JavaScript

### Datatypes:

1. Number: **Number** data type includes any positive or negative integer, as well as decimals. Entering a number into the console will return it right back to you.

#### Strings : Strings are a collection of characters enclosed inside double or single quotes. You can use strings to represent data like sentences, names, addresses, and more

## Indexing

Did you know that you can access individual characters in a string? To access an individual character, you can use the character's location in the string, called its **index**. Just put the index of the character inside square brackets (starting with [0] as the first character) immediately after the string. For example:

"James"[0];

***Returns:****"J"*

or more commonly, you will see it like this, using a variable:

**var** name = "James";

name[0];

***Returns:****"J"*

## Escaping strings

There are some cases where you might want to create a string that contains more than just numbers and letters.

"The man whispered, \"please speak to me.\""

| **Code** | **Character** |
| --- | --- |
| \\ | \ (backslash) |
| \" | '' (double quote) |
| \' | ' (single quote) |
| \n | newline |
| \t | tab |

### Boolean: A boolean variable can take either of two values - true or false

* [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#number_type)
* [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#bigint_type)
* [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#string_type)
* [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#boolean_type)
* [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#symbol_type) (new in ES2015)
* [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects)
  + [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function)
  + [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)
  + [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)
  + [RegExp](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp)
* [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#null_type)
* [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#undefined_type)

**Comments:**

*// this is a single-line comment*

*/\**

*this is*

*a multi-line*

*comment*

*\*/*

## What is NaN, null and undefined?

NaN stands for "Not-A-Number" and it's often returned indicating an error with number operations. For instance, if you wrote some code that performed a math calculation, and the calculation failed to produce a valid number, NaN might be returned.

*// calculating the square root of a negative number will return NaN*

Math.sqrt(-10)

*// trying to divide a string by 5 will return NaN*

"hello"/5

null refers to the "value of nothing", while undefined refers to the "absence of value/assignment".

## Equality

So far, you’ve seen how you can use == and != to compare numbers and strings for equality. However, if you use == and != in situations where the values that you're comparing have different data-types, it can lead to some interesting results. For example,

"1" == 1

***Returns:****true*

and

0 == false

***Returns:****true. The == operator is unable to differentiate 0 from false.*

' ' == false

***Returns:****true. Both the operands on either side of the == operator are first converted to zero, before comparison.*

All of the above three evaluate to true. The reason for such interesting outcomes is **Type Conversion**. In the case of regular comparison, the operands on either side of the == operator are first converted to numbers, before comparison. Therefore, a ' ', false, and 0 are all considered equal. Similarly, a '1' and 1 are also considered equal. If we don't want to convert the operands, before comparison, we have to use a **strict comparison** ===, that is explained below.

## Implicit type coercion

JavaScript is known as a **loosely typed language**.

Basically, this means that when you’re writing JavaScript code, you do not need to specify data types. Instead, when your code is interpreted by the JavaScript engine it will automatically be converted into the "appropriate" data type. This is called implicit type coercion and you’ve already seen examples like this before when you tried to concatenate strings with numbers.

"julia" + 1

***Returns:****"julia1"*

In this example, JavaScript takes the string "julia" and adds the number 1 to it resulting in the string "julia1". In other programming languages, this code probably would have returned an error, but in JavaScript the number 1 is converted into the string "1" and then is concatenated to the string "julia".

It’s behavior like this which makes JavaScript unique from other programming languages, but it can lead to some quirky behavior when doing operations and comparisons on mixed data types.

## Strict equality

Instead, in JavaScript it’s better to use **strict equality** to see if numbers, strings, or booleans, etc. are identical in type and value without doing the type conversion first. To perform a strict comparison, simply add an additional equals sign = to the end of the == and != operators.

"1" === 1

***Returns:****false*

This returns false because the string "1" is not the same type and value as the number 1.

0 === false

***Returns:****false*

This returns false because the number 0 is not the same type and value as the boolean false. Just like strict equality operator, there is also a **strict non-equality** operator !== that can be used instead of != if you don't want a type-conversion, before comparison. For example,

0 !== true

***Returns:****true*

and

'1' !== 1

***Returns:****true*

## Logical operators

| **Operator** | **Meaning** | **Example** | **How it works** |
| --- | --- | --- | --- |
| && | Logical AND | value1 && value2 | Returns true if **both** value1 **and** value2 evaluate to true. |
| || | Logical OR | value1 || value2 | Returns true if **either** value1 **or** value2 (**or even both!**) evaluates to true. |
| ! | Logical NOT | !value1 | Returns the **opposite** of value1. If value1 is true, then !value1 is false. |

### falsy

### Here’s the list of all of the falsy values:

1. false, 0, empty strings (""), NaN, null, and undefined all become false.

### Truthy

Here are some other examples of truthy values:

true

42

"pizza"

"0"

"null"

"undefined"

{}

[]

## Switch statement

A **switch statement** is another way to chain multiple else if statements that are based on the same value **without using conditional statements**. Instead, you just switch which piece of code is executed based on a value. By adding a **break** to each case clause, you fix the issue of the switch statement ***falling-through*** to other case clauses

**switch** (option) {

**case** 1:

console.log("You selected option 1.");

**break**;

**case** 2:

console.log("You selected option 2.");

**break**;

**case** 3:

console.log("You selected option 3.");

**break**;

**case** 4:

console.log("You selected option 4.");

**break**;

**case** 5:

console.log("You selected option 5.");

**break**;

**default**:

console.log("You have not selected valid option.");

}

## While Loop

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times.

Three main pieces of information that any loop should have are:

1. **When to start:** The code that sets up the loop — defining the starting value of a variable for instance.
2. **When to stop:** The logical condition to test whether the loop should continue.
3. **How to get to the next item:** The incrementing or decrementing step — for example, x = x \* 3 or x = x - 1

Here's a basic while loop example that includes all three parts.

**var** start = 0; *// when to start*

**while** (start < 10) { *// when to stop*

console.log(start);

start = start + 2; *// how to get to the next item*

}

## For Loop

The for loop explicitly forces you to define the start point, stop point, and each step of the loop. In fact, you'll get an Uncaught SyntaxError: Unexpected token ) if you leave out any of the three required pieces.

**for** ( start; stop; step ) {

*// do this thing*

}

## var fruits = ["Banana", "Orange", "Apple", "Mango"];

## var fk = fruits.keys();

## for (x of fk) {

## console.log(fruits[x]);

## }

* **for/in**- loops through the properties of an object
* [**for/of**](https://www.w3schools.com/jsref/jsref_forof.asp) - loops through the values of an iterable object

## Functions

**Functions** package up code so you can easily use (and reuse) a block of code. **Parameters** are variables that are used to store the data that's passed into a function for the function to use. **Arguments** are the actual data that's passed into a function when it is invoked. If you don't explicitly define a return value, the function will return undefined by default.

## Scope:

1. Global: If an identifier is declared in **global scope**, it's available everywhere.

For instance, global variables can conflict with other global variables of the same name. Once your programs get larger and larger, it'll get harder and harder to keep track and prevent this from happening.

1. Function/local scope: If an identifier is declared in **function scope**, it's available *in the function* it was declared in and also even in functions declared inside the function.

When trying to access an identifier, the JavaScript Engine will first look in the current function. If it doesn't find anything, it will continue to the next outer function to see if it can find the identifier there. It will keep doing this until it reaches the global scope.

Global identifiers are a bad idea. They can lead to bad variable names, conflicting variable names, and messy code.

## ****Function expression****

JavaScript, you can also store functions in variables. When a function is stored inside a variable it's called a **function expression**.

**var** catSays = **function**(max) {

**var** catMessage = "";

**for** (**var** i = 0; i < max; i++) {

catMessage += "meow ";

}

**return** catMessage;

};

## ****Inline function expressions****

A function expression is when a function is assigned to a variable. And, in JavaScript, this can also happen when you pass a function inline as an argument to another function. Take the favoriteMovie example for instance:

*// Function expression that assigns the function displayFavorite*

*// to the variable favoriteMovie*

**var** favoriteMovie = **function** **displayFavorite**(movieName) {

console.log("My favorite movie is " + movieName);

};

*// Function declaration that has two parameters: a function for displaying*

*// a message, along with a name of a movie*

**function** **movies**(messageFunction, name) {

messageFunction(name);

}

*// Call the movies function, pass in the favoriteMovie function and name of movie*

movies(favoriteMovie, "Finding Nemo");

***Returns:****My favorite movie is Finding Nemo*

**Arrow functions**

An **arrow function expression** is a compact alternative to a traditional [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function), but is limited and can't be used in all situations.

**Differences & Limitations:**

* Does not have its own bindings to [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) or [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), and should not be used as [methods](https://developer.mozilla.org/en-US/docs/Glossary/Method) for an object.
* Does not have [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), or [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target) keywords.
* Not suitable for [call](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call), [apply](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply) and [bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) methods, which generally rely on establishing a [scope](https://developer.mozilla.org/en-US/docs/Glossary/Scope).
* Can not be used as [constructors](https://developer.mozilla.org/en-US/docs/Glossary/Constructor).
* Can not use [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield), within its body

**// Traditional Function**

**function arrowFunc (a){**

**return a + 100;**

**}**

**//function expression**

**const arrowFunc = function (a){**

**return a + 100;**

**}**

**// Arrow Function Break Down**

**// 1. Remove the word "function" and place arrow between the argument and opening body bracket**

**const arrowFunc=(a) => {**

**return a + 100;**

**}**

**// 2. Remove the body brackets and word "return" -- the return is implied.**

**const arrowFunc =(a) => a + 100;**

# Difference between different Variable declaration keywords:



Hoisted – Variables declared with ‘var’ keyword can be accessed before the declaration line. But the value will still be undefined.

* JavaScript hoists function declarations and variable declarations to the top of the current scope.
* Variable assignments are not hoisted.
* Declare functions and variables at the top of your scripts, so the syntax and behavior are consistent with each other.

let & const cannot be accessed before declaration.[**Temporal Dead Zone**]—will get reference error, for var we will get “undefined” as hoisiting of variable declaration takes before variable assignment.

## Arrays

An **array** is useful because it stores multiple values into a single, organized data structure. You can define a new array by listing values separated with commas between square brackets []

You can also store numbers, booleans… and really anything!

*// creates a `mixedData` array with mixed data types*

**var** mixedData = ["abcd", 1, true, undefined, null, "all the things"];

You can even store an array in an array to create a **nested array**!

*// creates a `arraysInArrays` array with three arrays*

**var** arraysInArrays = [[1, 2, 3], ["Julia", "James"], [true, false, true, false]];

**Array.length**

You can find the **length** of an array by using its length property.

**Push**

You can use the push() method to add elements to the *end of an array*. the push() method returns the length of the array after an element has been added.

**Pop**

Alternatively, you can use the pop() method to remove elements from the end of an array. pop() returns the element that has been removed in case you need to use it.

**Shift**

The shift() method removes the first item of an array. The return value of the shift method is the removed item.

**Unshift**

The unshift() method adds new items to the beginning of an array, and returns the new length.

**Sort**

The sort() method sorts the items of an array. The Array object, with the items sorted is returned.

The sort order can be either alphabetic or numeric, and either ascending (up) or descending (down).

By default, the sort() method sorts the values as strings in alphabetical and ascending order.

This works well for strings ("Apple" comes before "Banana"). However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

Sort numbers in an array in descending order:

var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b-a});

Sort numbers in an array in ascending order:

points.sort(function(a, b){return a-b});

**Reverse**

The reverse() method reverses the order of the elements in an array. An Array, representing the array after it has been reversed is returned.

**Every**

The every() method checks if all elements in an array pass a test (provided as a function).

var ages = [32, 33, 16, 40];  
  
function checkAdult(age) {  
  return age >= 18;  
}  
function myFunction() {  
  document.getElementById("demo").innerHTML = ages.every(checkAdult);  
}

If it finds an array element where the function returns a *false* value, every() returns false (and does not check the remaining values)

* If no false occur, every() returns true

**Includes:**

The includes() method determines whether an array contains a specified element.

This method returns true if the array contains the element, and false if not.

**Splice**

splice() lets you specify the index location to add new elements, as well as the number of elements you'd like to delete (if any).

**Following is the syntax of splice() method**: arrayName.splice(arg1, arg2, item1, ....., itemX); where,

* arg1 = Mandatory argument. Specifies the starting index position to add/remove items. You can use a negative value to specify the position from the end of the array e.g., -1 specifies the last element.
* arg2 = Optional argument. Specifies the count of elements to be removed. If set to 0, no items will be removed.
* item1, ....., itemX are the items to be added at index position arg1

splice() method returns the item(s) that were removed.

## forEach() loop

an alternative way to iterate over an array, and manipulate each element in the array with an inline function expression.

**var** donuts = ["jelly donut", "chocolate donut", "glazed donut"];

donuts.forEach(**function**(donut) {

donut += " hole";

donut = donut.toUpperCase();

console.log(donut);

});

The function that you pass to the forEach() method can take up to three parameters. In the video, these are called element, index, and array, but you can call them whatever you like.

The forEach() method will call this function once for each element in the array (hence the name forEach.) Each time, it will call the function with different arguments. The element parameter will get the value of the array element. The index parameter will get the index of the element (starting with zero). The array parameter will get a reference to the whole array, which is handy if you want to modify the elements.

Here's another example:

words = ["cat", "in", "hat"];

words.forEach(**function**(word, num, all) {

console.log("Word " + num + " in " + all.toString() + " is " + word);

});

***Prints:*** *Word 0 in cat,in,hat is cat  
Word 1 in cat,in,hat is in  
Word 2 in cat,in,hat is hat*

forEach() always returns undefined

## Map

With the map() [method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map), you can take an array, perform some operation on each element of the array, and return a new array.

**var** donuts = ["jelly donut", "chocolate donut", "glazed donut"];

**var** improvedDonuts = donuts.map(**function**(donut) {

donut += " hole";

donut = donut.toUpperCase();

**return** donut;

});

***donuts array:****["jelly donut", "chocolate donut", "glazed donut"]****improvedDonuts array:****["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED DONUT HOLE"]*

Oh man, did you just see that? The map() method accepts one argument, a function that will be used to manipulate each element in the array.

## Filter:

The filter() method creates an array filled with all array elements that pass a test (provided as a function).

var ages = [32, 33, 16, 40];  
  
function checkAdult(age) {  
  return age >= 18;  
}  
  
function myFunction() {  
  document.getElementById("demo").innerHTML = ages.filter(checkAdult);  
}//returns 32,33,40

# Object

Objects are one of the most important data structures in JavaScript. Get ready to see them everywhere!

They have properties (information about the object) and methods (functions or capabilities the object has). Objects are an incredibly powerful data type and you will see them all over the place when working with JavaScript, or any other object-oriented programming language.

**Object-literal notation**

The syntax you see below is called **object-literal notation**. There are some important things you need to remember when you're structuring an object literal:

* The "key" (representing a **property** or **method** name) and its "value" are separated from each other by a **colon**
* The key: value *pairs* are separated from each other by **commas**
* The entire object is wrapped inside curly braces { }.

**var** sister = {

name: "Sarah",

age: 23,

parents: [ "alice", "andy" ],

siblings: ["julia"],

favoriteColor: "purple",

pets: true

};

*// two equivalent ways to use the key to return its value*

sister["parents"] *// returns [ "alice", "andy" ]*

sister.parents *// also returns ["alice", "andy"]*

Using sister["parents"] is called **bracket notation** (because of the brackets!) and using sister.parents is called **dot notation** (because of the dot!).

# Object() constructor

The **Object constructor** creates an object wrapper for the given value.

* If the value is [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null) or [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined), it will create and return an empty object.
* Otherwise, it will return an object of a Type that corresponds to the given value.
* If the value is an object already, it will return the value.

When called in a non-constructor context, Object behaves identically to new Object().

 Object() constructor function is a bit slower and more verbose. As such, the recommended way to create new objects in JavaScript is to use literal notation.

## Add property to an Object:

One can add the property to an object by simply giving it a value. Like below example, we are adding the property *husband* and giving the value directly. We can use bracket too while assigning the value.

var brooklynNineNine = {

name: 'Raymond Holt',

currentRole: 'Captian of brooklyn99',

}

brooklynNineNine.husband = 'Kevin' // Or brooklynNineNine['husband'] = 'Kevin'

console.log(brooklynNineNine)

One can use ES7 Syntax and functional approach and add the property which would yield the same result.

var list =

{

name: 'Michael Scott',

company: 'Dunder Mufflin',

designation: 'Regional Manager',

show: 'The Office',

},

new\_obj = { ...list, partner: 'Holly Flax' }

console.table(new\_obj)

## Delete property from an Object:

One can delete the property from the object using keyword delete . The delete keyword deletes both the value of the property and the property itself. After deletion, the property cannot be used before it is added back again.

var brooklynNineNine = {

name: 'Amy Santiago',

currentRole: 'Detective brooklyn99',

husband: 'Jake Peralta',

mentor: 'Raymond Holt'

}

delete brooklynNineNine.mentor;

console.log(brooklynNineNine)

## Update the value of the existing property:

One can update the value of the property just by reassigning the value to the same key.

var favChar = {

name: 'Michael Scott',

company: 'Dunder Mufflin',

designation: 'Regional Manager',

show: 'The Office'

}

favChar.designation = 'Hero of Threat Level Midnight'

console.table(favChar)

## Add the properties to the array of Object:

Consider we have an array of objects and we want to add the property to the objects in the array. We can achieve this using many array methods(also for loop) . Here I have used the array method .forEach to iterate through the array element and add the property to the object.

const list = [

{

name: 'Michael Scott',

company: 'Dunder Mufflin',

designation: 'Regional Manager',

show: 'The Office'

},

{

name: 'Barney Stinson',

company: 'Golaith National Bank',

designation: 'Please',

show: 'How I met your mother'

},

{

name: 'Jake Peralta',

company: 'NYPD',

designation: 'Detective',

show: 'Brooklyn 99'

},

]

list.forEach(function (element) {

element.favCharacter = "true";

});

console.table(list)

## Delete the properties from the array of Object:

Here, the delete is done as similar to the addition. The iteration is done using array method .forEach and then the deletion is done using keyword delete.

const list = [

{

name: 'Michael Scott',

company: 'Dunder Mufflin',

designation: 'Regional Manager',

show: 'The Office'

},

{

name: 'Barney Stinson',

company: 'Golaith National Bank',

designation: 'Please',

show: 'How I met your mother'

},

{

name: 'Jake Peralta',

company y: 'NYPD',

designation: 'Detective',

show: 'Brooklyn 99'

},

]

list.forEach(function (element) {

delete element.designation;

});

console.table(list)

## Update every values of the existing property in the array of Objects:

Here the array method .every is used to iterate through the elements of the array. The property 'responsibility' is reassigned (‘heart of the show to ‘making people laugh’) to different value.

const list = [

{

name: 'Michael Scott',

company: 'Dunder Mufflin',

designation: 'Regional Manager',

show: 'The Office',

responsibility: 'heart of the show'

},

{

name: 'Barney Stinson',

company: 'Golaith National Bank',

designation: 'Please',

show: 'How I met your mother',

responsibility: 'heart of the show'

},

{

name: 'Jake Peralta',

company: 'NYPD',

designation: 'Detective',

show: 'Brooklyn 99',

responsibility: 'heart of the show'

},

]

list.every(element => element.responsibility = 'making people laugh') // every method is used to iterate through the array

### Accessing Methods:

**const** developer = {

name: 'Andrew',

sayHello: **function** () {

console.log('Hi there!');

}

};

developer.sayHello();

*// 'Hi there!'*

developer['sayHello']();

*// 'Hi there!'*

### Accessing Properties:

**const** developer = {

name: 'Andrew',

sayHello: **function** () {

console.log('Hi there!');

}

};

developer.name;

*// ‘Andrew’*

developer['name'];

*// 'Andrew'*

### Object Keys and Values:

**const** dictionary = {

car: 'automobile',

apple: 'healthy snack',

cat: 'cute furry animal',

dog: 'best friend'

};

Object.keys(dictionary);

*// ['car', 'apple', 'cat', 'dog']*

Object.values(dictionary);

*// ['automobile', 'healthy snack', 'cute furry animal', 'best friend']*

## Functions are First-Class Functions

In JavaScript, functions are first-class functions. This means that you can do with a function just about anything that you can do with other elements, such as numbers, strings, objects, arrays, etc. JavaScript functions can:

1. Be stored in variables
2. Be returned from a function.
3. Be passed as arguments into another function.

## Function scope

 A function's scope is the set of variables available for use within that function. The scope of a function includes:

1. The function's arguments.
2. Local variables declared within the function.
3. Variables from its parent function's scope.
4. Global variables.

Closure:

A **closure** refers to the combination of a function and the lexical environment in which that function was declared. Every time a function is defined, closure is created for that function. This is especially powerful in situations where a function is defined within another function, allowing the nested function to access variables outside of it. Functions also keep a link to its parent's scope even if the parent has returned. This prevents data in its parents from being garbage collected.

## ****Immediately-invoked function expressions**** (IIFE)

One of the primary uses for IIFE's is to create private scope

(**function** **sayHi**(){

alert('Hi there!');

}

)();

(**function** (name){

alert(`Hi, ${name}`);

}

)('Andrew');

**const** myFunction = (

**function** () {

**const** hi = 'Hi!';

**return** **function** () {

console.log(hi);

}

}

)();

Note that an IIFE is only intended to be invoked once, to create a unique execution context. If you have some code that is expected to be re-used (e.g., a function meant to be executed more than once in the application), declaring the function and then invoking it might be a better option.

And since fewer variables are created, an IIFE will help to minimize pollution of the global environment, hindering the chances of variable name collisions.

## Constructor function:

JavaScript's class system is built directly on using functions and objects. Calling (i.e., invoking) a **constructor function** with new operator instantiates a new object. The same constructor function can be used to create different objects.

***function******SoftwareDeveloper****(name) {*

***this****.favoriteLanguage = 'JavaScript';*

***this****.name = name;*

*}*

***let*** *coder = SoftwareDeveloper('David');*

## *instanceof and the Prototype Chain 💡*

*In the above example, instanceof confirmed that a specific constructor function did in fact create a specific object. We know this because we directly instantiated the dev object after invoking the Developer() constructor function.*

Functions, objects, and this are all interconnected. When invoking constructor functions with the new operator, a this variable is set to the newly-created object. When invoking a method on an object, this is set to that object itself. And when invoking a function in a browser environment, this is set to window, otherwise known as the global object.

JavaScript provides three methods that allow us to set the value of this for a given function:

* call() invokes the function and has arguments passed in individually, separated by commas.
* apply() is similar to call(); it invokes the function just the same, but arguments are passed in as an array.
* bind() returns a new function with this bound to a specific object, allowing us to call it as a regular function.

For further research, we recommend checking out Kyle Simpson's **You Don't Know JS** series on this, linked below.

//call and apply

const mockingbird = {

title: 'To Kill a Mockingbird',

describe: function (name) {

if(this.title != undefined)

console.log(`${this.title} is a classic novel of ${name}`);

}

};

const pride = {

title: 'Pride and Prejudice'

};

mockingbird.describe.call(pride,'Parani');

mockingbird.describe.apply(pride,['Kavitha']);

//bind

function invokeTwice(cb) {

cb();

cb();

}

const dog = {

age: 5,

growOneYear: function () {

this.age += 1;

}

};

invokeTwice(dog.growOneYear.bind(dog));

console.log(dog.age);

## Prototype Chain

Each function has a prototype property, which is really just an object. When this function is invoked as a constructor using the new operator, it creates and returns a new object. This object is secretly linked to its constructor's prototype, and this secret link allows the object to access the prototype's properties and methods as if it were its own!

Since we know that the prototype property just points to a regular object, that object itself also has a secret link to its prototype. And that prototype object also has reference to its own prototype -- and so on. This is how the **prototype chain** is formed.

## Finding Properties and Methods on the Prototype Chain

Whether you're accessing a property (e.g., bailey.lives;) or invoking a method (e.g., bailey.meow();), the JavaScript interpreter looks for them along the prototype chain in a very particular order:

1. First, the JavaScript engine will look at the object's own properties. This means that any properties and methods defined directly in the object itself will take precedence over any properties and methods elsewhere if their names are the same (similar to variable shadowing in the scope chain).
2. If it doesn't find the property in question, it will then search the object's constructor's prototype for a match.
3. If the property doesn't exist in the prototype, the JavaScript engine will continue looking up the chain.
4. At the very end of the chain is the Object() object, or the top-level parent. If the property still cannot be found, the property is undefined.

Previously, we simply defined methods directly in a constructor function itself. Let's see how things look if we defined methods in the constructor's prototype instead!

Replacing conventional Object methods creation from constructor function to Prototype

**function** **Dalmatian** (name) {

**this**.name = name;

}

Dalmatian.prototype.bark = **function**() {

console.log(`${this.name} barks!`);

};

function Car(make, model ){

this.make=make;

this.model = model;

}

Car.prototype.switchOn = function(){

console.log(‘Car turned on!’);

}

### Replacing the prototype Object 💡

What happens if you completely replace a function's prototype object? How does this affect objects created by that function?

### hasOwnProperty()

hasOwnProperty() allows you to find the origin of a particular property. Upon passing in a string of the property name you're looking for, the method will return a boolean indicating whether or not the property belongs to the object itself (i.e., that property was not inherited).

**unction** **Phone**() {

**this**.operatingSystem = 'Android';

}

Phone.prototype.screenSize = 6;

**const** myPhone = **new** Phone();

**const** own = myPhone.hasOwnProperty('operatingSystem');

console.log(own);

*// true*

**const** inherited = myPhone.hasOwnProperty('screenSize');

console.log(inherited);

*// false*

### isPrototypeOf()

Objects also have access to the isPrototypeOf() method, which checks whether or not an object exists in another object's prototype chain.

### Object.getPrototypeOf()

Object.getPrototypeOf() is great for retrieving the prototype of a given object

### What happens when car.drive(); is executed? List the following events in the order that they occur:

**function** **Car** (color, year) {

**this**.color = color;

**this**.year = year;

}

Car.prototype.drive = **function** () {

console.log('Vroom vroom!');

};

**const** car = **new** Car('silver', 1988);

* 1. First, the JavaScript engine searches inside the car object for a property named drive
  2. The JavaScript engine does not find drive within the car object
  3. The JavaScript engine then accesses the car.\_\_proto\_\_ property.
  4. Since the car.\_\_proto\_\_ property points to Car.prototype , the JavaScript engine searches for drive in the prototype
  5. Because Car.prototype.drive \_is\_ a defined property, it is returned.
  6. Finally, since drive is invoked as a method on car , the value of this is set to car

# Inheritance

Inheritance in JavaScript is all about setting up the prototype chain. This allows us to **subclass**, that is, create a "child" object that inherits most or all of a "parent" object's properties and methods. We can then implement any of the child object's unique properties and methods separately, while still retaining data and functionality from its parent.

An object (instance) is secretly linked to its constructor function's prototype object through that instance's \_\_proto\_\_ property.

To efficiently manage inheritance in JavaScript, an effective approach is to avoid mutating the prototype completely. Object.create() allows us to do just that, taking in a parent object and returning a new object with its \_\_proto\_\_ property set to that parent object.

## Object.create()

Object.create() takes in a single object as an argument, and returns a new object with its \_\_proto\_\_ property set to what argument is passed into it.

**const** mammal = {

vertebrate: true,

earBones: 3

};

**const** rabbit = Object.create(mammal);

//rabbit is child and mammal is parent

console.log(rabbit);

*// {}*

console.log(rabbit.\_\_proto\_\_ === mammal);

*// true*

Object.create() gives us a clean method of establishing prototypal inheritance in JavaScript.

## Mixin

A mixin is a technique that takes the properties and methods from one object and copies them over to another object. In other words: a mixin is an technique that provides some useful functionality, but is not meant to be added to the prototype chain.

## Object.assign()

The simplest way to implement the mixin pattern is to use Object.assign(). Object.assign() is a method that copies an object's own (non-inherited) properties from one or more source objects into a target object, then returns the updated target object. In other words, Object.assign() adds to the target object by merging in the source object(s). Consider the following:

**let** target = {};

**let** source = { number: 7 };

Object.assign(target, source);

console.log(target);

*// { number: 7 }*

**const** duck = {

hasBill: true

};

**const** beaver = {

hasTail: true

};

**const** otter = {

hasFur: true,

feet: 'webbed'

};

**const** platypus = Object.assign({}, duck, beaver, otter);

console.log(platypus);

*// { hasBill: true, hasTail: true, hasFur: true, feet: 'webbed' }*

platypus.constructor;

*// Object()*

platypus.isPrototypeOf(duck);

*// false*

duck.isPrototypeOf(platypus);

*// false*

platypus.isPrototypeOf(beaver);

*// false*

beaver.isPrototypeOf(duck);

*// false*

platypus.isPrototypeOf(otter);

*// false*

otter.isPrototypeOf(platypus);

*// false*

## The Module Pattern: Recap

Before moving on to the next section, let's make sure we're on the same page regarding the Module Pattern. We'll be improving upon it in the very next section! Consider the following:

**let** diana = (**function** () {

**let** secretIdentity = 'Diana Prince';

**return** {

introduce: **function** () {

console.log(`Hi! I am ${secretIdentity}`);

}

};

})();

Recall that one of the key ingredients here is the IIFE! Not only does it prevent pollution of the global scope (which hinders the chance of variable name collisions) -- the IIFE helps prevent access to the secretIdentity variable.

console.log(diana.secretIdentity);

*// undefined*

And because the returned object's introduce() method retains access to its parent function's scope, we are given a public interface to interact with secretIdentity:

diana.introduce();

*// 'Hi! I am Diana Prince'*

The Module Pattern is commonly used to create private properties in JavaScript, Modules are a larger unit of organization than, say, functions or objects. This helps partition code and provide structure as an application scales.

## The Revealing Module Pattern

The underlying philosophy of the Revealing Module Pattern is that, while we still maintain encapsulation (as in the Module Pattern), we also reveal certain properties (and methods). The key ingredients to the Revealing Module Pattern are:

1. An IIFE (wrapper)
2. The module content (variables, methods, objects, etc.)
3. A returned object literal

References:

* JS Design patterns - <https://addyosmani.com/resources/essentialjsdesignpatterns/book/#whatisapattern>
* Kyle Simpson's **You Don't Know JS** series
* Read online (free!): ["Up & Going"](https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/up%20&%20going/README.md#you-dont-know-js-up--going)
* Read online (free!): ["Scope & Closures"](https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/scope%20&%20closures/README.md#you-dont-know-js-scope--closures)
* Read online (free!): ["this & Object Prototypes"](https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/this%20&%20object%20prototypes/README.md#you-dont-know-js-this--object-prototypes)
* Read online (free!): ["Types & Grammar"](https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/types%20&%20grammar/README.md#you-dont-know-js-types--grammar)
* Read online (free!): ["Async & Performance"](https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/async%20&%20performance/README.md#you-dont-know-js-async--performance)
* Read online (free!): ["ES6 & Beyond"](https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/es6%20&%20beyond/README.md#you-dont-know-js-es6--beyond)