# Intro

## Intro

**JavaScript Garden** is a growing collection of documentation about the most quirky parts of the JavaScript programming language. It gives advice to avoid common mistakes and subtle bugs, as well as performance issues and bad practices, that non-expert JavaScript programmers may encounter on their endeavours into the depths of the language.

JavaScript Garden does **not** aim to teach you JavaScript. Former knowledge of the language is strongly recommended in order to understand the topics covered in this guide. In order to learn the basics of the language, please head over to the excellent guide on the Mozilla Developer Network.

## The Authors

This guide is the work of two lovely Stack Overflow users, Ivo Wetzel (Writing) and Zhang Yi Jiang (Design).

It's currently maintained by Tim Ruffles.

## Contributors

- Too many to list here, see all contributors.

## Hosting

JavaScript Garden is hosted on GitHub, but Cramer Development supports us with a mirror at JavaScriptGarden.info.

## License

JavaScript Garden is published under the MIT license and hosted on GitHub. If you find errors or typos please file an issue or a pull request on the repository. You can also find us in the JavaScript room on Stack Overflow chat.

# Objects

## Object Usage And Properties

Everything in JavaScript acts like an object, with the only two exceptions being `null` and `undefined`.

```
false.toString(); // 'false'
[1, 2, 3].toString(); // '1,2,3'

function Foo(){}
Foo.bar = 1;
Foo.bar; // 1
```

A common misconception is that number literals cannot be used as objects. That is because a flaw in JavaScript's parser tries to parse the *dot notation* on a number as a floating point literal.

```
2.toString(); // raises SyntaxError
```

There are a couple of workarounds that can be used to make number literals act as objects too.

```
2..toString(); // the second point is correctly recognized
2 .toString(); // note the space left to the dot
(2).toString(); // 2 is evaluated first
```

### Objects As A Data Type

Objects in JavaScript can also be used as *Hashmaps*; they mainly consist of named properties mapping to values.

Using an object literal - `{}` notation - it is possible to create a plain object. This new object inherits from `Object.prototype` and does not have own properties defined.

```
var foo = {}; // a new empty object

// a new object with a 'test' property with value 12
var bar = {test: 12};
```

### Accessing Properties

The properties of an object can be accessed in two ways, via either the dot notation or the square bracket notation.

```
var foo = {name: 'kitten'}
foo.name; // kitten
foo['name']; // kitten

var get = 'name';
foo[get]; // kitten

foo.1234; // SyntaxError
foo['1234']; // works
```

The notations work almost identically, with the only difference being that the square bracket notation allows for dynamic setting of properties and the use of property names that would otherwise lead to a syntax error.

### Deleting Properties

The only way to remove a property from an object is to use the `delete` operator; setting the property to `undefined` or `null` only removes the *value* associated with the property, but not the *key*.

```
var obj = {
    bar: 1,
    foo: 2,
    baz: 3
};
obj.bar = undefined;
obj.foo = null;
delete obj.baz;

for(var i in obj) {
    if (obj.hasOwnProperty(i)) {
        console.log(i, '' + obj[i]);
    }
}
```

The above outputs both `bar undefined` and `foo null` - only `baz` was removed and is therefore missing from the output.

### Notation Of Keys

```
var test = {
    'case': 'I am a keyword, so I must be notated as a string',
    delete: 'I am a keyword, so me too' // raises SyntaxError
};
```

Object properties can be both notated as plain characters and as strings. Due to another mis-design in JavaScript's parser, the above will throw a `SyntaxError` prior to ECMAScript 5.

This error arises from the fact that `delete` is a *keyword*; therefore, it must be notated as a *string literal* to ensure that it will be correctly interpreted by older JavaScript engines.

---

## The Prototype

JavaScript does not feature a classical inheritance model; instead, it uses a *prototypal* one.

While this is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model is in fact more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model, while the other way around is a far more difficult task.

JavaScript is the only widely used language that features prototypal inheritance, so it can take time to adjust to the differences between the two models.

The first major difference is that inheritance in JavaScript uses *prototype chains*.

```
function Foo() {
    this.value = 42;
}
Foo.prototype = {
    method: function() {}
};

function Bar() {}

// Set Bar's prototype to a new instance of Foo
Bar.prototype = new Foo();
Bar.prototype.foo = 'Hello World';

// Make sure to list Bar as the actual constructor
Bar.prototype.constructor = Bar;

var test = new Bar(); // create a new bar instance

// The resulting prototype chain
test [instance of Bar]
    Bar.prototype [instance of Foo]
        { foo: 'Hello World' }
        Foo.prototype
            { method: ... }
            Object.prototype
                { toString: ... /* etc. */ }
```

**Note:** Simply using `Bar.prototype = Foo.prototype` will result in both objects sharing the **same** prototype. Therefore, changes to either object's prototype will affect the prototype of the other as well, which in most cases is not the desired effect.

In the code above, the object `test` will inherit from both `Bar.prototype` and `Foo.prototype`; hence, it will have access to the function `method` that was defined on `Foo`. It will also have access to the property `value` of the **one** `Foo` instance that is its prototype. It is important to note that `new Bar()` does **not** create a new `Foo` instance, but reuses the one assigned to its prototype; thus, all `Bar` instances will share the **same** `value` property.

### Property Lookup

When accessing the properties of an object, JavaScript will traverse the prototype chain **upwards** until it finds a property with the requested name.

If it reaches the top of the chain - namely `Object.prototype` - and still hasn't found the specified property, it will return the value undefined instead.

**Note:** Do **not** use `Bar.prototype = Foo`, since it will not point to the prototype of `Foo` but rather to the function object `Foo`. So the prototype chain will go over `Function.prototype` and not `Foo.prototype`; therefore, `method` will not be on the prototype chain.

### The Prototype Property

While the prototype property is used by the language to build the prototype chains, it is still possible to assign **any** given value to it. However, primitives will simply get ignored when assigned as a prototype.

```
function Foo() {}
Foo.prototype = 1; // no effect
```

Assigning objects, as shown in the example above, will work, and allows for dynamic creation of prototype chains.

### Performance

The lookup time for properties that are high up on the prototype chain can have a negative impact on performance, and this may be significant in code where performance is critical. Additionally, trying to access non-existent properties will always traverse the full prototype chain.

Also, when iterating over the properties of an object **every** property that is on the prototype chain will be enumerated.

### Extension Of Native Prototypes

One mis-feature that is often used is to extend `Object.prototype` or one of the other built in prototypes.

This technique is called monkey patching and breaks *encapsulation*. While used by popular frameworks such as Prototype, there is still no good reason for cluttering built-in types with additional *non-standard* functionality.

The **only** good reason for extending a built-in prototype is to backport the features of newer JavaScript engines; for example, `Array.forEach`.

### In Conclusion

It is **essential** to understand the prototypal inheritance model before writing complex code that makes use of it. Also, be aware of the length of the prototype chains in your code and break them up if necessary to avoid possible performance problems. Further, the native prototypes should **never** be extended unless it is for the sake of compatibility with newer JavaScript features.

---

## `hasOwnProperty`

To check whether an object has a property defined on *itself* and not somewhere on its prototype chain, it is necessary to use the `hasOwnProperty` method which all objects inherit from `Object.prototype`.

`hasOwnProperty` is the only thing in JavaScript which deals with properties and does **not** traverse the prototype chain.

**Note:** It is **not** enough to check whether a property is `undefined`. The property might very well exist, but its value just happens to be set to `undefined`.

```
// Poisoning Object.prototype
Object.prototype.bar = 1;
var foo = {goo: undefined};

foo.bar; // 1
'bar' in foo; // true

foo.hasOwnProperty('bar'); // false
foo.hasOwnProperty('goo'); // true
```

Only `hasOwnProperty` will give the correct and expected result. See the section on `for in` loops for more details on when to use `hasOwnProperty` when iterating over object properties.

### `hasOwnProperty` As A Property

JavaScript does not protect the property name `hasOwnProperty`; thus, if the possibility exists that an object might have a property with this name, it is necessary to use an *external* `hasOwnProperty` to get correct results.

```
var foo = {
    hasOwnProperty: function() {
        return false;
    },
    bar: 'Here be dragons'
};

foo.hasOwnProperty('bar'); // always returns false

// Use another Object's hasOwnProperty and call it with 'this' set to foo
({}).hasOwnProperty.call(foo, 'bar'); // true

// It's also possible to use hasOwnProperty from the Object
// prototype for this purpose
Object.prototype.hasOwnProperty.call(foo, 'bar'); // true
```

### In Conclusion

Using `hasOwnProperty` is the **only** reliable method to check for the existence of a property on an object. It is recommended that `hasOwnProperty` be used in many cases when iterating over object properties as described in the section on `for in` loops.

---

## The `for in` Loop

Just like the `in` operator, the `for in` loop traverses the prototype chain when iterating over the properties of an object.

**Note:** The `for in` loop will **not** iterate over any properties that have their `enumerable` attribute set to `false`; for example, the `length` property of an array.

```
// Poisoning Object.prototype
Object.prototype.bar = 1;

var foo = {moo: 2};
for(var i in foo) {
    console.log(i); // prints both bar and moo
}
```

Since it is not possible to change the behavior of the `for in` loop itself, it is necessary to filter out the unwanted properties inside the loop body. In ECMAScript 3 and older, this is done using the `hasOwnProperty` method of `Object.prototype`.

Since ECMAScript 5, `Object.defineProperty` can be used with `enumerable` set to `false` to add properties to objects (including `Object`) without these properties being enumerated. In this case it is reasonable to assume in application code that any enumerable properties have been added for a reason and to omit `hasOwnProperty`, since it makes code more verbose and less readable. In library code `hasOwnProperty` should still be used since assumptions cannot be made about which enumerable properties might reside on the prototype chain.

### Using `hasOwnProperty` For Filtering

```
// still the foo from above
for(var i in foo) {
    if (foo.hasOwnProperty(i)) {
        console.log(i);
    }
}
```

**Note:** Since `for in` always traverses the complete prototype chain, it will get slower with each additional layer of inheritance added to an object.

This version is the only correct one to use with older versions of ECMAScript. Due to the use of `hasOwnProperty`, it will **only** print out `moo`. When `hasOwnProperty` is left out, the code is prone to errors in cases where the native prototypes - e.g. `Object.prototype` - have been extended.

In newer versions of ECMAScript, non-enumerable properties can be defined with `Object.defineProperty`, reducing the risk of iterating over properties without using `hasOwnProperty`. Nonetheless, care must be taken when using older libraries like Prototype, which does not yet take advantage of new ECMAScript features. When this framework is included, `for in` loops that do not use `hasOwnProperty` are guaranteed to break.

### In Conclusion

It is recommended to **always** use `hasOwnProperty` in ECMAScript 3 or lower, as well as in library code. Assumptions should never be made in these environments about whether the native prototypes have been extended or not. Since ECMAScript 5, `Object.defineProperty` makes it possible to define non-enumerable properties and to omit `hasOwnProperty` in application code.

# Functions

## Function Declarations And Expressions

Functions in JavaScript are first class objects. That means they can be passed around like any other value. One common use of this feature is to pass an *anonymous function* as a callback to another, possibly an asynchronous function.

### The `function` Declaration

```
function foo() {}
```

The above function gets hoisted before the execution of the program starts; thus, it is available *everywhere* in the scope it was *defined*, even if called before the actual definition in the source.

```
foo(); // Works because foo was created before this code runs
function foo() {}
```

### The `function` Expression

```
var foo = function() {};
```

This example assigns the unnamed and *anonymous* function to the variable `foo`.

```
foo; // 'undefined'
foo(); // this raises a TypeError
var foo = function() {};
```

Due to the fact that `var` is a declaration that hoists the variable name `foo` before the actual execution of the code starts, `foo` is already declared when the script gets executed.

But since assignments only happen at runtime, the value of `foo` will default to undefined before the corresponding code is executed.

### Named Function Expression

Another special case is the assignment of named functions.

```
var foo = function bar() {
    bar(); // Works
}
bar(); // ReferenceError
```

Here, `bar` is not available in the outer scope, since the function only gets assigned to `foo`; however, inside of `bar`, it is available. This is due to how name resolution in JavaScript works, the name of the function is *always* made available in the local scope of the function itself.

## How `this` Works

JavaScript has a different concept of what the special name `this` refers to than most other programming languages. There are exactly **five** different ways in which the value of `this` can be bound in the language.

### The Global Scope

```
this;
```

When using `this` in global scope, it will simply refer to the *global* object.

### Calling A Function

```
foo();
```

Here, `this` will again refer to the *global* object.

ES5 Note: In strict mode, the global case **no longer** exists. `this` will instead have the value of `undefined` in that case.

### Calling A Method

```
test.foo();
```

In this example, `this` will refer to `test`.

### Calling A Constructor

```
new foo();
```

A function call that is preceded by the `new` keyword acts as a constructor. Inside the function, `this` will refer to a *newly created* `Object`.

### Explicit Setting Of `this`

```
function foo(a, b, c) {}

var bar = {};
foo.apply(bar, [1, 2, 3]); // array will expand to the below
foo.call(bar, 1, 2, 3); // results in a = 1, b = 2, c = 3
```

When using the `call` or `apply` methods of `Function.prototype`, the value of `this` inside the called function gets **explicitly set** to the first argument of the corresponding function call.

As a result, in the above example the *method case* does **not** apply, and `this` inside of `foo` will be set to `bar`.

**Note:** `this` **cannot** be used to refer to the object inside of an `Object` literal. So `var obj = {me: this}` will **not** result in `me` referring to `obj`, since `this` only gets bound by one of the five listed cases.

### Common Pitfalls

While most of these cases make sense, the first can be considered another mis-design of the language because it **never** has any practical use.

```
Foo.method = function() {
    function test() {
        // this is set to the global object
    }
    test();
}
```

A common misconception is that `this` inside of `test` refers to `Foo`; while in fact, it **does not**.

In order to gain access to `Foo` from within `test`, you can create a local variable inside of `method` that refers to `Foo`.

```
Foo.method = function() {
    var self = this;
    function test() {
        // Use self instead of this here
    }
    test();
}
```

`self` is just a normal variable name, but it is commonly used for the reference to an outer `this`. In combination with closures, it can also be used to pass `this` values around.

As of ECMAScript 5 you can use the `bind` method combined with an anonymous function to achieve the same result.

```
Foo.method = function() {
    var test = function() {
        // this now refers to Foo
    }.bind(this);
    test();
}
```

### Assigning Methods

Another thing that does **not** work in JavaScript is function aliasing, which is **assigning** a method to a variable.

```
var test = someObject.methodTest;
test();
```

Due to the first case, `test` now acts like a plain function call; therefore, `this` inside it will no longer refer to `someObject`.

While the late binding of `this` might seem like a bad idea at first, in fact, it is what makes prototypal inheritance work.

```
function Foo() {}
Foo.prototype.method = function() {};

function Bar() {}
Bar.prototype = Foo.prototype;

new Bar().method();
```

When `method` gets called on an instance of `Bar`, `this` will now refer to that very instance.

## Closures And References

One of JavaScript's most powerful features is the availability of *closures*. With closures, scopes **always** keep access to the outer scope, in which they were defined. Since the only scoping that JavaScript has is function scope, all functions, by default, act as closures.

### Emulating Private Variables

```
function Counter(start) {
    var count = start;
    return {
        increment: function() {
            count++;
        },

        get: function() {
            return count;
        }
    }
}

var foo = Counter(4);
foo.increment();
foo.get(); // 5
```

Here, `Counter` returns **two** closures: the function `increment` as well as the function `get`. Both of these functions keep a **reference** to the scope of `Counter` and, therefore, always keep access to the `count` variable that was defined in that scope.

### Why Private Variables Work

Since it is not possible to reference or assign scopes in JavaScript, there is **no** way of accessing the variable `count` from the outside. The only way to interact with it is via the two closures.

```
var foo = new Counter(4);
foo.hack = function() {
    count = 1337;
};
```

The above code will **not** change the variable `count` in the scope of `Counter`, since `foo.hack` was not defined in **that** scope. It will instead create - or override - the *global* variable `count`.

### Closures Inside Loops

One often made mistake is to use closures inside of loops, as if they were copying the value of the loop's index variable.

```
for(var i = 0; i < 10; i++) {
    setTimeout(function() {
        console.log(i);
    }, 1000);
}
```

The above will **not** output the numbers `0` through `9`, but will simply print the number `10` ten times.

The *anonymous* function keeps a **reference** to `i`. At the time `console.log` gets called, the `for loop` has already finished, and the value of `i` has been set to `10`.

In order to get the desired behavior, it is necessary to create a **copy** of the value of `i`.

### Avoiding The Reference Problem

In order to copy the value of the loop's index variable, it is best to use an anonymous wrapper.

```
for(var i = 0; i < 10; i++) {
    (function(e) {
        setTimeout(function() {
            console.log(e);
        }, 1000);
    })(i);
}
```

The anonymous outer function gets called immediately with `i` as its first argument and will receive a copy of the **value** of `i` as its parameter `e`.

The anonymous function that gets passed to `setTimeout` now has a reference to `e`, whose value does **not** get changed by the loop.

There is another possible way of achieving this, which is to return a function from the anonymous wrapper that will then have the same behavior as the code above.

```
for(var i = 0; i < 10; i++) {
    setTimeout((function(e) {
        return function() {
            console.log(e);
        }
    })(i), 1000)
}
```

The other popular way to achieve this is to add an additional argument to the `setTimeout` function, which passes these arguments to the callback.

```
for(var i = 0; i < 10; i++) {
    setTimeout(function(e) {
        console.log(e);
    }, 1000, i);
}
```

Some legacy JS environments (Internet Explorer 9 & below) do not support this.

There's yet another way to accomplish this by using `.bind`, which can bind a `this` context and arguments to function. It behaves identically to the code above

```
for(var i = 0; i < 10; i++) {
    setTimeout(console.log.bind(console, i), 1000);
}
```

## The `arguments` Object

Every function scope in JavaScript can access the special variable `arguments`. This variable holds a list of all the arguments that were passed to the function.

The `arguments` object is **not** an `Array`. While it has some of the semantics of an array - namely the `length` property - it does not inherit from `Array.prototype` and is in fact an `Object`.

Due to this, it is **not** possible to use standard array methods like `push`, `pop` or `slice` on `arguments`. While iteration with a plain `for` loop works just fine, it is necessary to convert it to a real `Array` in order to use the standard `Array` methods on it.

**Note:** In case `arguments` has already been defined inside the function's scope either via a `var` statement or being the name of a formal parameter, the `arguments` object will not be created.

### Converting To An Array

The code below will return a new `Array` containing all the elements of the `arguments` object.

```
Array.prototype.slice.call(arguments);
```

Because this conversion is **slow**, it is **not recommended** to use it in performance-critical sections of code.

### Passing Arguments

The following is the recommended way of passing arguments from one function to another.

```
function foo() {
    bar.apply(null, arguments);
}
function bar(a, b, c) {
    // do stuff here
}
```

Another trick is to use both `call` and `apply` together to turn methods - functions that use the value of `this` as well as their arguments - into normal functions which only use their arguments.

```
function Person(first, last) {
  this.first = first;
  this.last = last;
}

Person.prototype.fullname = function(joiner, options) {
  options = options || { order: "western" };
  var first = options.order === "western" ? this.first : this.last;
  var last =  options.order === "western" ? this.last  : this.first;
  return first + (joiner || " ") + last;
};

// Create an unbound version of "fullname", usable on any object with 'first'
// and 'last' properties passed as the first argument. This wrapper will
// not need to change if fullname changes in number or order of arguments.
Person.fullname = function() {
  // Result: Person.prototype.fullname.call(this, joiner, ..., argN);
  return Function.call.apply(Person.prototype.fullname, arguments);
};

var grace = new Person("Grace", "Hopper");

// 'Grace Hopper'
grace.fullname();

// 'Turing, Alan'
Person.fullname({ first: "Alan", last: "Turing" }, ", ", { order: "eastern" });
```

### Formal Parameters And Arguments Indices

The `arguments` object creates *getter* and *setter* functions for both its properties, as well as the function's formal parameters.

As a result, changing the value of a formal parameter will also change the value of the corresponding property on the `arguments` object, and the other way around.

```
function foo(a, b, c) {
    arguments[0] = 2;
    a; // 2

    b = 4;
    arguments[1]; // 4

    var d = c;
    d = 9;
    c; // 3
}
foo(1, 2, 3);
```

### Performance Myths And Truths

The only time the `arguments` object is not created is where it is declared as a name inside of a function or one of its formal parameters. It does not matter whether it is used or not.

Both *getters* and *setters* are **always** created; thus, using it has nearly no performance impact at all, especially not in real world code where there is more than a simple access to the `arguments` object's properties.

**ES5 Note:** These *getters* and *setters* are not created in strict mode.

However, there is one case which will drastically reduce the performance in modern JavaScript engines. That case is the use of `arguments.callee`.

```
function foo() {
    arguments.callee; // do something with this function object
    arguments.callee.caller; // and the calling function object
}
```

```
function bigLoop() {
    for(var i = 0; i < 100000; i++) {
        foo(); // Would normally be inlined...
    }
}
```

In the above code, `foo` can no longer be a subject to inlining since it needs to know about both itself and its caller. This not only defeats possible performance gains that would arise from inlining, but it also breaks encapsulation because the function may now be dependent on a specific calling context.

Making use of `arguments.callee` or any of its properties is **highly discouraged**.

**ES5 Note:** In strict mode, `arguments.callee` will throw a `TypeError` since its use has been deprecated.

## Constructors

Constructors in JavaScript are yet again different from many other languages. Any function call that is preceded by the `new` keyword acts as a constructor.

Inside the constructor - the called function - the value of `this` refers to a newly created object. The prototype of this **new** object is set to the `prototype` of the function object that was invoked as the constructor.

If the function that was called has no explicit `return` statement, then it implicitly returns the value of `this` - the new object.

```
function Person(name) {
    this.name = name;
}

Person.prototype.logName = function() {
    console.log(this.name);
};

var sean = new Person();
```

The above calls `Person` as constructor and sets the `prototype` of the newly created object to `Person.prototype`.

In case of an explicit `return` statement, the function returns the value specified by that statement, but **only** if the return value is an `Object`.

```
function Car() {
    return 'ford';
}
new Car(); // a new object, not 'ford'

function Person() {
    this.someValue = 2;

    return {
        name: 'Charles'
    };
}
new Person(); // the returned object ({name:'Charles'}), not including someValue
```

When the `new` keyword is omitted, the function will **not** return a new object.

```
function Pirate() {
    this.hasEyePatch = true; // gets set on the global object!
}
var somePirate = Pirate(); // somePirate is undefined
```

While the above example might still appear to work in some cases, due to the workings of `this` in JavaScript, it will use the *global object* as the value of `this`.

### Factories

In order to be able to omit the `new` keyword, the constructor function has to explicitly return a value.

```
function Robot() {
    var color = 'gray';
    return {
        getColor: function() {
            return color;
        }
    }
}

new Robot();
Robot();
```

Both calls to `Robot` return the same thing, a newly created object that has a property called `getColor`, which is a Closure.

It should also be noted that the call `new Robot()` does **not** affect the prototype of the returned object. While the prototype will be set on the newly created object, `Robot` never returns that new object.

In the above example, there is no functional difference between using and not using the `new` keyword.

### Creating New Objects Via Factories

It is often recommended to **not** use `new` because forgetting its use may lead to bugs.

In order to create a new object, one should rather use a factory and construct a new object inside of that factory.

```
function CarFactory() {
    var car = {};
    car.owner = 'nobody';

    var milesPerGallon = 2;
```

```
        car.setOwner = function(newOwner) {
            this.owner = newOwner;
        }

        car.getMPG = function() {
            return milesPerGallon;
        }

        return car;
    }
```

While the above is robust against a missing `new` keyword and certainly makes the use of private variables easier, it comes with some downsides.

1. It uses more memory since the created objects do **not** share the methods on a prototype.
2. In order to inherit, the factory needs to copy all the methods from another object or put that object on the prototype of the new object.
3. Dropping the prototype chain just because of a left out `new` keyword is contrary to the spirit of the language.

### In Conclusion

While omitting the `new` keyword might lead to bugs, it is certainly **not** a reason to drop the use of prototypes altogether. In the end it comes down to which solution is better suited for the needs of the application. It is especially important to choose a specific style of object creation and use it **consistently**.

---

## Scopes And Namespaces

Although JavaScript deals fine with the syntax of two matching curly braces for blocks, it does **not** support block scope; hence, all that is left in the language is *function scope*.

```
function test() { // a scope
    for(var i = 0; i < 10; i++) { // not a scope
        // count
    }
    console.log(i); // 10
}
```

**Note:** When not used in an assignment, return statement or as a function argument, the `{...}` notation will get interpreted as a block statement and **not** as an object literal. This, in conjunction with automatic insertion of semicolons, can lead to subtle errors.

There are also no distinct namespaces in JavaScript, which means that everything gets defined in one *globally shared* namespace.

Each time a variable is referenced, JavaScript will traverse upwards through all the scopes until it finds it. In the case that it reaches the global scope and still has not found the requested name, it will raise a `ReferenceError`.

### The Bane Of Global Variables

```
// script A
foo = '42';

// script B
var foo = '42'
```

The above two scripts do **not** have the same effect. Script A defines a variable called `foo` in the *global* scope, and script B defines a `foo` in the *current* scope.

Again, that is **not** at all the *same effect*: not using `var` can have major implications.

```
// global scope
var foo = 42;
function test() {
    // local scope
    foo = 21;
}
test();
foo; // 21
```

Leaving out the `var` statement inside the function `test` will override the value of `foo`. While this might not seem like a big deal at first, having thousands of lines of JavaScript and not using `var` will introduce horrible, hard-to-track-down bugs.

```
// global scope
var items = [/* some list */];
for(var i = 0; i < 10; i++) {
    subLoop();
}

function subLoop() {
    // scope of subLoop
    for(i = 0; i < 10; i++) { // missing var statement
        // do amazing stuff!
    }
}
```

The outer loop will terminate after the first call to `subLoop`, since `subLoop` overwrites the global value of `i`. Using a `var` for the second `for` loop would have easily avoided this error. The `var` statement should **never** be left out unless the *desired effect* is to affect the outer scope.

### Local Variables

The only source for local variables in JavaScript are function parameters and variables declared via the `var` statement.

```
// global scope
var foo = 1;
var bar = 2;
var i = 2;

function test(i) {
    // local scope of the function test
    i = 5;
```

```
    var foo = 3;
    bar = 4;
}
test(10);
```

While `foo` and `i` are local variables inside the scope of the function `test`, the assignment of `bar` will override the global variable with the same name.

### Hoisting

JavaScript **hoists** declarations. This means that both `var` statements and `function` declarations will be moved to the top of their enclosing scope.

```
bar();
var bar = function() {};
var someValue = 42;

test();
function test(data) {
    if (false) {
        goo = 1;

    } else {
        var goo = 2;
    }
    for(var i = 0; i < 100; i++) {
        var e = data[i];
    }
}
```

The above code gets transformed before execution starts. JavaScript moves the `var` statements, as well as `function` declarations, to the top of the nearest surrounding scope.

```
// var statements got moved here
var bar, someValue; // default to 'undefined'

// the function declaration got moved up too
function test(data) {
    var goo, i, e; // missing block scope moves these here
    if (false) {
        goo = 1;

    } else {
        goo = 2;
    }
    for(i = 0; i < 100; i++) {
        e = data[i];
    }
}

bar(); // fails with a TypeError since bar is still 'undefined'
someValue = 42; // assignments are not affected by hoisting
bar = function() {};

test();
```

Missing block scoping will not only move `var` statements out of loops and their bodies, it will also make the results of certain `if` constructs non-intuitive.

In the original code, although the `if` statement seemed to modify the *global variable* `goo`, it actually modifies the *local variable* - after hoisting has been applied.

Without knowledge of *hoisting*, one might suspect the code below would raise a `ReferenceError`.

```
// check whether SomeImportantThing has been initialized
if (!SomeImportantThing) {
    var SomeImportantThing = {};
}
```

But of course, this works due to the fact that the `var` statement is being moved to the top of the *global scope*.

```
var SomeImportantThing;

// other code might initialize SomeImportantThing here, or not

// make sure it's there
if (!SomeImportantThing) {
    SomeImportantThing = {};
}
```

### Name Resolution Order

All scopes in JavaScript, including the *global scope*, have the special name `this`, defined in them, which refers to the *current object*.

Function scopes also have the name `arguments`, defined in them, which contains the arguments that were passed to the function.

For example, when trying to access a variable named `foo` inside the scope of a function, JavaScript will look up the name in the following order:

1. In case there is a `var foo` statement in the current scope, use that.
2. If one of the function parameters is named `foo`, use that.
3. If the function itself is called `foo`, use that.
4. Go to the next outer scope, and start with **#1** again.

**Note:** Having a parameter called `arguments` will **prevent** the creation of the default `arguments` object.

### Namespaces

A common problem associated with having only one global namespace is the likelihood of running into problems where variable names clash. In JavaScript, this problem can easily be avoided with the help of *anonymous wrappers*.

```
(function() {
    // a self contained "namespace"

    window.foo = function() {
        // an exposed closure
    };

})(); // execute the function immediately
```

Unnamed functions are considered expressions; so in order to be callable, they must first be evaluated.

```
( // evaluate the function inside the parentheses
function() {}
) // and return the function object
() // call the result of the evaluation
```

There are other ways to evaluate and directly call the function expression which, while different in syntax, behave the same way.

```
// A few other styles for directly invoking the
!function(){}()
+function(){}()
(function(){}());
// and so on...
```

### In Conclusion

It is recommended to always use an *anonymous wrapper* to encapsulate code in its own namespace. This does not only protect code against name clashes, but it also allows for better modularization of programs.

Additionally, the use of global variables is considered **bad practice**. **Any** use of them indicates badly written code that is prone to errors and hard to maintain.

# Arrays

## Array Iteration And Properties

Although arrays in JavaScript are objects, there are no good reasons to use the `for in` loop. In fact, there are a number of good reasons **against** the use of `for in` on arrays.

**Note:** JavaScript arrays are **not** *associative arrays*. JavaScript only has objects for mapping keys to values. And while associative arrays **preserve** order, objects **do not**.

Because the `for in` loop enumerates all the properties that are on the prototype chain and because the only way to exclude those properties is to use `hasOwnProperty`, it is already up to **twenty times** slower than a normal `for` loop.

### Iteration

In order to achieve the best performance when iterating over arrays, it is best to use the classic `for` loop.

```
var list = [1, 2, 3, 4, 5, ...... 100000000];
for(var i = 0, l = list.length; i < l; i++) {
    console.log(list[i]);
}
```

There is one extra catch in the above example, which is the caching of the length of the array via `l = list.length`.

Although the `length` property is defined on the array itself, there is still an overhead for doing the lookup on each iteration of the loop. And while recent JavaScript engines **may** apply optimization in this case, there is no way of telling whether the code will run on one of these newer engines or not.

In fact, leaving out the caching may result in the loop being only **half as fast** as with the cached length.

### The `length` Property

While the *getter* of the `length` property simply returns the number of elements that are contained in the array, the *setter* can be used to **truncate** the array.

```
var arr = [1, 2, 3, 4, 5, 6];
arr.length = 3;
arr; // [1, 2, 3]

arr.length = 6;
arr.push(4);
arr; // [1, 2, 3, undefined, undefined, undefined, 4]
```

Assigning a smaller length truncates the array. Increasing it creates a sparse array.

### In Conclusion

For the best performance, it is recommended to always use the plain `for` loop and cache the `length` property. The use of `for in` on an array is a sign of badly written code that is prone to bugs and bad performance.

## The `Array` Constructor

Since the `Array` constructor is ambiguous in how it deals with its parameters, it is highly recommended to use the array literal - `[]` notation - when creating new arrays.

```
[1, 2, 3]; // Result: [1, 2, 3]
new Array(1, 2, 3); // Result: [1, 2, 3]

[3]; // Result: [3]
new Array(3); // Result: []
new Array('3') // Result: ['3']
```

In cases when there is only one argument passed to the `Array` constructor and when that argument is a `Number`, the constructor will return a new *sparse* array with the `length` property set to the value of the argument. It should be noted that **only** the `length` property of the new array will be set this way; the actual indexes of the array will not be initialized.

```
var arr = new Array(3);
arr[1]; // undefined
1 in arr; // false, the index was not set
```

Being able to set the length of the array in advance is only useful in a few cases, like repeating a string, in which it avoids the use of a loop.

```
new Array(count + 1).join(stringToRepeat);
```

### In Conclusion

Literals are preferred to the Array constructor. They are shorter, have a clearer syntax, and increase code readability.

# Types

## Equality And Comparisons

JavaScript has two different ways of comparing the values of objects for equality.

### The Equality Operator

The equality operator consists of two equal signs: `==`

JavaScript features *weak typing*. This means that the equality operator **coerces** types in order to compare them.

```
""           ==   "0"           // false
0            ==   ""            // true
0            ==   "0"           // true
false        ==   "false"       // false
false        ==   "0"           // true
false        ==   undefined     // false
false        ==   null          // false
null         ==   undefined     // true
" \t\r\n"    ==   0             // true
```

The above table shows the results of the type coercion, and it is the main reason why the use of `==` is widely regarded as bad practice. It introduces hard-to-track-down bugs due to its complicated conversion rules.

Additionally, there is also a performance impact when type coercion is in play; for example, a string has to be converted to a number before it can be compared to another number.

### The Strict Equality Operator

The strict equality operator consists of **three** equal signs: `===`.

It works like the normal equality operator, except that strict equality operator does **not** perform type coercion between its operands.

```
""           ===   "0"           // false
0            ===   ""            // false
0            ===   "0"           // false
false        ===   "false"       // false
false        ===   "0"           // false
false        ===   undefined     // false
false        ===   null          // false
null         ===   undefined     // false
" \t\r\n"    ===   0             // false
```

The above results are a lot clearer and allow for early breakage of code. This hardens code to a certain degree and also gives performance improvements in case the operands are of different types.

### Comparing Objects

While both `==` and `===` are called **equality** operators, they behave differently when at least one of their operands is an `Object`.

```
{} === {};                   // false
new String('foo') === 'foo'; // false
new Number(10) === 10;       // false
var foo = {};
foo === foo;                 // true
```

Here, both operators compare for **identity** and **not** equality; that is, they will compare for the same **instance** of the object, much like `is` in Python and pointer comparison in C.

### In Conclusion

It is highly recommended to only use the **strict equality** operator. In cases where types need to be coerced, it should be done explicitly and not left to the language's complicated coercion rules.

## The `typeof` Operator

The `typeof` operator (together with `instanceof`) is probably the biggest design flaw of JavaScript, as it is almost **completely broken**.

Although `instanceof` still has limited uses, `typeof` really has only one practical use case, which does **not** happen to be checking the type of an object.

**Note:** While `typeof` can also be called with a function like syntax, i.e. `typeof(obj)`, this is not a function call. The parentheses behave as normal and the return value will be used as the operand of the `typeof` operator. There is **no** `typeof` function.

### The JavaScript Type Table

```
Value               Class      Type
-------------------------------------
"foo"               String     string
new String("foo")   String     object
1.2                 Number     number
new Number(1.2)     Number     object
true                Boolean    boolean
new Boolean(true)   Boolean    object
new Date()          Date       object
new Error()         Error      object
[1,2,3]             Array      object
new Array(1, 2, 3)  Array      object
new Function("")    Function   function
/abc/g              RegExp     object (function in Nitro/V8)
new RegExp("meow")  RegExp     object (function in Nitro/V8)
{}                  Object     object
new Object()        Object     object
```

In the above table, *Type* refers to the value that the `typeof` operator returns. As can be clearly seen, this value is anything but consistent.

The *Class* refers to the value of the internal `[[Class]]` property of an object.

### The Class Of An Object

**From the Specification:** The value of `[[Class]]` can be one of the following strings. `Arguments`, `Array`, `Boolean`, `Date`, `Error`, `Function`, `JSON`, `Math`, `Number`, `Object`, `RegExp`, `String`.

The only way to determine an object's `[[Class]]` value is using `Object.prototype.toString`. It returns a string in the following format: `'[object ' + valueOfClass + ']'`, e.g `[object String]` or `[object Array]`:

```
function is(type, obj) {
    var clas = Object.prototype.toString.call(obj).slice(8, -1);
    return obj !== undefined && obj !== null && clas === type;
}

is('String', 'test'); // true
is('String', new String('test')); // true
```

In the above example, `Object.prototype.toString` gets called with the value of this being set to the object whose `[[Class]]` value should be retrieved.

**ES5 Note:** For convenience the return value of `Object.prototype.toString` for both `null` and `undefined` was **changed** from `Object` to `Null` and `Undefined` in ECMAScript 5.

### Testing For Undefined Variables

```
typeof foo !== 'undefined'
```

The above will check whether `foo` was actually declared or not; just referencing it would result in a `ReferenceError`. This is the only thing `typeof` is actually useful for.

### In Conclusion

In order to check the type of an object, it is highly recommended to use `Object.prototype.toString` because this is the only reliable way of doing so. As shown in the above type table, some return values of `typeof` are not defined in the specification; thus, they can differ between implementations.

Unless checking whether a variable is defined, `typeof` should be avoided.

## The `instanceof` Operator

The `instanceof` operator compares the constructors of its two operands. It is only useful when comparing custom made objects. Used on built-in types, it is nearly as useless as the typeof operator.

### Comparing Custom Objects

```
function Foo() {}
function Bar() {}
Bar.prototype = new Foo();

new Bar() instanceof Bar; // true
new Bar() instanceof Foo; // true

// This just sets Bar.prototype to the function object Foo,
// but not to an actual instance of Foo
Bar.prototype = Foo;
new Bar() instanceof Foo; // false
```

### Using `instanceof` With Native Types

```
new String('foo') instanceof String; // true
new String('foo') instanceof Object; // true

'foo' instanceof String; // false
'foo' instanceof Object; // false
```

One important thing to note here is that `instanceof` does not work on objects that originate from different JavaScript contexts (e.g. different documents in a web browser), since their constructors will not be the exact same object.

### In Conclusion

The `instanceof` operator should **only** be used when dealing with custom made objects that originate from the same JavaScript context. Just like the `typeof` operator, every other use of it should be **avoided**.

## Type Casting

JavaScript is a *weakly typed* language, so it will apply *type coercion* **wherever** possible.

```
// These are true
new Number(10) == 10; // Number object is converted
                      // to a number primitive via implicit call of
                      // Number.prototype.valueOf method

10 == '10';           // Strings gets converted to Number
10 == '+10 ';         // More string madness
10 == '010';          // And more
isNaN(null) == false; // null converts to 0
                      // which of course is not NaN

// These are false
10 == 010;
10 == '-10';
```

To avoid the issues above, use of the strict equal operator is **highly** recommended. Although this avoids a lot of common pitfalls, there are still many further issues that arise from JavaScript's weak typing system.

**ES5 Note:** Number literals that start with a `0` are interpreted as octal (Base 8). Octal support for these has been **removed** in ECMAScript 5 strict mode.

### Constructors Of Built-In Types

The constructors of the built in types like `Number` and `String` behave differently when being used with the `new` keyword and without it.

```
new Number(10) === 10;     // False, Object and Number
Number(10) === 10;         // True, Number and Number
new Number(10) + 0 === 10; // True, due to implicit conversion
```

Using a built-in type like `Number` as a constructor will create a new `Number` object, but leaving out the `new` keyword will make the `Number` function behave like a converter.

In addition, passing literals or non-object values will result in even more type coercion.

The best option is to cast to one of the three possible types **explicitly**.

### Casting To A String

```
'' + 10 === '10'; // true
```

By prepending an empty string, a value can easily be cast to a string.

### Casting To A Number

```
+'10' === 10; // true
```

Using the **unary** plus operator, it is possible to cast to a number.

### Casting To A Boolean

By using the **not** operator twice, a value can be converted to a boolean.

```
!!'foo';   // true
!!'';      // false
!!'0';     // true
!!'1';     // true
!!'-1'     // true
!!{};      // true
!!true;    // true
```

# Core

## Why Not To Use `eval`

The `eval` function will execute a string of JavaScript code in the local scope.

```
var number = 1;
function test() {
    var number = 2;
    eval('number = 3');
    return number;
}
test(); // 3
number; // 1
```

However, `eval` only executes in the local scope when it is being called directly *and* when the name of the called function is actually `eval`.

```
var number = 1;
function test() {
    var number = 2;
    var copyOfEval = eval;
    copyOfEval('number = 3');
    return number;
}
test(); // 2
number; // 3
```

The use of `eval` should be avoided. 99.9% of its "uses" can be achieved **without** it.

### `eval` In Disguise

The timeout functions `setTimeout` and `setInterval` can both take a string as their first argument. This string will **always** get executed in the global scope since `eval` is not being called directly in that case.

### Security Issues

`eval` also is a security problem, because it executes **any** code given to it. It should **never** be used with strings of unknown or untrusted origins.

### In Conclusion

`eval` should never be used. Any code that makes use of it should be questioned in its workings, performance and security. If something requires `eval` in order to work, it should **not** be used in the first place. A *better design* should be used, that does not require the use of `eval`.

---

## `undefined` And `null`

JavaScript has two distinct values for nothing, `null` and `undefined`, with the latter being more useful.

### The Value `undefined`

`undefined` is a type with exactly one value: `undefined`.

The language also defines a global variable that has the value of `undefined`; this variable is also called `undefined`. However, this variable is **neither** a constant nor a keyword of the language. This means that its *value* can be easily overwritten.

**ES5 Note:** `undefined` in ECMAScript 5 is **no longer** *writable* in strict mode, but its name can still be shadowed by for example a function with the name `undefined`.

Here are some examples of when the value `undefined` is returned:

- Accessing the (unmodified) global variable `undefined`.
- Accessing a declared *but not* yet initialized variable.
- Implicit returns of functions due to missing `return` statements.
- `return` statements that do not explicitly return anything.
- Lookups of non-existent properties.
- Function parameters that do not have any explicit value passed.
- Anything that has been set to the value of `undefined`.
- Any expression in the form of `void(expression)`

### Handling Changes To The Value Of `undefined`

Since the global variable `undefined` only holds a copy of the actual *value* of `undefined`, assigning a new value to it does **not** change the value of the *type* `undefined`.

Still, in order to compare something against the value of `undefined`, it is necessary to retrieve the value of `undefined` first.

To protect code against a possible overwritten `undefined` variable, a common technique used is to add an additional parameter to an anonymous wrapper that gets no argument passed to it.

```
var undefined = 123;
(function(something, foo, undefined) {
    // undefined in the local scope does
    // now again refer to the value `undefined`

})('Hello World', 42);
```

Another way to achieve the same effect would be to use a declaration inside the wrapper.

```
var undefined = 123;
(function(something, foo) {
    var undefined;
    ...

})('Hello World', 42);
```

The only difference here is that this version results in 4 more bytes being used in case it is minified, and there is no other `var` statement inside the anonymous wrapper.

### Uses Of `null`

While `undefined` in the context of the JavaScript language is mostly used in the sense of a traditional *null*, the actual `null` (both a literal and a type) is more or less just another data type.

It is used in some JavaScript internals (like declaring the end of the prototype chain by setting `Foo.prototype = null`), but in almost all cases, it can be replaced by `undefined`.

---

## Automatic Semicolon Insertion

Although JavaScript has C style syntax, it does **not** enforce the use of semicolons in the source code, so it is possible to omit them.

JavaScript is not a semicolon-less language. In fact, it needs the semicolons in order to understand the source code. Therefore, the JavaScript parser **automatically** inserts them whenever it encounters a parse error due to a missing semicolon.

```
var foo = function() {
} // parse error, semicolon expected
test()
```

Insertion happens, and the parser tries again.

```
var foo = function() {
}; // no error, parser continues
test()
```

The automatic insertion of semicolon is considered to be one of **biggest** design flaws in the language because it *can* change the behavior of code.

### How It Works

The code below has no semicolons in it, so it is up to the parser to decide where to insert them.

```
(function(window, undefined) {
    function test(options) {
        log('testing!')

        (options.list || []).forEach(function(i) {

        })

        options.value.test(
            'long string to pass here',
            'and another long string to pass'
        )

        return
        {
            foo: function() {}
        }
    }
    window.test = test

})(window)

(function(window) {
    window.someLibrary = {}

})(window)
```

Below is the result of the parser's "guessing" game.

```
(function(window, undefined) {
    function test(options) {

        // Not inserted, lines got merged
        log('testing!')(options.list || []).forEach(function(i) {

        }); // <- inserted

        options.value.test(
            'long string to pass here',
            'and another long string to pass'
        ); // <- inserted

        return; // <- inserted, breaks the return statement
        { // treated as a block

            // a label and a single expression statement
            foo: function() {}
        }; // <- inserted
    }
    window.test = test; // <- inserted

// The lines got merged again
})(window)(function(window) {
    window.someLibrary = {}; // <- inserted

})(window); //<- inserted
```

The parser drastically changed the behavior of the code above. In certain cases, it does the **wrong thing**.

**Note:** The JavaScript parser does not "correctly" handle return statements that are followed by a new line. While this is not necessarily the fault of the automatic semicolon insertion, it can still be an unwanted side-effect.

### Leading Parenthesis

In case of a leading parenthesis, the parser will **not** insert a semicolon.

```
log('testing!')
(options.list || []).forEach(function(i) {})
```

This code gets transformed into one line.

```
log('testing!')(options.list || []).forEach(function(i) {})
```

Chances are **very** high that `log` does **not** return a function; therefore, the above will yield a `TypeError` stating that `undefined is not a function`.

### In Conclusion

It is highly recommended to **never** omit semicolons. It is also recommended that braces be kept on the same line as their corresponding statements and to never omit them for single-line `if` / `else` statements. These measures will not only improve the consistency of the code, but they will also prevent the JavaScript parser from changing code behavior.

## The `delete` Operator

In short, it's *impossible* to delete global variables, functions and some other stuff in JavaScript which have a `DontDelete` attribute set.

### Global Code And Function Code

When a variable or a function is defined in a global or a function scope it is a property of either the Activation object or the Global object. Such properties have a set of attributes, one of which is `DontDelete`. Variable and function declarations in global and function code always create properties with `DontDelete`, and therefore cannot be deleted.

```
// global variable:
var a = 1; // DontDelete is set
delete a; // false
a; // 1

// normal function:
function f() {} // DontDelete is set
delete f; // false
typeof f; // "function"

// reassigning doesn't help:
f = 1;
delete f; // false
f; // 1
```

### Explicit Properties

Explicitly set properties can be deleted normally.

```
// explicitly set property:
var obj = {x: 1};
obj.y = 2;
delete obj.x; // true
delete obj.y; // true
obj.x; // undefined
obj.y; // undefined
```

In the example above, `obj.x` and `obj.y` can be deleted because they have no `DontDelete` attribute. That's why the example below works too.

```
// this works fine, except for IE:
var GLOBAL_OBJECT = this;
GLOBAL_OBJECT.a = 1;
a === GLOBAL_OBJECT.a; // true - just a global var
delete GLOBAL_OBJECT.a; // true
GLOBAL_OBJECT.a; // undefined
```

Here we use a trick to delete `a`. `this` here refers to the Global object and we explicitly declare variable `a` as its property which allows us to delete it.

IE (at least 6-8) has some bugs, so the code above doesn't work.

### Function Arguments And Built-Ins

Functions' normal arguments, `arguments` objects and built-in properties also have `DontDelete` set.

```
// function arguments and properties:
(function (x) {

  delete arguments; // false
  typeof arguments; // "object"

  delete x; // false
  x; // 1

  function f(){}
  delete f.length; // false
  typeof f.length; // "number"

})(1);
```

### Host Objects

The behaviour of `delete` operator can be unpredictable for hosted objects. Due to the specification, host objects are allowed to implement any kind of behavior.

### In Conclusion

The `delete` operator often has unexpected behaviour and can only be safely used to delete explicitly set properties on normal objects.

# Other

## `setTimeout` And `setInterval`

Since JavaScript is asynchronous, it is possible to schedule the execution of a function using the `setTimeout` and `setInterval` functions.

```
function foo() {}
var id = setTimeout(foo, 1000); // returns a Number > 0
```

**Note:** Timeouts are **not** part of the ECMAScript standard. They were implemented in BOM, or DOM Level 0, which are never defined nor documented

When `setTimeout` is called, it returns the ID of the timeout and schedule `foo` to run **approximately** one thousand milliseconds in the future. `foo` will then be executed **once**.

Depending on the timer resolution of the JavaScript engine running the code, as well as the fact that JavaScript is single threaded and other code that gets executed might block the thread, it is by **no means** a safe bet that one will get the exact delay specified in the `setTimeout` call.

The function that was passed as the first parameter will get called by the *global object*, which means that `this` inside the called function refers to the global object.

```
function Foo() {
    this.value = 42;
    this.method = function() {
        // this refers to the global object
        console.log(this.value); // will log undefined
    };
    setTimeout(this.method, 500);
}
new Foo();
```

formally. No recommended specification has been published so far, however, they are currently being standardized by HTML5. Due to this nature, the implementation may vary from browsers and engines.

### Stacking Calls With `setInterval`

While `setTimeout` only runs the function once, `setInterval` - as the name suggests - will execute the function **every** `X` milliseconds, but its use is discouraged.

When code that is being executed blocks the timeout call, `setInterval` will still issue more calls to the specified function. This can, especially with small intervals, result in function calls stacking up.

```
function foo(){
    // something that blocks for 1 second
}
setInterval(foo, 100);
```

**Note:** As `setTimeout` takes a **function object** as its first parameter, a common mistake is to use `setTimeout(foo(), 1000)`, which will use the **return value** of the call `foo` and **not** `foo`. This is, most of the time, a silent error, since when the function returns `undefined` `setTimeout` will **not** raise any error.

In the above code, `foo` will get called once and will then block for one second.

While `foo` blocks the code, `setInterval` will still schedule further calls to it. Now, when `foo` has finished, there will already be **ten** further calls to it waiting for execution.

### Dealing With Possible Blocking Code

The easiest solution, as well as most controllable solution, is to use `setTimeout` within the function itself.

```
function foo(){
    // something that blocks for 1 second
    setTimeout(foo, 100);
}
foo();
```

Not only does this encapsulate the `setTimeout` call, but it also prevents the stacking of calls and gives additional control. `foo` itself can now decide whether it wants to run again or not.

### Manually Clearing Timeouts

Clearing timeouts and intervals works by passing the respective ID to `clearTimeout` or `clearInterval`, depending on which `set` function was used in the first place.

```
var id = setTimeout(foo, 1000);
clearTimeout(id);
```

### Clearing All Timeouts

As there is no built-in method for clearing all timeouts and/or intervals, it is necessary to use brute force in order to achieve this functionality.

```
// clear "all" timeouts
for(var i = 1; i < 1000; i++) {
    clearTimeout(i);
}
```

But there might still be timeouts that are unaffected by this arbitrary number. Another way of doing this is to consider that the ID given to a timeout is incremented by one every time you call `setTimeout`.

```
// clear "all" timeouts
var biggestTimeoutId = window.setTimeout(function(){}, 1),
i;
for(i = 1; i <= biggestTimeoutId; i++) {
    clearTimeout(i);
}
```

Even though this works on all major browsers today, it isn't specified that the IDs should be ordered that way and it may change. Therefore, it is instead recommended to keep track of all the timeout IDs, so they can be cleared specifically.

### Hidden Use Of `eval`

`setTimeout` and `setInterval` can also take a string as their first parameter. This feature should **never** be used because it internally makes use of `eval`.

```
function foo() {
    // will get called
}

function bar() {
    function foo() {
        // never gets called
    }
    setTimeout('foo()', 1000);
}
bar();
```

**Note:** The exact workings when a string is passed to them might differ in various JavaScript implementations. For example, Microsoft's JScript uses the `Function` constructor in place of `eval`.

Since `eval` is not getting called directly in this case, the string passed to `setTimeout` will be executed in the *global scope*; thus, it will not use the local variable `foo` from the scope of `bar` .

It is further recommended to **not** use a string to pass arguments to the function that will get called by either of the timeout functions.

```
function foo(a, b, c) {}

// NEVER use this
setTimeout('foo(1, 2, 3)', 1000)

// Instead use an anonymous function
setTimeout(function() {
    foo(1, 2, 3);
}, 1000)
```

### In Conclusion

A string should **never** be used as the parameter of `setTimeout` or `setInterval` . It is a clear sign of **really** bad code, when arguments need to be supplied to the function that gets called. An *anonymous function* should be passed that then takes care of the actual call.

Furthermore, the use of `setInterval` should be avoided because its scheduler is not blocked by executing JavaScript.

**Note:** While it is also possible to use `setTimeout(foo, 1000, 1, 2, 3)` syntax, it is not recommended, as its use may lead to subtle errors when used with methods. Furthermore, the syntax might not work in some JavaScript implementations. For example, Microsoft's Internet Explorer does **not** pass the arguments directly to the callback.