

Is JavaScript's "new" keyword considered harmful?

Asked 16 years ago Modified 2 years ago Viewed 122k times



597



In [another question](#), a user pointed out that the `new` keyword was dangerous to use and proposed a solution to object creation that did not use `new`. I didn't believe that was true, mostly because I've used [Prototype](#), [Script.aculo.us](#) and other excellent JavaScript libraries, and everyone of them used the `new` keyword.

In spite of that, yesterday I was watching Douglas Crockford's talk at YUI theater and he said the exactly same thing, that he didn't use the `new` keyword anymore in his code ([Crockford on JavaScript - Act III: Function the Ultimate - 50:23 minutes](#)).

Is it 'bad' to use the `new` keyword? What are the advantages and disadvantages of using it?

javascript

Share

Improve this question

Follow

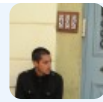
edited Dec 3, 2022 at 21:18



Peter Mortensen

31.6k ● 22 ● 109 ● 133

asked Dec 20, 2008 at 15:20



Pablo Fernandez


105k ● 58 ● 195 ● 233

97 It is NOT 'bad' to use the new keyword. But if you forget it, you will be calling the object constructor as a regular function. If your constructor doesn't check its execution context then it won't notice that 'this' points to different object (ordinarily the global object) instead of the new instance. Therefore your constructor will be adding properties and methods to the global object (window). If you always check that 'this' is an instance of your object in the object function, then you won't ever have this problem. – [Kris](#) Jan 7, 2010 at 4:01

6 I don't understand this. On one hand doug discourages use of `new`. But when you look at the YUI library. You have to use `new` everywhere. Such as `var myDataSource = new Y.DataSource.IO({source:"./myScript.php"});`.
– [aditya_gaur](#) Aug 8, 2011 at 18:14 ✎

71 I don't think this should have been closed. Yes, it's likely to inspire some Crockford anti-fan venom but we are talking about popular advice to basically avoid a major language feature here. The mechanism that makes every JQuery object weigh almost nothing (where memory is concerned) involves using the 'new' keyword. Prefer factory methods to constantly invoking new, but don't drastically reduce your architectural options and performance potential by only using object literals. They have their place and constructors have their place. It is hands-down dated and lousy advice.
– [Erik Reppen](#) Jul 19, 2012 at 20:09

5 TLDR: Using `new` isn't dangerous. Omitting `new` is dangerous, and therefore [bad](#). But in ES5 you can use [Strict Mode](#), which protects you from that danger and many others.
– [jkdev](#) Jul 11, 2016 at 8:08 ✎

- 3 This entire thread is outdated. The current best practice is to use ES6 modules and, for inheritance, ES6 classes. They're in strict mode by default, which would imply that `this` won't be `globalThis` in a function call, but it goes beyond that: classes cannot be used without `new` at all. Since ES6, the language has moved on to *encourage* the use of `new`, as seen by newer constructors such as `Map` or `Set` not being usable (anymore) without `new`. Going against this doesn't seem reasonable. Note that *old* constructors, e.g. `new String`, `new Date`, `new Array`, have *old* rules.
- [Sebastian Simon](#) Jan 10, 2022 at 0:14 

13 Answers

Sorted by:

Highest score (default)



627



Crockford has done a lot to popularize good JavaScript techniques. His opinionated stance on key elements of the language have sparked many useful discussions. That said, there are far too many people that take each proclamation of "bad" or "harmful" as gospel, refusing to look beyond one man's opinion. It can be a bit frustrating at times.

Use of the functionality provided by the `new` keyword has several advantages over building each object from scratch:

1. [Prototype inheritance](#). While often looked at with a mix of suspicion and derision by those accustomed to class-based OO languages, JavaScript's native inheritance technique is a simple and surprisingly effective means of code re-use. And the `new`

keyword is the canonical (and only available cross-platform) means of using it.

2. Performance. This is a side-effect of #1: if I want to add 10 methods to every object I create, I *could* just write a creation function that manually assigns each method to each new object... Or, I could assign them to the creation function's `prototype` and use `new` to stamp out new objects. Not only is this faster (no code needed for each and every method on the prototype), it avoids ballooning each object with separate properties for each method. On slower machines (or especially, slower JS interpreters) when many objects are being created this can mean a significant savings in time and memory.

And yes, `new` has one crucial disadvantage, ably described by other answers: if you forget to use it, your code will break without warning. Fortunately, that disadvantage is easily mitigated - simply add a bit of code to the function itself:

```
function foo()  
{  
  // if user accidentally omits the new keyword, this  
  // silently correct the problem...  
  if ( !(this instanceof foo) )  
    return new foo();  
  
  // constructor logic follows...  
}
```

Now you can have the advantages of `new` without having to worry about problems caused by accidentally misuse.

John Resig goes into detail on this technique in his [Simple "Class" Instantiation](#) post, as well as including a means of building this behavior into your "classes" by default. Definitely worth a read... as is his upcoming book, [Secrets of the JavaScript Ninja](#), which finds hidden gold in this and many other "harmful" features of the JavaScript language (the **chapter** on `with` is especially enlightening for those of us who initially dismissed this much-maligned feature as a gimmick).

A general-purpose sanity check

You could even add an assertion to the check if the thought of broken code silently working bothers you. Or, as [some](#) commented, use the check to introduce a runtime exception:

```
if ( !(this instanceof arguments.callee) )  
    throw new Error("Constructor called as a function")
```

Note that this snippet is able to avoid hard-coding the constructor function name, as unlike the previous example it has no need to actually instantiate the object - therefore, it can be copied into each target function without modification.

ES5 taketh away

As [Sean McMillan](#), [stephenbez](#) and [jrh](#) noted, the use of `arguments.callee` is invalid in ES5's [strict mode](#). So the

above pattern will throw an error if you use it in that context.

ES6 and an entirely harmless new

ES6 introduces [Classes](#) to JavaScript - no, not in the weird Java-aping way that old-school Crockford did, but in spirit much more like the light-weight way he (and others) later adopted, taking the best parts of prototypal inheritance and baking common patterns into the language itself.

...and part of that includes a safe new:

```
class foo
{
  constructor()
  {
    // constructor logic that will ONLY be hit
    // if properly constructed via new
  }
}

// bad invocation
foo(); // throws,
// Uncaught TypeError: class constructors must be invo
```

But what if you don't *want* to use the new sugar? What if you just want to update your perfectly fine old-style prototypal code with the sort of safety checks shown above such that they keep working in strict mode?

Well, as [Nick Parsons notes](#), ES6 provides a handy check for that as well, in the form of `new.target`:

```
function foo()
{
  if ( !(new.target) )
    throw new Error("Constructor called as a function");

  // constructor logic follows...
}
```

So whichever approach you choose, you can - with a bit of thought and good hygiene - use `new` without harm.

Share Improve this answer

edited Apr 13, 2022 at 2:42

Follow

answered Dec 20, 2008 at 17:17




[Shog9](#)

159k ● 36 ● 235 ● 240

103 if (!(this instanceof arguments.callee)) throw Error("Constructor called as a function");// More generic, don't require knowledge of the constructors name, make the user fix the code. – [some](#) Dec 20, 2008 at 17:47

5 If you're worried about performance - and actually have reason to be - then don't do the check at all. Either remember to use `new`, or use a wrapper function to remember it for you. I *suspect* that if you're at the point where it matters, you've already exhausted other optimizations such as memoization, so your creation calls will be localized anyway... – [Shog9](#) Nov 16, 2009 at 21:23

66 Using `arguments.callee` to check if you've been called with `new` may not be so good, since `arguments.callee` is not available in strict mode. Better to use the function name. – [Sean McMillan](#) Jul 5, 2011 at 16:26

74 If I misspell an identifier, my code breaks. Should I not use identifiers? Not using `new` because you can forget to write it in your code is just as ridiculous as my example.
– [Thomas Eding](#) Nov 8, 2011 at 0:30 

17 If you turn on strict mode, if you forget to use `new` you will get an exception when you try to use this:
yuiblog.com/blog/2010/12/14/strict-mode-is-coming-to-town
That is easier than adding an instance of check to each constructor. – [stephenbez](#) Mar 7, 2013 at 23:27



190

I have just read some parts of [Crockford](#)'s book "[JavaScript: The Good Parts](#)". I get the feeling that he considers everything that ever has bitten him as harmful:



About switch fall through:



I never allow switch cases to fall through to the next case. I once found a bug in my code caused by an unintended fall through immediately after having made a vigorous speech about why fall through was sometimes useful. (page 97, ISBN 978-0-596-51774-8)

About ++ and --:

The ++ (increment) and -- (decrement) operators have been known to contribute to bad code by encouraging excessive trickiness. They are

second only to faulty architecture in enabling viruses and other security menaces. (page 122)

About new:

If you forget to include the *new* prefix when calling a constructor function, then *this* will not be bound to the new object. Sadly, *this* will be bound to the global object, so instead of augmenting your new object, you will be clobbering global variables. That is really bad. There is no compile warning, and there is no runtime warning. (page 49)

There are more, but I hope you get the picture.

My answer to your question: **No, it's not harmful.** but if you forget to use it when you should you could have some problems. If you are developing in a good environment you notice that.

In the 5th edition of ECMAScript there is support for [strict mode](#). In strict mode, `this` is no longer bound to the global object, but to `undefined`.

Share Improve this answer

Follow

edited Dec 3, 2022 at 21:29



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Dec 20, 2008 at 15:43



some

49.5k ● 14 ● 81 ● 95

4 I completely agree. The solution: Always document how users have to instantiate your objects. Use an example and users will likely cut/paste. There are constructs/features in every language that can be misused resulting in unusual/unexpected behavior. It doesn't make them harmful. – [nicerobot](#) Dec 20, 2008 at 15:52

48 There is a convention to always start constructors with an upper case letter, and all other functions with a lower case letter. – [some](#) Dec 20, 2008 at 15:59

64 I just realized that Crockford don't consider WHILE harmful... I don't know how many time I have created an infinitive loop because I have forgot to increment a variable... – [some](#) Dec 20, 2008 at 16:01

5 Same goes for ++, --. They express exactly what I intend in the clearest possible language. I love'em! switch fall throughs might be clear to some, but I'm weary about those. I use them when they are clearly more clear ('cuse the pun, couldn't resist). – [PEZ](#) Dec 20, 2008 at 16:23

33 My answer to Crockford: Programming is hard, lets go shopping. Sheesh! – [Jason Jackson](#) Dec 20, 2008 at 16:49



93

JavaScript being a dynamic language, there are a zillion ways to mess up where another language would stop you.



Avoiding a fundamental language feature such as `new` on the basis that you might mess up is a bit like removing





your shiny new shoes before walking through a minefield just in case you might get your shoes muddy.

I use a convention where function names begin with a lowercase letter and 'functions' that are actually class definitions begin with an uppercase letter. The result is a really quite compelling visual clue that the 'syntax' is wrong:

```
var o = MyClass(); // This is clearly wrong.
```

On top of this, good naming habits help. After all, functions do things and therefore there should be a verb in its name whereas classes represent objects and are nouns and adjectives without any verb.

```
var o = chair() // Executing chair is daft.  
var o = createChair() // Makes sense.
```

It's interesting how Stack Overflow's syntax colouring has interpreted the code above.

[Share](#) [Improve this answer](#)

[Follow](#)

edited Dec 3, 2022 at 21:56



[Peter Mortensen](#)

31.6k ● 22 ● 109 ● 133

answered Dec 20, 2008 at 17:10



[AnthonyWJones](#)

189k ● 35 ● 235 ● 307

8 Yeah, I was just thinking the same thing about the syntax coloring. – [BobbyShaftoe](#) Dec 20, 2008 at 18:00

5 Once I forgot to type the word 'function'. The word should be avoided at all costs. Another time I didn't use curly braces in a multi-line if/then statement. So now I don't use curly braces and just write one-line conditionals. – [Hal50000](#) Dec 2, 2015 at 15:49

1 "This is clearly wrong". Why? For my classes (which simply do `if (! this instanceof MyClass) return new MyClass())` I actually prefer the `new`-less syntax. Why? Because *functions* are more generic than *constructor functions*. Leaving out `new` makes it easy to change a constructor function into a regular function... Or the other way around. Adding in `new` makes the code more specific than it needs to be. And hence less flexible. Compare with Java where it's recommended to accept `List` instead of `ArrayList` so the caller can choose the implementation. – [Stijn de Witt](#) Apr 17, 2017 at 19:33

I don't know about JavaScript, but for Java, [the \(new\) syntax highlighter](#) definitely expects Java code conventions to be followed, incl. casing of identifiers and space before and after the equal sign. – [Peter Mortensen](#) Dec 3, 2022 at 22:03



41

I am newbie to JavaScript so maybe I am just not too experienced in providing a good view point to this. Yet I want to share my view on this "new" thing.



I have come from the C# world where using the keyword "new" is so natural that it is the factory design pattern that looks weird to me.





When I first code in JavaScript, I don't realize that there is the "new" keyword and code like the one in [YUI](#) pattern and it doesn't take me long to run into disaster. I lose track of what a particular line is supposed to be doing when looking back the code I've written. More chaotic is that my mind can't really transit between object instances boundaries when I am "dry-running" the code.

Then, I found the "new" keyword which, to me, "separates" things. With the *new* keyword, it creates things. Without the *new* keyword, I know I won't confuse it with creating things unless the function I am invoking gives me strong clues of that.

For instance, with `var bar=foo();` I don't have any clues as what *bar* could possibly be.... Is it a return value or is it a newly created object? But with `var bar = new foo();` I know for sure *bar* is an object.

Share Improve this answer

Follow

edited Dec 8, 2022 at 18:28



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Dec 22, 2008 at 9:22



Conrad

733 ● 4 ● 7

4 Agreed, and I believe the factory pattern should follow a naming convention like makeFoo() – [pluckyglen](#) May 14, 2009 at 21:55

3 +1 - the presence of 'new' gives a clearer statement of intent. – [belugabob](#) Jun 17, 2009 at 7:24

- 4 This response is kind of weird, when almost everything in JS is an object. Why do you need to know for sure that a function is an object when all functions are objects?
– [Joshua Ramirez](#) Jul 9, 2012 at 15:28 ✎
-

@JoshuaRamirez The point is not that `typeof new foo() == "object"`. It's that `new` returns an instance of `foo`, and you know you can call `foo.bar()` and `foo.bang()`. However that can easily be mitigated by using [JsDoc's @return](#) Not that I'm advocating for procedural code (avoiding the word `new`) – [Ruan Mendes](#) Mar 8, 2013 at 17:18 ✎

- 1 @JuanMendes Hmm... Your post made it seem to me that you liked `new` because of its explicit keyword in your code base. I can dig that. That's why I use a module pattern. I'll have a function called `createFoo` or `NewFoo` or `MakeFoo`, doesn't matter as long as it's explicit. Inside that I declare variables which are used as closures inside of an object literal that is returned from the function. That object literal ends up being your object, and the function was just a construction function. – [Joshua Ramirez](#) Apr 16, 2013 at 18:20
-



40

Another case **for** `new` is what I call [Pooh Coding](#). [Winnie-the-Pooh](#) follows his tummy. I say go **with** the language you are using, not **against** it.



Chances are that the maintainers of the language will optimize the language for the idioms they try to encourage. If they put a new keyword into the language they probably think it makes sense to be clear when creating a new instance.



Code written following the language's intentions will increase in efficiency with each release. And code avoiding the key constructs of the language will suffer with time.

And this goes well beyond performance. I can't count the times I've heard (or said) "why the hell did they do **that**?" when finding strange looking code. It often turns out that at the time when the code was written there was some "good" reason for it. Following the Tao of the language is your best insurance for not having your code ridiculed some years from now.

Share Improve this answer

Follow

edited Dec 8, 2022 at 18:21



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Dec 20, 2008 at 18:25



PEZ

17k ● 7 ● 47 ● 66

3 +1 for the Pooh Coding link - now i need to find excuses to work this into my conversations... – [Shog9](#) Dec 20, 2008 at 19:06

LOL! I have years of experience in that particular field and I can assure you that you'll find no difficulties. They often call me Pooh over at robowiki.net. =) – [PEZ](#) Dec 20, 2008 at 19:22

1 Don't know if you'll ever see this comment, but that Pooh Coding link is dead. – [Calvin](#) Apr 26, 2009 at 23:50

Thanks for the heads up. We migrated robowiki.net to a new wiki last week and the old content is unreachable at the



I wrote a post on how to mitigate the problem of calling a constructor without the *new* keyword.

25



It's mostly didactic, but it shows how you can create constructors that work with or without *new* and doesn't require you to add [boilerplate code](#) to test *this* in every constructor.



[Constructors without using "new"](#)

Here's the gist of the technique:

```
/**
 * Wraps the passed in constructor so it works with
 * or without the new keyword
 * @param {Function} realCtor The constructor function
 * Note that this is going to be wrapped
 * and should not be used directly
 */
function ctor(realCtor) {
  // This is going to be the actual constructor
  return function wrapperCtor() {
    var obj; // The object that will be created
    if (this instanceof wrapperCtor) {
      // Called with new
      obj = this;
    } else {
      // Called without new. Create an empty object of
      // correct type without running that constructor
      surrogateCtor.prototype = wrapperCtor.prototype;
      obj = new surrogateCtor();
    }
    // Call the real constructor function
    realCtor.apply(obj, arguments);
  };
}
```



```
    return obj;
  }

  function surrogateCtor() {}
}
```

Here's how to use it:

```
// Create our point constructor
Point = ctor(function(x, y) {
  this.x = x;
  this.y = y;
});

// This is good
var pt = new Point(20, 30);
// This is OK also
var pt2 = Point(20, 30);
```

Share Improve this answer

edited Dec 8, 2022 at 18:40

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jun 20, 2011 at 22:52



Ruan Mendes

92.2k ● 31 ● 158 ● 220

6 avoiding `arguments.callee` is awesome! – Gregg Lind
Feb 1, 2012 at 19:29

I maintained a collection of similar Crockford-inspired utilities, and found that it always had me running to find the implementations when I start a new project. And this wraps such a fundamental part of programming: object instantiation. All this work, just to enable haphazard instantiation code? I appreciate the ingenuity of this wrapper, honestly, but it just

seems to engender careless coding. – [Joe Coder](#) Jun 25, 2016 at 1:25

- 1 @joecoder I did mention this was for didactic purposes only, I do not subscribe to this paranoia style of coding. However, if I was writing a class library, then yes I would add this feature in a way that is transparent to callers – [Ruan Mendes](#) Jun 25, 2016 at 14:41
-



The rationale behind not using the *new* keyword, is simple:

22



By not using it at all, you avoid the pitfall that comes with accidentally omitting it. The construction pattern that [YUI](#) uses, is an example of how you can avoid the *new* keyword altogether:



```
var foo = function () {  
    var pub = { };  
    return pub;  
}  
var bar = foo();
```

Alternatively, you could do this:

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

But by doing so you run risk of someone forgetting to use the **new** keyword, and the **this** operator being all [FUBAR](#). As far as I know, there isn't any advantage to doing this (other than you being used to it).

At The End Of The Day: **It's about being defensive.** Can you use the new statement? Yes. Does it make your code more dangerous? Yes.

If you have ever written C++, it's akin to setting pointers to NULL after you delete them.

Share Improve this answer

edited Dec 3, 2022 at 21:22

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Dec 20, 2008 at 15:42



Greg Dean

30k ● 15 ● 69 ● 79

-
- 6 No. With "new foo()" some properties are set on the returned object, like constructor. – [some](#) Dec 20, 2008 at 16:44
-
- 37 So, just to make this clear: you shouldn't use "new" because you might forget it? You are kidding, right? – [Bombe](#) Dec 20, 2008 at 17:08
-
- 18 @Bombe : in other languages forgetting "new" would result in an error. In Javascript, it just keeps on trucking. You can forget, and never realise. And simply looking at erroneous code *wont* be obvious at all whats going wrong.
– [Kent Fredric](#) Dec 20, 2008 at 17:14
-
- 4 @Greg: I don't see how the first technique allows for use of prototype chains - object literals are great, but throwing away the performance and other advantages provided by prototypes out of fear seems a bit silly. – [Shog9](#) Dec 20, 2008 at 17:19
-

5 @Bombe - You should use "new" because you (and anyone that uses your code) will never make a mistake? You are kidding, right? – [Greg Dean](#) Dec 20, 2008 at 17:34



20



I think "new" adds clarity to the code. And clarity is worth everything. It is good to know there are pitfalls, but avoiding them by avoiding clarity doesn't seem like the way for me.

Share Improve this answer

edited Dec 3, 2022 at 21:30



Follow



[Peter Mortensen](#)

31.6k ● 22 ● 109 ● 133



answered Dec 20, 2008 at 16:12



[PEZ](#)

17k ● 7 ● 47 ● 66



15



Case 1: `new` isn't required and should be avoided

```
var str = new String('asd'); // type: object
var str = String('asd');    // type: string

var num = new Number(12);    // type: object
var num = Number(12);        // type: number
```



Case 2: `new` is required, otherwise you'll get an error

```
new Date().getFullYear(); // correct, returns the
Date().getFullYear();    // invalid, returns an e
```

Share Improve this answer

edited Feb 26, 2011 at 17:27

Follow

answered Nov 13, 2010 at 17:04



user492203

-
- 5 It should be noted that in case 1, calling the constructor as a function only makes sense if you are intending type conversion (and do not know if the hull object has a conversion method, such as `toString()`). In all other cases, use literals. Not `String('asd')`, but simply `'asd'`, and not `Number(12)`, but simply `12`.
- [PointedEars](#) Sep 5, 2012 at 11:21

-
- 1 @PointedEars One exception to this rule would be `Array`: `Array(5).fill(0)` is obviously more readable than `[undefined, undefined, undefined, undefined, undefined].fill(0)` and `(var arr = []).length = 5; arr.fill(0);` – [yyny](#) Dec 14, 2015 at 21:58

@YoYoYonnY My statement was made in context of case 1 of the answer, which refers to the use of `new` with constructors for object types corresponding to *primitive* types. The literal that you are suggesting is *not* equivalent to the `Array` call (try `0 in ...`), and the rest is a syntax error :) Otherwise you are correct, although it should be noted as well that `Array(5)` can be ambiguous if you consider *non-conforming* implementations (and therefore an *emulated* `Array.prototype.fill(...)`). – [PointedEars](#) Dec 17, 2015 at 15:51

Note that `Date` has special rules to differentiate between `new` being used vs. not being used. It's kind of an outlier. Also, all of these constructors are part of the oldest language specifications. Newer language features are more behaved. All of the answers here need to be updated or deleted...

– [Sebastian Simon](#) Jan 10, 2022 at 0:04



12

Here is the briefest summary I could make of the two strongest arguments for and against using the `new` operator:



Arguments against `new`



1. Functions designed to be instantiated as objects using the `new` operator can have disastrous effects if they are incorrectly invoked as normal functions. A function's code in such a case will be executed in the scope where the function is called, instead of in the scope of a local object as intended. This can cause global variables and properties to get overwritten with disastrous consequences.
2. Finally, writing `function Func()`, and then calling `Func.prototype` and adding stuff to it so that you can call `new Func()` to construct your object seems ugly to some programmers, who would rather use another style of object inheritance for architectural and stylistic reasons.

For more on this argument check out [Douglas Crockford's](#) great and concise book [JavaScript: The Good Parts](#). In

fact, check it out anyway.

Arguments in favor of new

1. Using the new operator along with [prototypal](#) assignment is fast.
2. That stuff about accidentally running a constructor function's code in the global namespace can easily be prevented if you always include a bit of code in your constructor functions to check to see if they are being called correctly, and, in the cases where they aren't, handling the call appropriately as desired.

See [John Resig's post](#) for a simple explanation of this technique, and for a generally deeper explanation of the inheritance model he advocates.

Share Improve this answer

Follow

edited Dec 8, 2022 at 18:38



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Dec 28, 2010 at 23:47



alegscogs

6,715 ● 2 ● 18 ● 9

-
- 1 An update on the against arguments: #1 can be mitigated by using `'use strict';` mode (or the method in your link) #2 Has syntactic sugar in **ES6**, however the behavior is the same as before. – [ninMonkey](#) Nov 11, 2015 at 12:39
-

I agree [with PEZ](#) and some here.



9

It seems obvious to me that "new" is self descriptive object creation, where the [YUI](#) pattern Greg Dean describes is *completely obscured*.



The possibility someone could write `var bar = foo;` or `var bar = baz();` where *baz* isn't an object creating method seems *far* more dangerous.



Share Improve this answer

edited Dec 3, 2022 at 22:07

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Dec 20, 2008 at 17:10



annakata

75.7k ● 18 ● 115 ● 180



6

I think *new* is evil, not because if you forget to use it by mistake it might cause problems, but because it screws up the inheritance chain, making the language tougher to understand.



JavaScript is [prototype-based](#) object-oriented. Hence every object *must* be created from another object like so:



`var newObj=Object.create(oldObj)`. Here *oldObj* is called the prototype of *newObj* (hence "prototype-based"). This implies that if a property is not found in *newObj* then it will be searched in *oldObj*. *newObj* by default will thus be an empty object, but due to its prototype chain, it appears to have all the values of *oldObj*.

On the other hand, if you do `var newObj=new oldObj()`, the prototype of *newObj* is *oldObj.prototype*, which is unnecessarily difficult to understand.

The trick is to use

```
Object.create=function(proto){
  var F = function(){};
  F.prototype = proto;
  var instance = new F();
  return instance;
};
```

It is inside this function and it is only here that *new* should be used. After this, simply use the *Object.create()* method. The method resolves the prototype problem.

Share Improve this answer

edited Dec 8, 2022 at 18:35

Follow

community wiki

3 revs, 3 users 52%

Mihir Gogate

-
- 7 To be honest, I'm not wild about this technique - it doesn't add anything new, and as your answer illustrates it can even end up being a crutch. IMHO, understanding prototype chains and object instantiation is crucial to understanding JavaScript... But if it makes you uncomfortable to use them directly, then using a helper function to take care of some details is fine, so long as you remember what you're doing. FWIW: a (somewhat more useful) variation on this is part of the ECMAScript 5th ed. std, and already available in some

browsers - **so you should be careful not to blindly redefine it!** – [Shog9](#) Aug 31, 2010 at 1:29 

BTW: I'm not sure why you made this CW, but if you want to re-post it with the formatting corrections I made, be careful to avoid that checkbox... – [Shog9](#) Aug 31, 2010 at 1:34

- 3 Unnecessarily difficult to understand? `var c = new Car()` is the same as doing `var c = Object.create(Car.prototype); Car.call(c)`
– [Ruan Mendes](#) Mar 4, 2016 at 18:53



0



In my not-so-humble opinion, "new" is a flawed concept in 2021 JavaScript. It adds words where none are needed.

It makes the return value of a function/constructor implicit and forces the use of *this* in the function/constructor.

Adding noise to code is never a good thing.



```
// With new
function Point(x, y) {
  this.x = x
  this.y = y
}
let point = new Point(0, 0)
```

Vs.

```
// Without new
function Point(x, y) {
  return { x, y }
}
let point = Point(0, 0)
```

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Apr 5, 2021 at 13:23



Chris Buck

755 ● 5 ● 15
