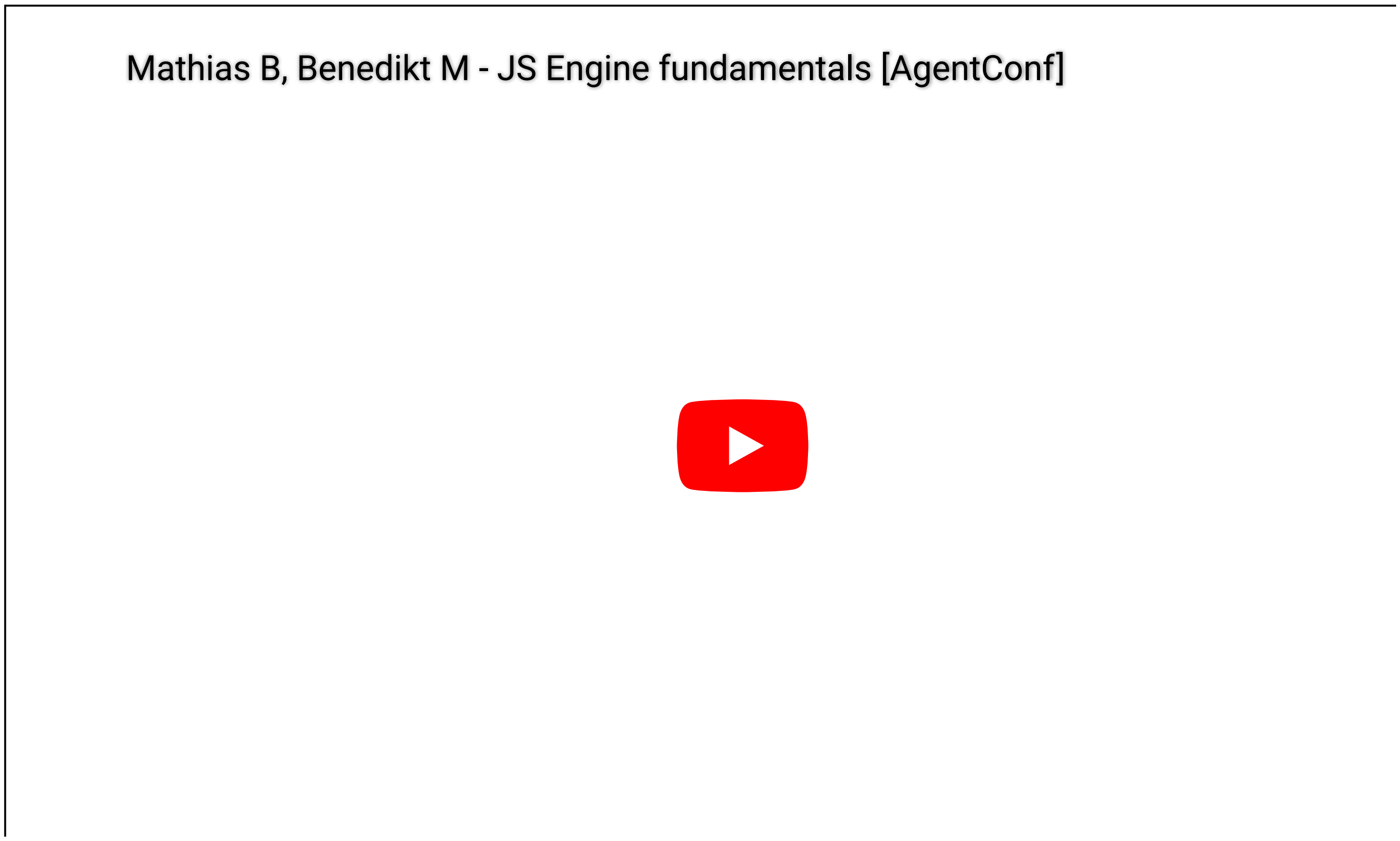
The story of a V8 performance cliff in React

Published 28 August 2019 · Tagged with internals presentations

Previously, we discussed how JavaScript engines optimize object and array access through the use of Shapes and Inline Caches, and we've explored how engines speed up prototype property access in particular. This article describes how V8 chooses optimal in-memory representations for various JavaScript values, and how that impacts the shape machinery — all of which helps explain a recent V8 performance cliff in React core.

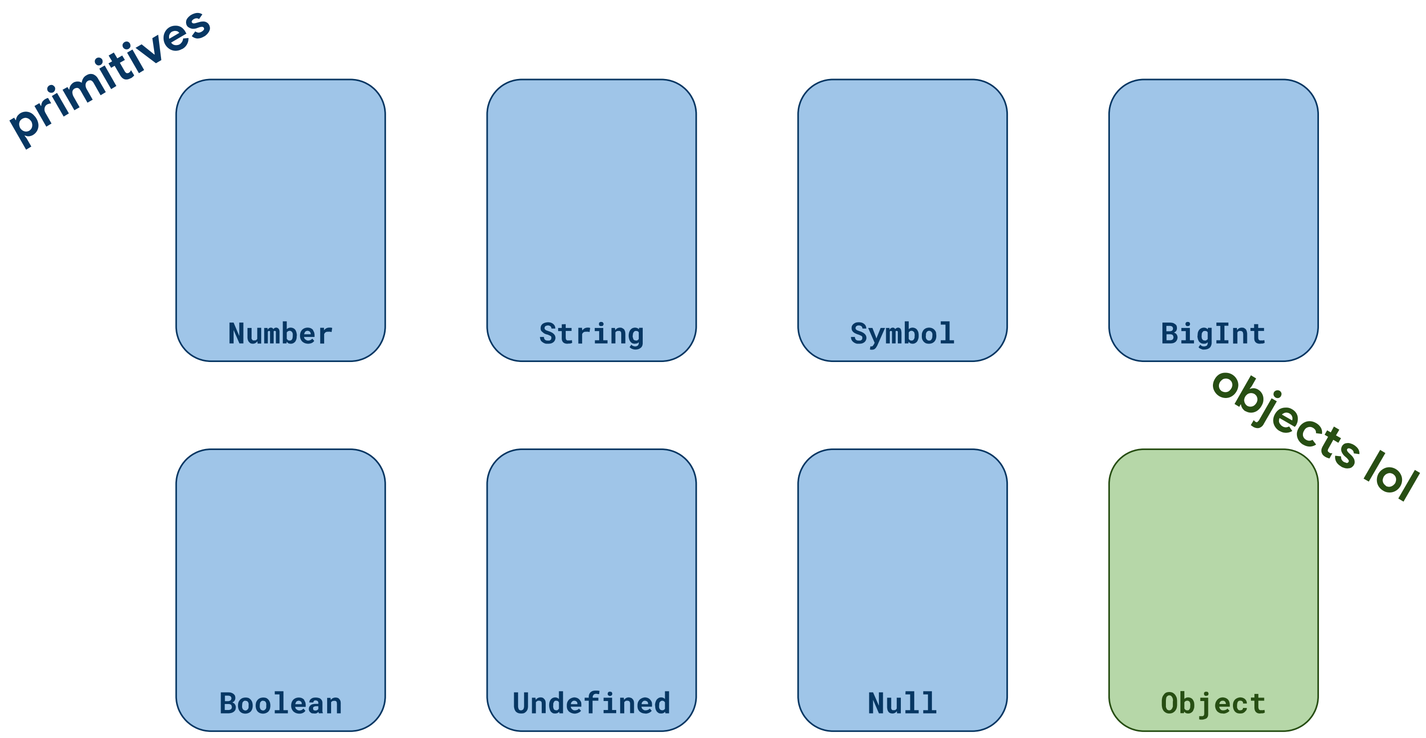
Note: If you prefer watching a presentation over reading articles, then enjoy the video below! If not, skip the video and read on.



"JavaScript engine fundamentals: the good, the bad, and the ugly" as presented by Mathias Bynens and Benedikt Meurer at AgentConf 2019.

JavaScript types

Every JavaScript value has exactly one of (currently) eight different types: `Number`, `String`, `Symbol`, `BigInt`, `Boolean`, `Undefined`, `Null`, and `Object`.



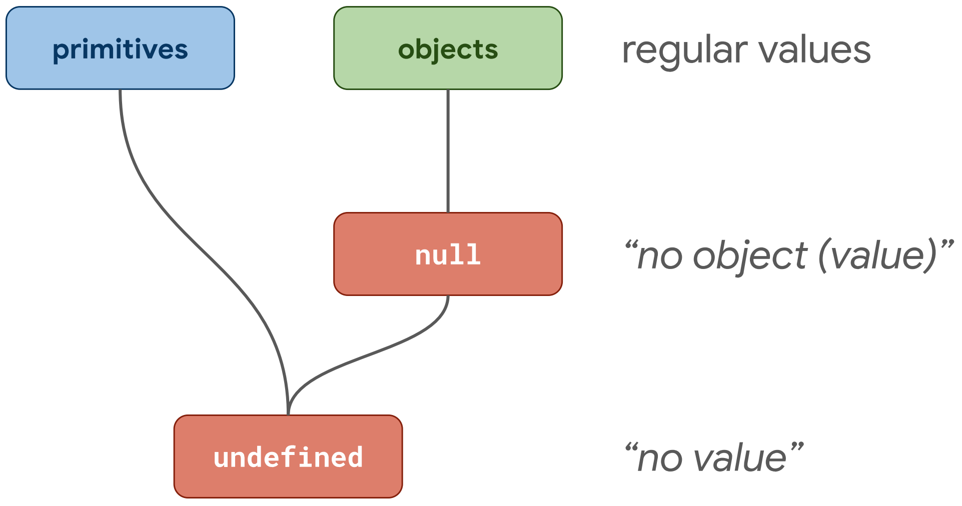
With one notable exception, these types are observable in JavaScript through the `typeof` operator:

```
typeof 42;
// → 'number'
typeof 'foo';
// → 'string'
typeof Symbol('bar');
// → 'symbol'
typeof 42n;
// → 'bigint'
typeof true;
// → 'boolean'
typeof undefined;
// → 'undefined'
typeof null;
// → 'object' 🤔
typeof { x: 42 };
// → 'object'
```

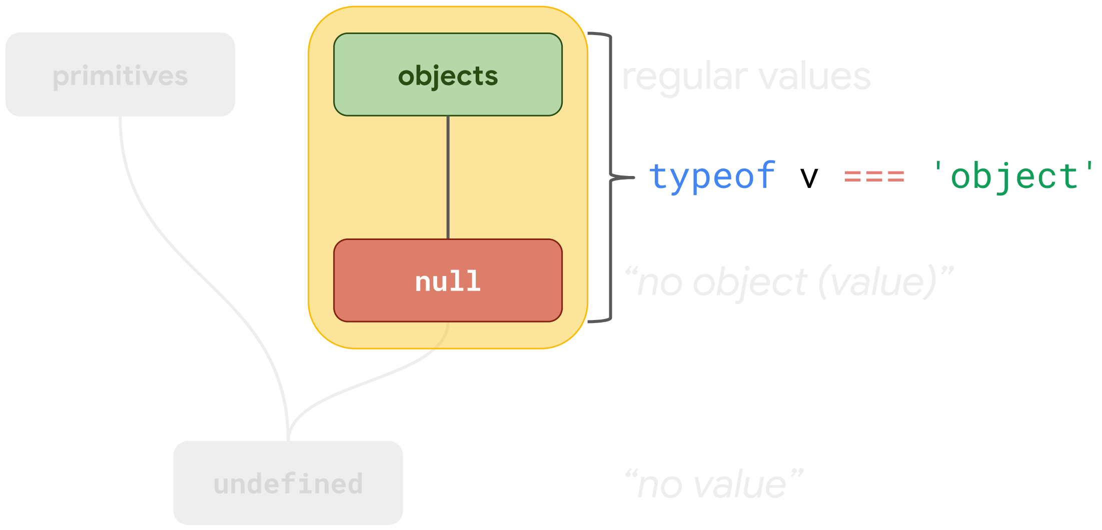
`typeof null` returns `'object'`, and not `'null'`, despite `Null` being a type of its own. To understand why, consider that the set of all JavaScript types is divided into two groups:

- *objects* (i.e. the `Object` type)
- *primitives* (i.e. any non-object value)

As such, `null` means "no object value", whereas `undefined` means "no value".



Following this line of thought, Brendan Eich designed JavaScript to make `typeof` return `'object'` for all values on the right-hand side, i.e. all objects and `null` values, in the spirit of Java. That's why `typeof null === 'object'` despite the spec having a separate `Null` type.



Value representation

JavaScript engines must be able to represent arbitrary JavaScript values in memory. However, it's important to note that the JavaScript type of a value is separate from how JavaScript engines represent that value in memory.

The value `42`, for example, has type `number` in JavaScript.

```
typeof 42;
// → 'number'
```

There are several ways to represent an integer number like `42` in memory:

representation	bits
two's complement 8-bit	0010 1010
two's complement 32-bit	0000 0000 0000 0000 0000 0000 0010 1010
packed binary-coded decimal (BCD)	0100 0010
32-bit IEEE-754 floating-point	0100 0010 0010 1000 0000 0000 0000 0000
64-bit IEEE-754 floating-point	0100 0000 0100 0101 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

ECMAScript standardizes numbers as 64-bit floating-point values, also known as *double precision floating-point* or *Float64*. However, that doesn't mean that JavaScript engines store numbers in Float64 representation all the time — doing so would be terribly inefficient! Engines can choose other internal representations, as long as the observable behavior matches Float64 exactly.

Most numbers in real-world JavaScript applications happen to be valid ECMAScript array indices, i.e. integer values in the range from 0 to $2^{32}-2$.

```
array[0]; // Smallest possible array index.
array[42];
array[2**32-2]; // Greatest possible array index.
```

JavaScript engines can choose an optimal in-memory representation for such numbers to optimize code that accesses array elements by index. For the processor to do the memory access operation, the array index must be available in two’s complement. Representing array indices as Float64 instead would be wasteful, as the engine would then have to convert back and forth between Float64 and two’s complement every time someone accesses an array element.

The 32-bit two’s complement representation is not just useful for array operations. In general, **processors execute integer operations much faster than floating-point operations**. That’s why in the next example, the first loop is easily twice as fast compared to the second loop.

```
for (let i = 0; i < 1000; ++i) {
  // fast 🚀
}

for (let i = 0.1; i < 1000.1; ++i) {
  // slow 🐌
}
```

The same goes for operations as well. The performance of the modulo operator in the next piece of code depends on whether you’re dealing with integers or not.

```
const remainder = value % divisor;
// Fast 🚀 if `value` and `divisor` are represented as integers,
// slow 🐌 otherwise.
```

If both operands are represented as integers, the CPU can compute the result very efficiently. V8 has additional fast-paths for the cases where the `divisor` is a power of two. For values represented as floats, the computation is much more complex and takes a lot longer.

Because integer operations generally execute much faster than floating-point operations, It would seem that engines could just always use two’s complement for all integers and all results of integer operations. Unfortunately, that would be a violation of the ECMAScript specification! ECMAScript standardizes on Float64, and so **certain integer operations actually produce floats**. It’s important that JS engines produce the correct results in such cases.

```
// Float64 has a safe integer range of 53 bits. Beyond that range,
// you must lose precision.
```

```
2**53 === 2**53+1;
// → true

// Float64 supports negative zeros, so -1 * 0 must be -0, but
// there’s no way to represent negative zero in two’s complement.
-1*0 === -0;
// → true

// Float64 has infinities which can be produced through division
// by zero.
1/0 === Infinity;
// → true
-1/0 === -Infinity;
// → true

// Float64 also has NaNs.
0/0 === NaN;
```

Even though the values on the left-hand side are integers, all the values on the right are floats. This is why none of the above operations can be performed correctly using 32-bit two’s complement. JavaScript engines have to take special care to make sure that integer operations fall back appropriately to produce the fancy Float64 results.

For small integers in the 31-bit signed integer range, V8 uses a special representation called `Smi`. Anything that is not a `Smi` is represented as a `HeapObject`, which is the address of some entity in memory. For numbers, we use a special kind of `HeapObject`, the so-called `HeapNumber`, to represent numbers that aren’t inside the `Smi` range.

```
 -Infinity // HeapNumber
-(2**30)-1 // HeapNumber
  -(2**30) // Smi
       -42 // Smi
        -0 // HeapNumber
         0 // Smi
       4.2 // HeapNumber
        42 // Smi
   2**30-1 // Smi
```

```
     2**30 // HeapNumber
  Infinity // HeapNumber
       NaN // HeapNumber
```

As the above example shows, some JavaScript numbers are represented as `Smi`s, and others are represented as `HeapNumber`s. V8 is specifically optimized for `Smi`s, because small integers are so common in real-world JavaScript programs. `Smi`s don't need to be allocated as dedicated entities in memory, and enable fast integer operations in general.

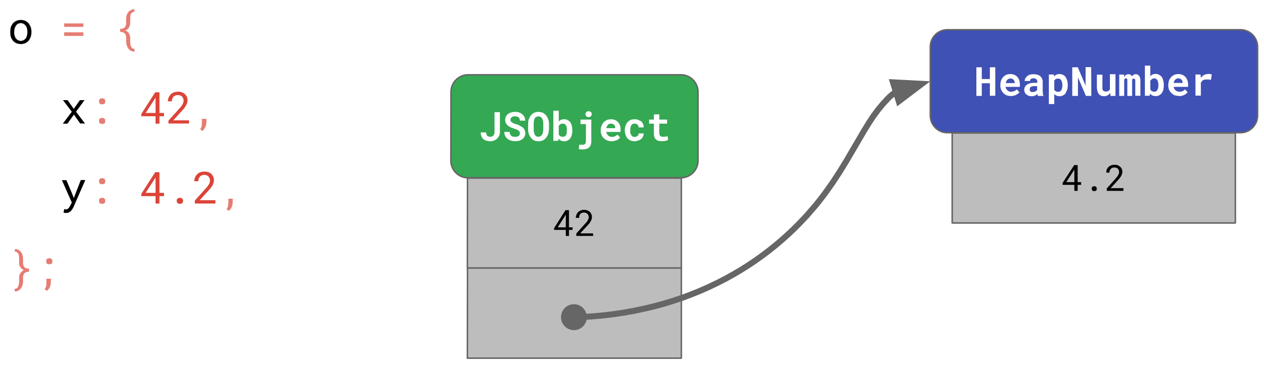
The important take-away here is that **even values with the same JavaScript type can be represented in completely different ways** behind the scenes, as an optimization.

`Smi` vs. `HeapNumber` vs. `MutableHeapNumber`

Here's how that works under the hood. Let's say you have the following object:

```
const o = {
  x: 42,  // Smi
  y: 4.2, // HeapNumber
};
```

The value `42` for `x` can be encoded as `Smi`, so it can be stored inside of the object itself. The value `4.2` on the other hand needs a separate entity to hold the value, and the object points to that entity.

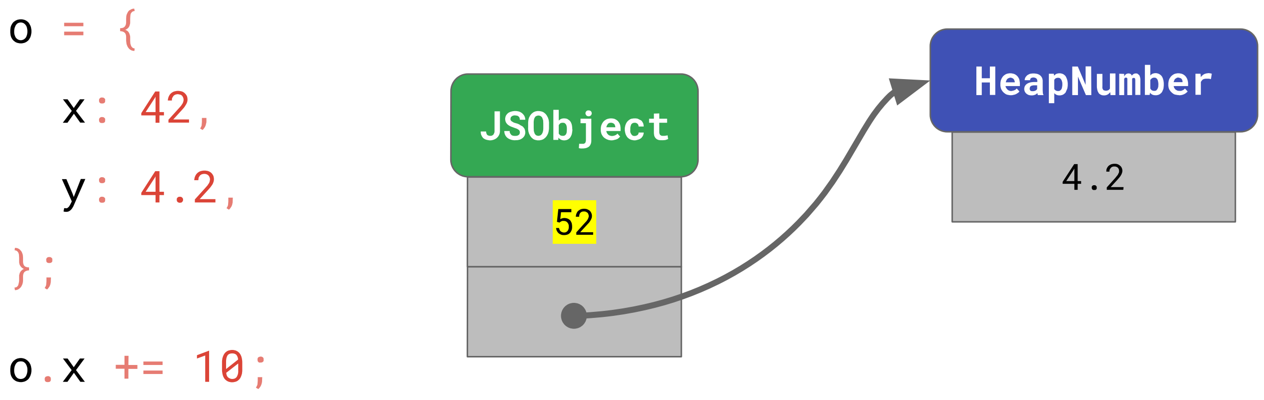

```
o = {
  x: 42,
  y: 4.2,
};
```

Now, let's say we run the following JavaScript snippet:

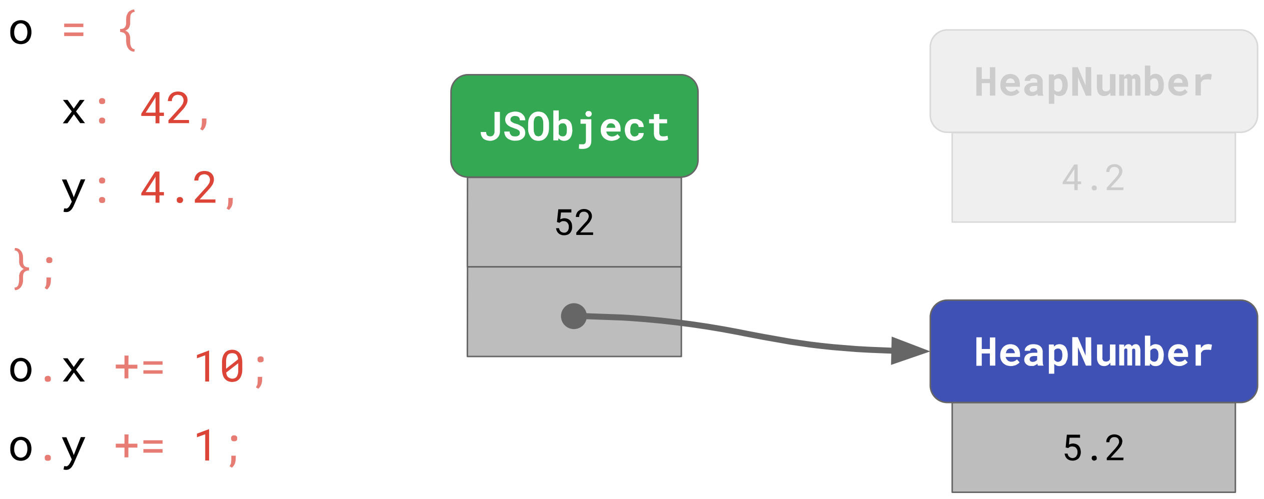
```
o.x += 10;
// → o.x is now 52
```

```
o.y += 1;
// → o.y is now 5.2
```

In this case, the value of `x` can be updated in-place, since the new value `52` also fits the `Smi` range.



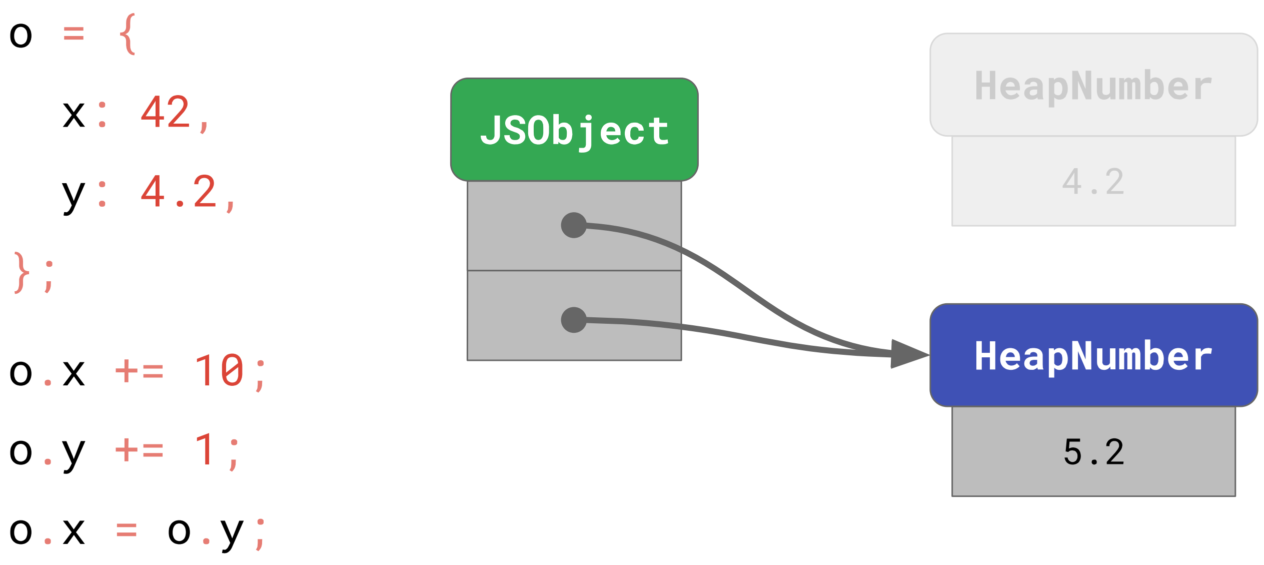
However, the new value of `y=5.2` does not fit into a `Smi` and is also different from the previous value `4.2`, so V8 has to allocate a new `HeapNumber` entity for the assignment to `y`.



`HeapNumber`s are not mutable, which enables certain optimizations. For example, if we assign `y`s value to `x`:

```
o.x = o.y;
// → o.x is now 5.2
```

…we can now just link to the same `HeapNumber` instead of allocating a new one for the same value.

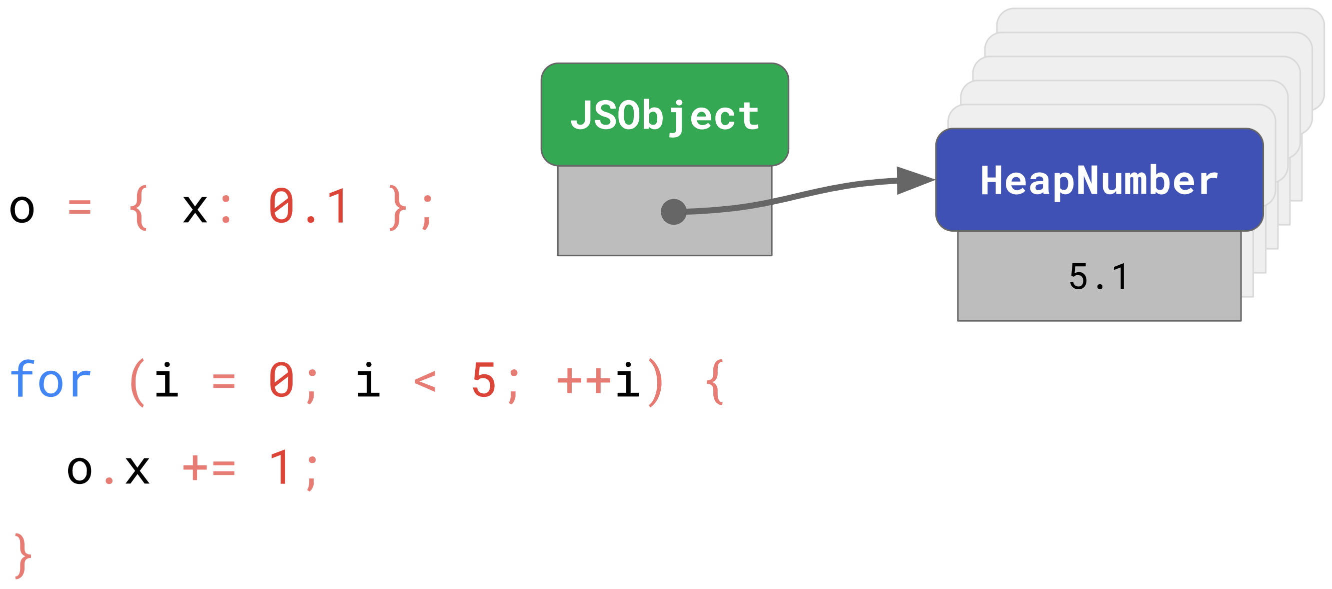


```
o = {
  x: 42,
  y: 4.2,
};
o.x += 10;
o.y += 1;
o.x = o.y;
```

One downside to `HeapNumber`s being immutable is that it would be slow to update fields with values outside the `Smi` range often, like in the following example:

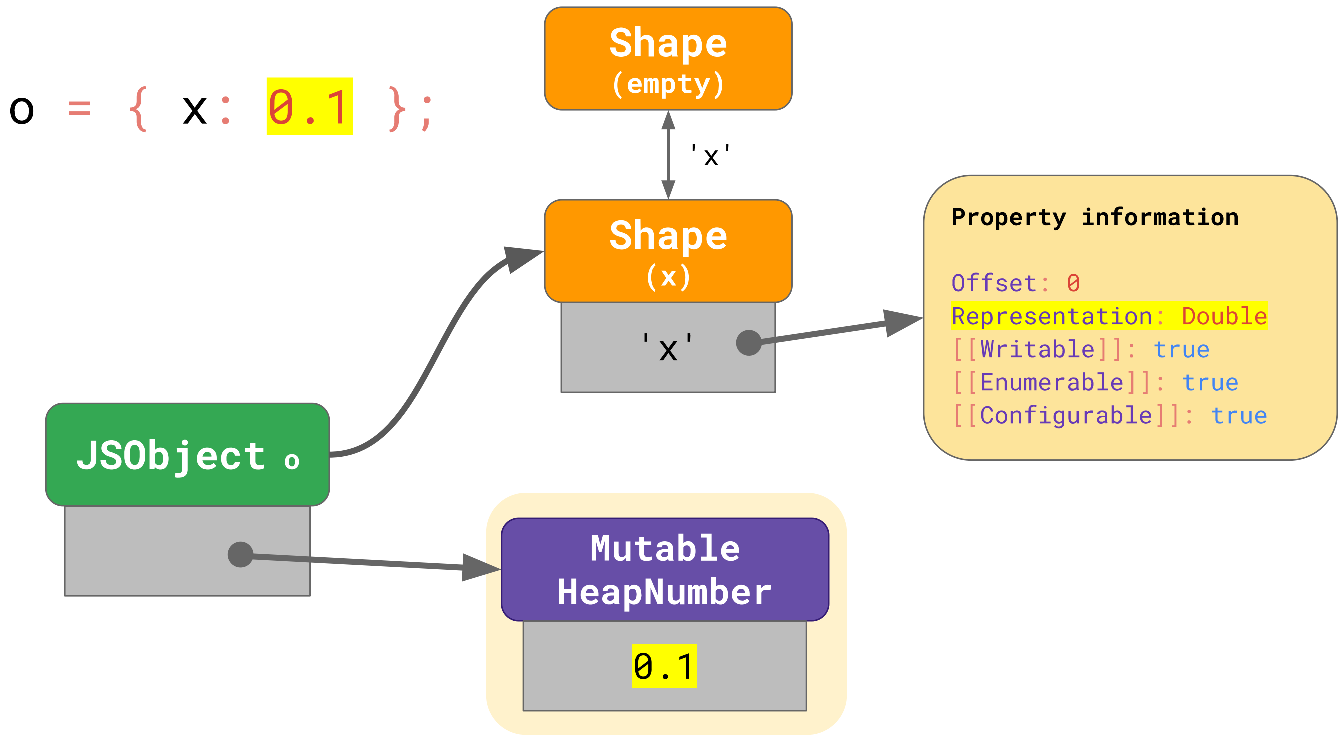
```
// Create a `HeapNumber` instance.
const o = { x: 0.1 };

for (let i = 0; i < 5; ++i) {
  // Create an additional `HeapNumber` instance.
  o.x += 1;
}
```

The first line would create a `HeapNumber` instance with the initial value `0.1`. The loop body changes this value to `1.1`, `2.1`, `3.1`, `4.1`, and finally `5.1`, creating a total of six `HeapNumber` instances along the way, five of which are garbage once the loop finishes.



```
o = { x: 0.1 };

for (i = 0; i < 5; ++i) {
  o.x += 1;
}
```

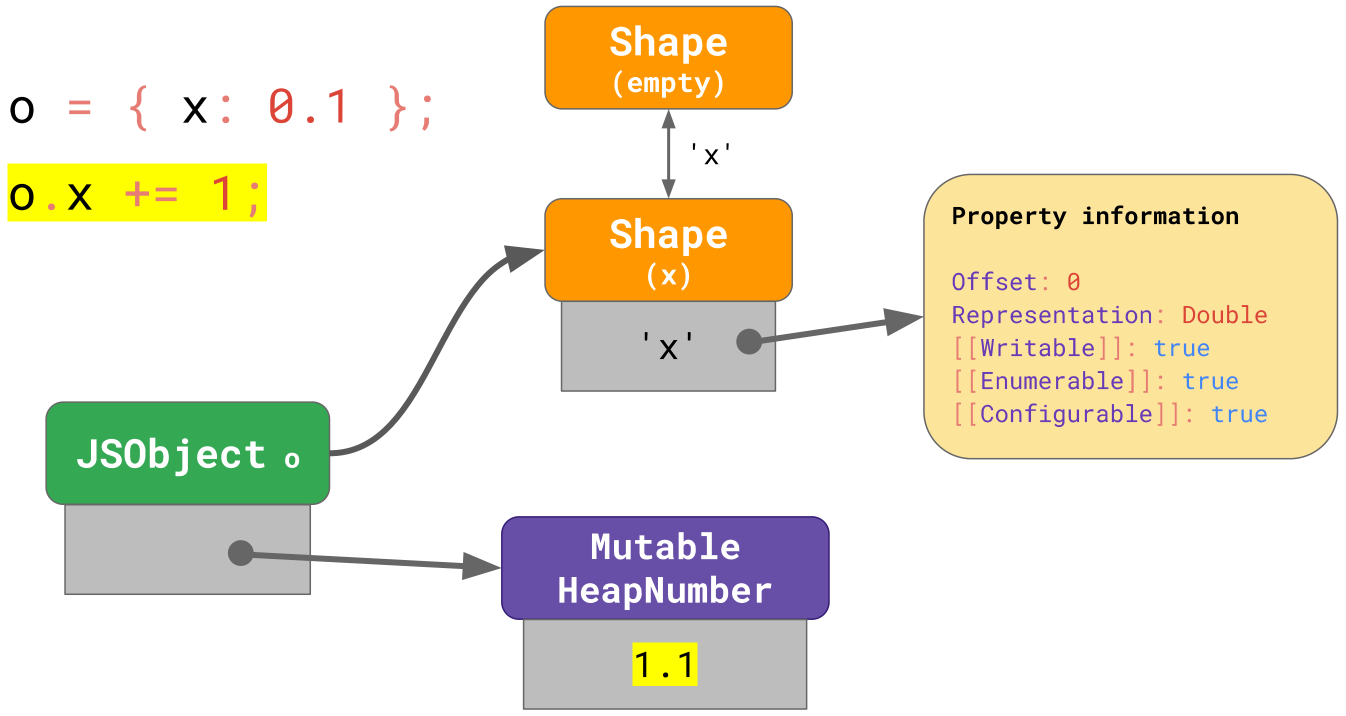
To avoid this problem, V8 provides a way to update non-`Smi` number fields in-place as well, as an optimization. When a numeric field holds values outside the `Smi` range, V8 marks that field as a `Double` field on the shape, and allocates a so-called `MutableHeapNumber` that holds the actual value encoded as Float64.



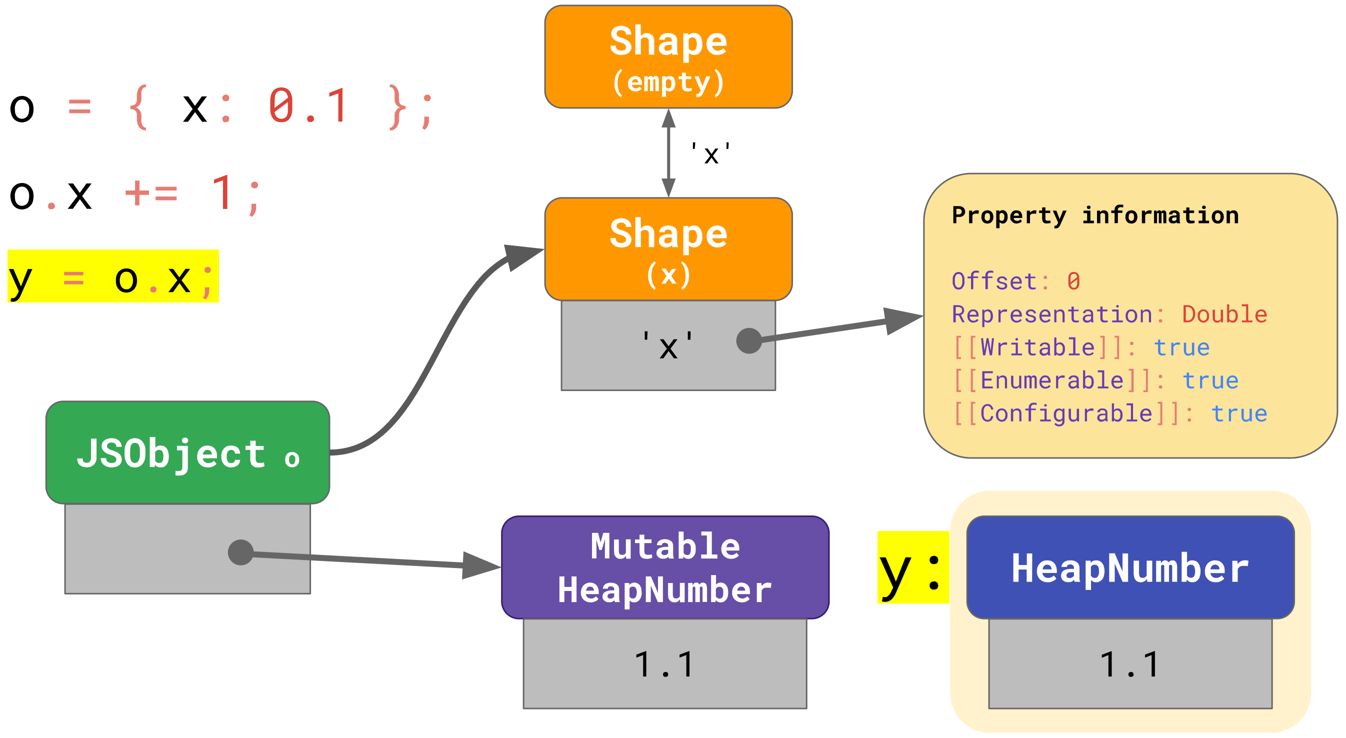
```
o = { x: 0.1 };
```

```
Property information

Offset: 0
Representation: Double
[[Writable]]: true
[[Enumerable]]: true
[[Configurable]]: true
```

When your field’s value changes, V8 no longer needs to allocate a new `HeapNumber`, but instead can just update the `MutableHeapNumber` in-place.



However, there's a catch to this approach as well. Since the value of a `MutableHeapNumber` can change, it's important that these are not passed around.

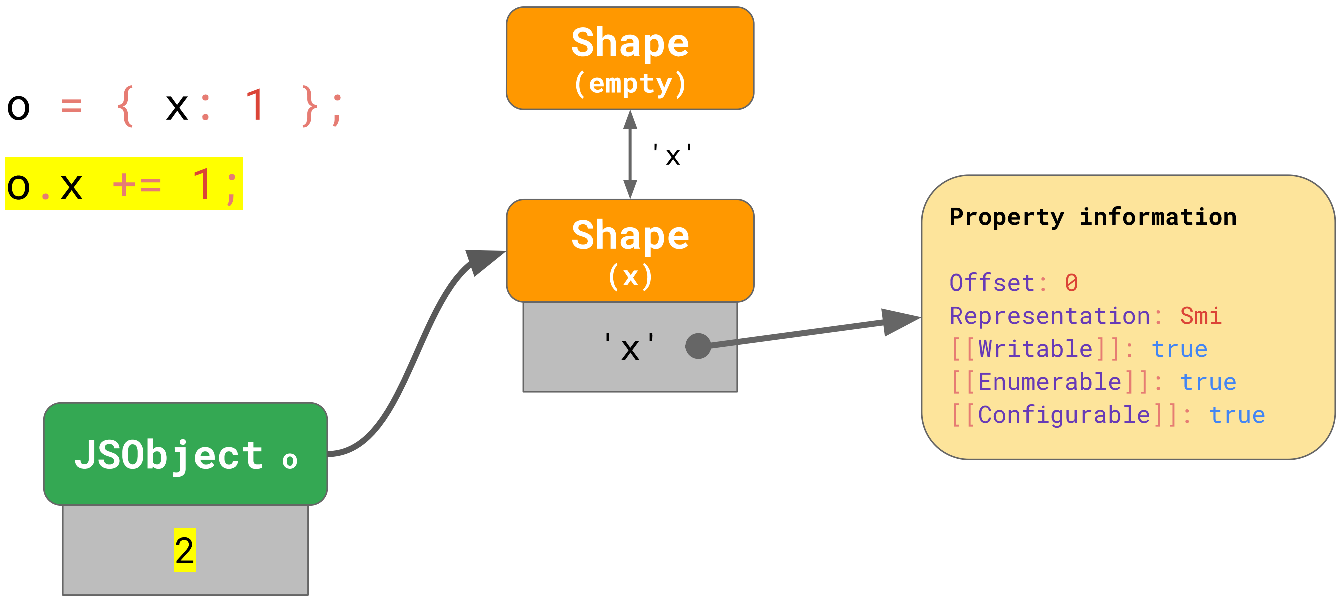


For example, if you assign `o.x` to some other variable `y`, you wouldn't want the value of `y` to change the next time `o.x` changes — that would be a violation of the JavaScript spec! So when `o.x` is accessed, the number must be *re-boxed* into a regular `HeapNumber` before assigning it to `y`.

For floats, V8 performs all the above-mentioned “boxing” magic behind the scenes. But for small integers it would be wasteful to go with the `MutableHeapNumber` approach, since `Smi` is a more efficient representation.

```
const object = { x: 1 };
// → no “boxing” for `x` in object

object.x += 1;
// → update the value of `x` inside object
```

To avoid the inefficiency, all we have to do for small integers is mark the field on the shape as `Smi` representation, and simply update the number value in place as long as it fits the small integer range.



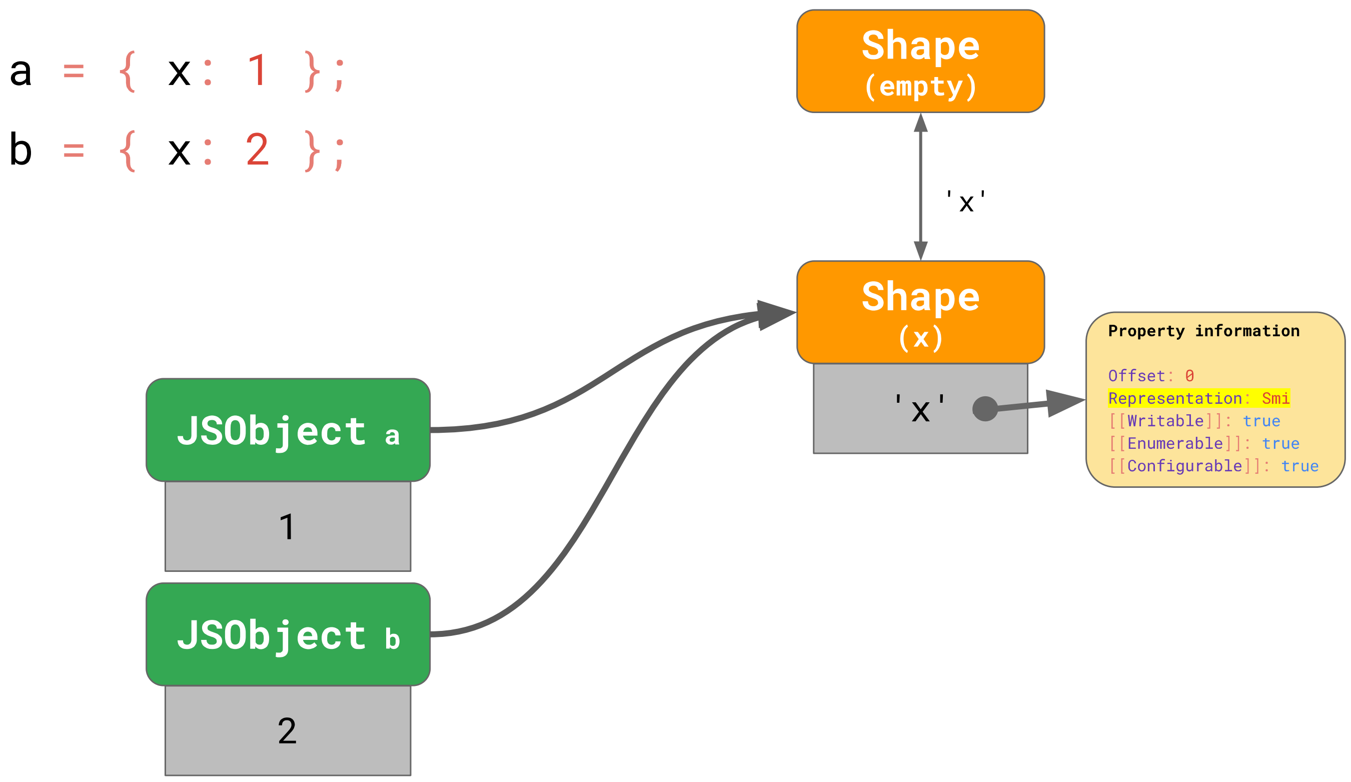
Shape deprecations and migrations

So what if a field initially contains a `Smi`, but later holds a number outside the small integer range? Like in this case, with two objects both using the same shape where `x` is represented as `Smi` initially:

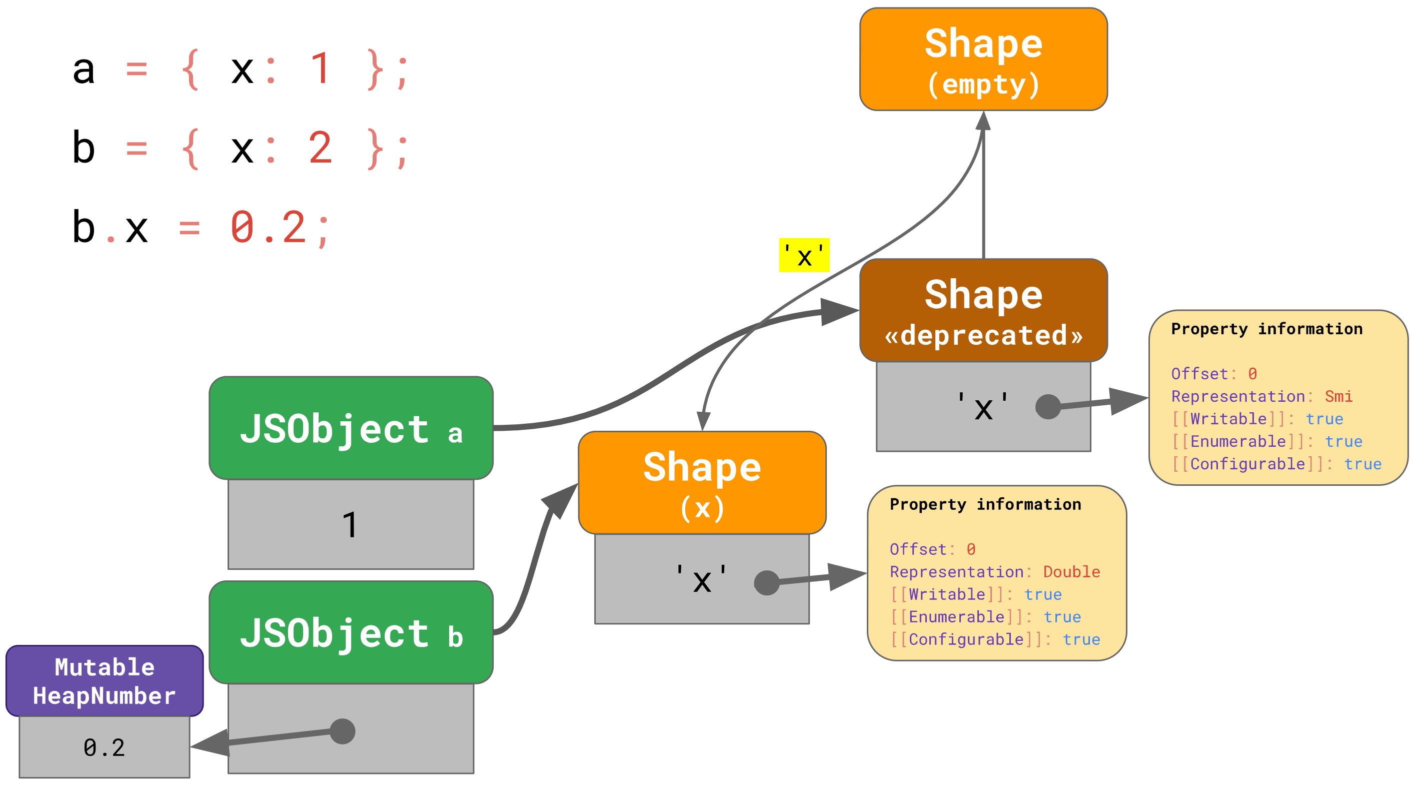
```
const a = { x: 1 };
const b = { x: 2 };
// → objects have `x` as `Smi` field now

b.x = 0.2;
// → `b.x` is now represented as a `Double`
```

```
y = a.x;
```

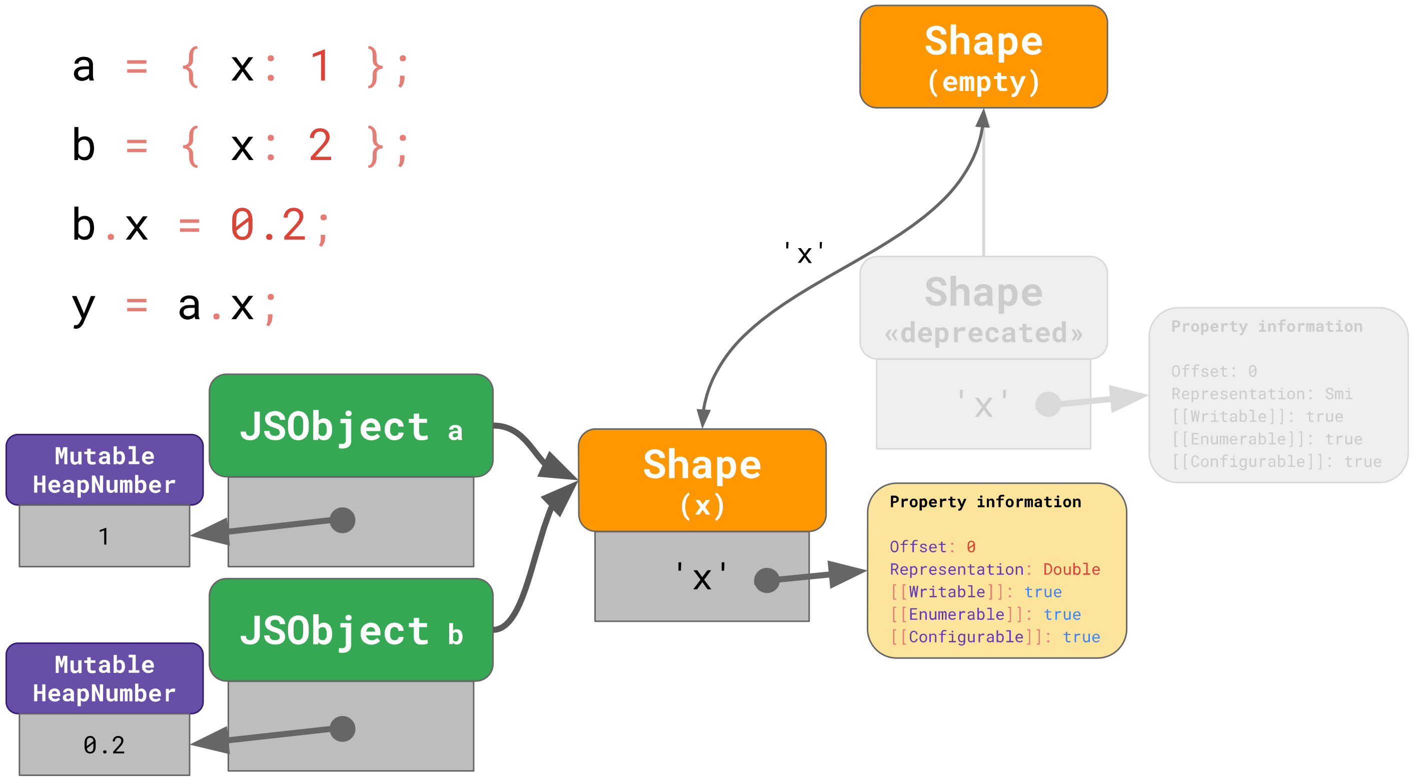
This starts out with two objects pointing to the same shape, where `x` is marked as `Smi` representation:



When `b.x` changes to `Double` representation, V8 allocates a new shape where `x` is assigned `Double` representation, and which points back to the empty shape. V8 also allocates a `MutableHeapNumber` to hold the new value `0.2` for the `x` property. Then we update the object `b` to point to this new shape, and change the slot in the object to point to the previously allocated `MutableHeapNumber` at offset 0. And finally, we mark the old shape as deprecated and unlink it from the transition tree. This is done by having a new transition for `'x'` from the empty shape to the newly-created shape.



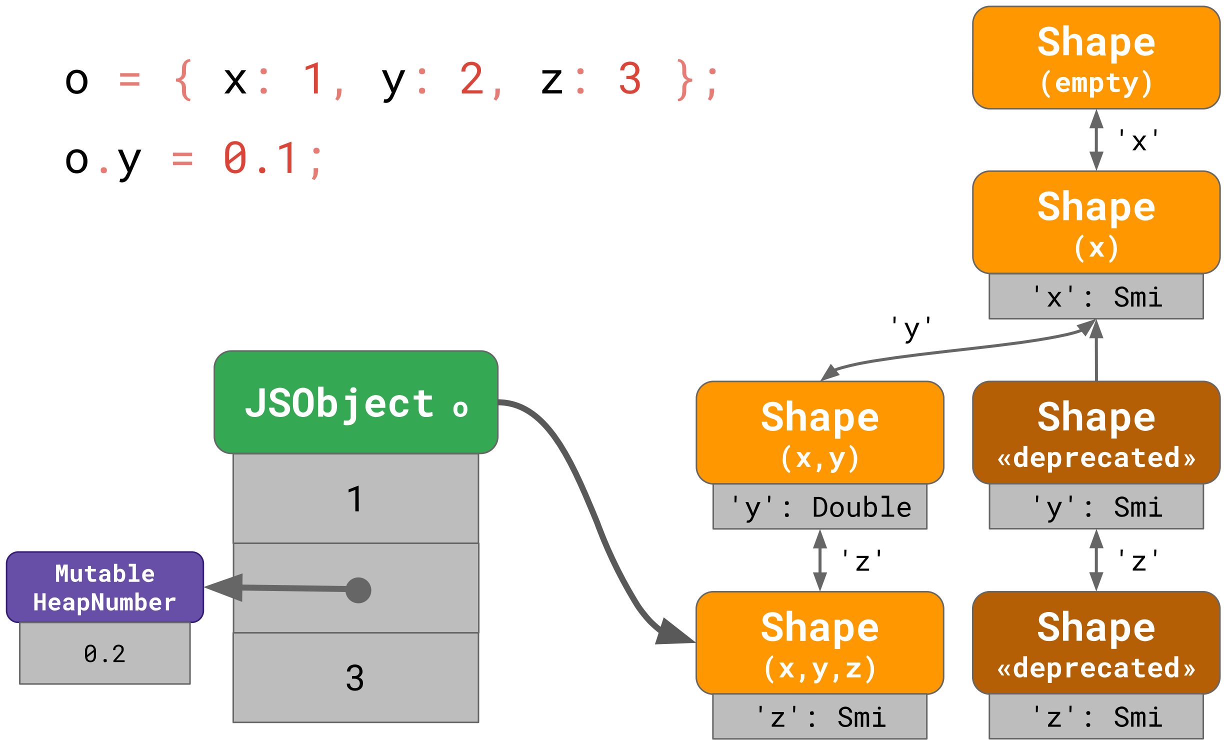
We cannot completely remove the old shape at this point, since it is still used by `a`, and it would be way too expensive to traverse the memory to find all objects pointing to the old shape and update them eagerly. Instead V8 does this lazily: any property access or assignment to `a` migrates it to the new shape first. The idea is to eventually make the deprecated shape unreachable and to have the garbage collector remove it.



A trickier case occurs if the field that changes representation is *not* the last one in the chain:

```
const o = {
  x: 1,
  y: 2,
  z: 3,
};

o.y = 0.1;
```

In that case V8 needs to find the so-called *split shape*, which is the last shape in the chain before the relevant property gets introduced. Here we're changing `y`, so we need to find the last shape that doesn't have `y`, which in our example is the shape that introduced `x`.



Starting from the split shape, we create a new transition chain for `y` which replays all the previous transitions, but with `'y'` being marked as `Double` representation. And we use this new transition chain for `y`, marking the old subtree as deprecated. In the last step we migrate the instance `o` to the new shape, using a `MutableHeapNumber` to hold the value of `y` now. This way, new objects do not take the old path, and once all references to the old shape are gone, the deprecated shape part of the tree disappears.

Extensibility and integrity-level transitions

`Object.preventExtensions()` prevents new properties from ever being added to an object. If you try, it throws an exception. (If you're not in strict mode, it doesn't throw but it silently does nothing.)

```
const object = { x: 1 };
Object.preventExtensions(object);
object.y = 2;
// TypeError: Cannot add property y;
//            object is not extensible
```

`Object.seal` does the same as `Object.preventExtensions`, but it also marks all properties as non-configurable, meaning you can't delete them, or change their enumerability, configurability, or writability.

```
const object = { x: 1 };
Object.seal(object);
object.y = 2;
// TypeError: Cannot add property y;
//            object is not extensible
delete object.x;
// TypeError: Cannot delete property x
```

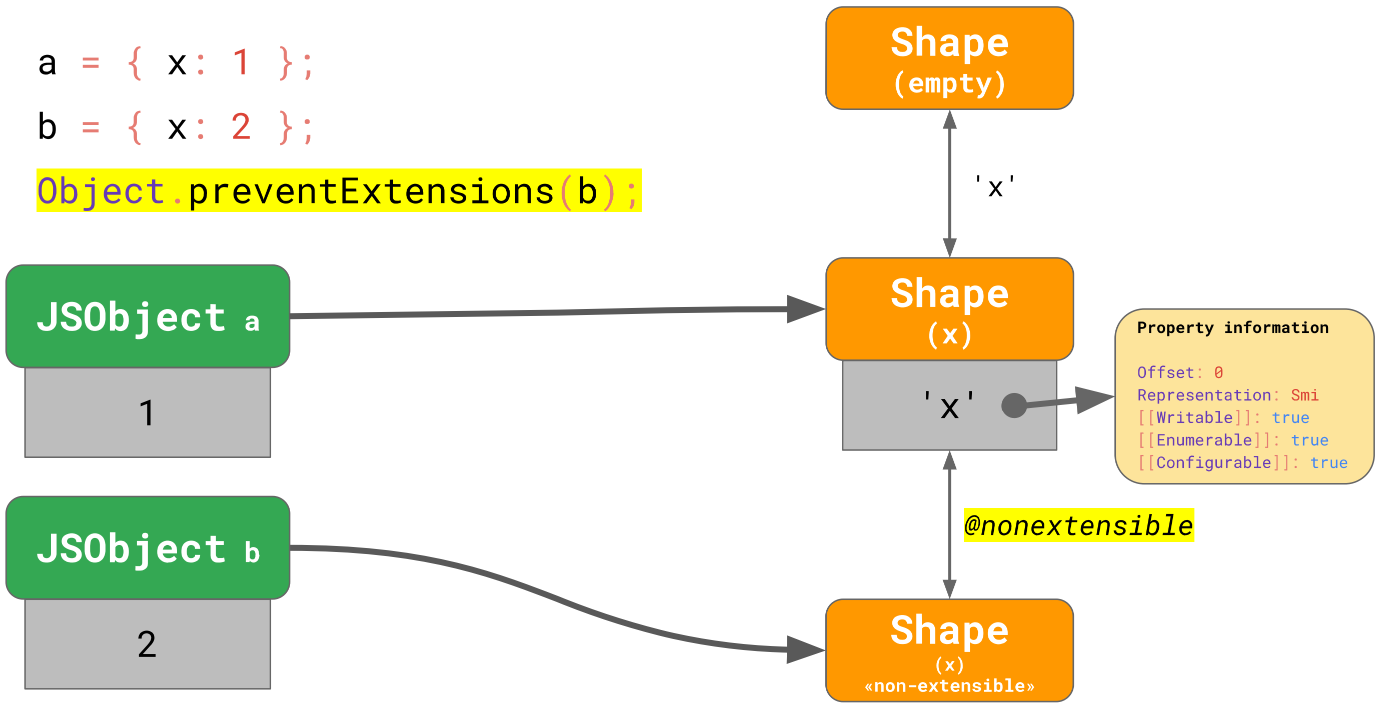
`Object.freeze` does the same as `Object.seal`, but it also prevents the values of existing properties from being changed by marking them non-writable.

```
const object = { x: 1 };
Object.freeze(object);
object.y = 2;
// TypeError: Cannot add property y;
//            object is not extensible
delete object.x;
// TypeError: Cannot delete property x
object.x = 3;
// TypeError: Cannot assign to read-only property x
```

Let's consider this concrete example, with two objects which both have a single property `x`, and where we then prevent any further extensions to the second object.

```
const a = { x: 1 };
const b = { x: 2 };

Object.preventExtensions(b);
```

It starts out like we already know, transitioning from the empty shape to a new shape that holds the property `'x'` (represented as `Smi`). When we prevent extensions to `b`, we perform a special transition to a new shape which is marked as non-extensible. This special transition doesn't introduce any new property — it's really just a marker.



Note how we can't just update the shape with `x` in-place, since that is needed by the other object `a`, which is still extensible.

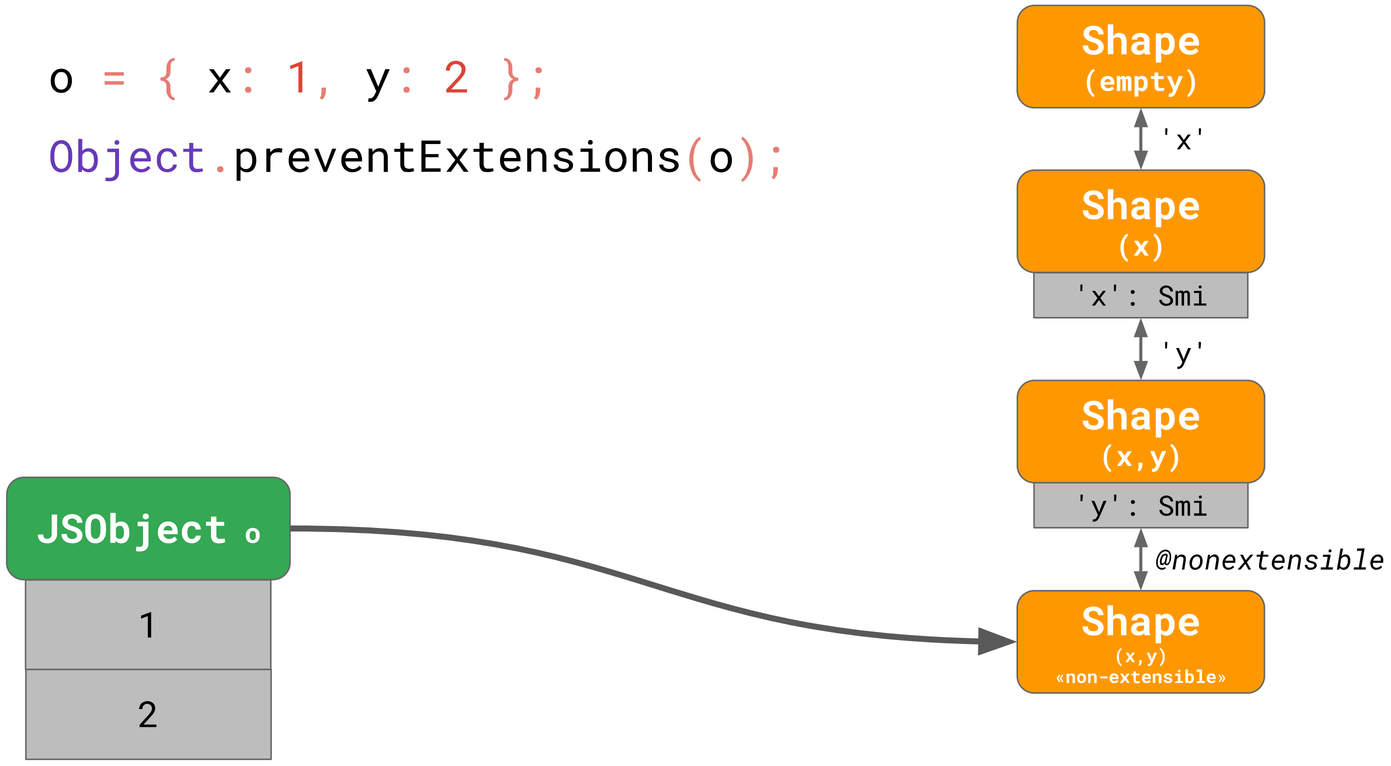
The React performance issue

Let's put it all together and use what we learned to understand [the recent React issue #14365](). When the React team profiled a real-world application, they spotted an odd V8 performance cliff that affected React's core. Here's a simplified repro for the bug:

```
const o = { x: 1, y: 2 };
Object.preventExtensions(o);
o.y = 0.2;
```

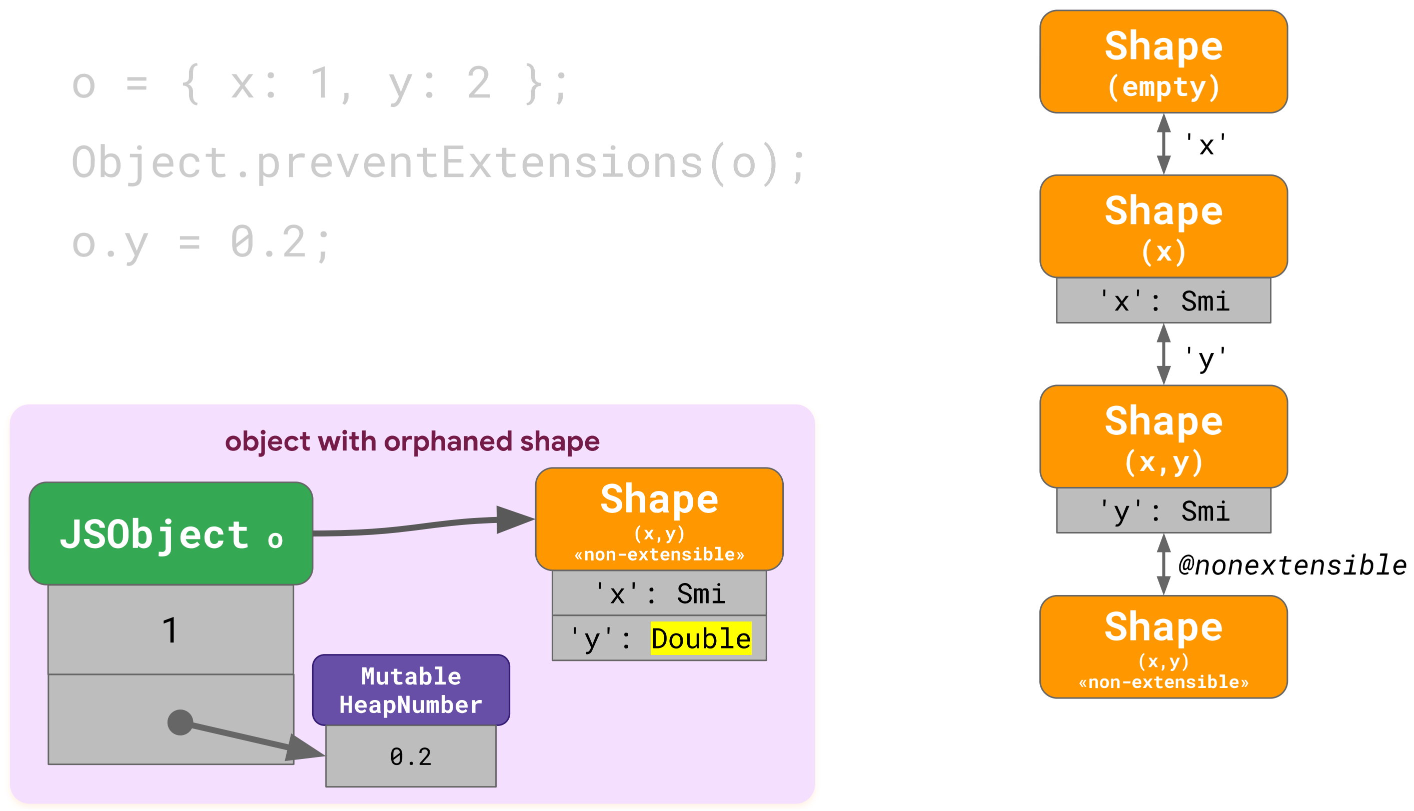
We have an object with two fields that have `Smi` representation. We prevent any further extensions to the object, and eventually force the second field to `Double` representation.

As we learned before, this creates roughly the following setup:



Both properties are marked as `Smi` representation, and the final transition is the extensibility transition to mark the shape as non-extensible.

Now we need to change `y` to `Double` representation, which means we need to again start by finding the split shape. In this case, it's the shape that introduced `x`. But now V8 got confused, since the split shape was extensible while the current shape was marked as non-extensible. And V8 didn't really know how to replay the transitions properly in this case. So V8 essentially just gave up trying to make sense of this, and instead created a separate shape that is not connected to the existing shape tree and not shared with any other objects. Think of it as an *orphaned shape*:



```
o = { x: 1, y: 2 };
Object.preventExtensions(o);
o.y = 0.2;
```

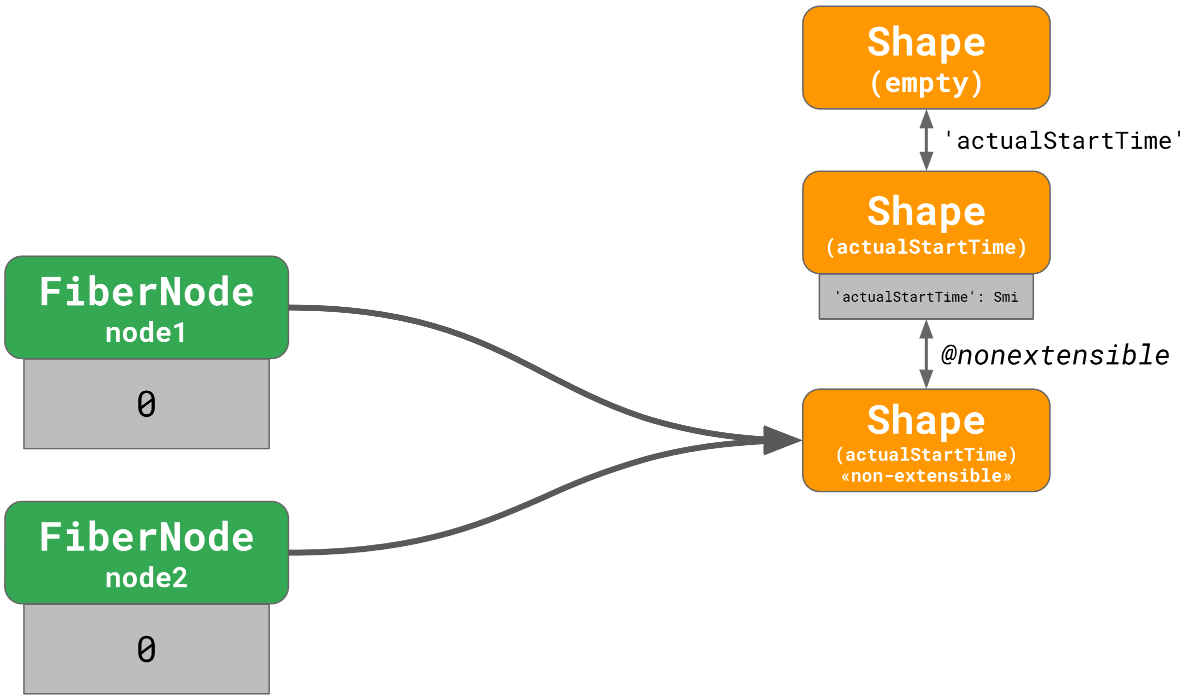
You can imagine it's pretty bad if this happens to lots of objects, since that renders the whole shape system useless.

In the case of React, here's what happened: each `FiberNode` has a couple of fields that are supposed to hold timestamps when profiling is turned on.

```
class FiberNode {
  constructor() {
    this.actualStartTime = 0;
    Object.preventExtensions(this);
  }
}

const node1 = new FiberNode();
const node2 = new FiberNode();
```

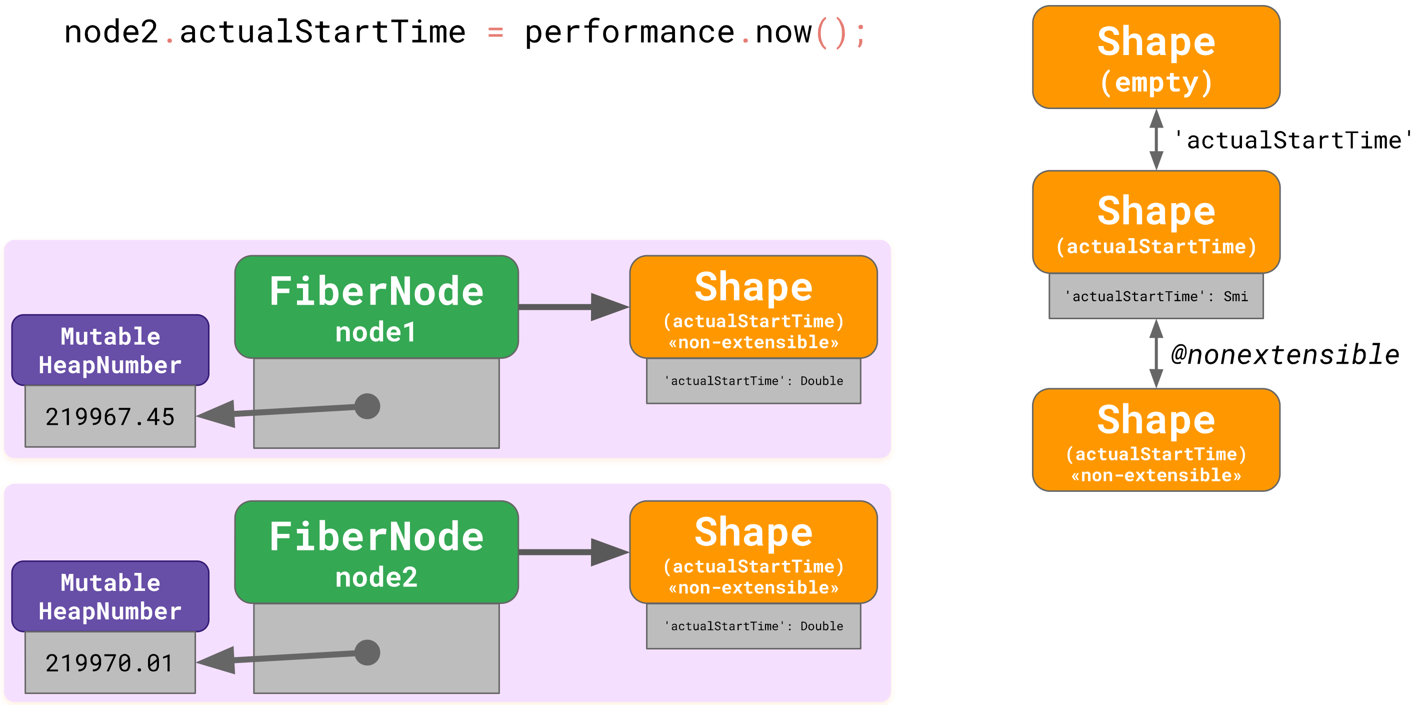
These fields (such as `actualStartTime`) are initialized with `0` or `-1`, and thus start out with `Smi` representation. But later, actual floating-point timestamps from [`performance.now()`] are stored in these fields, causing them to go to `Double` representation, since they don't fit into a `Smi`. On top of that, React also prevents extensions to `FiberNode` instances.

Initially the simplified example above looked like this:



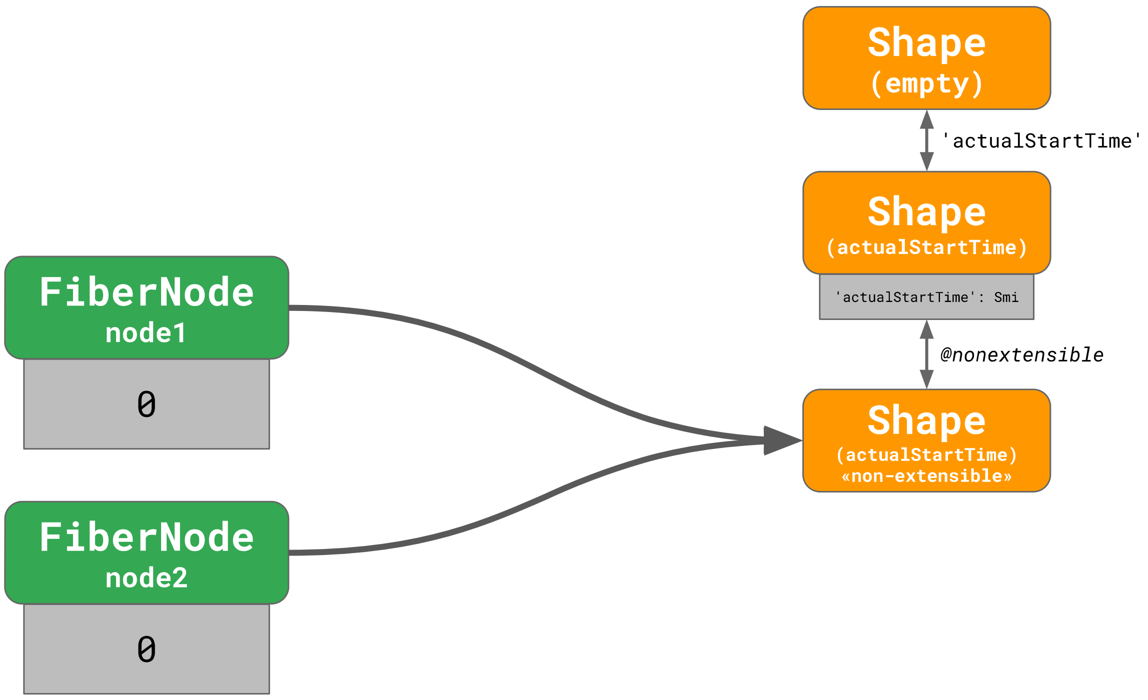
There are two instances sharing a shape tree, all working as intended. But then, as you store the real timestamp, V8 gets confused finding the split shape:

```
node1.actualStartTime = performance.now();
node2.actualStartTime = performance.now();
```



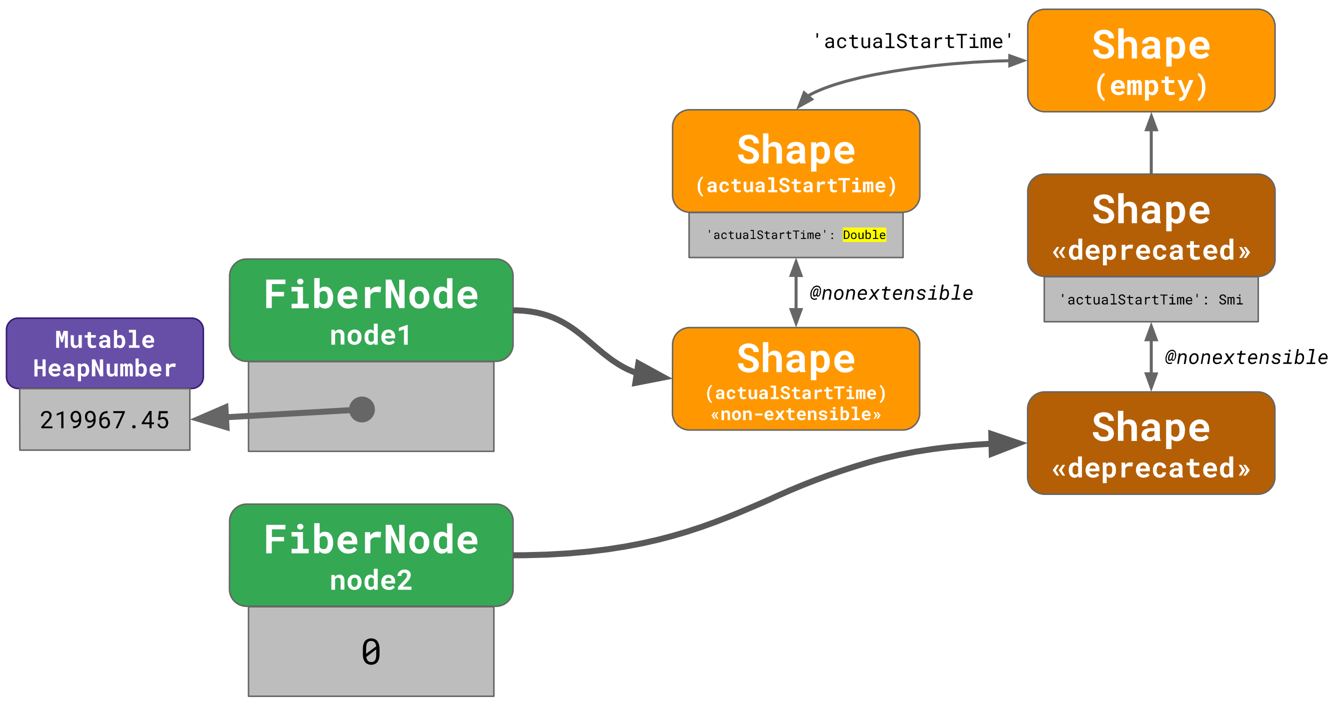
V8 assigns a new orphaned shape to `node1`, and the same thing happens to `node2` some time later, resulting in two *orphan islands*, each with their own disjoint shapes. Many real-world React apps don't just have two, but rather tens of thousands of these `FiberNode`s. As you can imagine, this situation was not particularly great for V8's performance.

Luckily, we've fixed this performance cliff in V8 v7.4, and we're looking into making field representation changes cheaper to remove any remaining performance cliffs. With the fix, V8 now does the right thing:



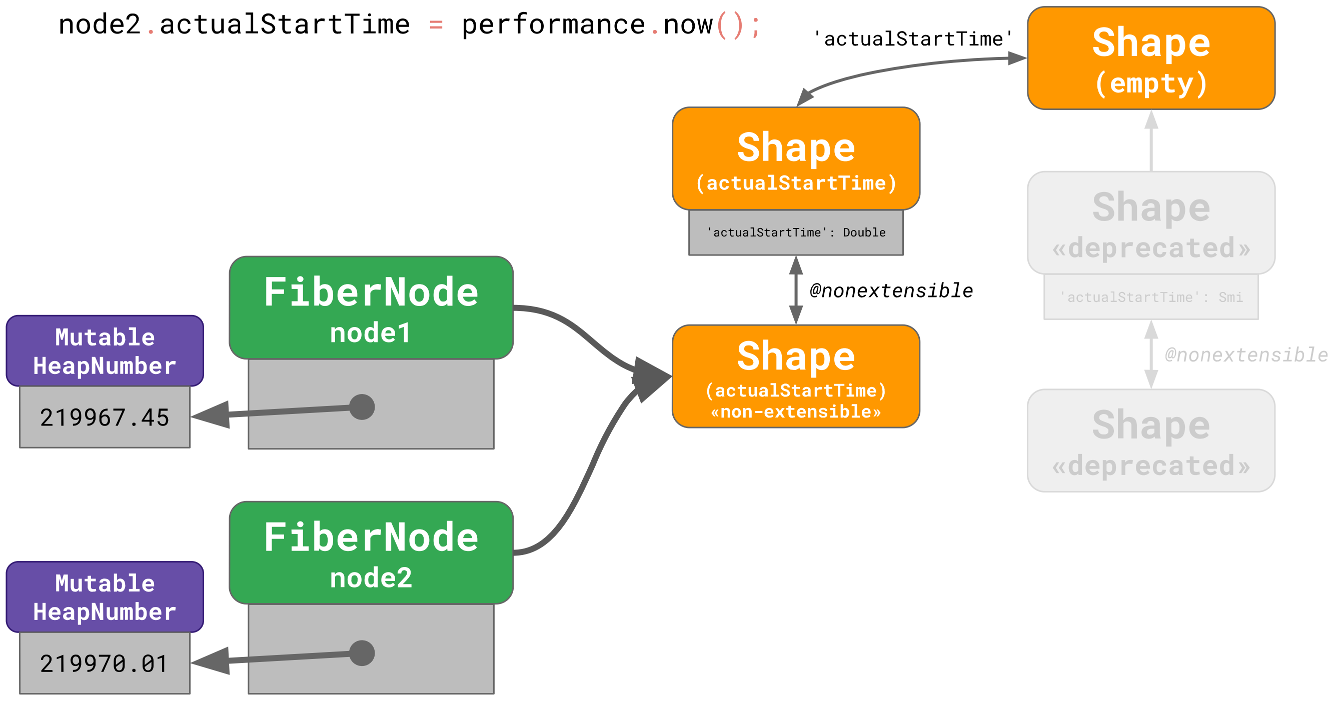
The two `FiberNode` instances point to the non-extensible shape where `'actualStartTime'` is a `Smi` field. When the first assignment to `node1.actualStartTime` happens, a new transition chain is created and the previous chain is marked as deprecated:

```
node1.actualStartTime = performance.now();
```



Note how the extensibility transition is now properly replayed in the new chain.

```
node1.actualStartTime = performance.now();
node2.actualStartTime = performance.now();
```



After the assignment to `node2.actualStartTime`, both nodes refer to the new shape, and the deprecated part of the transition tree can be cleaned up by the garbage collector.

Note: You might think all this shape deprecation/migration is complex, and you'd be right. In fact, we have a suspicion that on real-world websites it causes more issues (in terms of performance, memory use, and complexity) than it helps, particularly since with pointer compression we'll no longer be able to use it to store double-valued fields in-line in the object. So, we're hoping to remove V8's shape deprecation mechanism entirely. You could say it's **puts on sunglasses** being deprecated. *YEEEAAAHHH…*

The React team mitigated the problem on their end by making sure that all the time and duration fields on `FiberNode`s start out with `Double` representation:

```
class FiberNode {
  constructor() {
    // Force `Double` representation from the start.
    this.actualStartTime = Number.NaN;
    // Later, you can still initialize to the value you want:
    this.actualStartTime = 0;
    Object.preventExtensions(this);
  }
}

const node1 = new FiberNode();
const node2 = new FiberNode();
```

Instead of `Number.NaN`, any floating-point value that doesn't fit the `Smi` range could be used. Examples include `0.000001`, `Number.MIN_VALUE`, `-0`, and `Infinity`.

It's worth pointing out that the concrete React bug was V8-specific and that in general, developers shouldn't optimize for a specific version of a JavaScript engine. Still, it's nice to have a handle when things don't work.

Keep in mind that the JavaScript engine performs some magic under the hood, and you can help it by not mixing types if possible. For example, don't initialize your numeric fields with `null`, as that disables all the benefits from the field representation tracking, and it makes your code more readable:

```
// Don't do this!
class Point {
  x = null;
  y = null;
}

const p = new Point();
p.x = 0.1;
p.y = 402;
```

In other words, **write readable code, and performance will follow!**

Take-aways

We’ve covered the following in this deep-dive:

- JavaScript distinguishes between “primitives” and “objects”, and `typeof` is a liar.
- Even values with the same JavaScript type can have different representations behind the scenes.
- V8 tries to find the optimal representation for every property in your JavaScript programs.
- We’ve discussed how V8 deals with shape deprecations and migrations, including extensibility transitions.

Based on this knowledge, we identified some practical JavaScript coding tips that can help boost performance:

- Always initialize your objects in the same way, so that shapes can be effective.
- Choose sensible initial values for your fields to help JavaScript engines with representation selection.

Posted by Benedikt Meurer (@bmeurer) and Mathias Bynens (@mathias).

Retweet this article!

Branding · Terms · Privacy · Twitter · Edit this page on GitHub   Dark Theme