

---

## Operator overloading in JavaScript

Should JavaScript support operator overloading? It's not clear one way or another whether operator overloading has enough benefit in terms of the complexity, in terms of language design, implementation work, and security surface. At the same time, there's an argument that it's better to provide general operator overloading than to overload operators for specific, additional, built-in constructs (and partly for this reason, Decimal was not added as part of ES6, although it might have been useful for JS developers).

This article tries to examine how operator overloading *could* look, if we want to go in this direction. Hopefully, the concreteness will help us decide whether to go down this path, which can help move the committee towards concrete next steps on long-standing feature requests, one way or another.

Status: **Withdrawn**

### Case studies

Operator overloading is all about enabling richer libraries. This section gives four motivating use cases of such rich libraries.

### Numeric types

JavaScript has a very restricted set of numeric types. Traditionally, it had just Number: an IEEE-754 double-precision binary float. The Stage 4 BigInt proposal added a single new numeric type for arbitrary-size integers. But there are more numeric types that developers need in practice, such as decimals, rationals, complex numbers, etc. Operator overloading can provide these, with intuitive syntax for their use.

```
1 // Usage example
2 import Decimal from "./decimal.mjs";
3 with operators from Decimal; // Enable operator overloading for
  decimals
4                                     // Declaration may use some other syntax
5
6 Decimal(1) + Decimal(2)           // ==> Decimal(3)
7 Decimal(3) * Decimal(2)           // ==> Decimal(6)
8 Decimal(1) == Decimal(1)           // ==> true
9 Decimal(1) == 1                     // ==> true
10 1 == Decimal(1)                    // ==> true
11 Decimal(1) === 1                    // ==> false (not overloadable)
```

A possible implementation of this module:

---

```

1 // -----
2 // decimal.mjs
3
4 import Big from './big.mjs'; // https://github.com/MikeMcI/big.js/
5
6 const DecimalOperators = Operators({
7   "+"(a, b) { return a._big.plus(b._big); },
8   "*" (a, b) { return a._big.times(b._big); },
9   "=="(a, b) { return a._big.eq(b._big); },
10 }, {
11   left: Number, {
12     "=="(a, b) { return b._big.eq(a); }
13   }
14 }, {
15   right: Number, {
16     "=="(a, b) { return a._big.eq(b); }
17   }
18 });
19
20 export default
21 class Decimal extends DecimalOperators {
22   _big;
23   constructor(arg) { this._big = new Big(arg); }
24 }
25 Object.preventExtensions(Decimal); // ensure the operators don't
    change

```

## Matrix/vector computations

JavaScript is increasingly used for data processing and analysis, with libraries like `stdlib`. These calculations are made a bit more awkward because things involving vector, matrix and tensor calculations need to be done all via method chaining, rather than more naturally using operators as they can in many other programming languages. Operator overloading could provide that natural phrasing.

```

1 // Usage example
2 import { Vector } from './vector.mjs';
3 with operators from Vector;
4
5 new Vector([1, 2, 3]) + new Vector([4, 5, 6]) // ==> new Vector([5,
    7, 9])
6 3 * new Vector([1, 2, 3]) // ==> new Vector([3,
    6, 9])
7 new Vector([1, 2, 3]) == new Vector([1, 2, 3]) // ==> true
8 (new Vector([1, 2, 3]))[1] // ==> 2

```

A possible implementation:

---

---

```

1 // -----
2 // vector.mjs
3 // This example uses the "Imperative API".
4 // It would also be possible to write with the decorator-based API of
  // the previous example.
5
6 const VectorOps = Operators({
7   "+"(a, b) {
8     return new Vector(a.contents.map((elt, i) => elt + b.contents[i]));
9   },
10  "=="(a, b) {
11    return a.contents.length === b.contents.length &&
12           a.contents.every((elt, i) => elt == b.contents[i]);
13  },
14  "[]"(a, b) {
15    return a.contents[b];
16  }
17 }, {
18   left: Number,
19   "*" (a, b) {
20     return new Vector(b.contents.map(elt => elt * a));
21   }
22 });
23
24 export class Vector extends VectorOps {
25   contents;
26   constructor(contents) {
27     super();
28     this.contents = contents;
29   }
30   get length() { return this.contents.length; }
31 }
32 Object.preventExtensions(Vector); // ensure the operators don't change

```

## Equation DSLs

JavaScript is used in systems with equation-based DSLs, such as TensorFlow.js. In systems like TensorFlow, operators can be used to construct an abstract formula in other programming languages. Operator overloading could allow these formula DSLs to be expressed as infix expressions, as people naturally think of them.

For example, in TensorFlow.js's introductory tutorials, there is an example of an equation definition as follows:

```

1 function predict(x) {
2   // y = a * x ^ 3 + b * x ^ 2 + c * x + d
3   return tf.tidy(() => {
4     return a.mul(x.pow(tf.scalar(3, 'int32')))

```

---

```
5     .add(b.mul(x.square()))
6     .add(c.mul(x))
7     .add(d);
8   });
9 }
```

It's unfortunate that the equation has to be written twice, once to explain it and once to write it in code. With operator overloading and extensible literals, it might be written as follows instead:

```
1 function predict(x) {
2   with operators from tf.equation;
3
4   // y = a * x ^ 3 + b * x ^ 2 + c * x + d
5   return tf.tidy(() => {
6     return a * x ** tf.scalar(3, 'int32')
7       + b * x.square()
8       + c * x
9       + d;
10  });
11 }
```

At this point, maybe you don't even need that comment!

### Ergonomic CSS units calculations

Tab Atkins proposed that CSS support syntax in JavaScript for CSS unit literals and operators. The CSS Typed OM turned out a bit different, with ergonomic affordances but without using new types of literals or operator overloading. With this proposal, in conjunction with extended numeric literals, we could have some more intuitive units calculations than the current function- and method-based solution.

In this case, the `CSSNumericValue` platform objects would come with operator overloading already enabled. Their definition in the CSS Typed OM specification would, indirectly, make use of the same JavaScript mechanism that

```
1 with operators from CSSNumericValue;
2
3 document.querySelector("#element").style.paddingLeft = Css.em(3) + CSS.
  px(2);
```

### Design goals

- Expressivity

- 
- Support operator overloading on both mutable and immutable objects, and in the future, typed objects and value types.
  - Support operands of different types and the same type, as in the above examples.
  - Explain all of JS's behavior on existing types in terms of operator overloading.
  - Available in both strict and sloppy mode, with and without class syntax.
- Predictability
    - The meaning of operators on existing objects shouldn't be overridable or monkey-patchable, both for built-in types and for objects defined in other libraries.
    - It should not be possible to change the behavior of existing code using operators by unexpectedly passing it an object which overloads operators. (*If this is feasible.*)
    - Don't encourage a crazy coding style in the ecosystem.
  - Efficiently implementable
    - In native implementations, don't slow down code which doesn't take advantage of operator overloading (including within a module that uses operator overloading in some other paths).
    - When operator overloading is used, it should lend itself to relatively efficient native implementations, including - In the startup path, when code is run just a few times - Lends itself well to inline caching (for both monomorphic and polymorphic cases) to reduce any overhead of the dispatch - Feasible to optimize in a JIT (for both monomorphic and polymorphic cases), with a minimal number of cheap hidden class checks, and without extremely complicated cases for when things become invalid - Don't create too much complexity in the implementation to support such performance
    - When enough type declarations are present, it should be feasible to implement efficiently in TypeScript, similarly to BigInt's implementation.
  - Operator overloading should be a way of 'explaining the language' and providing hooks into something that's already there, rather than adding something which is a very different pattern from built-in operator definitions.

### **Avoiding classic pitfalls of operator overloading and readability**

The accusation is frequently made at C++ and Haskell that they are unreadable due to excessive use of obscure operators. On the other hand, in the Python ecosystem, operators are generally considered to make code significantly cleaner, e.g., in their use in NumPy.

This proposal includes several subtle design decisions to nudge the ecosystem in a direction of not using operator overloading in an excessive way, while still supporting the motivating case studies: -

---

Operators can be overloaded for one operand being a new user-defined type and the other operand being a previously defined type only in certain circumstances: - Strings only support overloading for + and the comparison operators. - Non-numeric, non-string primitives don't support overloading at all. - When one operand is an ordinary Object and the other is an Object with overloaded operators, the ordinary object is first coerced to some kind of primitive, making it less useful unless both operands were set up for overloading. - ToPrimitive, ToNumber, ToString, etc are *not* extended to ever return non-primitives. - Only built-in operators are supported; there are no user-defined operators. - Using overloaded operators requires the `@use: operators` statement, adding a little bit of friction, so overloaded operators are more likely to be used when they “pay for” that friction themselves from the perspective of a library user.

## Usage documentation

This section includes high-level for how to use and define overloaded operators, targeted at JavaScript programmers potentially using the feature. For low-level spec-like text, see PROTO-SPEC.md.

## Using operators

With this proposal, operators can be overloaded on certain JavaScript objects that declare themselves as having overloaded operators.

The following operators may have overloaded behavior: - Mathematical operators: unary +, −, ++, −−; binary +, −, \*, /, %, \*\* - Bitwise operators: unary ~; binary &, ^, |, <<, >>, >>> - Comparison operators: ==, <, >, <=, >= - Possibly, integer-indexed property access: [], [] =

The definition of >, <= and >= is derived from <, and the definition of assigning operators like += is derived their corresponding binary operator, for example +.

The following operators do not support overloading: - !, &&, || (boolean operations—always does ToBoolean first, and then works with the boolean) - === and the built-in SameVale and SameValueZero operations (always uses the built-in strict equality definition) - . and [] with non-integer values (these are property access; use Proxy to overload) - ( ) (calling a function—use a Proxy to overload) - , (just returns the right operand) - With future proposals, |>, ?. , ?. [ , ?. ( , ?? (based on function calls, property access, and checks against the specific null/undefined values, so similar to the above)

To use operator overloading, import a module that exports a class, and enable operators on it using a `@use: operators` declaration.

---

## with operators from declarations

Operator overloading is only enabled for the classes that you specifically opt in to. To do this overloading, use a `@use: operators` declaration, followed by a comma-separated list of classes that overload operators that you want to enable. This declaration is a form of built-in decorator.

For example, if you have two classes, `Vector` and `Scalar`, which support overloaded operators, you can

```
1 import { Vector, Scalar } from "./module.mjs";
2
3 new Vector([1, 2, 3]) * new Scalar(3); // TypeError: operator
   overloading on Vector and Scalar is not enabled
4
5 with operators from Vector, Scalar;
6
7 new Vector([1, 2, 3]) * new Scalar(3); // Works, returning new Vector
   ([3, 6, 9])
```

The scope of enabling operators is based on JavaScript blocks (e.g., you can enable operators within a specific function, rather than globally). By default, built-in types like `String`, `Number` and `BigInt` already have operators enabled.

## The Operators factory function

Recommended usage:

```
1 const operators = Operators(operatorDefinitions [,
   leftOrRightOperatorDefinitions...])
2 class MyClass extends operators { /* ... */ }
3 Object.preventExtensions(MyClass);
```

The `Operators` function is called with one required argument, which is a dictionary of operator definitions. The property keys are operator names like `+` and the values are functions, which take two arguments, which implement the operator. The dictionary may also have an `open` property, as described for `@Operators.overloaded` above.

The subsequent parameters of `Operators` are similar dictionaries of operator definitions, used for defining the behavior of operators when one of the parameters is of a type declared previously: they must have either a `left:` or `right:` property, indicating the type of the other operand.

Note: The `Operators` function and the above decorators could be exposed from a built-in module rather than being a property of the global object, depending on how that proposal goes.

---

## Q/A

### How does this proposal compare to operator overloading in other languages?

For a detailed investigation, see LANGCOMP.md. tl;dr: - It's a pretty popular design choice to conservatively support overloading only on some operators, and to define some in terms of others, as this proposal does. User-defined operators have been difficult to varying extents in other programming languages. - The way this proposal dispatches on the two operands is somewhat novel, most similar to Matlab. Unfortunately, of the established, popular mechanisms meet the design goals articulated in this document.

### Can this work with subclasses, rather than only defining overloading on base classes?

That would be equivalent to giving overloading behavior to existing objects. For example, imagine `SubclassOperators` as a sort of mixin for `Operators`, taking the superclass as its first argument, and then added operator overloading behavior to the return value of the superclass's constructor. Then, the following code would add operator overloading behavior to an unsuspecting object if we permitted operator overloading to be triggered by a decorator on a class that inherited from any other class!

```
1 function addOverloads(obj) {
2   class SuperClass { constructor() { return obj; } }
3   class SubClass extends SubclassOperators(SuperClass, {"+"(a, b) { /*
4     ... */ }}) {}
5   new SubClass(obj);
6   return obj;
7 }
```

The reason that this would modify the existing instance is that `SubClass` would put operator overloading behavior on whatever is returned from the super constructor, and that super constructor returns the existing object! Even if you don't use `with operators from`, there is suddenly different behavior when using operators on the object (throwing exceptions).

Let's avoid this level of dynamic-ness, and make the language more predictable by keeping it a static, unchange-able property of an object whether it overloads operators or not.

### Can't we allow monkey-patching, for mocking, etc?

You can do mocking by creating a separate operator-overloaded class which works like the one you're trying to mock, or even interacts with it. Or, you can make your own hooks into the operator defini-



---

tions to allow mocking. But letting any code reach into the definition of the operators for any other type risks making operators much less reliable than JavaScript programmers are accustomed to.

### Why does this have to be based on classes? I don't like classes!

It doesn't *have* to be based on classes, but the reason the above examples use inheritance is that a base class constructor gives a chance to return a truly unique object, with an internal slot that guides the overloading behavior. It's not clear how to get that out of object literals, but you can use the above API in a way like this, if you'd like:

```
1 function makePoint(obj) { return Object.assign(new pointOps, obj); }
2 const pointOps = Operators({ "+"(a, b) { return makePoint({x: a.x + b.x
  , y: a.y + b.y}); });
3 let point = makePoint({ x: 1, y: 2 });
4 with operators from pointOps;
5 (point + point).y; // 4
```

In the future, value types and/or typed objects could give a more ergonomic syntax which might not involve classes.

### Why not use symbols instead of a whole new dispatch mechanism?

Symbols would allow monkey-patching and a general lack of robustness. They don't give a clear way to dispatch on the right operand, without requiring a *second* property access (like Python). The Python-style dispatch also has a left-to-right bias, which is unfortunate. Symbols also don't let us avoid doing additional property accesses on existing objects, the way that this proposal does enable us to do.

### Why doesn't this let me define my own operator token?

This proposal only allows overloading built-in operators, because:

- There's significant concern about "punctuation overload" in JavaScript programs, already growing more fragile with private fields/methods (`#`) and decorators (`@`). Too many kinds of punctuation could make programs hard to read.
- User-defined precedence for such tokens is unworkable to parse.
- Hopefully the pipeline operator and optional chaining will solve many of the cases that would motivate these operators.
- We deliberately want to limit the syntactic divergence of JavaScript programs.

---

User-defined operator tokens may be a worthwhile proposal, but I (littledan) would be somewhat uncomfortable championing them for the above reasons.

### **Why doesn't this proposal allow ordinary functions to be called in infix contexts?**

For example, Haskell permits this capability, using backticks.

Such a capability could be useful, but it incurs the issues with adding more punctuation (see the previous Q/A entry), while not providing as terse results as overloading built-in operators. Method chaining or the pipeline operator can be used in many of the cases where infix function application could also be used.

### **Should operator overloading use inheritance-based multiple dispatch involving the prototype chain?**

This proposal has opted against using something like Slate's Prototype Multiple Dispatch, because:

- This is really complicated to implement and optimize reliably.
- It's not clear what important use cases there are that aren't solved by single-level dispatch.

### **If you define your other-type overloads based on a previously defined type, how do you know which type came first?**

If you have operators defined in two different modules, then to define overloads between them, import one module from the other, and don't make a circularity between the two. If you do this, the loading order will be deterministic. The one that imports the other one is responsible for defining the overloads between the two types.

### **How does operator overloading relate to other proposals?**

**Decorators** Decorators (Stage 2) could be used for a more ergonomic way to define operator overloading, as described in a previous version of this README. However, the decorators proposal is not yet stable, with changes still being discussed, so it's a bit early to propose a concrete decorator syntax for operator overloading.

**BigInt and BigDecimal** BigInt (Stage 4) provides definitions for how operators work on just one new type, representing arbitrary-precision integers. BigDecimal (Stage 0) represents another, for arbitrary-precision decimals. This proposal generalizes that to types defined in JavaScript.

---

**Records and Tuples** Records and Tuples (Stage 1) provides a deeply immutable compound primitive notion, superficially analogous to Objects and Arrays, with value semantics.

If either operator overloading and records and tuples advances past Stage 1, then a next step for the other proposal would be to define semantics for the two features to be used together. One possibility is that for the return value of [Operators](#) to have a method that would take a Record or Tuple and return a new one with operators overloaded (details TBD).

**Typed Objects and Value Types** Typed Objects is a proposal for efficient, fixed-shape objects.

Value Types is an idea in TC39 about user-definable primitive types. At some points in the past, it was proposed that operator overloading be tied to value types.

Neither proposal is currently championed in TC39, but Records and Tuples presents a sort of first step towards Value Types.

When these proposals mature more, it will be good to look into how operator overloading can be enabled for Typed Objects and Value Types. The idea in this repository is to not limit operator overloading to those two types of values, but to also permit operator overloading for ordinary objects.

**Extended numeric literals** The extended numeric literals proposal (Stage 1) allows numeric-like types such as `3@px` or `5.2@m` to be defined and used ergonomically. Extended numeric literals and operator overloading could fit well together, but they don't depend on each other and can each be used separately. This README omits use of extended numeric literals, for simplicity and to focus on the operator overloading aspects.

### **How does operator overloading interact with Proxy and membrane systems?**

In this proposal, operators remain *not* operations visible in the meta-object protocol. Objects with overloaded operators don't even undergo the typical object coercion. However, this proposal still attempts to mesh well with membrane systems.

All operator-overloaded values are objects, so any technique that's used to create or access them can be mediated by membrane wrapping. The value returned from the membrane can be overloaded in a separate, membrane-mediated way, assuming collaboration between the overloaded object and the membrane system (otherwise there's no introspection API to see which operators to overload).

A membrane system which runs early in the program's execution (like the freeze-the-world systems) can monkey-patch and replace the [Operators](#) object to provide this collaboration; therefore, there

---

is no need for any particular additional hooks. At a minimum, even without replacing the `Operators` object, the membrane can deny use of overloaded operators for the object on the other side of the membrane.

`with operator from` declarations provide a further defense: Those declarations prove that the piece of the program has access to (a piece of) the class defining overloaded operators. This works because the lookup of the internal slot `[[OperatorSetDefinition]]` does is not transparent to Proxies. A membrane system can deny access to that original operator set, and instead replace it with a separate class which overloads operators in a membrane-mediated way. In this way, even if an overloaded value “leaks”, the right to call its operators is controlled by the class, which forms a capability object.

### **Could this proposal allow overloading `[]`?**

Possibly. `[]` is property access, not an operator as such. ES6 introduced Proxy, which lets developers define custom semantics for property access. Unfortunately, this capability has certain issues: - This proposal is based on `Operators` factories, which produce constructors that return new objects with operator overloading. `Proxy` also returns new object instances. It’s not possible to use both Proxy and the mechanism in this repository together, since operator overloading doesn’t forward through Proxy traps. Therefore, some other mechanism is needed. - Proxy has so far not yet been optimized in JavaScript engines as much as some might hope. It’s not clear exactly what the cause is, but one factor may be how general Proxy capabilities are. The proposal for overloading `[]` is much more restricted in its power, potentially making it more easily optimizable. - From the perspective of many JavaScript developers and even library authors, at a high level, it’s an “implementation detail” that array index access is based on JavaScript property access; for them, overloading this way is consistent with their mental model.

The overloading of `[]` proposed here would be based on the semantics of Integer Indexed Exotic Objects. It wouldn’t add or change anything about JavaScript’s meta-object protocol, and it would only change the semantics of objects with overloading of `[]` declared.