**Javascript concepts**

**Object.seal( ) In JavaScript**

**Object and Object Constructors in JavaScript?**  
In the living world of object-oriented programming we already know the importance of classes and objects but unlike other programming languages, JavaScript does not have the traditional classes as seen in other languages. But JavaScript has objects and constructors which work mostly in the same way to perform the same kind of operations.

* Constructors are general JavaScript functions which are used with the “new” keyword. Constructors are of two types in JavaScript i.e. built-in constructors(array and object) and custom constructors(define properties and methods for specific objects).
* Constructors can be useful when we need a way to create an object “type” that can be used multiple times without having to redefine the object every time and this could be achieved using the Object Constructor function**. It’s a convention to capitalize the name of constructors** to distinguish them from regular functions.

For instance, consider the following code:

filter\_none

brightness\_5

|  |
| --- |
| function Automobile(color) {    this.color=color;  }    var vehicle1 = new Automobile ("red"); |

The function “Automobile()” is an object constructor, and its properties and methods i.e “color” is declared inside it by prefixing it with the keyword “this”. Objects defined using an object constructor are then made instants using the keyword “new”.

When new Automobile() is called, JavaScript does two things:

1. It creates a fresh new object(instance) Automobile() and assigns it to a variable.
2. It sets the constructor property i.e “color” of the object to Automobile.

**Object.seal() Method**  
Among the Object constructor methods, there is a method Object.seal() which is used to seal an object. Sealing an object does not allow new properties to be added and marks all existing properties as non-configurable. Although values of present properties can be changed as long as they are writable.  
The object to be sealed is passed as an argument and the method returns the object which has been sealed.

**Difference between Object.freeze() Method and Object.seal() Method**

If an object is frozen using the Object.freeze() method then its **properties become immutable** and no changes can be made in them whereas if an object is sealed using the Object.seal() method then the **changes can be made**`in the existing properties of the object.

**Applications:**

* Object.seal() is used for sealing objects and arrays.
* Object.freeze() is used to make an object immutable.

**Syntax:**

Object.seal(obj)

***Parameters Used:***

1. *obj : It is the object which has to be sealed.*

***Return Value:*** *Object.sealed() returns the object that was passed to the function.*

**Decorators**[**#**](http://www.typescriptlang.org/docs/handbook/decorators.html#decorators)

A *Decorator* is a special kind of declaration that can be attached to a [class declaration](http://www.typescriptlang.org/docs/handbook/decorators.html#class-decorators), [method](http://www.typescriptlang.org/docs/handbook/decorators.html#method-decorators), [accessor](http://www.typescriptlang.org/docs/handbook/decorators.html#accessor-decorators), [property](http://www.typescriptlang.org/docs/handbook/decorators.html#property-decorators), or [parameter](http://www.typescriptlang.org/docs/handbook/decorators.html#parameter-decorators). Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

**Decorator Composition**[**#**](http://www.typescriptlang.org/docs/handbook/decorators.html#decorator-composition)

Multiple decorators can be applied to a declaration, as in the following examples:

* On a single line:
* @f @g x
* On multiple lines:
* @f
* @g

x

When multiple decorators apply to a single declaration, their evaluation is similar to [function composition in mathematics](http://en.wikipedia.org/wiki/Function_composition). In this model, when composing functions f and g, the resulting composite (f ∘ g)(x) is equivalent to f(g(x)).

As such, the following steps are performed when evaluating multiple decorators on a single declaration in TypeScript:

1. The expressions for each decorator are evaluated top-to-bottom.
2. The results are then called as functions from bottom-to-top.

If we were to use [decorator factories](http://www.typescriptlang.org/docs/handbook/decorators.html#decorator-factories), we can observe this evaluation order with the following example:

**function** **f**() {

console.log("f(): evaluated");

**return** **function** (target, propertyKey: string, descriptor: PropertyDescriptor) {

console.log("f(): called");

}

}

**function** **g**() {

console.log("g(): evaluated");

**return** **function** (target, propertyKey: string, descriptor: PropertyDescriptor) {

console.log("g(): called");

}

}

**class** C {

@f()

@g()

method() {}

}

Which would print this output to the console:

f(): evaluated

g(): evaluated

g(): called

f(): called

**Class Decorators**[**#**](http://www.typescriptlang.org/docs/handbook/decorators.html#class-decorators)

A *Class Decorator* is declared just before a class declaration. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition.

The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument.

If the class decorator returns a value, it will replace the class declaration with the provided constructor function.

NOTE  Should you choose to return a new constructor function, you must take care to maintain the original prototype. The logic that applies decorators at runtime will **not** do this for you.

The following is an example of a class decorator (@sealed) applied to the Greeter class:

@sealed

**class** Greeter {

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

greet() {

**return** "Hello, " + **this**.greeting;

}

}

We can define the @sealed decorator using the following function declaration:

**function** **sealed**(**constructor**: Function) {

Object.seal(**constructor**);

Object.seal(**constructor**.prototype);

}

Next we have an example of how to override the constructor.

**function** **classDecorator**<**T** **extends** {**new**(...args:any[]):{}}>(**constructor**:T) {

**return** **class** extends **constructor** {

newProperty = "new property";

hello = "override";

}

}

@classDecorator

**class** Greeter {

property = "property";

hello: string;

**constructor**(m: string) {

**this**.hello = m;

}

}

console.log(**new** Greeter("world"));

output:-

Greeter {

property: 'property',

hello: 'override',

newProperty: 'new property' }

## Method Decorators [#](http://www.typescriptlang.org/docs/handbook/decorators.html#method-decorators)

A Method Decorator is declared just before a method declaration. The decorator is applied to the Property Descriptor for the method, and can be used to observe, modify, or replace a method definition. A method decorator cannot be used in a declaration file, on an overload, or in any other ambient context (such as in a declare class).

The expression for the method decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The Property Descriptor for the member.

NOTE  The Property Descriptor will be undefined if your script target is less than ES5.

If the method decorator returns a value, it will be used as the Property Descriptor for the method.

NOTE  The return value is ignored if your script target is less than ES5.

The following is an example of a method decorator (@enumerable) applied to a method on the Greeter class:

**class** Greeter {

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

@enumerable(false)

greet() {

**return** "Hello, " + **this**.greeting;

}

}

We can define the @enumerable decorator using the following function declaration:

**function** **enumerable**(value: boolean) {

**return** **function** (target: any, propertyKey: string, descriptor: PropertyDescriptor) {

descriptor.enumerable = value;

};

}

The @enumerable(false) decorator here is a [decorator factory](http://www.typescriptlang.org/docs/handbook/decorators.html#decorator-factories). When the @enumerable(false) decorator is called, it modifies the enumerable property of the property descriptor.

Example:-

function enumerable(value: boolean) {

return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {

console.log(target,propertyKey,descriptor)

descriptor.enumerable = value;

};

}

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

@enumerable(false)

greet() {

return "Hello, " + this.greeting;

}

}

var kk=new Greeter("kk");

console.log(kk.greet())

output:-

Greeter {} 'greet' { value: [Function: greet],

writable: true,

enumerable: false,

configurable: true }

Hello, kk

# Polyfills

In web development, a polyfill is code that implements a feature on web browsers that do not support the feature.

**[Babel](https://javascript.info/polyfills" \l "babel)**

When we use modern features of the language, some engines may fail to support such code. Just as said, not all features are implemented everywhere.

Here Babel comes to the rescue.

[Babel](https://babeljs.io/) is a [transpiler](https://en.wikipedia.org/wiki/Source-to-source_compiler). It rewrites modern JavaScript code into the previous standard.

Actually, there are two parts in Babel:

1. First, the transpiler program, which rewrites the code. The developer runs it on their own computer. It rewrites the code into the older standard. And then the code is delivered to the website for users. Modern project build system like [webpack](http://webpack.github.io/) provide means to run transpiler automatically on every code change, so that very easy to integrate into development process.
2. Second, the polyfill.

New language features may include new built-in functions and syntax constructs. The transpiler rewrites the code, transforming syntax constructs into older ones. But as for new built-in functions, we need to implement them. JavaScript is a highly dynamic language, scripts may add/modify any functions, so that they behave according to the modern standard.

A script that updates/adds new functions is called “polyfill”. It “fills in” the gap and adds missing implementations.

Two interesting polyfills are:

* + [babel polyfill](https://babeljs.io/docs/usage/polyfill/) that supports a lot, but is big.
  + [polyfill.io](http://polyfill.io/) service that allows to load/construct polyfills on-demand, depending on the features we need.

So, if we’re going to use modern language features, a transpiler and a polyfill are necessary.

## **Examples**

sessionStorage is available in all the latest browsers (IE8 and upwards) but isn't in IE7 and below.

A polyfill can be used to [plug the support](http://gist.github.com/350433) for older browsers that don't provide sessionStorage.

Now with the polyfiller in place, as a developer I can rely on using the Web Storage API (for sessions) and not have to feature test in my code or fork to handle different situations.

# Mixins

In JavaScript we can only inherit from a single object. There can be only one [[Prototype]] for an object. And a class may extend only one other class.

But sometimes that feels limiting. For instance, I have a class StreetSweeper and a class Bicycle, and want to make a StreetSweepingBicycle.

Or, talking about programming, we have a class User and a class EventEmitter that implements event generation, and we’d like to add the functionality of EventEmitter to User, so that our users can emit events.

There’s a concept that can help here, called “mixins”.

As defined in Wikipedia, a [mixin](https://en.wikipedia.org/wiki/Mixin) is a class that contains methods for use by other classes without having to be the parent class of those other classes.

In other words, a mixin provides methods that implement a certain behavior, but we do not use it alone, we use it to add the behavior to other classes.

## [A mixin example](https://javascript.info/mixins" \l "a-mixin-example)

The simplest way to make a mixin in JavaScript is to make an object with useful methods, so that we can easily merge them into a prototype of any class.

For instance here the mixin sayHiMixin is used to add some “speech” for User:

// mixin

let sayHiMixin = {

sayHi() {

alert(`Hello ${this.name}`);

},

sayBye() {

alert(`Bye ${this.name}`);

}

};

// usage:

class User {

constructor(name) {

this.name = name;

}

}

// copy the methods

Object.assign(User.prototype, sayHiMixin);

// now User can say hi

new User("Dude").sayHi(); // Hello Dude!

There’s no inheritance, but a simple method copying. So User may inherit from another class and also include the mixin to “mix-in” the additional methods, like this:

class User extends Person {

// ...

}

Object.assign(User.prototype, sayHiMixin);

Mixins can make use of inheritance inside themselves.

For instance, here sayHiMixin inherits from sayMixin:

let sayMixin = {

say(phrase) {

alert(phrase);

}

};

let sayHiMixin = {

\_\_proto\_\_: sayMixin, // (or we could use Object.create to set the prototype here)

sayHi() {

// call parent method

super.say(`Hello ${this.name}`);

},

sayBye() {

super.say(`Bye ${this.name}`);

}

};

class User {

constructor(name) {

this.name = name;

}

}

// copy the methods

Object.assign(User.prototype, sayHiMixin);

// now User can say hi

new User("Dude").sayHi(); // Hello Dude!

Please note that the call to the parent method super.say() from sayHiMixin looks for the method in the prototype of that mixin, not the class.

## [EventMixin](https://javascript.info/mixins" \l "eventmixin)

Now let’s make a mixin for real life.

An important feature of many browser objects (not only) is that they can generate events. Events is a great way to “broadcast information” to anyone who wants it. So let’s make a mixin that allows to easily add event-related functions to any class/object.

* The mixin will provide a method .trigger(name, [...data]) to “generate an event” when something important happens to it. The name argument is a name of the event, optionally followed by additional arguments with event data.
* Also the method .on(name, handler) that adds handler function as the listener to events with the given name. It will be called when an event with the given name triggers, and get the arguments from .trigger call.
* …And the method .off(name, handler) that removes handler listener.

After adding the mixin, an object user will become able to generate an event "login" when the visitor logs in. And another object, say, calendar may want to listen to such events to load the calendar for the logged-in person.

Or, a menu can generate the event "select" when a menu item is selected, and other objects may assign handlers to react on that event. And so on.

Here’s the code:

let eventMixin = {

/\*\*

\* Subscribe to event, usage:

\* menu.on('select', function(item) { ... }

\*/

on(eventName, handler) {

if (!this.\_eventHandlers) this.\_eventHandlers = {};

if (!this.\_eventHandlers[eventName]) {

this.\_eventHandlers[eventName] = [];

}

this.\_eventHandlers[eventName].push(handler);

},

/\*\*

\* Cancel the subscription, usage:

\* menu.off('select', handler)

\*/

off(eventName, handler) {

let handlers = this.\_eventHandlers && this.\_eventHandlers[eventName];

if (!handlers) return;

for (let i = 0; i < handlers.length; i++) {

if (handlers[i] === handler) {

handlers.splice(i--, 1);

}

}

},

/\*\*

\* Generate an event with the given name and data

\* this.trigger('select', data1, data2);

\*/

trigger(eventName, ...args) {

if (!this.\_eventHandlers || !this.\_eventHandlers[eventName]) {

return; // no handlers for that event name

}

// call the handlers

this.\_eventHandlers[eventName].forEach(handler => handler.apply(this, args));

}

};

* .on(eventName, handler) – assigns function handler to run when the event with that name happens. Technically, there’s \_eventHandlers property, that stores an array of handlers for each event name. So it just adds it to the list.
* .off(eventName, handler) – removes the function from the handlers list.
* .trigger(eventName, ...args) – generates the event: all handlers from \_eventHandlers[eventName] are called, with a list of arguments ...args.

Usage:

// Make a class

class Menu {

choose(value) {

this.trigger("select", value);

}

}

// Add the mixin with event-related methods

Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// add a handler, to be called on selection:

menu.on("select", value => alert(`Value selected: ${value}`));

// triggers the event => the handler above runs and shows:

// Value selected: 123

menu.choose("123");

Now if we’d like any code to react on menu selection, we can listen to it with menu.on(...).

And eventMixin mixin makes it easy to add such behavior to as many classes as we’d like, without interfering with the inheritance chain.

We can use mixins as a way to augment a class by multiple behaviors, like event-handling as we have seen above.

Mixins may become a point of conflict if they occasionally overwrite existing class methods. So generally one should think well about the naming methods of a mixin, to minimize the probability of that.

# Async/await

There’s a special syntax to work with promises in a more comfortable fashion, called “async/await”. It’s surprisingly easy to understand and use.

## [Async functions](https://javascript.info/async-await" \l "async-functions)

Let’s start with the async keyword. It can be placed before a function, like this:

async function f() {

return 1;

}

The word “async” before a function means one simple thing: a function always returns a promise. Even If a function actually returns a non-promise value, prepending the function definition with the “async” keyword directs JavaScript to automatically wrap that value in a resolved promise.

For instance, the code above returns a resolved promise with the result of 1, let’s test it:

async function f() {

return 1;

}

f().then(alert); // 1

…We could explicitly return a promise, that would be the same as:

async function f() {

return Promise.resolve(1);

}

f().then(alert); // 1

So, async ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There’s another keyword, await, that works only inside async functions, and it’s pretty cool.

## [Await](https://javascript.info/async-await" \l "await)

The syntax:

// works only inside async functions

let value = await promise;

The keyword await makes JavaScript wait until that promise settles and returns its result.

Here’s an example with a promise that resolves in 1 second:

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("done!"), 1000)

});

let result = await promise; // wait till the promise resolves (\*)

alert(result); // "done!"

}

f();

The function execution “pauses” at the line (\*) and resumes when the promise settles, with result becoming its result. So the code above shows “done!” in one second.

Let’s emphasize: await literally makes JavaScript wait until the promise settles, and then go on with the result. That doesn’t cost any CPU resources, because the engine can do other jobs meanwhile: execute other scripts, handle events etc.

It’s just a more elegant syntax of getting the promise result than promise.then, easier to read and write.

**await won’t work in the top-level code**

People who are just starting to use await tend to forget the fact that we can’t use await in top-level code. For example, this will not work:

// syntax error in top-level code

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

We can wrap it into an anonymous async function, like this:

(async () => {

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

...

})();

**await accepts “thenables”**

Like promise.then, await allows to use thenable objects (those with a callable then method). The idea is that a 3rd-party object may not be a promise, but promise-compatible: if it supports .then, that’s enough to use with await.

Here’s a demo Thenable class, the await below accepts its instances:

class Thenable {

constructor(num) {

this.num = num;

}

then(resolve, reject) {

alert(resolve);

// resolve with this.num\*2 after 1000ms

setTimeout(() => resolve(this.num \* 2), 1000); // (\*)

}

};

async function f() {

// waits for 1 second, then result becomes 2

let result = await new Thenable(1);

alert(result);

}

f();

If await gets a non-promise object with .then, it calls that method providing native functions resolve, reject as arguments. Then await waits until one of them is called (in the example above it happens in the line (\*)) and then proceeds with the result.

# **JavaScript Object Accessors**

## **JavaScript Function or Getter?**

What is the differences between these two examples?

### **Example 1**

var person = {  
  firstName: "John",  
  lastName : "Doe",  
  fullName : function() {  
    return this.firstName + " " + this.lastName;  
  }  
};  
  
// Display data from the object using a method:  
document.getElementById("demo").innerHTML = person.fullName();

### **Example 2**

var person = {  
  firstName: "John",  
  lastName : "Doe",  
  get fullName() {  
    return this.firstName + " " + this.lastName;  
  }  
};  
  
// Display data from the object using a getter:  
document.getElementById("demo").innerHTML = person.fullName;

Example 1 access fullName as a function: person.fullName().

Example 2 access fullName as a property: person.fullName.

## **Object.defineProperty()**

The Object.defineProperty() method can also be used to add Getters and Setters:

### **Example**

// Define object  
var obj = {counter : 0};  
  
// Define setters  
Object.defineProperty(obj, "reset", {  
  get : function () {this.counter = 0;}  
});  
Object.defineProperty(obj, "increment", {  
  get : function () {this.counter++;}  
});  
Object.defineProperty(obj, "decrement", {  
  get : function () {this.counter--;}  
});  
Object.defineProperty(obj, "add", {  
  set : function (value) {this.counter += value;}  
});  
Object.defineProperty(obj, "subtract", {  
  set : function (value) {this.counter -= value;}  
});  
  
// Play with the counter:  
obj.reset;  
obj.add = 5;  
obj.subtract = 1;  
obj.increment;  
obj.decrement;

# **JavaScript Object Constructors**

### **Example**

function Person(first, last, age, eye) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.eyeColor = eye;  
}

It is considered good practice to name constructor functions with an upper-case first letter.

## **Object Types (Blueprints) (Classes)**

Sometimes we need a "**blueprint**" for creating many objects of the same "type".

The way to create an "object type", is to use an **object constructor function**.

In the example above, function Person() is an object constructor function.

Objects of the same type are created by calling the constructor function with the new keyword:

var myFather = new Person("John", "Doe", 50, "blue");  
var myMother = new Person("Sally", "Rally", 48, "green");

Note:- Math is a global object. The new keyword cannot be used on Math.

# **JavaScript Object Prototypes**

## **Prototype Inheritance**

All JavaScript objects inherit properties and methods from a prototype:

* Date objects inherit from Date.prototype
* Array objects inherit from Array.prototype
* Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

## **Adding Properties and Methods to Objects**

Sometimes you want to add new properties (or methods) to all existing objects of a given type.

Sometimes you want to add new properties (or methods) to an object constructor.

## **Using the prototype Property**

The JavaScript prototype property allows you to add new properties to object constructors:

### **Example**

function Person(first, last, age, eyecolor) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.eyeColor = eyecolor;  
}  
  
Person.prototype.nationality = "English";

The JavaScript prototype property also allows you to add new methods to objects constructors:

### **Example**

function Person(first, last, age, eyecolor) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.eyeColor = eyecolor;  
}  
  
Person.prototype.name = function() {  
  return this.firstName + " " + this.lastName;  
};

### **ES5 New Object Methods**

// Adding or changing an object property  
Object.defineProperty(object, property, descriptor)  
  
// Adding or changing many object properties  
Object.defineProperties(object, descriptors)  
  
// Accessing Properties  
Object.getOwnPropertyDescriptor(object, property)  
  
// Returns all properties as an array  
Object.getOwnPropertyNames(object)  
  
// Returns enumerable properties as an array  
Object.keys(object)  
  
// Accessing the prototype  
Object.getPrototypeOf(object)  
  
// Prevents adding properties to an object  
Object.preventExtensions(object)  
// Returns true if properties can be added to an object  
Object.isExtensible(object)  
  
// Prevents changes of object properties (not values)  
Object.seal(object)  
// Returns true if object is sealed  
Object.isSealed(object)  
  
// Prevents any changes to an object  
Object.freeze(object)  
// Returns true if object is frozen  
Object.isFrozen(object)

## **Changing a Property Value**

### **Syntax**

Object.defineProperty(object, property, {value : value})

Ex-

Object.defineProperty(person, "language", {value : "NO"});

## **Changing Meta Data**

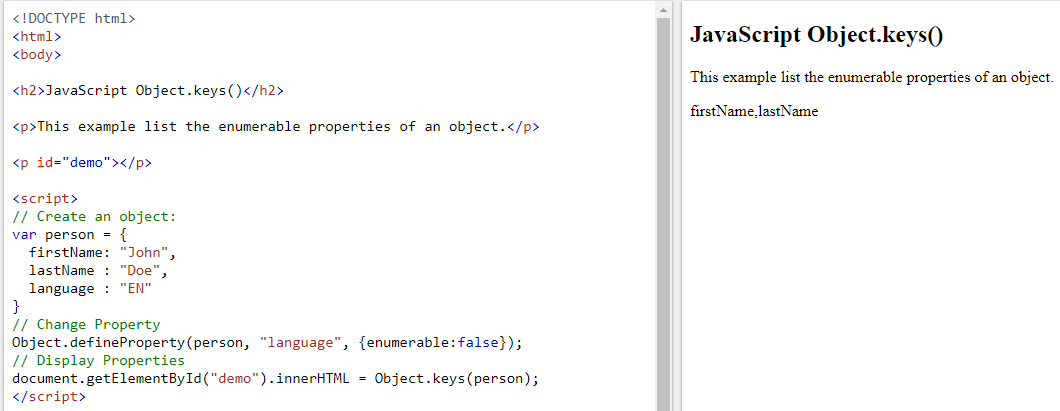
ES5 allows the following property meta data to be changed:

writable : true      // Property value can be changed  
enumerable : true    // Property can be enumerated  
configurable : true  // Property can be reconfigured

## **Listing All Properties**



## **Listing Enumerable Properties**



## **Adding Getters and Setters**

The Object.defineProperty() method can also be used to add Getters and Setters:

### **Example**

//Create an object  
var person = {firstName:"John", lastName:"Doe"};  
  
// Define a getter  
Object.defineProperty(person, "fullName", {  
  get : function () {return this.firstName + " " + this.lastName;}  
});

## Nesting Functions in Normal Functions

The first example of this anti-pattern is nesting a function inside a normal function. Here is an oversimplified example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function foo(a, b) {      function bar() {          return a + b;      }        return bar();  }    foo(1, 2); |

You may not write this exact code, but it's important to recognize the pattern. An outer function, foo(), contains an inner function, bar(), and calls that inner function to do work. Many developers forget that functions are values in JavaScript. When you declare a function in your code, the JavaScript engine creates a corresponding function object—a value that can be assigned to a variable or passed to another function. The act of creating a function object resembles that of any other type of value; the JavaScript engine doesn't create it until it needs to. So in the case of the above code, the JavaScript engine doesn't create the inner bar() function until foo() executes. When foo() exits, the bar() function object is destroyed.

The fact that foo() has a name implies it will be called multiple times throughout the application. While one execution of foo() would be considered OK, subsequent calls cause unnecessary work for the JavaScript engine because it has to recreate a bar() function object for every foo() execution. So, if you call foo() 100 times in an application, the JavaScript engine has to create and destroy 100 bar()function objects. Big deal, right? The engine has to create other local variables within a function every time it's called, so why care about functions?

*Unlike other types of values, functions typically don't change; a function is created to perform a specific task. So it doesn’t make much sense to waste CPU cycles recreating a somewhat static value over and over again.*

Ideally, the bar() function object in this example should only be created once, and that's easy to achieve—although naturally, more complex functions may require extensive refactoring. The idea is to move the bar() declaration outside of foo() so that the function object is only created once, like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function foo(a, b) {      return bar(a, b);  }    function bar(a, b) {      return a + b;  }    foo(1, 2); |

Notice that the new bar() function isn't exactly as it was inside of foo(). Because the old bar() function used the a and b parameters in foo(), the new version needed refactoring to accept those arguments in order to do its work.

Depending upon the browser, this optimized code is anywhere from 10% to 99% faster than the nested version. You can view and run the test for yourself at [jsperf.com/nested-named-functions](http://jsperf.com/nested-named-functions). Do keep in mind the simplicity of this example. A 10% (at the lowest end of the performance spectrum) performance gain doesn't seem like a lot, but it would be higher as more nested and complex functions are involved.

To perhaps confuse the issue, wrap this code in an anonymous, self-executing function, like this:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13 | (function() {    function foo(a, b) {      return bar(a, b);  }    function bar(a, b) {      return a + b;  }    foo(1, 2);    }()); |

Wrapping code in an anonymous function is a common pattern, and at first glance it might appear this code replicates the aforementioned performance issue by wrapping the optimized code in an anonymous function. While there is a slight performance hit by executing the anonymous function, this code is perfectly acceptable. The self-executing function serves only to contain and protect the foo() and bar()functions, but more importantly, the anonymous function executes only once—thus the inner foo() and bar() functions are created only once. However, there are some cases where anonymous functions are just as (or more so) problematic as named functions.

### Nesting Functions in Constructor Functions

Another variation of this anti-pattern is nesting functions within constructors, as shown below:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11 | function Person(firstName, lastName) {      this.firstName = firstName;      this.lastName = lastName;        this.getFullName = function() {          return this.firstName + " " + this.lastName;      };  }    var jeremy = new Person("Jeremy", "McPeak"),      jeffrey = new Person("Jeffrey", "Way"); |

This code defines a constructor function called Person(), and it represents (if it wasn't obvious) a person. It accepts arguments containing a person's first and last name and stores those values in firstName and lastName properties, respectively. The constructor also creates a method called getFullName(); it concatenates the firstName and lastName properties and returns the resulting string value.

*When you create any object in JavaScript, the object is stored in memory*

This pattern has become quite common in today's JavaScript community because it can emulate privacy, a feature that JavaScript isn't currently designed for (note that privacy isn’t in the above example; you’ll look at that later). But in using this pattern, developers create inefficiency not only in execution time, but in memory usage. When you create any object in JavaScript, the object is stored in memory. It stays in memory until all references to it are either set to null or are out of scope. In the case of the jeremyobject in the above code, the function assigned to getFullName is typically stored in memory for as long as the jeremy object is in memory. When the jeffrey object is created, a new function object is created and assigned to jeffrey's getFullName member, and it too consumes memory for as long as jeffrey is in memory. The problem here is that jeremy.getFullName is a different function object than jeffrey.getFullName (jeremy.getFullName === jeffrey.getFullName results in false; run this code at <http://jsfiddle.net/k9uRN/>). They both have the same behavior, but they are two completely different function objects (and thus each consume memory).

 Imagine creating 100 Person objects: if each getFullName() method consumes 4KB of memory, then 100 Person objects would consume at least 400KB of memory. That can add up, but it can be drastically reduced by using the prototype object.

### Use the Prototype

As mentioned earlier, functions are objects in JavaScript. All function objects have a prototype property, but it is only useful for constructor functions. In short, the prototype property is quite literally a prototype for creating objects; whatever is defined on a constructor function's prototype is shared among all objects created by that constructor function.

*Unfortunately, prototypes are not stressed enough in JavaScript education.*

Unfortunately, prototypes are not stressed enough in JavaScript education, yet they are absolutely essential to JavaScript because it’s based on and built with prototypes—it’s a prototypal language. Even if you never typed the word prototype in your code, they are being used behind the scenes. For example, every native string-based method, like split(), substr(), or replace(), are defined on String()'s prototype. Prototypes are so important to the JavaScript language that if you do not embrace JavaScript’s prototypal nature, you're writing inefficient code. Consider the above implementation of the Person data type: creating a Person object requires the JavaScript engine to do more work and allocate more memory.

So, how can using the prototype property make this code more efficient? Well, first take a look at the refactored code:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11 | function Person(firstName, lastName) {      this.firstName = firstName;      this.lastName = lastName;  }    Person.prototype.getFullName = function() {      return this.firstName + " " + this.lastName;  };    var jeremy = new Person("Jeremy", "McPeak"),      jeffrey = new Person("Jeffrey", "Way"); |

Here, the getFullName() method definition is moved out of the constructor and onto the prototype. This simple change has the following effects:

* The constructor performs less work, and thus, executes faster (18%-96% faster).
* The getFullName() method is created only once and shared among all Person objects (jeremy.getFullName === jeffrey.getFullName results in true; run this code at <http://jsfiddle.net/Pfkua/>). Because of this, each Person object uses less memory.

Note: The jeremy and jeffrey objects no longer have their own getFullName() method, but the JavaScript engine will find it on Person()'s prototype. In older JavaScript engines, the process of finding a method on the prototype could incur a performance hit, but not so in today's JavaScript engines. The speed at which modern engines find prototyped methods is extremely fast.

### Privacy

But what about privacy? After all, this anti-pattern was birthed out of a perceived need for private object members. If you’re not familiar with the pattern, take a look at the following code:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | function Foo(paramOne) {      var thisIsPrivate = paramOne;        this.bar = function() {          return thisIsPrivate;      };  }    var foo = new Foo("Hello, Privacy!");  alert(foo.bar()); // alerts "Hello, Privacy!" |

This code defines a constructor function called Foo(), and it has one parameter called paramOne. The value passed to Foo() is stored in a local variable called thisIsPrivate. Note that thisIsPrivate is a variable, not a property; so, it is inaccessible outside of Foo(). There's also a method defined inside the constructor, and it's called bar(). Because bar() is defined within Foo(), it has access to the thisIsPrivate variable. So when you create a Foo object and call bar(), the value assigned to thisIsPrivate is returned.

The value assigned to thisIsPrivate is preserved. It cannot be accessed outside of Foo(), and thus, it is protected from outside modification. That's great, right? Well, yes and no. It's understandable why some developers want to emulate privacy in JavaScript: you can ensure that an object's data is secured from outside tampering. But at the same time, you introduce inefficiency to your code by not using the prototype.

So again, what about privacy? Well that's simple: don't do it. The language currently does not officially support private object members—although that may change in a future revision of the language. Instead of using closures to create private members, the convention to denote "private members" is to prepend the identifier with an underscore (ie: \_thisIsPrivate). The following code rewrites the previous example using the convention:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | function Foo(paramOne) {      this.\_thisIsPrivate = paramOne;  }    Foo.prototype.bar = function() {      return this.\_thisIsPrivate;  };    var foo = new Foo("Hello, Convention to Denote Privacy!");  alert(foo.bar()); // alerts "Hello, Convention to Denote Privacy!" |

No, it's not private, but the underscore convention basically says "don't touch me." Until JavaScript fully supports private properties and methods,  wouldn't you rather have more efficient and performant code than privacy? The correct answer is: yes!

# **Understanding Classes in JavaScript**

JavaScript is a prototype-based language, and every object in JavaScript has a hidden internal property called [[Prototype]] that can be used to extend object properties and methods.

Classes in JavaScript do not actually offer additional functionality, and are often described as providing "syntactical sugar" over prototypes and inheritance in that they offer a cleaner and more elegant syntax.

## **Classes Are Functions**

A JavaScript class is a type of function. Classes are declared with the class keyword. We will use function expression syntax to initialize a function and class expression syntax to initialize a class.

// Initializing a function with a function expression

**const** x = **function**() {}

// Initializing a class with a class expression

**const** y = **class** {}

We can access the [[Prototype]] of an object using the [Object.getPrototypeOf() method](https://www.digitalocean.com/community/tutorials/understanding-prototypes-and-inheritance-in-javascript" \l "javascript-prototypes). Let's use that to test the empty **function** we created.

**Object**.getPrototypeOf(x);

Output

ƒ () { [native code] }

We can also use that method on the **class** we just created.

**Object**.getPrototypeOf(y);

Output

ƒ () { [native code] }

The code declared with function and class both return a function [[Prototype]]. With prototypes, any function can become a constructor instance using the new keyword.

**const** x = **function**() {}

// Initialize a constructor from a function

**const** constructorFromFunction = **new** x();

console.log(constructorFromFunction);

Output

x {}

constructor: ƒ ()

This applies to classes as well.

**const** y = **class** {}

// Initialize a constructor from a class

**const** constructorFromClass = **new** y();

console.log(constructorFromClass);

Output

y {}

constructor: class

These prototype constructor examples are otherwise empty,

## **Defining a Class**

A constructor function is initialized with a number of parameters, which would be assigned as properties of this, referring to the function itself. The first letter of the identifier would be capitalized by convention.

constructor.js

// Initializing a constructor function

**function** **Hero**(name, level) {

**this**.name = name;

**this**.level = level;

}

When we translate this to the class syntax, shown below, we see that it is structured very similarly.

class.js

// Initializing a class definition

**class** Hero {

constructor(name, level) {

**this**.name = name;

**this**.level = level;

}

}

The only difference in the syntax of the initialization is using the class keyword instead of function, and assigning the properties inside a constructor() method.

## **Defining Methods**

The common practice with constructor functions is to assign methods directly to the prototypeinstead of in the initialization, as seen in the greet() method below.

constructor.js

**function** **Hero**(name, level) {

**this**.name = name;

**this**.level = level;

}

// Adding a method to the constructor

Hero.prototype.greet = **function**() {

**return** `${**this**.name} says hello.`;

}

With classes this syntax is simplified, and the method can be added directly to the class. Using the [method definition shorthand](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions) introduced in ES6, defining a method is an even more concise process.

class.js

**class** Hero {

constructor(name, level) {

**this**.name = name;

**this**.level = level;

}

// Adding a method to the constructor

greet() {

**return** `${**this**.name} says hello.`;

}

}

**const** hero1 = **new** Hero('Varg', 1);

If we print out more information about our new object with console.log(hero1), we can see more details about what is happening with the class initialization.

Output

Hero {name: "Varg", level: 1}

\_\_proto\_\_:

▶ constructor: class Hero

▶ greet: ƒ greet()

We can see in the output that the constructor() and greet() functions were applied to the \_\_proto\_\_, or [[Prototype]] of hero1, and not directly as a method on the hero1 object. While this is clear when making constructor functions, it is not obvious while creating classes. Classes allow for a more simple and succinct syntax, but sacrifice some clarity in the process.

## **Extending a Class**

With ES6 classes, the [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) keyword is used in place of call to access the parent functions. We will use extends to refer to the parent class.

class.js

// Creating a new class from the parent

**class** Mage extends Hero {

constructor(name, level, spell) {

// Chain constructor with super

super(name, level);

// Add a new property

**this**.spell = spell;

}

}

Now we can create a new Mage instance in the same manner.

**const** hero2 = **new** Mage('Lejon', 2, 'Magic Missile');

We will print hero2 to the console and view the output.

Output

Mage {name: "Lejon", level: 2, spell: "Magic Missile"}

\_\_proto\_\_: Hero

▶ constructor: class Mage

the class construction the [[Prototype]] is linked to the parent, in this case Hero.

#### Arrow functions note:

#### Important quirks to be aware of when using Arrow functions

If you use the ‘new’ keyword with => functions it will throw an error. Arrow functions can’t be used as a constructor — normal functions support the ‘new’ via the property prototype and internal method [[Construct]]. Arrow functions don’t use neither, thus the new (() => {}) throws an error.

# Arrow functions

var elements = [

'Hydrogen',

'Helium',

'Lithium',

'Beryllium'

];

// In this case, because we only need the length property, we can use destructuring parameter:

// Notice that the `length` corresponds to the property we want to get whereas the

// obviously non-special `lengthFooBArX` is just the name of a variable which can be changed

// to any valid variable name you want

elements.map(({ length :lengthFooBArX }) => lengthFooBArX); // [8, 6, 7, 9]

// This destructuring parameter assignment can also be written as seen below. However, note that in

// this example we are not assigning `length` value to the made up property. Instead, the literal name

// itself of the variable `length` is used as the property we want to retrieve from the object.

elements.map(({ length }) => length); // [8, 6, 7, 9]

### **No separate this**[**Section**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#No_separate_this)

Before arrow functions, every new function defined its own [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) value based on how the function was called:

* A new object in the case of a constructor.
* undefined in [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) function calls.
* The base object if the function was called as an "object method".
* etc.

This proved to be less than ideal with an object-oriented style of programming.

function Person() {

// The Person() constructor defines `this` as an instance of itself.

this.age = 0;

setInterval(function growUp() {

// In non-strict mode, the growUp() function defines `this`

// as the global object (because it's where growUp() is executed.),

// which is different from the `this`

// defined by the Person() constructor.

this.age++;

}, 1000);

}

var p = new Person();

In ECMAScript 3/5, the this issue was fixable by assigning the value in this to a variable that could be closed over.

function Person() {

var that = this;

that.age = 0;

setInterval(function growUp() {

// The callback refers to the `that` variable of which

// the value is the expected object.

that.age++;

}, 1000);

}

An arrow function does not have its own this. The this value of the enclosing lexical scope is used; arrow functions follow the normal variable lookup rules. So while searching for this which is not present in current scope, an arrow function ends up finding the this from its enclosing scope.

Thus, in the following code, the this within the function that is passed to setInterval has the same value as the this in the lexically enclosing function:

function Person(){

this.age = 0;

setInterval(() => {

this.age++; // |this| properly refers to the Person object

}, 1000);

}

var p = new Person();

# **Window setInterval() Method**

### **Example**

Alert "Hello" every 3 seconds (3000 milliseconds):

setInterval(function(){ alert("Hello"); }, 3000);

## **Definition and Usage**

The setInterval() method calls a function or evaluates an expression at specified intervals (in milliseconds).

The setInterval() method will continue calling the function until [clearInterval()](https://www.w3schools.com/jsref/met_win_clearinterval.asp) is called, or the window is closed.

The ID value returned by setInterval() is used as the parameter for the clearInterval() method.

**Tip:** 1000 ms = 1 second.

**Tip:** To execute a function only once, after a specified number of milliseconds, use the [setTimeout()](https://www.w3schools.com/jsref/met_win_settimeout.asp) method.

## **Syntax**

setInterval(function, milliseconds, param1, param2, ...)

## **Parameter Values**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| param1, param2, ... | Optional. Additional parameters to pass to the function (Not supported in IE9 and earlier) |

### **Example**

Using clearInterval() to stop time in the previous example:

var myVar = setInterval(myTimer, 1000);  
  
function myTimer() {  
  var d = new Date();  
  var t = d.toLocaleTimeString();  
  document.getElementById("demo").innerHTML = t;  
}  
  
function myStopFunction() {  
  clearInterval(myVar);  
}

### **Example**

Using setInterval() and clearInterval() to create a dynamic progress bar:



# [**Is it valid to define functions in JSON results?**](https://stackoverflow.com/questions/2001449/is-it-valid-to-define-functions-in-json-results)

A short answer is **NO**...

But **wait**...

There is still ways to store your function, it's widely **not recommended** to that, but still possible:

We said, you can save a string... how about converting your function to a string then?

const data = {func: '()=>"a FUNC"'};

Then you can stringify data using JSON.stringify(data) and then using JSON.parse to parse it (if this step needed)...

And **eval** to execute a string function (before doing that, just let you know using eval widely not recommended):

eval(data.func)();

# Event.composed

The read-only **composed** property of the [Event](https://developer.mozilla.org/en-US/docs/Web/API/Event) interface returns a [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean) which indicates whether or not the event will propagate across the shadow DOM boundary into the standard DOM.

## Syntax[Section](https://developer.mozilla.org/en-US/docs/Web/API/Event/composed#Syntax)

var composed = Event.composed;

### **Value**[**Section**](https://developer.mozilla.org/en-US/docs/Web/API/Event/composed#Value)

A [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean) which is true if the event will cross from the shadow DOM into the standard DOM after reaching the shadow root (that is, the first node in the shadow DOM in which the event began to propagate). All UA-dispatched UI events are composed (click/touch/mouseover/copy/paste, etc.) — most other types of events are not composed and so will return false, for example synthetic events that have been created without their composed option being set to true.

Propagation only occurs if the [bubbles](https://developer.mozilla.org/en-US/docs/Web/API/Event/bubbles) property is also true. However, capturing only composed events are also handled at host as if they were in AT\_TARGET phase. You can determine the path the event will follow through the shadow root to the DOM root by calling [composedPath()](https://developer.mozilla.org/en-US/docs/Web/API/Event/composedPath" \o "The composedPath() method of the Event interface returns the event’s path which is an array of the objects on which listeners will be invoked. This does not include nodes in shadow trees if the shadow root was created with its ShadowRoot.mode closed.).

If this value is false, the shadow root will be the last node to be offered the event.

By default, custom events stop at shadow DOM boundaries. To make a custom event pass through shadow DOM boundaries, set the composed flag to true when you create the event

# **HTML | bubbles Event Property**

The **bubbles event** property is used for returning a Boolean value which indicates whether an event is a bubbling event or not. The event is triggered if an event handler is set for that object, if not so then the event bubble up to the object’s parent. The event bubble up until it reaches the document object or handled.  
 **What is even bubbling?** Consider a situation an element is present inside another element and both of them handle an event. When an event occurs, in bubbling, the innermost element handles the event first, then outer, and so on. Please refer [this](https://www.quirksmode.org/js/events_order.html) for details.

**Return Value:**The **bubbles event** property returns true if the event can bubble up through the DOM, else it returns false

## **Definition and Usage**

The bubbles event property returns a Boolean value that indicates whether or not an event is a bubbling event.

Event bubbling directs an event to its intended target, it works like this:

* A button is clicked and the event is directed to the button
* If an event handler is set for that object, the event is triggered
* If no event handler is set for that object, the event bubbles up (like a bubble in water) to the objects parent

The event bubbles up from parent to parent until it is handled, or until it reaches the document object.

**Call method**

## **The JavaScript call() Method**

The call() method is a predefined JavaScript method.

It can be used to invoke (call) a method with an owner object as an argument (parameter).

With call(), an object can use a method belonging to another object.

This example calls the **fullName** method of person, using it on **person1**:

### **Example**

var person = {  
  **fullName**: function() {  
    return this.firstName + " " + this.lastName;  
  }  
}  
var person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
var person2 = {  
  firstName:"Mary",  
  lastName: "Doe"  
}  
person.fullName.call(**person1**);  // Will return "John Doe"

**Apply method**

# **JavaScript | Function Apply**

The apply() method is used to write methods, which can be used on different objects. It is different from the function call() because it takes arguments as an array.

**Syntax:**

apply()

**Return Value:** It returns the method values of a given function.

Example:

<script>

        var student = {

            details: function() {

                return this.name + "<br>" + this.class;

            }

        }

        var stud1 = {

            name:"Dinesh",

            class: "11th",

        }

        var stud2 = {

            name:"Vaibhav",

            class: "11th",

        }

        var x = student.details.apply(stud2);

        document.getElementById("GFG").innerHTML = x;

    </script>

Example2:

<script>

        var student = {

            details: function(section, rollnum) {

                return this.name + "<br>" + this.class

                    + " " + section + "<br>" + rollnum;

            }

        }

        var stud1 = {

            name:"Dinesh",

            class: "11th",

        }

        var stud2 = {

            name:"Vaibhav",

            class: "11th",

        }

        var x = student.details.apply(stud2, ["A", "24"]);

        document.getElementById("GFG").innerHTML = x;

    </script>

**Throttle and Debounce**

For throttling, let’s consider that first use case, stopping click spamming. We have a button in our app that when clicked, makes an API call of some kind. Let’s say to enter a competition. With throttling we can restrict the amount the API would get hit. The user may be clicking 20 times a second but we only fire the handler once per second.

For debouncing, let’s consider the auto save feature. Auto save tries to save the state of the application every time the user makes an update or interacts. We can debounce the save until a user hasn’t made any updates or interacted for a set period of time. That way we don’t spam the save function and make unnecessary saves. This will help performance.

# Implementing throttle and debounce

There are various implementations of throttle and debounce. The majority will achieve the same goal. Their implementations revolve around the use of setTimeout.

## Debounce

Debounce is the simpler of the two to implement.

const debounce = (func, delay) => {  
 let inDebounce  
 return function() {  
 const context = this  
 const args = arguments  
 clearTimeout(inDebounce)  
 inDebounce = setTimeout(() => func.apply(context, args), delay)  
 }  
}

We are passing a function(func) and a delay(delay) into the debounce function. inDebounce is a variable that we use to track the delay period.

If we are invoking for the first time, our function will execute at the end of our delay. If we invoke and then invoke again before the end of our delay, the delay restarts. It’s much easier to understand by reading the code and playing with the demo.

## Throttle

Throttle can be a little taxing as its desired behaviour has different interpretations. Let’s start by limiting the rate at which we execute a function.

const throttle = (func, limit) => {  
 let inThrottle  
 return function() {  
 const args = arguments  
 const context = this  
 if (!inThrottle) {  
 func.apply(context, args)  
 inThrottle = true  
 setTimeout(() => inThrottle = false, limit)  
 }  
 }  
}

The first call to our function will execute and sets the limit period inThrottle. We can call our function during this period but it will not fire until the throttle period has passed. Once it has passed, the next invocation will fire and the process repeats.

But what about our last call? If it’s in the limit period it’s ignored and what if we don’t want that? For example, if we bound to mouse movement for a resize and missed the last call we’d never get the desired result. We need to catch this and execute it after the limit period (Thanks to [worldwar](https://medium.com/u/d867200d89a8?source=post_page-----b01cad5c8edf----------------------" \t "_blank) for questioning the previous implementation which didn’t always work 100% as expected).

const throttle = (func, limit) => {  
 let lastFunc  
 let lastRan  
 return function() {  
 const context = this  
 const args = arguments  
 if (!lastRan) {  
 func.apply(context, args)  
 lastRan = Date.now()  
 } else {  
 clearTimeout(lastFunc)  
 lastFunc = setTimeout(function() {  
 if ((Date.now() - lastRan) >= limit) {  
 func.apply(context, args)  
 lastRan = Date.now()  
 }  
 }, limit - (Date.now() - lastRan))  
 }  
 }  
}

This implementation ensures that we catch and execute that last invocation. We also invoke it at the correct time. We do this by creating a variable lastRan which is a timestamp of the last invocation. We can then use this to determine if the last invocation took place within the throttle limit. We can also use lastRan to determine whether the throttled function has ran at all. This makes the previous variable inThrottle redundant.

One way to think about this implementation of throttle is like a chaining debounce. Each time the debounce waiting period lessens.  
throttle has some interesting possibilities. For example, you could store all the ignored executions and run them all at the end in order.

AJAX Implementation

**XMLHttpRequest(XHR) Object**

It is a special JavaScript object, which helps in achieving the Asynchronous communication in your webpage.

Instead of reloading entire page, partial content can be loaded with respective element through this Object.

This Object is a predecessor which was introduced as an ActiveX Object called XMLHttp by Microsoft as a part of Internet Explorer 5.0 browser.

At the same time other browsers like Safari, Mozilla and all, started implementing XMLHttpRequest object which internally supports all the methods and properties of originally Microsoft ActiveX object, due to this enhancement automatically Microsoft also implemented XMLHttpRequest object.

It has capacity of transferring various data formats like plain text, xml, html , JSON over HTTP request.

Let us have a glance over XMLHttpRequest supporting properties and methods and its usage in short.

**Array.prototype.join()**

The **join()** method creates and returns a new string by concatenating all of the elements in an array (or an [array-like object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Working_with_array-like_objects)), separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

Example:-

const elements = ['Fire', 'Air', 'Water'];

console.log(elements.join());

// expected output: "Fire,Air,Water"

console.log(elements.join(''));

// expected output: "FireAirWater"

console.log(elements.join('\n'));

// expected output: "Fire-Air-Water"

## Syntax

arr.join([separator])

### **Parameters**

**separator**Optional

Specifies a string to separate each pair of adjacent elements of the array. The separator is converted to a string if necessary. If omitted, the array elements are separated with a comma (","). If separator is an empty string, all elements are joined without any characters in between them.

### **Return value**

A string with all array elements joined. If arr.length is 0, the empty string is returned.

## Description

The string conversions of all array elements are joined into one string.

If an element is undefined, null or an empty array [], it is converted to an empty string.

### **Joining an array-like object**

The following example joins array-like object ([arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments)), by calling [Function.prototype.call](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call" \o "The call() method calls a function with a given this value and arguments provided individually.) on Array.prototype.join.

function f(a, b, c) {

var s = Array.prototype.join.call(arguments);

console.log(s); // '1,a,true'

}

f(1, 'a', true);

//expected output: "1,a,true"

**Array.prototype.reduce()**

The **reduce()** method executes a **reducer** function (that you provide) on each element of the array, resulting in a single output value.

Example:

const array1 = [1, 2, 3, 4];

const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4

console.log(array1.reduce(reducer));

// expected output: 10

// 5 + 1 + 2 + 3 + 4

console.log(array1.reduce(reducer, 5));

// expected output: 15

The **reducer** function takes four arguments:

1. Accumulator (acc)
2. Current Value (cur)
3. Current Index (idx)
4. Source Array (src)

Your **reducer** function's returned value is assigned to the accumulator, whose value is remembered across each iteration throughout the array and ultimately becomes the final, single resulting value.

arr.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])

### **Parameters**

**callback**

A function to execute on each element in the array (except for the first, if no initialValueis supplied), taking four arguments:

**accumulator**

The accumulator accumulates the callback's return values. It is the accumulated value previously returned in the last invocation of the callback, or initialValue, if supplied (see below).

**currentValue**

The current element being processed in the array.

**index**Optional

The index of the current element being processed in the array. Starts from index 0 if an initialValue is provided. Otherwise, starts from index 1.

**array**Optional

The array reduce() was called upon.

**initialValue**Optional

A value to use as the first argument to the first call of the callback. If no initialValue is supplied, the first element in the array will be used and skipped. Calling reduce() on an empty array without an initialValue will throw a TypeError.

### **Return value**

The single value that results from the reduction.

## Description

The reduce() method executes the callback once for each assigned value present in the array, taking four arguments:

* accumulator
* currentValue
* currentIndex
* array

The first time the callback is called, accumulator and currentValue can be one of two values. If initialValue is provided in the call to reduce(), then accumulator will be equal to initialValue, and currentValue will be equal to the first value in the array. If no initialValue is provided, then accumulator will be equal to the first value in the array, and currentValue will be equal to the second.

**Note:** If initialValue is not provided, reduce() will execute the callback function starting at index 1, skipping the first index. If initialValue is provided, it will start at index 0.

If the array is empty and no initialValue is provided, [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError" \o "The TypeError object represents an error when an operation could not be performed, typically (but not exclusively) when a value is not of the expected type.) will be thrown. If the array only has one element (regardless of position) and no initialValue is provided, or if initialValue is provided but the array is empty, the solo value will be returned without calling*callback*.

It is usually safer to provide an initialValue because there are three possible outputs without initialValue, as shown in the following example.

var maxCallback = ( acc, cur ) => Math.max( acc.x, cur.x );

var maxCallback2 = ( max, cur ) => Math.max( max, cur );

// reduce() without initialValue

[ { x: 22 }, { x: 42 } ].reduce( maxCallback ); // 42

[ { x: 22 } ].reduce( maxCallback ); // { x: 22 }

[ ].reduce( maxCallback ); // TypeError

// map/reduce; better solution, also works for empty or larger arrays

[ { x: 22 }, { x: 42 } ].map( el => el.x )

.reduce( maxCallback2, -Infinity );

### **How reduce() works**

Suppose the following use of reduce() occurred:

[0, 1, 2, 3, 4].reduce(function(accumulator, currentValue, currentIndex, array) {

return accumulator + currentValue;

});

The callback would be invoked four times, with the arguments and return values in each call being as follows:

| **callback** | **accumulator** | **currentValue** | **currentIndex** | **array** | **return value** |
| --- | --- | --- | --- | --- | --- |
| **first call** | 0 | 1 | 1 | [0, 1, 2, 3, 4] | 1 |
| **second call** | 1 | 2 | 2 | [0, 1, 2, 3, 4] | 3 |
| **third call** | 3 | 3 | 3 | [0, 1, 2, 3, 4] | 6 |
| **fourth call** | 6 | 4 | 4 | [0, 1, 2, 3, 4] | 10 |

The value returned by reduce() would be that of the last callback invocation (10).

Example 1:

var initialValue = 0;

var sum = [{x: 1}, {x: 2}, {x: 3}].reduce(function (accumulator, currentValue) {

console.log(accumulator)

return accumulator + currentValue.x;

},initialValue)

console.log(sum)

Output:

> 0

> 1

> 3

> 6

Example 2:

var initialValue = 0;

var sum = [{x: 1}, {x: 2}, {x: 3}].reduce(function (accumulator, currentValue) {

console.log(accumulator)

return accumulator + currentValue.x;

})

console.log(sum)

output:

Object { x: 1 }

> "[object Object]2"

> "[object Object]23"

### **Counting instances of values in an object**

var names = ['Alice', 'Bob', 'Tiff', 'Bruce', 'Alice'];

var countedNames = names.reduce(function (allNames, name) {

if (name in allNames) {

allNames[name]++;

}

else {

allNames[name] = 1;

}

return allNames;

}, {});

// countedNames is:

// { 'Alice': 2, 'Bob': 1, 'Tiff': 1, 'Bruce': 1 }

# [**JavaScript array random index insertion and deletion**](https://stackoverflow.com/questions/4567002/javascript-array-random-index-insertion-and-deletion)

Will these numeric gaps be somehow allocated (and therefore take some memory) even when they do not have assigned values?

No. JavaScript arrays aren't really arrays at all (see below), and the unused indexes consume no memory.

When I delete myArray[456] from upper example, would items below this item be relocated?

If you're talking about array indexes, it depends on how you delete it: If you use the delete keyword, no. If you use the splice function or similar, yes. In terms of memory, no, other entries are not relocated (regardless), and any memory that was referenced by an entry that no longer exists (whether because of a delete or a splice or a pop or similar) becomes available to be reclaimed by the garbage collector. Linked lists have virtually no advantage in JavaScript over arrays or plain old objects, and you rarely see them. Adding to a JavaScript array (or object) is likely to be a near-constant-time operation (implementations will probably need to do hashing and possibly some traversal of B-tree structures or similar, but that's totally implementation-specific), as is deletion.

For what you're describing, as Zevon pointed out, you may not want an array at all. You really only need an array if you need a length property or one of the array functions that relies on it. Otherwise, you're better off with a plain old object:

var obj = {};

obj[123] = "foo";

obj[456] = "bar";

obj[789] = "baz";

That's perfectly valid JavaScript. The values you're using in brackets (123, etc.) are coerced to strings (whether you're dealing with an array or a plain object), and so the key is really "123", etc. (whether you're using an Array or an Object). You can even loop through them, with the for..in control structure ([details here](http://blog.niftysnippets.org/2010/11/myths-and-realities-of-forin.html)).

What do I mean by "...aren't really arrays at all"? Literally that. JavaScript objects are key->value maps, and JavaScript arrays are nothing more than objects that have keys and values and special handling for keys that are numeric strings, and a special length property. Although we conventionally write array "indexes" as numbers, like all property names they are strings — a[0] is converted to a["0"] (although implementations are free to optimize this as long as the behavior remains as per the spec)

Array objects give special treatment to a certain class of property names. A property name P (in the form of a String value) is an array index if and only if ToString(ToUint32(P)) is equal to P and ToUint32(P) is not equal to 2^32−1. A property whose property name is an array index is also called an element. Every Array object has a length property whose value is always a nonnegative integer less than 2^32. The value of the length property is numerically greater than the name of every property whose name is an array index; whenever a property of an Array object is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever a property is added whose name is an array index, the length property is changed, if necessary, to be one more than the numeric value of that array index; and whenever the length property is changed, every property whose name is an array index whose value is not smaller than the new length is automatically deleted. This constraint applies only to own properties of an Array object and is unaffected by length or array index properties that may be inherited from its prototypes.

**Working with objects**

 Objects are sometimes called associative arrays, since each property is associated with a string value that can be used to access it.

## Enumerate the properties of an object

Starting with [ECMAScript 5](https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_5_support_in_Mozilla), there are three native ways to list/traverse object properties:

* [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loops  
  This method traverses all enumerable properties of an object and its prototype chain
* [Object.keys(o)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys)  
  This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object o.
* [Object.getOwnPropertyNames(o)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames)  
  This method returns an array containing all own properties' names (enumerable or not) of an object o.

# [**Object.getOwnPropertyNames vs Object.keys**](https://stackoverflow.com/questions/22658488/object-getownpropertynames-vs-object-keys)

There is a little difference. Object.getOwnPropertyNames(a) returns all own properties of the object a. Object.keys(a) returns all enumerable own properties. It means that if you define your object properties without making some of them enumerable: false these two methods will give you the same result.

It's easy to test:

var a = {};

Object.defineProperties(a, {

one: {enumerable: true, value: 'one'},

two: {enumerable: false, value: 'two'},

});

Object.keys(a); // ["one"]

Object.getOwnPropertyNames(a); // ["one", "two"]

If you define a property without providing property attributes descriptor (meaning you don't use Object.defineProperties), for example:

a.test = 21;

then such property becomes an enumerable automatically and both methods produce the same array.

Another difference is in case of array Object.getOwnPropertyNames method will return an extra property that is length.

var x = ["a", "b", "c", "d"];

Object.keys(x); //[ '0', '1', '2', '3' ]

Object.getOwnPropertyNames(x); //[ '0', '1', '2', '3', 'length' ]

### **Using a constructor function**

Alternatively, you can create an object with these two steps:

1. Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.
2. Create an instance of the object with new.

### **Using the Object.create method**

Objects can also be created using the [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create" \o "The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.) method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function.

// Animal properties and method encapsulation

var Animal = {

type: 'Invertebrates', // Default value of properties

displayType: function() { // Method which will display type of Animal

console.log(this.type);

}

};

// Create new animal type called animal1

var animal1 = Object.create(Animal);

animal1.displayType(); // Output:Invertebrates

// Create new animal type called Fishes

var fish = Object.create(Animal);

fish.type = 'Fishes';

fish.displayType(); // Output:Fishes

## Defining properties for an object type

You can add a property to a previously defined object type by using the prototype property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a color property to all objects of type Car, and then assigns a value to the color property of the object car1.

Car.prototype.color = null;

car1.color = 'black';

## Defining getters and setters

A [getter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get) is a method that gets the value of a specific property. A [setter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set) is a method that sets the value of a specific property. You can define getters and setters on any predefined core object or user-defined object that supports the addition of new properties. The syntax for defining getters and setters uses the object literal syntax.

The following illustrates how getters and setters could work for a user-defined object o.

var o = {

a: 7,

get b() {

return this.a + 1;

},

set c(x) {

this.a = x / 2;

}

};

console.log(o.a); // 7

console.log(o.b); // 8

o.c = 50;

console.log(o.a); // 25

The o object's properties are:

* o.a — a number
* o.b — a getter that returns o.a plus 1
* o.c — a setter that sets the value of o.a to half of the value o.c is being set to

Please note that function names of getters and setters defined in an object literal using "[gs]et property()" (as opposed to \_\_define[GS]etter\_\_ ) are not the names of the getters themselves, even though the [gs]et propertyName(){ } syntax may mislead you to think otherwise. To name a function in a getter or setter using the "[gs]et property()" syntax, define an explicitly named function programmatically using [Object.defineProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty" \o "The static method Object.defineProperty() defines a new property directly on an object, or modifies an existing property on an object, and returns the object.) (or the [Object.prototype.\_\_defineGetter\_\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineGetter) legacy fallback).

The following code illustrates how getters and setters can extend the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) prototype to add a year property to all instances of the predefined Date class. It uses the Date class's existing getFullYear and setFullYear methods to support the year property's getter and setter.

These statements define a getter and setter for the year property:

var d = Date.prototype;

Object.defineProperty(d, 'year', {

get: function() { return this.getFullYear(); },

set: function(y) { this.setFullYear(y); }

});

In principle, getters and setters can be either

* defined using [object initializers](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Object_initializers), or
* added later to any object at any time using a getter or setter adding method.

When defining getters and setters using [object initializers](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Object_initializers) all you need to do is to prefix a getter method with get and a setter method with set. Of course, the getter method must not expect a parameter, while the setter method expects exactly one parameter (the new value to set). For instance:

var o = {

a: 7,

get b() { return this.a + 1; },

set c(x) { this.a = x / 2; }

};

Getters and setters can also be added to an object at any time after creation using the Object.defineProperties method. This method's first parameter is the object on which you want to define the getter or setter. The second parameter is an object whose property names are the getter or setter names, and whose property values are objects for defining the getter or setter functions. Here's an example that defines the same getter and setter used in the previous example:

var o = { a: 0 };

Object.defineProperties(o, {

'b': { get: function() { return this.a + 1; } },

'c': { set: function(x) { this.a = x / 2; } }

});

o.c = 10; // Runs the setter, which assigns 10 / 2 (5) to the 'a' property

console.log(o.b); // Runs the getter, which yields a + 1 or 6

# Details of the object model

JavaScript is an object-based language based on prototypes, rather than being class-based. Because of this different basis, it can be less apparent how JavaScript allows you to create hierarchies of objects and to have inheritance of properties and their values. This chapter attempts to clarify the situation.

## Class-based vs. prototype-based languages

Class-based object-oriented languages, such as Java and C++, are founded on the concept of two distinct entities: classes and instances.

* A class defines all of the properties that characterize a certain set of objects (considering methods and fields in Java, or members in C++, to be properties). A class is abstract rather than any particular member in a set of objects it describes. For example, the Employee class could represent the set of all employees.
* An instance, on the other hand, is the instantiation of a class; that is, one of its members. For example, Victoria could be an instance of the Employee class, representing a particular individual as an employee. An instance has exactly the same properties of its parent class (no more, no less).

A prototype-based language, such as JavaScript, does not make this distinction: it simply has objects. A prototype-based language has the notion of a prototypical object, an object used as a template from which to get the initial properties for a new object. Any object can specify its own properties, either when you create it or at run time. In addition, any object can be associated as the prototype for another object, allowing the second object to share the first object's properties.

### **Defining a class**

In class-based languages, you define a class in a separate class definition. In that definition you can specify special methods, called constructors, to create instances of the class. A constructor method can specify initial values for the instance's properties and perform other processing appropriate at creation time. You use the new operator in association with the constructor method to create class instances.

JavaScript follows a similar model, but does not have a class definition separate from the constructor. Instead, you define a constructor function to create objects with a particular initial set of properties and values. Any JavaScript function can be used as a constructor. You use the new operator with a constructor function to create a new object.

Note that ECMAScript 2015 introduces a [class declaration](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes):

JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

Subclasses and inheritance

JavaScript implements inheritance by allowing you to associate a prototypical object with any constructor function. So, you can create exactly the Employee — Manager example, but you use slightly different terminology. First you define the Employee constructor function, specifying the name and dept properties. Next, you define the Manager constructor function, calling the Employee constructor and specifying the reports property. Finally, you assign a new object derived from Employee.prototype as the prototype for the Manager constructor function. Then, when you create a new Manager, it inherits the name and dept properties from the Employee object.

### **Adding and removing properties**

In class-based languages, you typically create a class at compile time and then you instantiate instances of the class either at compile time or at run time. You cannot change the number or the type of properties of a class after you define the class. In JavaScript, however, at run time you can add or remove properties of any object. If you add a property to an object that is used as the prototype for a set of objects, the objects for which it is the prototype also get the new property.

### **Summary of differences**

The following table gives a short summary of some of these differences. The rest of this chapter describes the details of using JavaScript constructors and prototypes to create an object hierarchy and compares this to how you would do it in Java.

| **Comparison of class-based (Java) and prototype-based (JavaScript) object systems** | | |
| --- | --- | --- |
| **Category** | **Class-based (Java)** | **Prototype-based (JavaScript)** |
| **Class vs. Instance** | Class and instance are distinct entities. | All objects can inherit from another object. |
| **Definition** | Define a class with a class definition; instantiate a class with constructor methods. | Define and create a set of objects with constructor functions. |
| **Creation of new object** | Create a single object with the new operator. | Same. |
| **Construction of object hierarchy** | Construct an object hierarchy by using class definitions to define subclasses of existing classes. | Construct an object hierarchy by assigning an object as the prototype associated with a constructor function. |
| **Inheritance model** | Inherit properties by following the class chain. | Inherit properties by following the prototype chain. |
| **Extension of properties** | Class definition specifies all properties of all instances of a class. Cannot add properties dynamically at run time. | Constructor function or prototype specifies an initial set of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects. |

**Creating the hierarchy**

The following Java and JavaScript Employee definitions are similar. The only difference is that you need to specify the type for each property in Java but not in JavaScript (this is due to Java being a [strongly typed language](http://en.wikipedia.org/wiki/Strong_and_weak_typing) while JavaScript is a weakly typed language).

#### JavaScript

class Employee {

constructor() {

this.name = '';

this.dept = 'general';

}

}

#### Java

public class Employee {

public String name = "";

public String dept = "general";

}

The Manager and WorkerBee definitions show the difference in how to specify the next object higher in the inheritance chain. In JavaScript, you add a prototypical instance as the value of the prototype property of the constructor function, then override the prototype.constructor to the constructor function. You can do so at any time after you define the constructor. In Java, you specify the superclass within the class definition. You cannot change the superclass outside the class definition.

#### JavaScript

function Manager() {

Employee.call(this);

this.reports = [];

}

Manager.prototype = Object.create(Employee.prototype);

Manager.prototype.constructor = Manager;

function WorkerBee() {

Employee.call(this);

this.projects = [];

}

WorkerBee.prototype = Object.create(Employee.prototype);

WorkerBee.prototype.constructor = WorkerBee;

#### Java

public class Manager extends Employee {

public Employee[] reports =

new Employee[0];

}

public class WorkerBee extends Employee {

public String[] projects = new String[0];

}

The Engineer and SalesPerson definitions create objects that descend from WorkerBee and hence from Employee. An object of these types has properties of all the objects above it in the chain. In addition, these definitions override the inherited value of the dept property with new values specific to these objects.

#### JavaScript

function SalesPerson() {

WorkerBee.call(this);

this.dept = 'sales';

this.quota = 100;

}

SalesPerson.prototype = Object.create(WorkerBee.prototype);

SalesPerson.prototype.constructor = SalesPerson;

function Engineer() {

WorkerBee.call(this);

this.dept = 'engineering';

this.machine = '';

}

Engineer.prototype = Object.create(WorkerBee.prototype)

Engineer.prototype.constructor = Engineer;

**Note:** The term instance has a specific technical meaning in class-based languages. In these languages, an instance is an individual instantiation of a class and is fundamentally different from a class. In JavaScript, "instance" does not have this technical meaning because JavaScript does not have this difference between classes and instances. However, in talking about JavaScript, "instance" can be used informally to mean an object created using a particular constructor function. So, in this example, you could informally say that jane is an instance of Engineer. Similarly, although the terms parent, child, ancestor, and descendant do not have formal meanings in JavaScript; you can use them informally to refer to objects higher or lower in the prototype chain.

### **Inheriting properties**

Suppose you create the mark object as a WorkerBee with the following statement:

var mark = new WorkerBee;

When JavaScript sees the new operator, it creates a new generic object and implicitly sets the value of the internal property [[Prototype]] to the value of WorkerBee.prototype and passes this new object as the value of the *this* keyword to the WorkerBee constructor function. The internal [[Prototype]] property determines the prototype chain used to return property values. Once these properties are set, JavaScript returns the new object and the assignment statement sets the variable mark to that object.

This process does not explicitly put values in the mark object (*local* values) for the properties that mark inherits from the prototype chain. When you ask for the value of a property, JavaScript first checks to see if the value exists in that object. If it does, that value is returned. If the value is not there locally, JavaScript checks the prototype chain (using the internal [[Prototype]] property). If an object in the prototype chain has a value for the property, that value is returned. If no such property is found, JavaScript says the object does not have the property. In this way, the mark object has the following properties and values:

mark.name = '';

mark.dept = 'general';

mark.projects = [];

The mark object is assigned local values for the name and dept properties by the Employee constructor. It is assigned a local value for the projects property by the WorkerBeeconstructor. This gives you inheritance of properties and their values in JavaScript.

Because these constructors do not let you supply instance-specific values, this information is generic. The property values are the default ones shared by all new objects created from WorkerBee. You can, of course, change the values of any of these properties. So, you could give specific information for mark as follows:

mark.name = 'Doe, Mark';

mark.dept = 'admin';

mark.projects = ['navigator'];

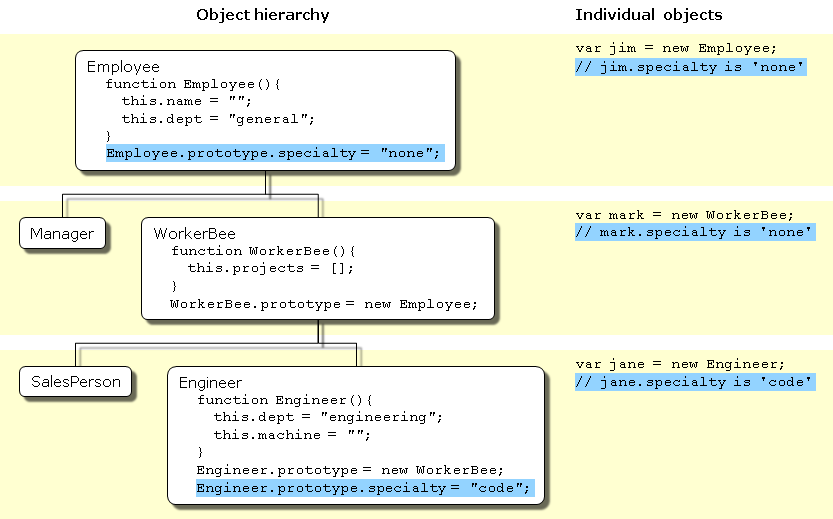
### **Adding properties**

Now, the mark object has a bonus property, but no other WorkerBee has this property.

If you add a new property to an object that is being used as the prototype for a constructor function, you add that property to all objects that inherit properties from the prototype. For example, you can add a specialty property to all employees with the following statement:

Employee.prototype.specialty = 'none';

As soon as JavaScript executes this statement, the mark object also has the specialtyproperty with the value of "none". The following figure shows the effect of adding this property to the Employee prototype and then overriding it for the Engineer prototype.



**More flexible constructors**

These JavaScript definitions use a special idiom for setting default values:

this.name = name || '';

The JavaScript logical OR operator (||) evaluates its first argument. If that argument converts to true, the operator returns it. Otherwise, the operator returns the value of the second argument. Therefore, this line of code tests to see if name has a useful value for the nameproperty. If it does, it sets this.name to that value. Otherwise, it sets this.name to the empty string.

**Note:** This may not work as expected if the constructor function is called with arguments which convert to false (like 0 (zero) and empty string (""). In this case the default value will be chosen.

With these definitions, when you create an instance of an object, you can specify values for the locally defined properties. You can use the following statement to create a new Engineer:

var jane = new Engineer('belau');

Jane's properties are now:

jane.name == '';

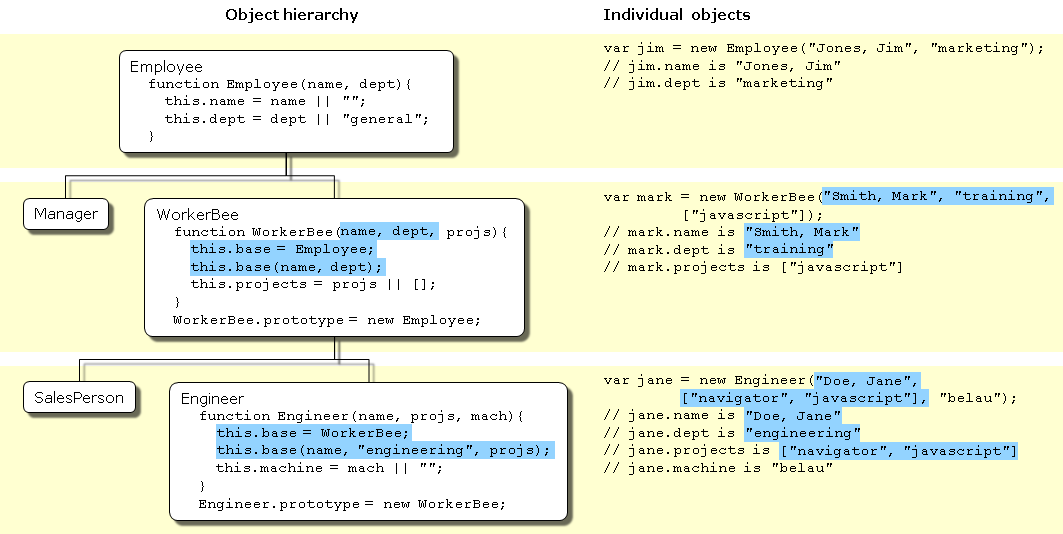
jane.dept == 'engineering';

jane.projects == [];

jane.machine == 'belau';

Notice that with these definitions, you cannot specify an initial value for an inherited property such as name. If you want to specify an initial value for inherited properties in JavaScript, you need to add more code to the constructor function.

So far, the constructor function has created a generic object and then specified local properties and values for the new object. You can have the constructor add more properties by directly calling the constructor function for an object higher in the prototype chain. The following figure shows these new definitions.



# **What is the difference between prototype and \_\_proto\_\_ in JavaScript?**

In reality, the only *true* difference between prototype and \_\_proto\_\_ is that the former is a property of a class constructor, while the latter is a property of a class instance.

In other words, while iPhone.prototype provides a blueprint for building an iPhone, newPhone.\_\_proto\_\_ affirms that the iPhone has indeed been built according to that specific blueprint. But with regards to the properties and methods present in those two objects… well, they’re exactly the same.

\_\_proto\_\_ is an object in every class *instance* that points to the prototype it was created from. Here, newPhone.\_\_proto\_\_ is a reference to iPhone.prototype, and thus holds the exact same contents as well. By having a \_\_proto\_\_ property identical to iPhone.prototype, newPhone is essentially saying, “Look, since I’m an iPhone 11, I have the exact same features as any other iPhone 11! I’ve got Face ID, 4K video, you name it.”

Let's look at one of these definitions in detail. Here's the new definition for the Engineerconstructor:

function Engineer(name, projs, mach) {

this.base = WorkerBee;

this.base(name, 'engineering', projs);

this.machine = mach || '';

}

Suppose you create a new Engineer object as follows:

var jane = new Engineer('Doe, Jane', ['navigator', 'javascript'], 'belau');

JavaScript follows these steps:

1. The new operator creates a generic object and sets its \_\_proto\_\_ property to Engineer.prototype.
2. The new operator passes the new object to the Engineer constructor as the value of the this keyword.
3. The constructor creates a new property called base for that object and assigns the value of the WorkerBee constructor to the base property. This makes the WorkerBeeconstructor a method of the Engineer object. The name of the base property is not special. You can use any legal property name; base is simply evocative of its purpose.
4. The constructor calls the base method, passing as its arguments two of the arguments passed to the constructor ("Doe, Jane" and ["navigator", "javascript"]) and also the string "engineering". Explicitly using "engineering" in the constructor indicates that all Engineer objects have the same value for the inherited dept property, and this value overrides the value inherited from Employee.
5. Because base is a method of Engineer, within the call to base, JavaScript binds the this keyword to the object created in Step 1. Thus, the WorkerBee function in turn passes the "Doe, Jane" and "engineering" arguments to the Employee constructor function. Upon return from the Employee constructor function, the WorkerBee function uses the remaining argument to set the projects property.
6. Upon return from the base method, the Engineer constructor initializes the object's machine property to "belau".
7. Upon return from the constructor, JavaScript assigns the new object to the jane variable.

You might think that, having called the WorkerBee constructor from inside the Engineerconstructor, you have set up inheritance appropriately for Engineer objects. This is not the case. Calling the WorkerBee constructor ensures that an Engineer object starts out with the properties specified in all constructor functions that are called. However, if you later add properties to the Employee or WorkerBee prototypes, those properties are not inherited by the Engineer object.

For example, assume you have the following statements:

function Engineer(name, projs, mach) {

this.base = WorkerBee;

this.base(name, 'engineering', projs);

this.machine = mach || '';

}

var jane = new Engineer('Doe, Jane', ['navigator', 'javascript'], 'belau');

Employee.prototype.specialty = 'none';

The jane object does not inherit the specialty property. You still need to explicitly set up the prototype to ensure dynamic inheritance.

Assume instead you have these statements:

function Engineer(name, projs, mach) {

this.base = WorkerBee;

this.base(name, 'engineering', projs);

this.machine = mach || '';

}

Engineer.prototype = new WorkerBee;

var jane = new Engineer('Doe, Jane', ['navigator', 'javascript'], 'belau');

Employee.prototype.specialty = 'none';

Now the value of the jane object's specialty property is "none".

Another way of inheriting is by using the [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) methods. Below are equivalent:

function Engineer(name, projs, mach) {

this.base = WorkerBee;

this.base(name, 'engineering', projs);

this.machine = mach || '';

}

function Engineer(name, projs, mach) {

WorkerBee.call(this, name, 'engineering', projs);

this.machine = mach || '';

}

Using the javascript call() method makes a cleaner implementation because the base is not needed anymore.

**Property inheritance revisited**

This section discusses some subtleties that were not necessarily apparent in the earlier discussions.

### **Local versus inherited values**

When you access an object property, JavaScript performs these steps, as described earlier in this chapter:

1. Check to see if the value exists locally. If it does, return that value.
2. If there is not a local value, check the prototype chain (using the \_\_proto\_\_ property).
3. If an object in the prototype chain has a value for the specified property, return that value.
4. If no such property is found, the object does not have the property.

The outcome of these steps depends on how you define things along the way. The original example had these definitions:

function Employee() {

this.name = '';

this.dept = 'general';

}

function WorkerBee() {

this.projects = [];

}

WorkerBee.prototype = new Employee;

With these definitions, suppose you create amy as an instance of WorkerBee with the following statement:

var amy = new WorkerBee;

The amy object has one local property, projects. The values for the name and deptproperties are not local to amy and so derive from the amy object's \_\_proto\_\_ property. Thus, amy has these property values:

amy.name == '';

amy.dept == 'general';

amy.projects == [];

Now suppose you change the value of the name property in the prototype associated with Employee:

Employee.prototype.name = 'Unknown';

At first glance, you might expect that new value to propagate down to all the instances of Employee. However, it does not.

When you create any instance of the Employee object, that instance gets a **local value** for the name property (the empty string). This means that when you set the WorkerBee prototype by creating a new Employee object, WorkerBee.prototype has a local value for the nameproperty. Therefore, when JavaScript looks up the name property of the amy object (an instance of WorkerBee), JavaScript finds the local value for that property in WorkerBee.prototype. It therefore does not look further up the chain to Employee.prototype.

If you want to change the value of an object property at run time and have the new value be inherited by all descendants of the object, you cannot define the property in the object's constructor function. Instead, you add it to the constructor's associated prototype. For example, assume you change the preceding code to the following:

function Employee() {

this.dept = 'general'; // Note that this.name (a local variable) does not appear here

}

Employee.prototype.name = ''; // A single copy

function WorkerBee() {

this.projects = [];

}

WorkerBee.prototype = new Employee;

var amy = new WorkerBee;

Employee.prototype.name = 'Unknown';

In this case, the name property of amy becomes "Unknown".

As these examples show, if you want to have default values for object properties and you want to be able to change the default values at run time, you should set the properties in the constructor's prototype, not in the constructor function itself.

### **Determining instance relationships**

Property lookup in JavaScript looks within an object's own properties and, if the property name is not found, it looks within the special object property \_\_proto\_\_. This continues recursively; the process is called "lookup in the prototype chain".

The special property \_\_proto\_\_ is set when an object is constructed; it is set to the value of the constructor's prototype property. So the expression new Foo() creates an object with \_\_proto\_\_ == Foo.prototype. Consequently, changes to the properties of Foo.prototypealters the property lookup for all objects that were created by new Foo().

Every object has a \_\_proto\_\_ object property (except Object); every function has a prototype object property. So objects can be related by 'prototype inheritance' to other objects. You can test for inheritance by comparing an object's \_\_proto\_\_ to a function's prototype object. JavaScript provides a shortcut: the instanceof operator tests an object against a function and returns true if the object inherits from the function prototype. For example,

var f = new Foo();

var isTrue = (f instanceof Foo);

a more detailed example,

Create an Engineer object as follows:

var chris = new Engineer('Pigman, Chris', ['jsd'], 'fiji');

With this object, the following statements are all true:

chris.\_\_proto\_\_ == Engineer.prototype;

chris.\_\_proto\_\_.\_\_proto\_\_ == WorkerBee.prototype;

chris.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ == Employee.prototype;

chris.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ == Object.prototype;

chris.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ == null;

### **Global information in constructors**

When you create constructors, you need to be careful if you set global information in the constructor. For example, assume that you want a unique ID to be automatically assigned to each new employee. You could use the following definition for Employee:

var idCounter = 1;

function Employee(name, dept) {

this.name = name || '';

this.dept = dept || 'general';

this.id = idCounter++;

}

With this definition, when you create a new Employee, the constructor assigns it the next ID in sequence and then increments the global ID counter. So, if your next statement is the following, victoria.id is 1 and harry.id is 2:

var victoria = new Employee('Pigbert, Victoria', 'pubs');

var harry = new Employee('Tschopik, Harry', 'sales');

At first glance that seems fine. However, idCounter gets incremented every time an Employee object is created, for whatever purpose. If you create the entire Employee hierarchy shown in this chapter, the Employee constructor is called every time you set up a prototype. Suppose you have the following code:

var idCounter = 1;

function Employee(name, dept) {

this.name = name || '';

this.dept = dept || 'general';

this.id = idCounter++;

}

function Manager(name, dept, reports) {...}

Manager.prototype = new Employee;

function WorkerBee(name, dept, projs) {...}

WorkerBee.prototype = new Employee;

function Engineer(name, projs, mach) {...}

Engineer.prototype = new WorkerBee;

function SalesPerson(name, projs, quota) {...}

SalesPerson.prototype = new WorkerBee;

var mac = new Engineer('Wood, Mac');

Further assume that the definitions omitted here have the base property and call the constructor above them in the prototype chain. In this case, by the time the mac object is created, mac.id is 5.

Depending on the application, it may or may not matter that the counter has been incremented these extra times. If you care about the exact value of this counter, one possible solution involves instead using the following constructor:

function Employee(name, dept) {

this.name = name || '';

this.dept = dept || 'general';

if (name)

this.id = idCounter++;

}

When you create an instance of Employee to use as a prototype, you do not supply arguments to the constructor. Using this definition of the constructor, when you do not supply arguments, the constructor does not assign a value to the id and does not update the counter. Therefore, for an Employee to get an assigned id, you must specify a name for the employee. In this example, mac.id would be 1.

Alternatively, you can create a copy of Employee's prototype object to assign to WorkerBee:

WorkerBee.prototype = Object.create(Employee.prototype);

// instead of WorkerBee.prototype = new Employee

### **No multiple inheritance**

Some object-oriented languages allow multiple inheritance. That is, an object can inherit the properties and values from unrelated parent objects. JavaScript does not support multiple inheritance.

Inheritance of property values occurs at run time by JavaScript searching the prototype chain of an object to find a value. Because an object has a single associated prototype, JavaScript cannot dynamically inherit from more than one prototype chain.

In JavaScript, you can have a constructor function call more than one other constructor function within it. This gives the illusion of multiple inheritance. For example, consider the following statements:

function Hobbyist(hobby) {

this.hobby = hobby || 'scuba';

}

function Engineer(name, projs, mach, hobby) {

this.base1 = WorkerBee;

this.base1(name, 'engineering', projs);

this.base2 = Hobbyist;

this.base2(hobby);

this.machine = mach || '';

}

Engineer.prototype = new WorkerBee;

var dennis = new Engineer('Doe, Dennis', ['collabra'], 'hugo');

# **Why let and var bindings behave differently using setTimeout function?**

According to MDN,

***let****allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used. This is unlike the*[*var*](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/var)*keyword, which defines a variable globally, or locally to an entire function regardless of block scope.*

Below is a simple code snippet which should console log out the value of i after 100ms on each loop.

for(var i = 1; i <= 5; i++) { setTimeout(function() {  
 console.log('Value of i : ' + i);   
 },100);}

The **desired** output of the above code is

Value of i : 1  
Value of i : 2  
Value of i : 3  
Value of i : 4  
Value of i : 5

But the **actual** output is

Value of i : 6  
Value of i : 6  
Value of i : 6  
Value of i : 6  
Value of i : 6

The above result is because, **var**definesvariable globally, or locally to an entire function regardless of block scope.

To fix the issue in ES5, we used Immediately Invoked Function Expression (IIFE) to capture the state of variable.

for(var i = 1; i <= 5; i++) {  
   
 (function(i){  
 setTimeout(function(){  
 console.log('Value of i : ' + i);  
 },100);  
 })(i);} **Output:**Value of i : 1  
Value of i : 2  
Value of i : 3  
Value of i : 4  
Value of i : 5

With the introduction of let which defines the variable in block scope, the above code can be refactored as shown below

for(let i = 1; i <= 5; i++) {  
   
 setTimeout(function(){ console.log('Value of i : ' + i); },100);}**Output:**  
Value of i : 1  
Value of i : 2  
Value of i : 3  
Value of i : 4  
Value of i : 5

**let**creates a variable declaration for each loop which is block level declaration. So basically it creates a scope within { }.

# **Variable declaration and scoping rules**

### **Declaring variables with the var keyword**

When you declare a variable using the var keyword, the scope is as follows:

* If the variable is declared outside of any functions, the variable is available in the global scope.
* If the variable is declared within a function, the variable is available from its point of declaration until the end of the function definition.

Unlike what you might be used to from other languages, variables declared with the var keyword have no block scope. In concrete terms, this means that if you declare a variable using var within a for, or any non-function block, the variable's scope extends beyond the block to the end of the block's parent scope. On the other hand, if you declare a variable inside a function with the var keyword, the variable is only available within the function definition, and cannot be accessed outside of the function. We, therefore, say that variables declared with var are function-scoped.

### **Declaring variables with the let keyword**

variables declared using the let keyword have three important characteristics.

* They are **block scoped**
* They are **not** accessible before they are assigned
* They **cannnot** be re-declared within the same scope

Let's see what this means using some examples.

### **Declaring variables with the const keyword**

Variables declared with the const keyword share all the characteristics of variables declared using the let keyword, plus one important distinguishing characteristic:

* They can't be reassigned

const a = 2;

a = 3 // Error, reassignment is not allowed

const a = 2;

const a = 3 // Error, re-declaration is not allowed

## **Variable Mutability**

Regardless of how you declare a variable, using any of the keywords we have discussed, the variable is mutable. Mutability must not be confused with reassignment.

 This difference is highlighted when working with arrays or objects. An example or two will clarify what this means.

### **Object example:**

const person = {

name: 'Michael'

};

person.name = 'Jamie' // OK! person variable mutated, not completely re-assigned

console.log(person.name); // "Jamie"

person = "Newton" // Error, re-assignment is not allowed with const declared variables

## **Accessing a variable before its declaration**

In the section on [declaring variables with let](https://dev.to/simplymichael/variable-declaration-and-scoping-rules-c0k#declaring-variables-with-the-let-keyword), we noted that one of the characteristics of let declared variables is that they are not accessible before they are declared. What does this mean? Let's see.

Consider this piece of code:

console.log(a); // undefined, but no error raised

var a = 2;

In the above snippet, we attempt to read the value of the a variable before its declaration. Instead of getting an error, we get undefined. Why is that? The answer is that var declared variables are moved to the top of the scope at execution.

At runtime, this code is interpreted as:

var a;

console.log(a); // undefined: a is declared, but hasn't been assigned a value, hence no errors raised

a = 2;

This phenomenon is what is referred to as hoisting.

If we try to do a similar thing with a variable declared using let or const, we will get a reference error thrown.

console.log(a); // ReferenceError

let a = 2;

### **nitialization of several variables**

var x = 0;

function f() {

var x = y = 1; // Declares x locally; declares y globally.

}

f();

console.log(x, y); // 0 1

# **Types of dependencies**

Most people only have dependencies and devDependencies, but each of these are important to understand.

##### dependencies

These are your normal dependencies, or rather ones that you need when running your code (e.g. React or ImmutableJS).

##### devDependencies

These are your development dependencies. Dependencies that you need at some point in the development workflow but not while running your code (e.g. Babel or Flow).

##### peerDependencies

Peer dependencies are a special type of dependency that would only ever come up if you were publishing your own package.

Having a peer dependency means that your package needs a dependency that is the same exact dependency as the person installing your package. This is useful for packages like react that need to have a single copy of react-dom that is also used by the person installing it.

##### optionalDependencies

Optional dependencies are just that: optional. If they fail to install, Yarn will still say the install process was successful.

This is useful for dependencies that won’t necessarily work on every machine and you have a fallback plan in case they are not installed (e.g. Watchman).

##### bundledDependencies

Array of package names that will be bundled when publishing the package.

Bundled dependencies should be inside your project. The functionality is basically the same as normal dependencies. They will also be packed when running yarn pack.

Normal dependencies are usually installed from the npm registry. Bundled dependencies are useful in cases normal dependencies are not sufficient:

* When you want to re-use a third party library that doesn’t come from the npm registry or that was modified.
* When you want to re-use your own projects as modules.
* When you want to distribute some files with your module.

# **JS dot-notation vs. bracket notation**

var cat = {

property1: "meow",

property2: "hiss"

}

cat.x  
*undefined*

But why? Why undefined? Is there anyway to access a property by using a variable? Surely there must be a way! Let’s try bracket notation:

cat[x]  
**hiss**

Great Scott— it works! But why didn’t I type quotation marks around x like I did with property1?

cat[x]  
**hiss**cat["property2"]  
**hiss**

Because in a JS object, **all property keys are strings**. When you use dot notation, JS is thinking you are looking for a key whose value is a string of whatever is after the dot. So

cat.x

looks for a property of cat with a key of “x”. But

cat  
*Object {property1: "meow", property2: "hiss"}*

our object cat doesn’t have a property called “x”. Now bracket notation can handle cat[x] because, as said before, the way JS interprets or “unboxes” statements. It runs along and starts to evaluate the first complete statement it encounters. It then evaluates this statement and runs along till it finds the next complete statement, which it then evaluates.

 Let’s illustrate by introducing another wrinkle in our code by way of a new object

var dog = {attribute1: "loving", attribute2: "trueBlue"}

Let’s also give cats a new property with the following value:

cat["loving"] = "something cat's are not"

What if we wanted to access this value of “something cat’s are not” by using the dog object? Can it be done with dot notation?

cat.dog.attribute1  
**TypeError: Cannot read property 'attribute1' of undefined**

Dog is not an attribute of cat, so this will not work. With dot notation we can only move down an object’s property tree by way of its explicit property keys.

But this can be done with bracket notation:

cat[dog["attribute1"]]  
"something cat's are not"

Javascript has seen the opening brackets, but keeps moving till it encounters the first closed bracket. Then it evaluates this statement. In our case above, after the first evaluation, we get

cat["loving"]

Next, JavaScript evaluates this statement, revealing the value stored inside the “loving” key of this cat property:

"something cat's are not"

**Type=module in script tag**

**Note:**The import and export are part of ES6 and not yet supported in the Node version used. To use this in the browser, we need to add type="module" attribute in the script tag.

For example:

<script type="module" src="travel.js"></script>

To learn more:

<https://javascript.info/modules-intro>