); // Still just prints 5
```

Data Types

In JavaScript, there are seven fundamental data types:

1. *Number*: it includes any numbers or with decimals: 4, 8, 1516, 23.42.
2. *String*: Any grouping of characters on your keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: ' ... ' or double quotes " ... ".
3. *Boolean*: This data type only has two possible values—either `true` or `false` (without quotes).
4. *Null*: This data type represents the intentional absence of a value, and is represented by the keyword `null` (without quotes).
5. *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`.
6. *Symbol*: A newer feature to the language, symbols are unique identifiers, useful in more complex coding.
7. *Object*: Collections of related data.

The first 6 of those types are considered *primitive data types*.

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

Arithmetic Operators

An *operator* is a character that performs a task in our code. We perform mathematical calculations on numbers using Arithmetic Operator.

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

```
console.log(3.5+23); //26.6
console.log(2019-1969); // 50
console.log(65/240); // 0.270833333333
console.log(0.2708*100); // 27.08
console.log(11 % 3); // Prints 2
```

String Concatenation

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

This process of appending one string to another is called *concatenation*.

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

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.

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

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

the value saved to the `length` property is retrieved from the instance of the string, `'Hello'`.

Methods

Remember that methods are actions we can perform.

We *call*, or use, these methods by appending an instance with (the dot operator), the name of the method, and opening and closing parentheses: ie. `'example string'.methodName()`.

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

You can find a list of built-in string methods in the (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/prototype).

Built-in Objects

In addition to `console`, there are other ("https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects").

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
```

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

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

`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.

```
console.log(Math.floor(Math.random() * 100));  
console.log(Math.ceil(43.8));  
console.log(Number.isInteger(2017));
```

Variables

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.

Create a Variable: var

These 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
```

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*.

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

Create a Variable: let

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
```

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

Create a Variable: const

Just like with `var` and `let` you can store, declare a `const` variable and assign value in a `const` variable is same. 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`.

Mathematical Assignment Operators

In the example below, 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`.

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

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

Another way we could have reassigned `w` after performing some mathematical operation on it is to use built-in *mathematical assignment operators*.

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

// These console.log() statements below will help you check the values of the
// variables.
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);

// 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.

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);
```

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
```

String Concatenation with 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.'
```

String Interpolation

we can insert, or *interpolate*, variables into strings using *template literals*.

```
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 (`).
- Inside the template literal, you'll see a placeholder, `${myPet}`.
- 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.

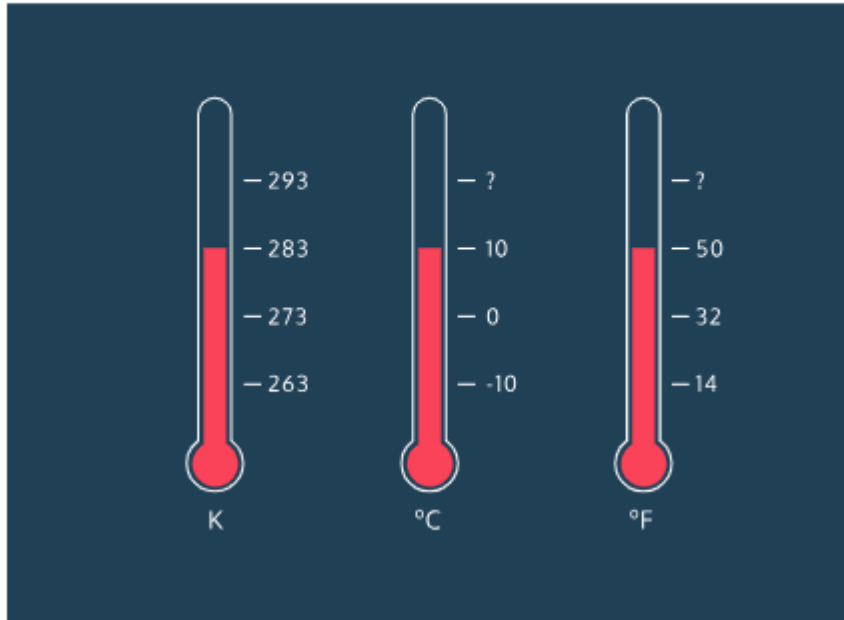
typeof operator

If you need to check the data type of a variable's value, you can use the `typeof` operator.

```
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
```

Project-1:

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



```
// forecast today is 283 Kelvin will stay constant.
const kelvin=283;
console.log(`kelvin :${kelvin}`);

//Celsius is similar to Kelvin – the only difference is that Celsius is 273
degrees less than Kelvin.
let celsius = kelvin-273;
console.log(`kelvin to celsius:${celsius}`);

//now,convert celsius to fahrenheit
let fahrenheit=celsius*(9/5)+32;
console.log("celsius to fahrenheit is :"+" "+fahrenheit);

//to get round of value use floor print in result variable
let result=Math.floor(fahrenheit);
console.log("round of fehrenheit value is:"+" "+result);

console.log(`The temperature is ${kelvin} degrees ${fahrenheit}`);
```

Project-2:

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.

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.

```
// human age
let myAge=10;

/*
first 2 years of dog age is increased by
10.5 years that means --
2year age human == 10.5+ 10.5=21
*/
let earlyYears=2;

/*
first 2 years of dog age is increased by
10.5 years that means --
2year age human == 10.5+ 10.5=21
*/
earlyYears *= 10.5;

// HUMAN AGE - EARLY YEARS AGE RATE THAT IS 2 YEARS
let laterYears=myAge-2;

//calculate the number of dog years accounted for by your later years.
laterYears *=4;
console.log(" laterYears: "+" "+laterYears);

// Add earlyYears and laterYears together
let myAgeInDogYears;
myAgeInDogYears=earlyYears+laterYears;
```

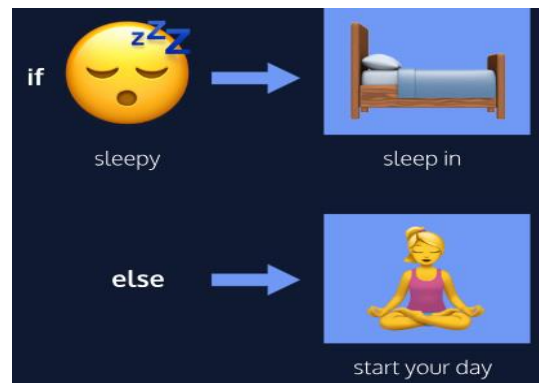
```
// change Capital letter to small letter
let myName='SHUBHAM MATHUR'.toLowerCase();

console.log(`My name is ${myName}. I am ${myAgeInDogYears} years old in dog
years.`);
```

What are Conditional Statements?

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

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



The if keyword

We often perform a task based on a condition. For example, if the weather is nice today, then we will go outside.

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!"
```

- 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.

If...Else Statements

In many cases, we'll have code we want to run if our condition evaluates to `false`.

```
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!"
```

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

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. Produce Boolean values.

```
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'.

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

Logical Operators

There are operators that work with **boolean values** known as *logical operators*.

- the *and* operator (`&&`)

```
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.

- the *or* 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.

- the *not* operator, otherwise known as the *bang operator* (!)

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 truevalue and pass back false, or it will take a falsevalue and pass back true.

```
let mood = 'sleepy';  
let tirednessLevel = 6;  
  
//ANOTHER WAY : WE CAN ADD ((CONDITION)>= (CONDITION))  
  
if((mood==='sleepy')!= (tirednessLevel>8))  
{  
  console.log('time to sleep');  
}  
else{  
  console.log('not bed time yet');  
}
```

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;

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!'
```

Example :

```
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.');
```

Truthy and Falsy Assignment

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 defaultName;
if (username) {
  defaultName = username;
} else {
  defaultName = '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 defaultName = username || '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*.

```
let tool = '';

tool = 'maker';

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

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

Ternary Operator

```
let isCorrect = true;
/*
if (isCorrect) {
  console.log('Correct!');
} else {
  console.log('Incorrect!');
}
*/
isCorrect ? console.log('Correct!') : console.log('Incorrect!');
let favoritePhrase = 'Love That!';
/*
if (favoritePhrase === 'Love That!') {
  console.log('I love that!');
} else {
  console.log("I don't love that!");
}
*/
favoritePhrase==='Love That!' ? console.log('I love that!') : console.log("I
don't love that!");
```

- 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.

Else If Statements

We can add more conditions to our `if...else` with an `else if` statement.

The `else if` statement always comes after the `if` statement and before the `else` statement.

```
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.');
```

...

```
}
```

The switch keyword

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

A `switch` statement looks like this:

```
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;
```

What are functions?

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*.

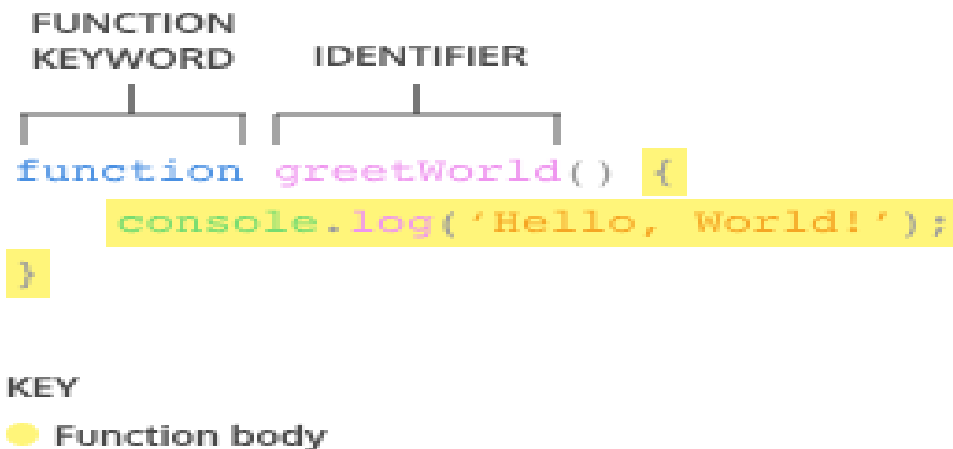
Imagine being asked to calculate the area of three different rectangles:

```
// 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;
```

Function Declarations



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, `{ }`.

```
console.log(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:

<https://developer.mozilla.org/en-US/docs/Glossary/Hoisting>

Calling a Function

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

IDENTIFIER



```
greetWorld();
```

+



We can call the same function as many times as needed.

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

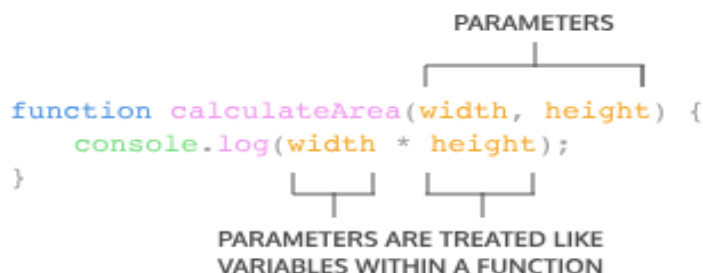


```
sayThanks();  
sayThanks();  
sayThanks();
```

Parameters and Arguments

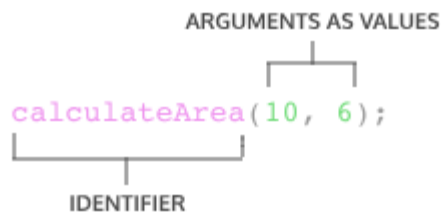
Some functions can take inputs and use the inputs to perform a task.

We use parameters as placeholders for information that will be passed to the function when it is called.

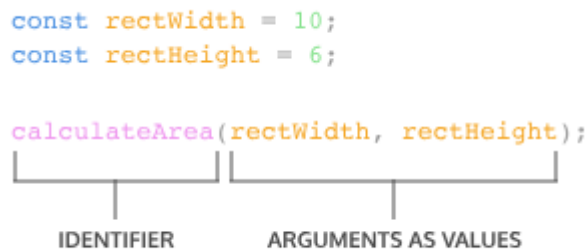


Inside the function body these parameter are act just like regular variables. `width` and `height` act as placeholders for values that will be multiplied together.

When calling a function that has parameters, we specify the values in the parentheses that follow the function name. The values or variable that are passed to the function when it is called are called *arguments*.



Notice that the order in which arguments are passed and assigned follows the order that the parameters are declared.



The variables `rectWidth` and `rectHeight` are initialized with the values for the height and width of a rectangle before being used in the function call.

```
function sayThanks(name) {  
  console.log('Thank you for your purchase ' + name + '! We appreciate your  
business.');
```

```
  }  
  
  sayThanks('Cole');
```

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.

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

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

Example :

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

makeShoppingList();
```

Return


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`!

```
function calculateArea(width, height) {
  const area = width * height;
  return area;
}
```



RETURN KEYWORD RETURN VALUE

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.

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

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*.

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)`.

Example:

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

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.

```
const calculateArea = function(width, height) {  
  const area = width * height;  
  return area;  
};
```

The diagram labels the parts of the function expression: 'calculateArea' is the IDENTIFIER, 'function' is the FUNCTION KEYWORD, and '(width, height)' are the PARAMETERS.

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)
```

Example:

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

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  
}
```

Example :

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

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*.

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.

ZERO PARAMETERS

```
const functionName = () => {};
```

ONE PARAMETER

```
const functionName = paramOne => {};
```

TWO OR MORE PARAMETERS

```
const functionName = (paramOne, paramTwo) => {};
```

1. 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*.

SINGLE-LINE BLOCK

```
const sumNumbers = number => number + number;
```

MULTI-LINE BLOCK

```
const sumNumbers = number => {  
  const sum = number + number;  
  return sum; } — RETURN STATEMENT  
};
```

So if we have a function:

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

We can refactor the function to:

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

Example:

```
const plantNeedsWater = day => day === 'Wednesday' ? true : false;  
  
console.log(plantNeedsWater('Wednesday'));
```