← Asynchronous stack traces: why `await` beats `Promise#then()`

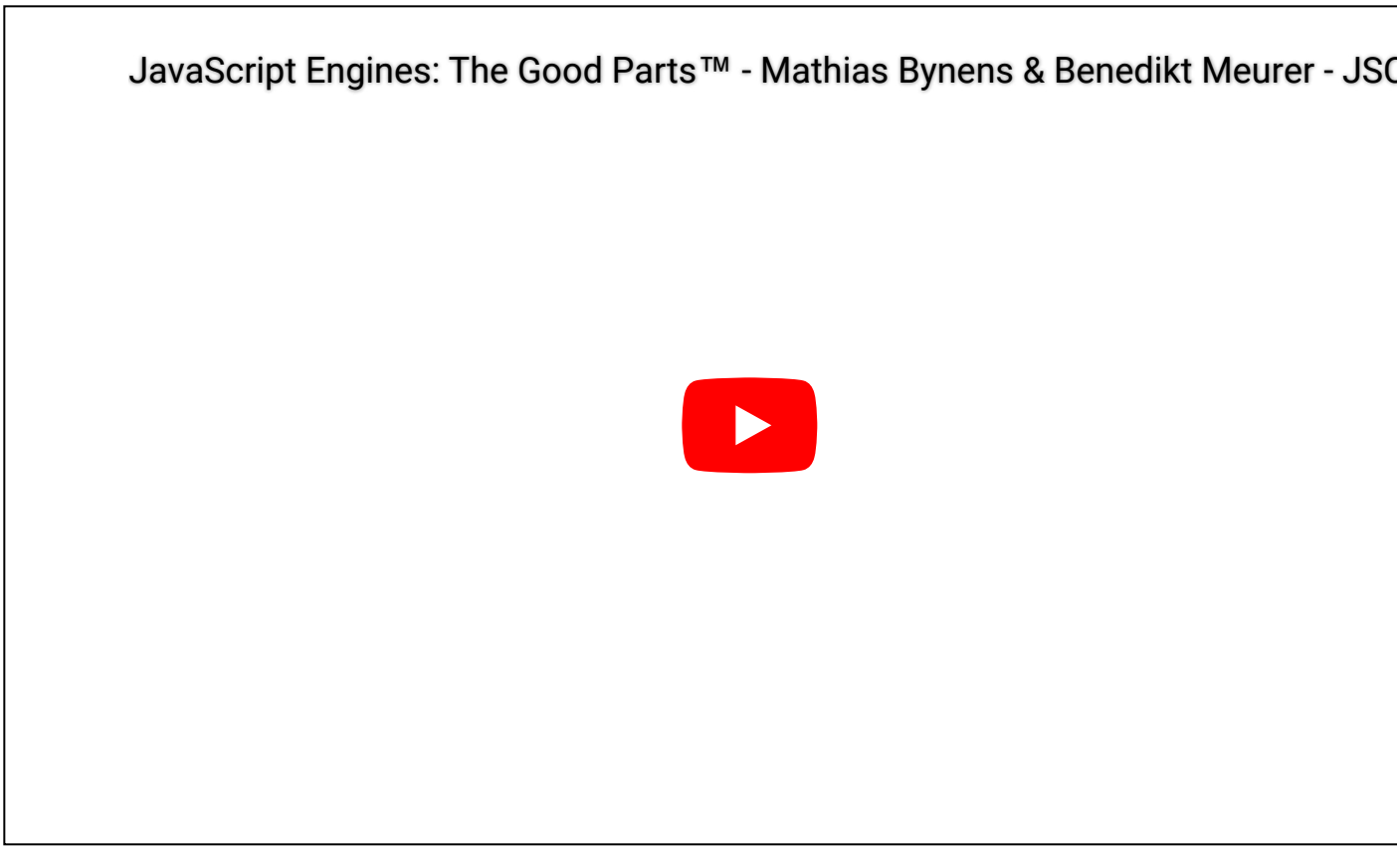
JavaScript engine fundamentals: optimizing prototypes →

JavaScript engine fundamentals: Shapes and Inline Caches

Published 14th June 2018 · tagged with JavaScript, performance

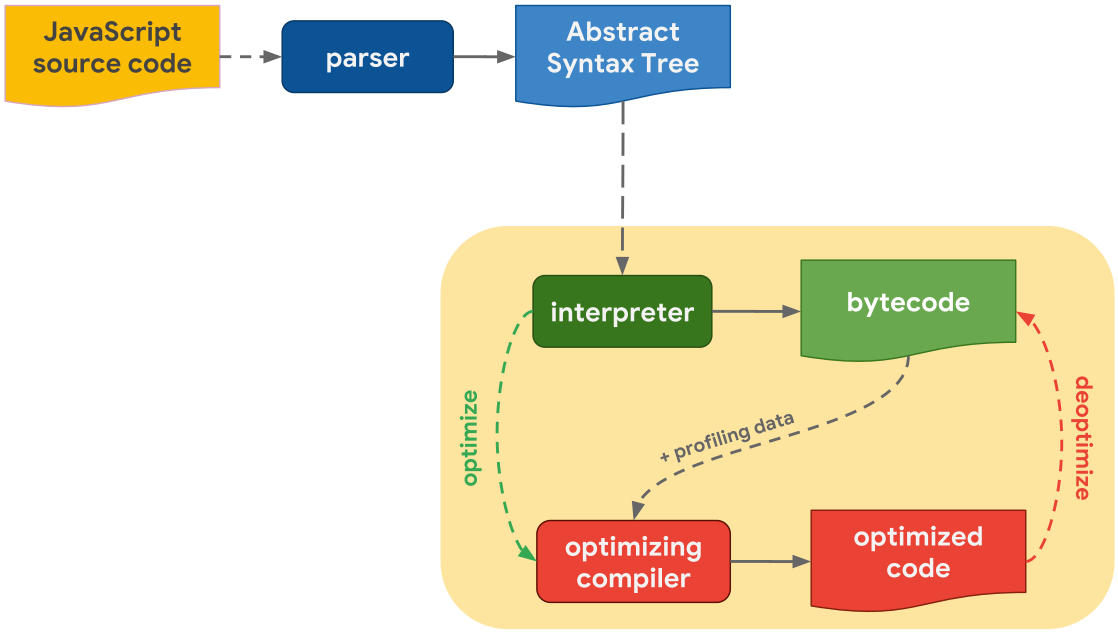
This article describes some key fundamentals that are common to all JavaScript engines — and not just V8, the engine the authors (Benedikt and Mathias) work on. As a JavaScript developer, having a deeper understanding of how JavaScript engines work helps you reason about the performance characteristics of your code.

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



The JavaScript engine pipeline

It all starts with the JavaScript code you write. The JavaScript engine parses the source code and turns it into an Abstract Syntax Tree (AST). Based on that AST, the interpreter can start to do its thing and produce bytecode. Great! At that point the engine is actually running the JavaScript code.



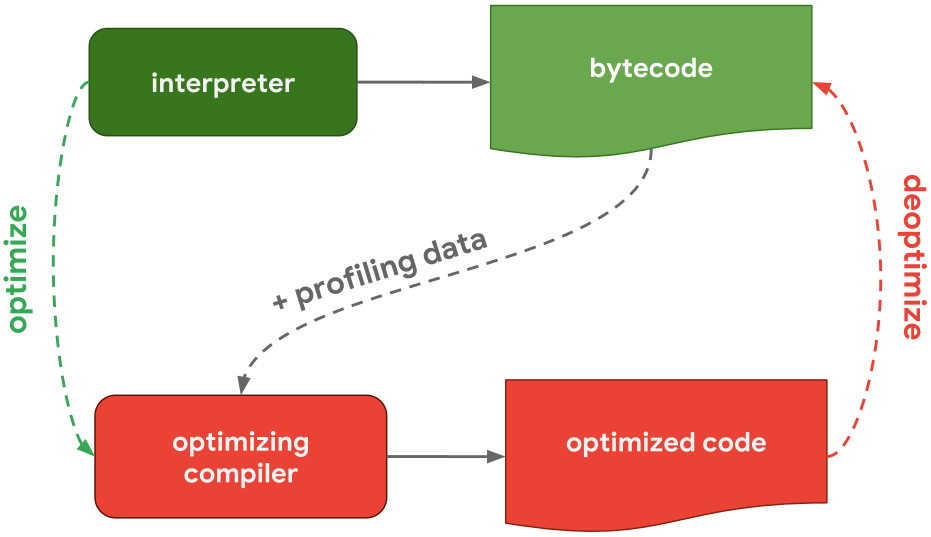
To make it run faster, the bytecode can be sent to the optimizing compiler along with profiling data. The optimizing compiler makes certain assumptions based on the profiling data it has, and then produces highly-optimized machine code.

If at some point one of the assumptions turns out to be incorrect, the optimizing compiler deoptimizes and goes back to the interpreter.

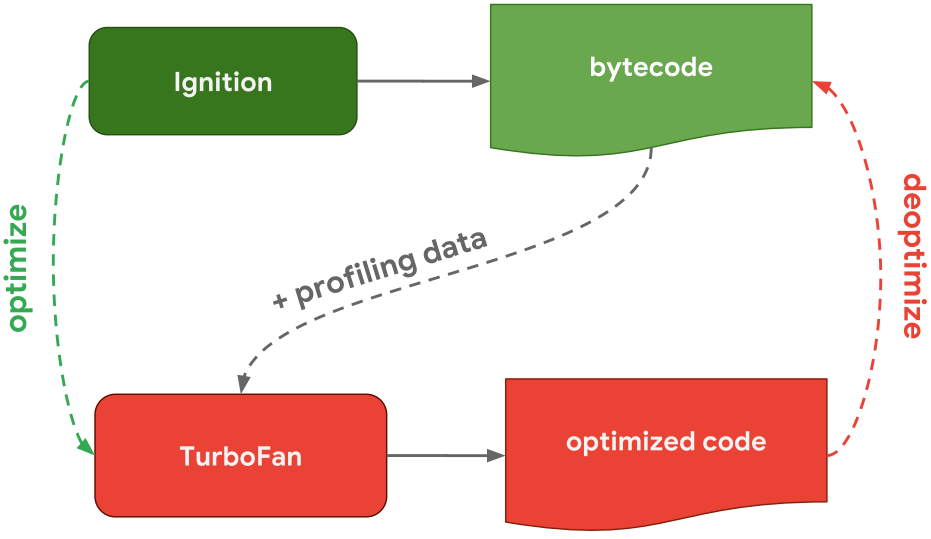
Interpreter/compiler pipelines in JavaScript engines

Now, let's zoom in on the parts of this pipeline that actually run your JavaScript code, i.e. where code gets interpreted and optimized, and go over some of the differences between major JavaScript engines.

Generally speaking, there's a pipeline containing an interpreter and an optimizing compiler. The interpreter generates unoptimized bytecode quickly, and the optimizing compiler takes a little longer but eventually produces highly-optimized machine code.

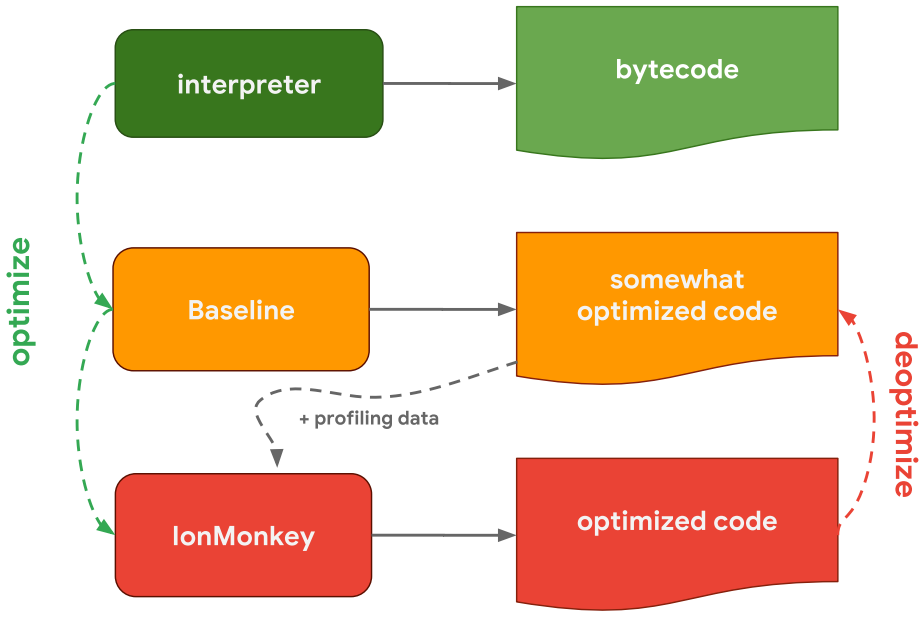


This generic pipeline is pretty much exactly how V8, the JavaScript engine used in Chrome and Node.js, works:

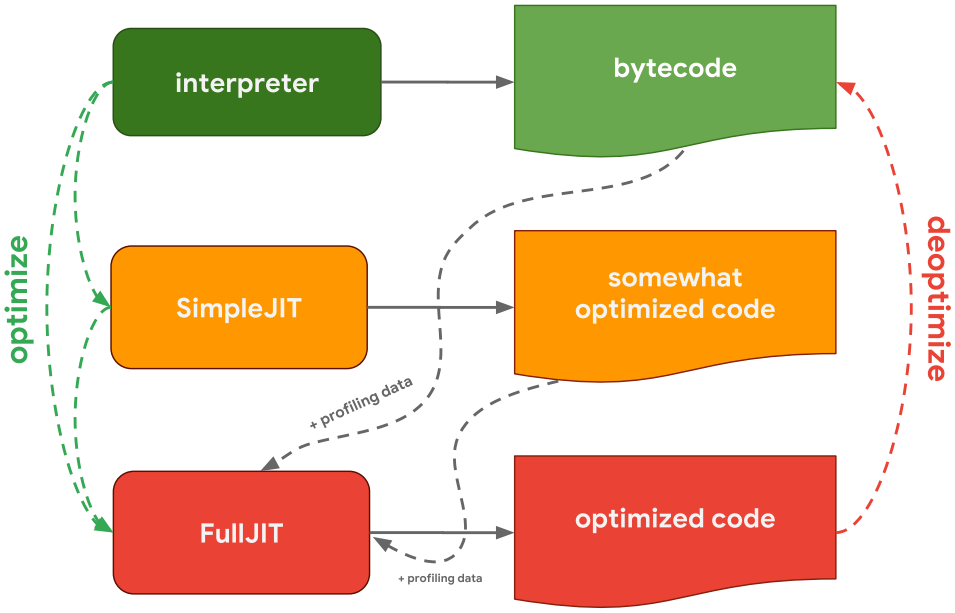


The interpreter in V8 is called Ignition, and is responsible for generating and executing bytecode. While it runs the bytecode, it collects profiling data, which can be used to speed up

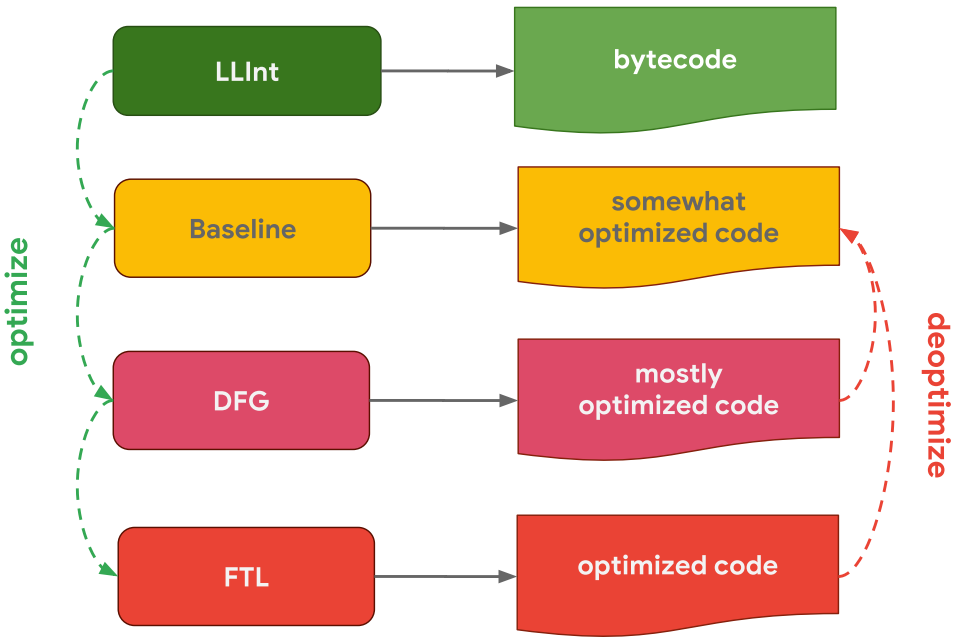
the execution later. When a function becomes *hot*, for example when it's run often, the generated bytecode **and** the profiling data are passed on to TurboFan, our optimizing compiler, to generate highly-optimized machine code based on the profiling data.



SpiderMonkey, Mozilla's JavaScript engine as used in Firefox and in SpiderNode, does it a little differently. They have not one but two optimizing compilers. The interpreter optimizes into the Baseline compiler, which produces somewhat optimized code. Combined with profiling data gathered while running the code, the IonMonkey compiler can produce heavily-optimized code. If the speculative optimization fails, IonMonkey falls back to the Baseline code.



Chakra, Microsoft's JavaScript engine as used in Edge and Node-ChakraCore, has a very similar setup with two optimizing compilers. The interpreter optimizes into SimpleJIT — where JIT stands for Just-In-Time compiler — which produces somewhat optimized code. Combined with profiling data, the FullJIT can produce more-heavily-optimized code.



JavaScriptCore (abbreviated as JSC), Apple's JavaScript engine as used in Safari and React Native, takes it to the extreme with three different optimizing compilers. LLInt, the Low-Level Interpreter, optimizes into the Baseline compiler, which can then optimize into the DFG (Data Flow Graph) compiler, which can in turn optimize into the FTL (Faster Than Light) compiler.

Why do some engines have more optimizing compilers than others? It's all about trade-offs. An interpreter can produce bytecode quickly, but bytecode is generally not very efficient. An optimizing compiler on the other hand takes a little longer, but eventually produces much more efficient machine code. There is a trade-off between quickly getting code to run (interpreter) or taking some more time, but eventually running the code with optimal performance (optimizing compiler). Some engines choose to add multiple optimizing compilers with different time/efficiency characteristics, allowing for more fine-grained control over these trade-offs at the cost of additional complexity. Another trade-off relates to memory usage; see our follow-up article for details on that.

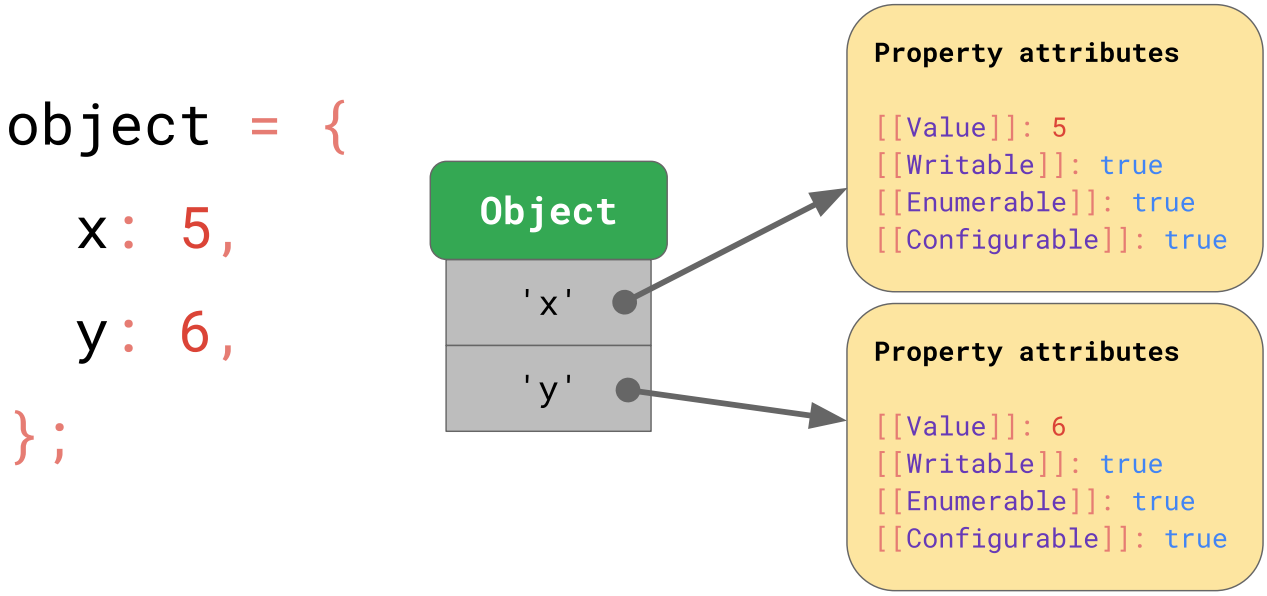
We've just highlighted the main differences in the interpreter and optimizing compiler pipelines for each JavaScript engine. But besides these differences, at a high level, **all JavaScript engines have the same architecture**: there's a parser and some kind of interpreter/compiler pipeline.

JavaScript's object model

Let's look at what else JavaScript engines have in common by zooming in on how some aspects are implemented.

For example, how do JavaScript engines implement the JavaScript object model, and which tricks do they use to speed up accessing properties on JavaScript objects? As it turns out, all major engines implement this very similarly.

The ECMAScript specification essentially defines all objects as dictionaries, with string keys mapping to *property attributes*.



Other than the `[[Value]]` itself, the spec defines these properties:

- `[[Writable]]` which determines whether the property can be reassigned to,
- `[[Enumerable]]` which determines whether the property shows up in `for`-`in` loops,
- and `[[Configurable]]` which determines whether the property can be deleted.

The `[[double square brackets]]` notation looks funky, but that's just how the spec represents properties that aren't directly exposed to JavaScript. You can still get to these property attributes for any given object and property in JavaScript by using the `Object.getOwnPropertyDescriptor` API:

```
const object = { foo: 42 };
Object.getOwnPropertyDescriptor(object, 'foo');
// → { value: 42, writable: true, enumerable: true, configurable: true }
```

Ok, so that's how JavaScript defines objects. What about arrays?

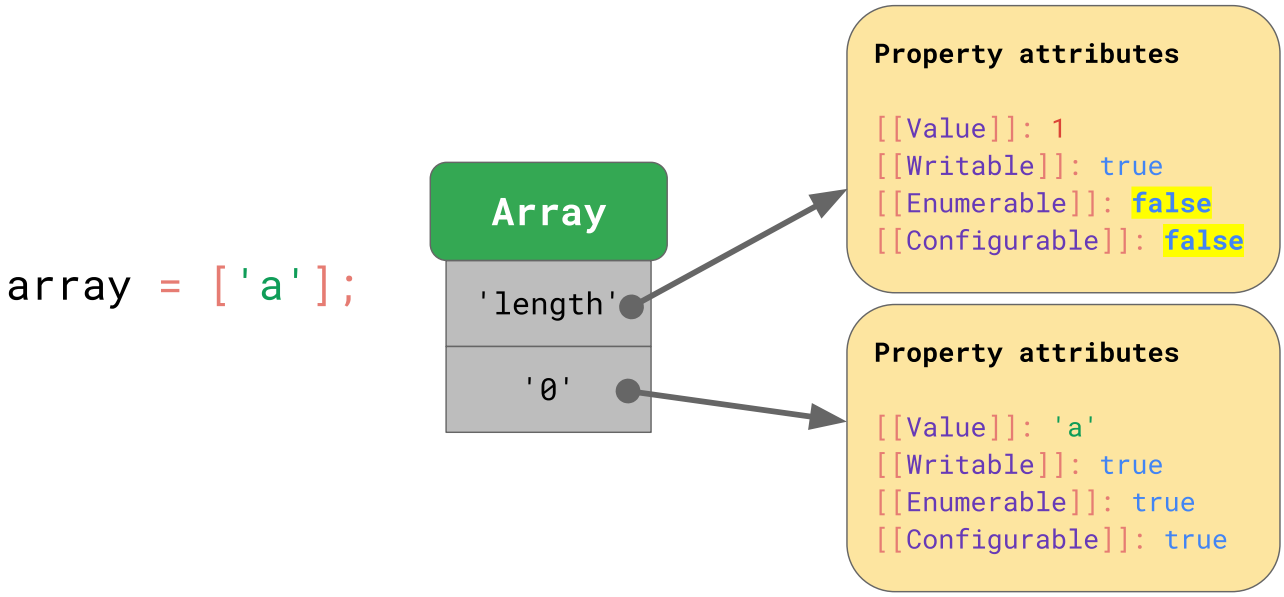
You can think of arrays as a special case of objects. One difference is that arrays have special handling of array indices. Here *array index* is a special term in the ECMAScript specification. Arrays are limited to $2^{32}-1$ items in JavaScript. An array index is any valid index within that limit, i.e. any integer number from 0 to $2^{32}-2$.

Another difference is that arrays also have a magical `length` property.

```
const array = ['a', 'b'];
array.length; // → 2
array[2] = 'c';
array.length; // → 3
```

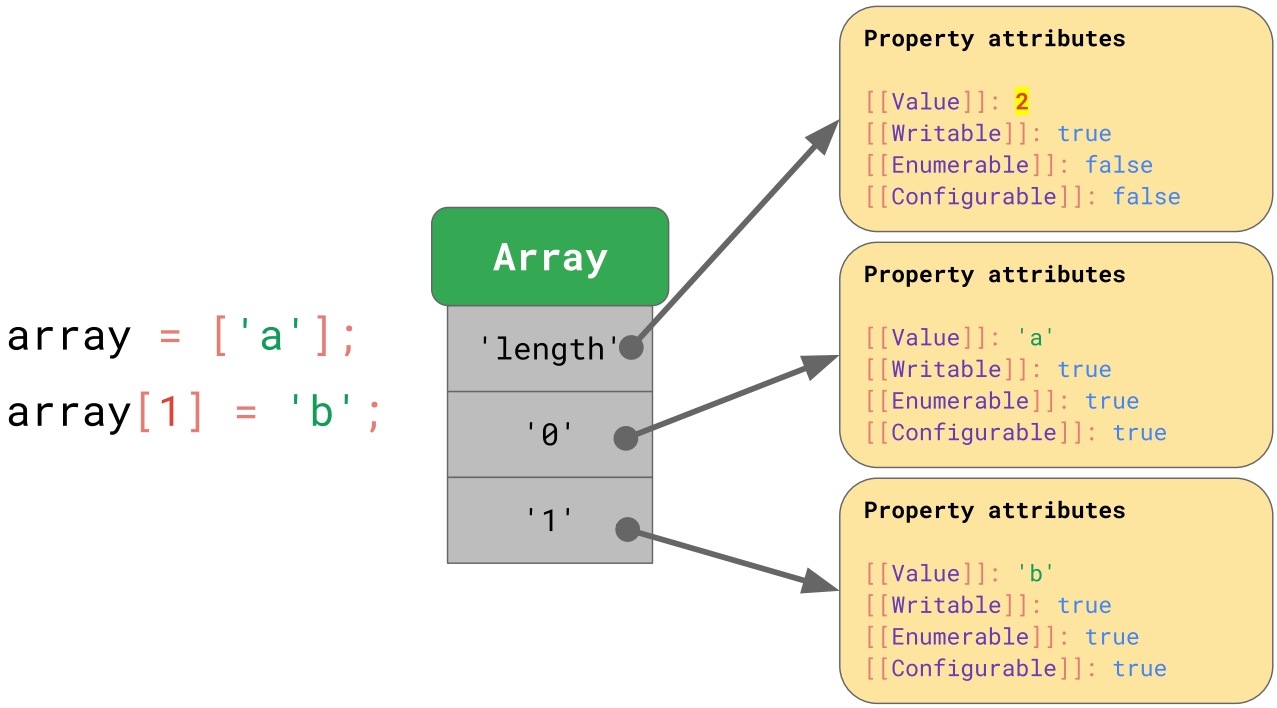
In this example, the array has a `length` of `2` when it's created. Then we assign another element to index `2`, and the `length` automatically updates.

JavaScript defines arrays similarly to objects. For example, all the keys including array indices are represented as strings explicitly. The first element in the array is stored under the key `'0'`.



The `'length'` property is just another property that happens to be non-enumerable and non-configurable.

Once an element is added to the array, JavaScript automatically updates the `[[Value]]` property attribute of the `'length'` property.



Generally speaking, arrays behave pretty similarly to objects.

Optimizing property access

Now that we know how objects are defined in JavaScript, let's dive into how JavaScript engines enable working with objects efficiently.

Looking at JavaScript programs in the wild, accessing properties is by far the most common operation. It's crucial for JavaScript engines to make property access fast.

```
const object = {
  foo: 'bar',
  baz: 'qux',
};

// Here, we're accessing the property `foo` on `object`:
doSomething(object.foo);
//          ^^^^^^^^^^
```

Shapes

In JavaScript programs, it's common to have multiple objects with the same property keys. Such objects have the same *shape*.

```
const object1 = { x: 1, y: 2 };
const object2 = { x: 3, y: 4 };
// `object1` and `object2` have the same shape.
```

It's also very common to access the same property on objects with the same shape:

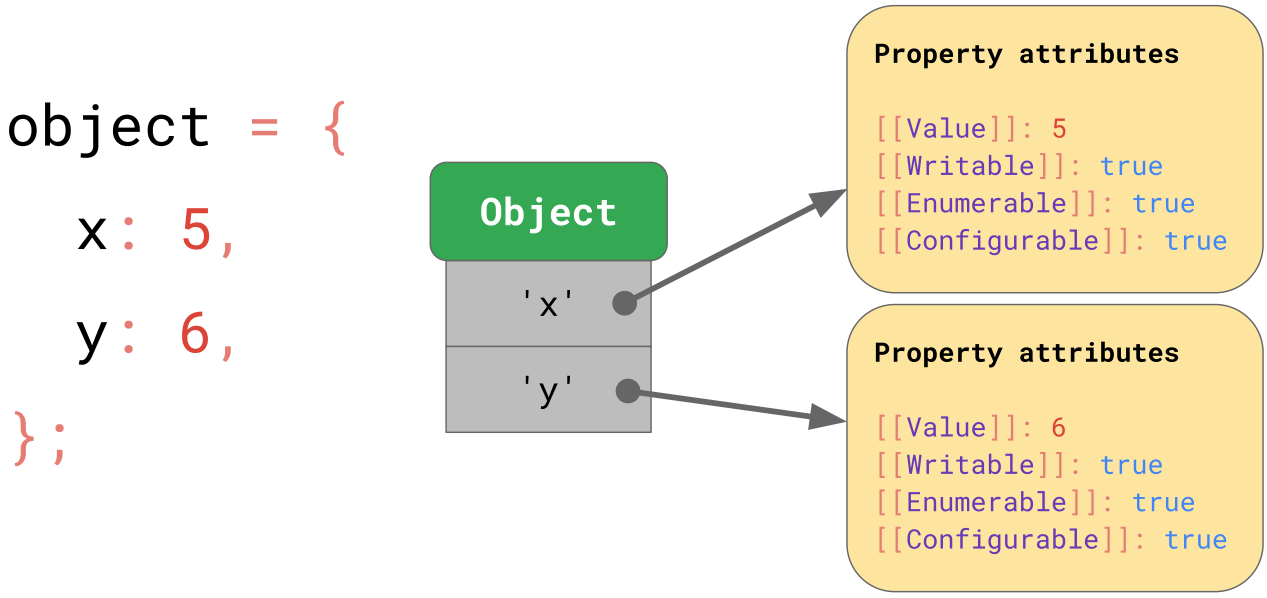
```
function logX(object) {
  console.log(object.x);
  //          ^^^^^^^^
}

const object1 = { x: 1, y: 2 };
const object2 = { x: 3, y: 4 };

logX(object1);
logX(object2);
```

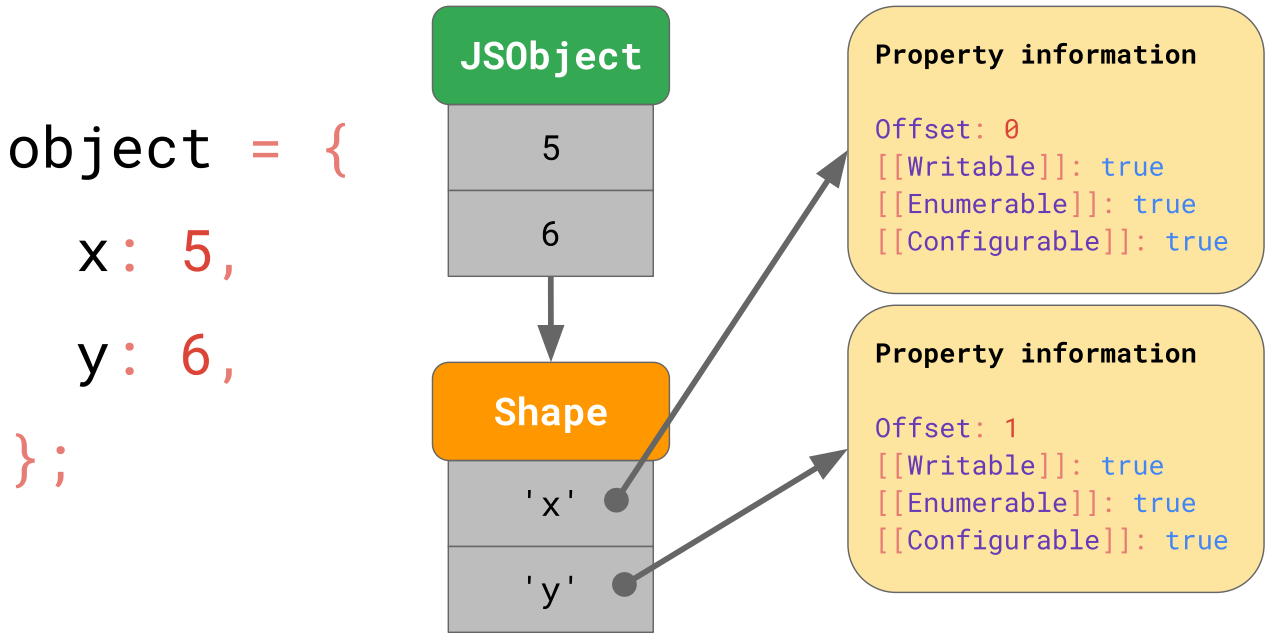
With that in mind, JavaScript engines can optimize object property access based on the object's shape. Here's how that works.

Let's assume we have an object with the properties `x` and `y`, and it uses the dictionary data structure we discussed earlier: it contains the keys as strings, and those point to their respective property attributes.

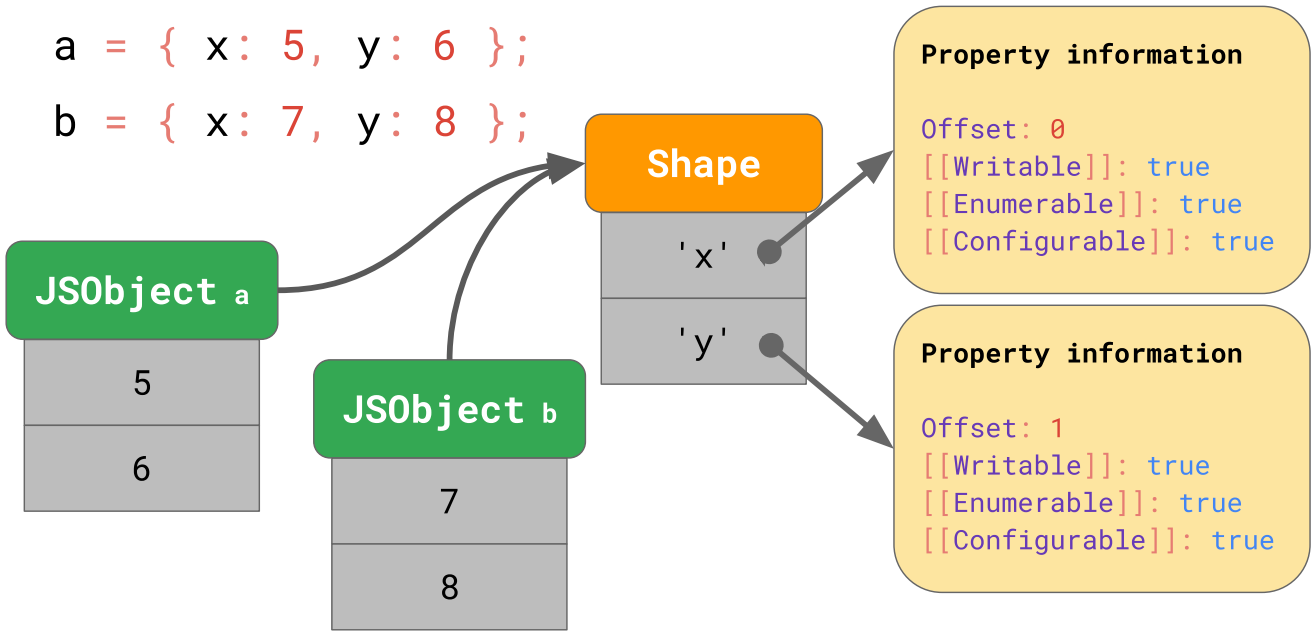


If you access a property, e.g. `object.y`, the JavaScript engine looks in the `JSObject` for the key `'y'`, then loads the corresponding property attributes, and finally returns the `[[value]]`.

But where are these property attributes stored in memory? Should we store them as part of the `JSObject`? If we assume that we'll be seeing more objects with this shape later, then it's wasteful to store the full dictionary containing the property names and attributes on the `JSObject` itself, as the property names are repeated for all objects with the same shape. That's a lot of duplication and unnecessarily memory usage. As an optimization, engines store the `Shape` of the object separately.



This `Shape` contains all the property names and the attributes, except for their `[[value]]`s. Instead the `Shape` contains the offset of the values inside of the `JSObject`, so that the JavaScript engine knows where to find the values. Every `JSObject` with this same shape points to exactly this `Shape` instance. Now every `JSObject` only has to store the values that are unique to this object.



The benefit becomes clear when we have multiple objects. No matter how many objects there are, as long as they have the same shape, we only have to store the shape and property information once!

All JavaScript engines use shapes as an optimization, but they don't all call them shapes:

- Academic papers call them *Hidden Classes* (confusing w.r.t. JavaScript classes)
- V8 calls them *Maps* (confusing w.r.t. JavaScript `Map`s)
- Chakra calls them *Types* (confusing w.r.t. JavaScript's dynamic types and `typeof`)
- JavaScriptCore calls them *Structures*
- SpiderMonkey calls them *Shapes*

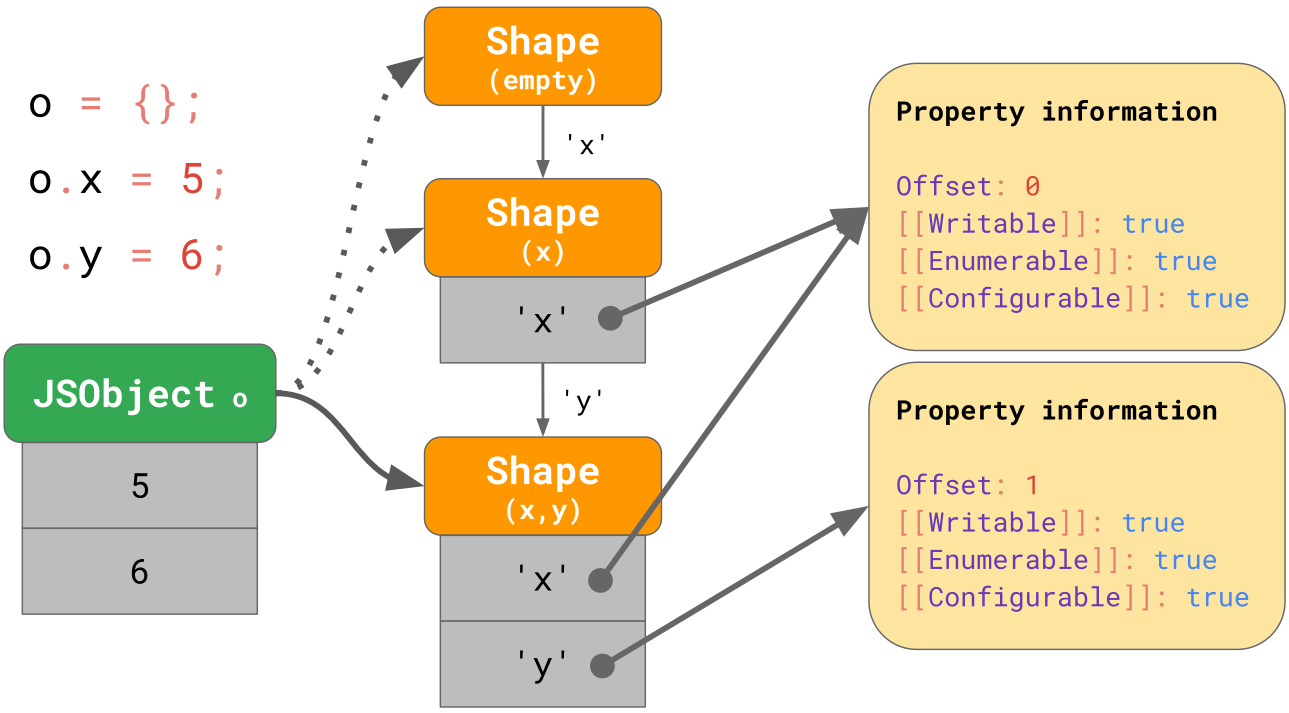
Throughout this article, we'll continue to use the term *shapes*.

Transition chains and trees

What happens if you have an object with a certain shape, but then you add a property to it? How does the JavaScript engine find the new shape?

```
const object = {};
object.x = 5;
object.y = 6;
```

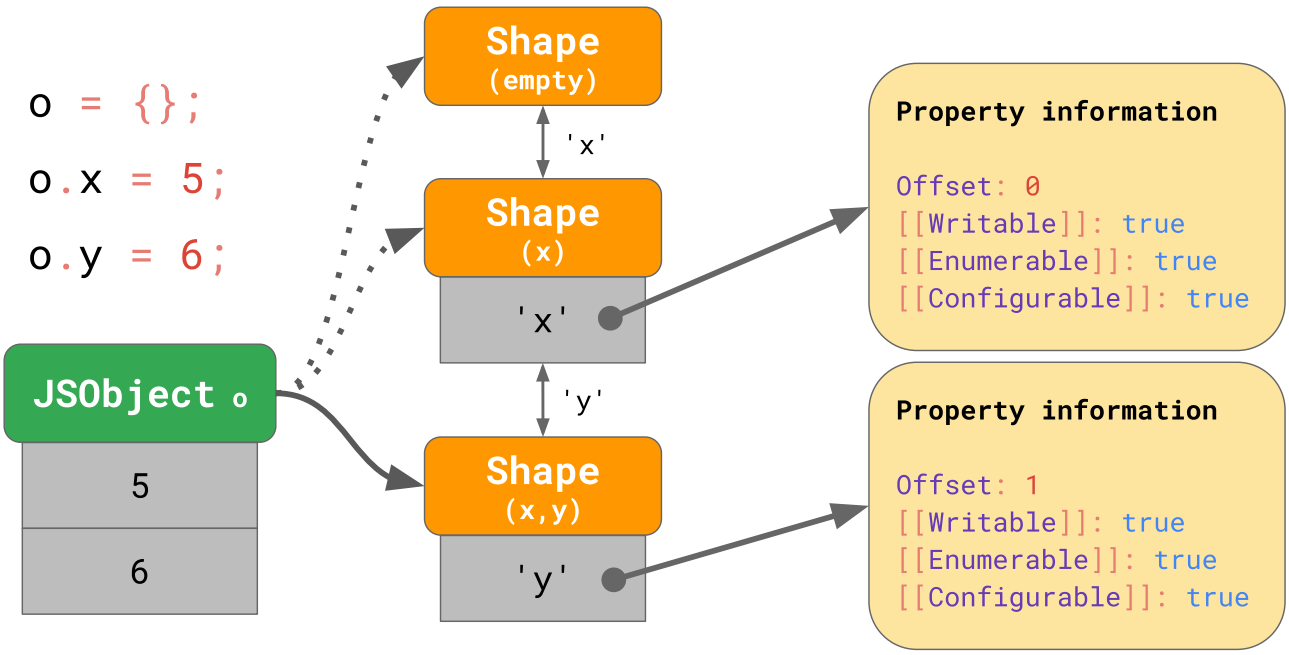
The shapes form so-called *transition chains* in the JavaScript engine. Here's an example:



The object starts out without any properties, so it points to the empty shape. The next statement adds a property `'x'` with a value `5` to this object, so the JavaScript engine transitions to a shape that contains the property `'x'` and a value `5` is added to the `JSObject` at the first offset `0`. The next line adds a property `'y'`, so the engine transitions to yet another shape that contains both `'x'` and `'y'`, and appends the value `6` to the `JSObject` (at offset `1`).

> **Note:** The order in which properties are added impacts the shape. For example, `{ x: 4, y: 5 }` results in a different shape than `{ y: 5, x: 4 }`.

We don't even need to store the full table of properties for each `Shape`. Instead, every `Shape` only needs to know about the new property it introduces. For example, in this case we don't have to store the information about `'x'` in that last shape, because it can be found earlier in the chain. To make this work, every `Shape` links back to its previous shape:

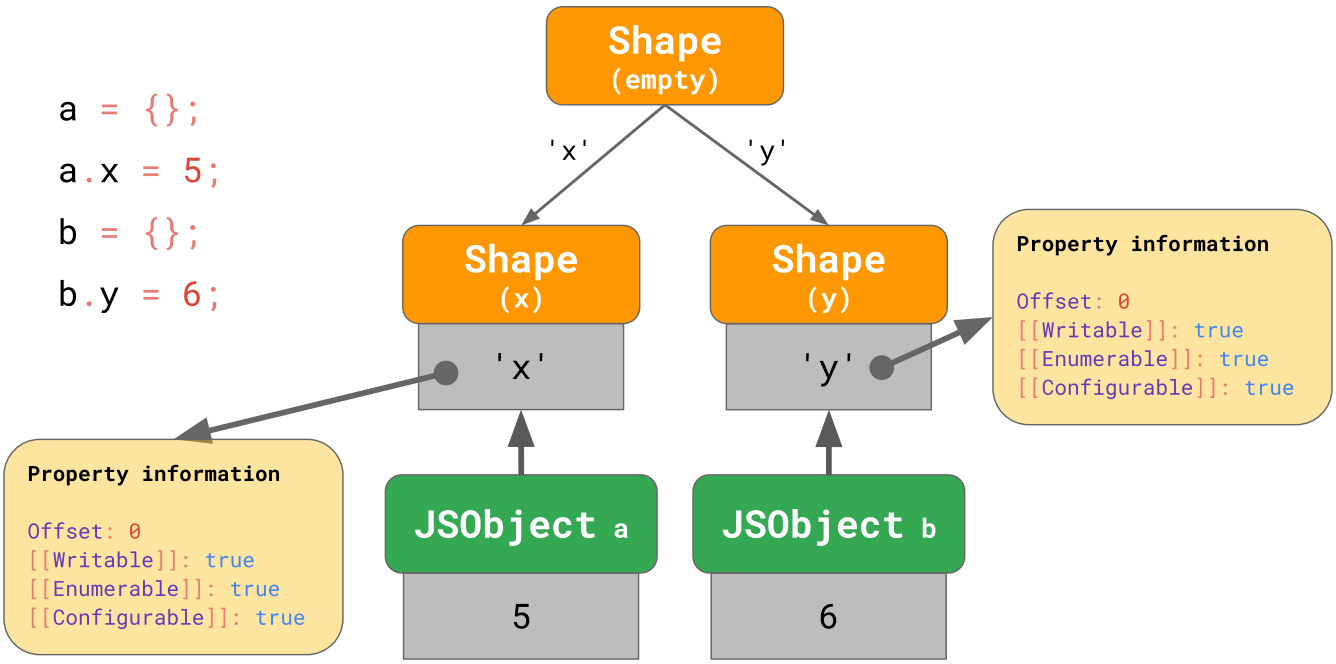


If you write `o.x` in your JavaScript code, the JavaScript engine looks up the property `'x'` by walking up the transition chain until it finds the `Shape` that introduced property `'x'`.

But what happens if there's no way to create a transition chain? For example, what if you have two empty objects, and you add a different property to each of them?

```
const object1 = {};
object1.x = 5;
const object2 = {};
object2.y = 6;
```

In that case we have to branch, and instead of a chain, we end up with a *transition tree*:



Here, we create an empty object `a`, and then add a property `'x'` to it. We end up with a `JSObject` containing a single value, and two `Shapes`: the empty shape, and the shape with only a property `x`.

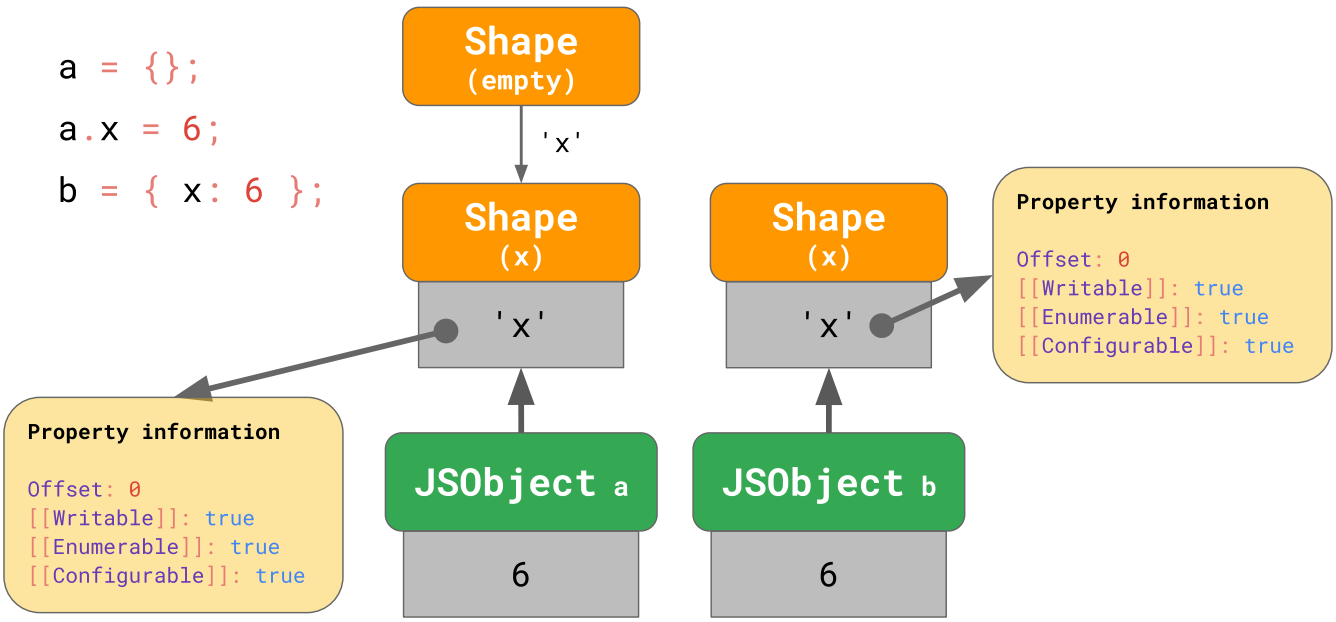
The second example starts with an empty object `b` too, but then adds a different property `'y'`. We end up with two shape chains, and a total of three shapes.

Does that mean we always start at the empty shape? Not necessarily. Engines apply some optimizations for object literals that already contain properties. Let's say we either add `x` starting from the empty object literal, or have an object literal that already contains `x`:

```
const object1 = {};
object1.x = 5;
const object2 = { x: 6 };
```

In the first example, we start at the empty shape and transition to the shape that also contains `x`, just as we saw before.

In the case of `object2`, it makes sense to directly produce objects that already have `x` from the beginning instead of starting from an empty object and transitioning.



The object literal that contains the property `'x'` starts at a shape that contains `'x'` from the beginning, effectively skipping the empty shape. This is what (at least) V8 and SpiderMonkey do. This optimization shortens the transition chains and makes it more efficient to construct objects from literals.

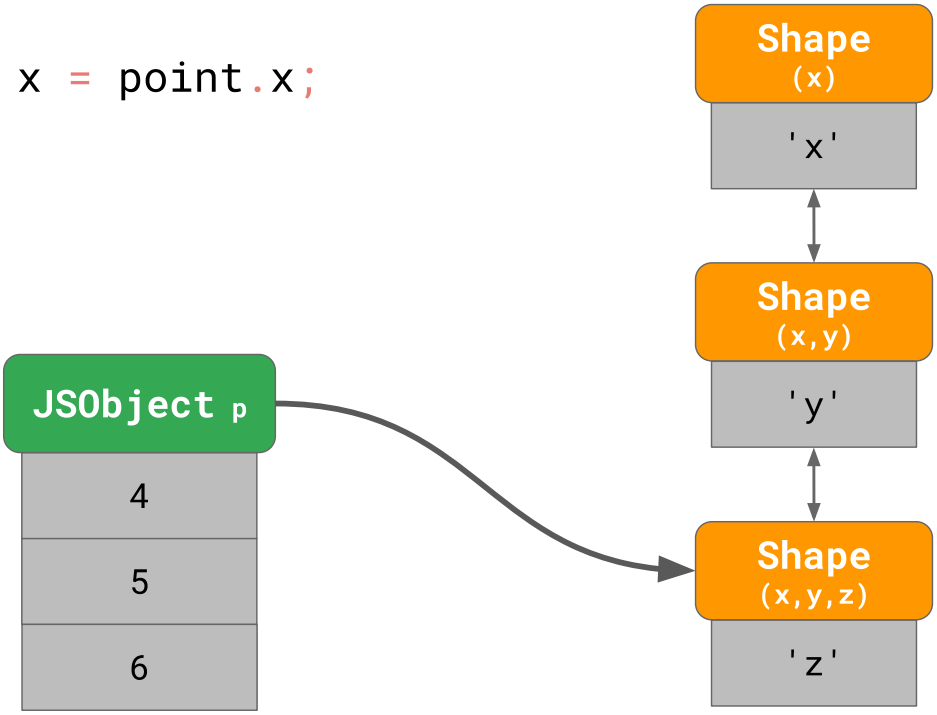
Benedikt's blog post on surprising polymorphism in React applications discusses how these subtleties can affect real-world performance.

Here's an example of a 3D point object with properties `'x'`, `'y'`, and `'z'`.

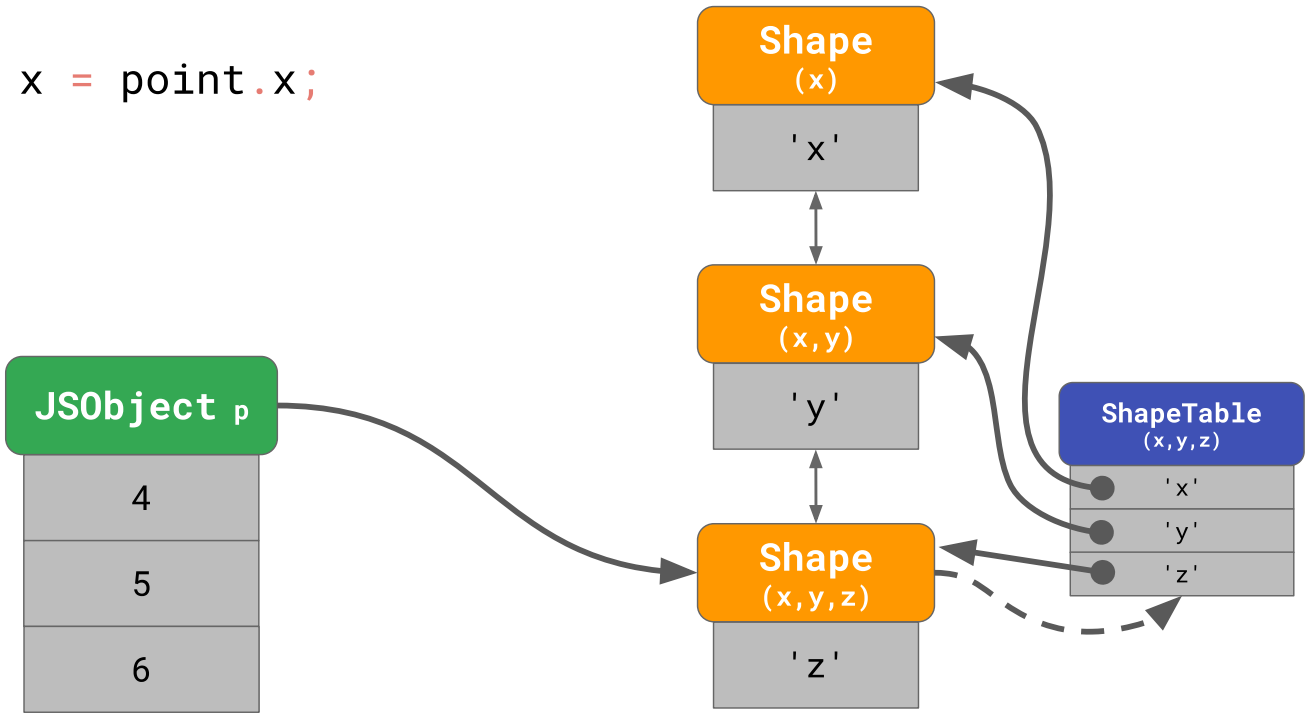
```
const point = {};
point.x = 4;
point.y = 5;
point.z = 6;
```

As we learned before, this creates an object with 3 shapes in memory (not counting the empty shape). To access the property `'x'` on that object, e.g. if you write `point.x` in your program, the JavaScript engine needs to follow the linked list: it starts at the `Shape` at the bottom, and then works its way up to the `Shape` that introduced `'x'` at the top.

```
x = point.x;
```



That's going to be really slow if we do this more often, especially when the objects have lots of properties. The time to find the property is `O(n)`, i.e. linear in the number of properties on the object. To speed up searching for properties, JavaScript engines add a `ShapeTable` data structure. This `ShapeTable` is a dictionary, mapping property keys to the respective `Shape`s that introduce the given property.



Wait a minute, now we're back to dictionary lookups… That's where we were before we started adding `Shape`s in the first place! So why do we bother with shapes at all?

The reason is that shapes enable another optimization called *Inline Caches*.

Inline Caches (ICs)

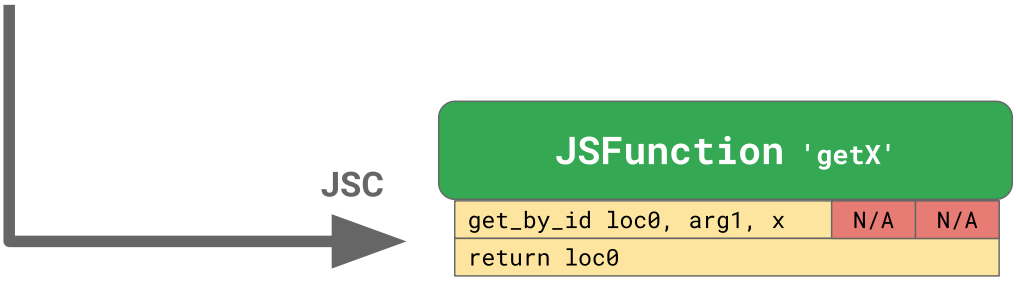
The main motivation behind shapes is the concept of Inline Caches or ICs. ICs are the key ingredient to making JavaScript run fast! JavaScript engines use ICs to memorize information on where to find properties on objects, to reduce the number of expensive lookups.

Here's a function `getX` that takes an object and loads the property `x` from it:

```
function getX(o) {
  return o.x;
}
```

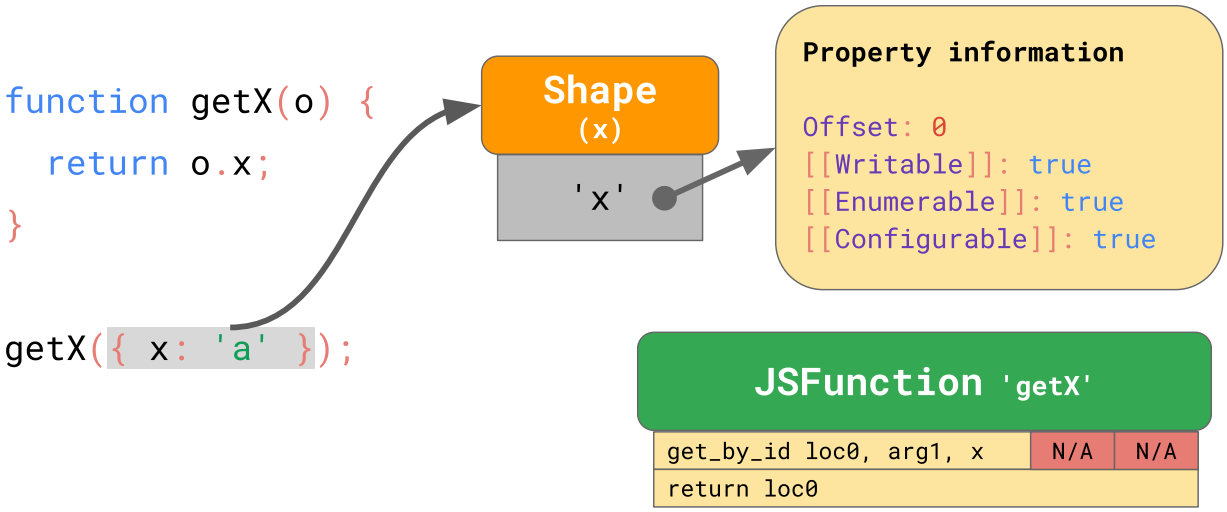
If we run this function in JSC, it generates the following bytecode:

```
function getX(o) {
  return o.x;
}
```

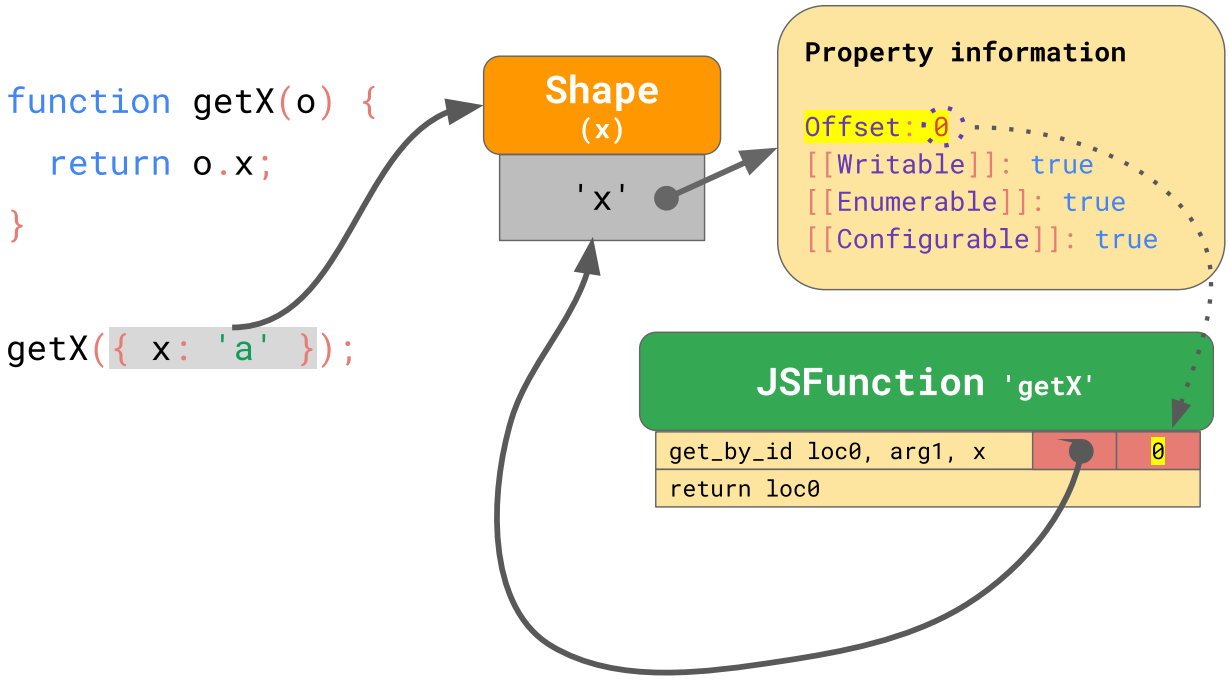


The first `get_by_id` instruction loads the property `'x'` from the first argument (`arg1`) and stores the result into `loc0`. The second instruction returns what we stored to `loc0`.

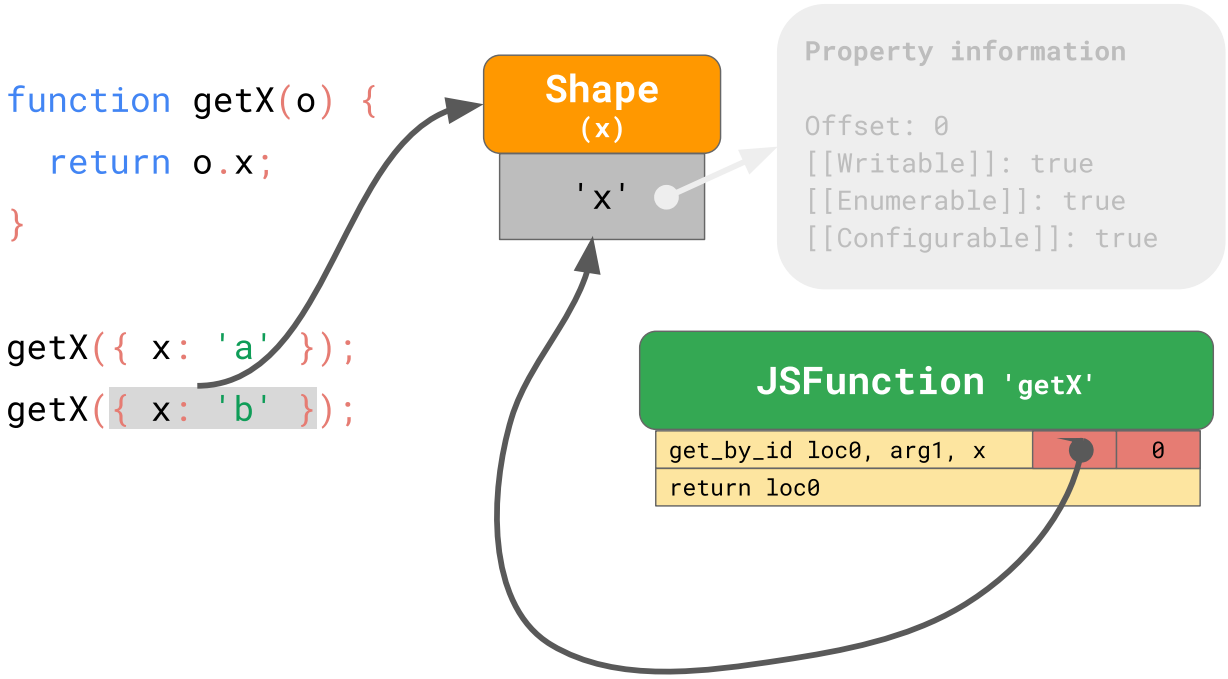
JSC also embeds an Inline Cache into the `get_by_id` instruction, which consists of two uninitialized slots.



Now let's assume we call `getX` with an object `{ x: 'a' }`. As we learned, this object has a shape with property `'x'` and the `Shape` stores the offset and attributes for that property `x`. When you execute the function for the first time, the `get_by_id` instruction looks up the property `'x'` and finds that the value is stored at offset `0`.



The IC embedded into the `get_by_id` instruction memorizes the shape and the offset at which the property was found:



For subsequent runs, the IC only needs to compare the shape, and if it's the same as before, just load the value from the memorized offset. Specifically, if the JavaScript engine sees objects with a shape that the IC recorded before, it no longer needs to reach out to the property information at all — instead, the expensive property information lookup can be skipped entirely. That's significantly faster than looking up the property each time.

Storing arrays efficiently

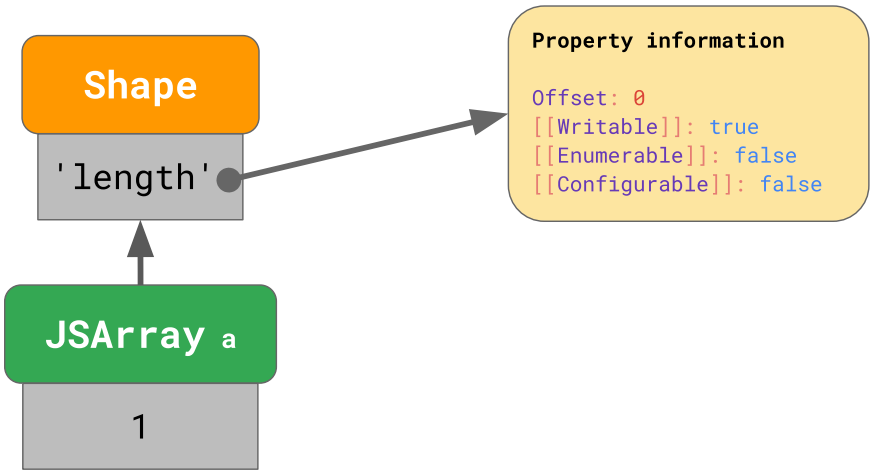
For arrays, it's common to store properties that are *array indices*. The values for such properties are called array elements. It would be wasteful memory-wise to store property attributes for each and every array element in every single array. Instead, JavaScript engines use the fact that array-indexed properties are writable, enumerable, and configurable by default, and store array elements separately from other named properties.

Consider this array:

```
const array = [
  '#jsconfeu',
];
```

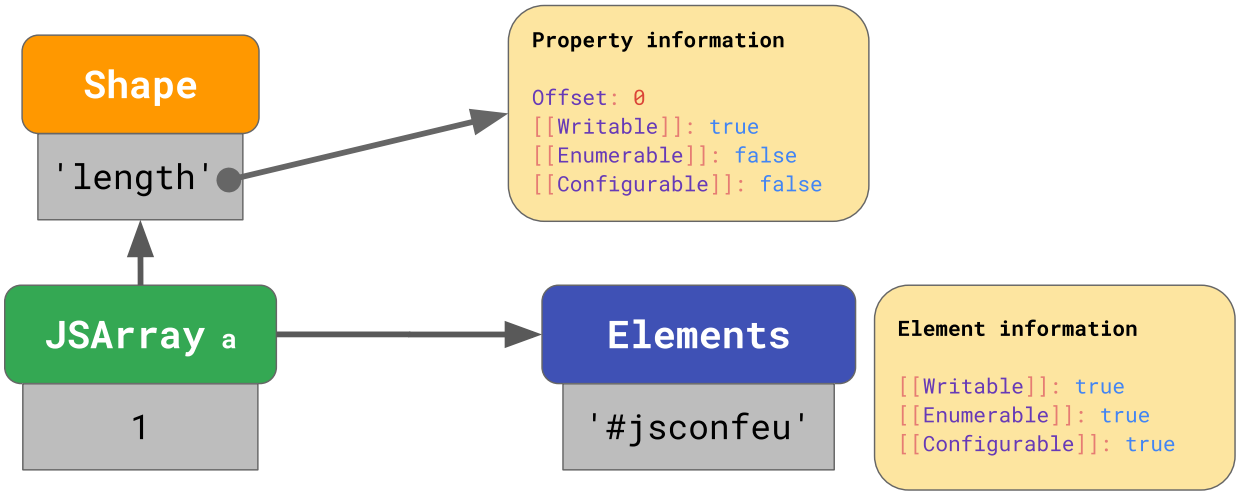
The engine stores the array length (`1`), and points to the `Shape` that contains the offset and the attributes for the `'length'` property.

```
array = ['#jsconfeu'];
```



This is similar to what we've seen before… but where are the array values stored?

```
array = ['#jsconfeu'];
```



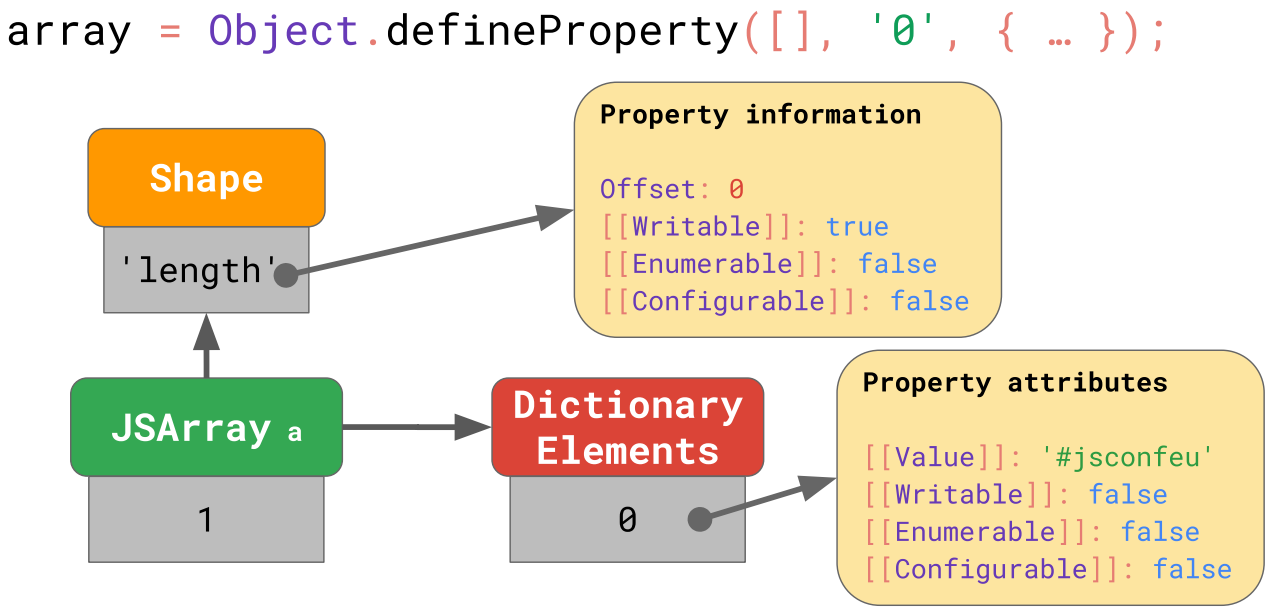
Every array has a separate *elements backing store* that contains all the array-indexed property values. The JavaScript engine doesn't have to store any property attributes for array elements, because usually they are all writable, enumerable, and configurable.

What happens in the unusual case, though? What if you change the property attributes of array elements?

```
// Please don't ever do this!
const array = Object.defineProperty(
  [],
  '0',
  {
    value: 'Oh noes!!1',
    writable: false,
    enumerable: false,
    configurable: false,
  }
);
```

The above snippet defines a property named `'0'` (which happens to be an array index), but sets it attributes to non-default values.

In such edge cases, the JavaScript engine represents the *entire* elements backing store as a dictionary that maps array indices to property attributes.



Even when just a single array element has non-default attributes, the entire array's backing store goes into this slow and inefficient mode. **Avoid `Object.defineProperty` on array**

indices! (I'm not sure why you would even want to do this. It seems like a weird, non-useful thing to do.)

Take-aways

We've learned how JavaScript engines store objects and arrays, and how Shapes and ICs help to optimize common operations on them. 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 they don't end up having different shapes.
- Don't mess with property attributes of array elements, so they can be stored and operated on efficiently.

Translations

- Chinese
- Korean

Note: This article was co-authored by Benedikt Meurer.

Looking for more? If you're interested in how JavaScript engines speed up accesses to prototype properties, read the next article in this series, "JavaScript engine fundamentals: optimizing prototypes".

About me

Hi there! I'm Mathias. I work on Chrome DevTools and the V8 JavaScript engine at Google. HTML, CSS, JavaScript, Unicode, performance, and security get me excited. Follow me on Twitter, Mastodon, and GitHub.

Comments

No comments yet.

Leave a comment