codeburst.io

Node.js V8 internals: an illustrative primer - codeburst

Vardan Grigoryan (vardanator)

15-19 minutes

When you switch from C++ to a more “friendly” language, you get this strange feeling of being fooled by this new language, its environment or its runtime, or something else (can’t tell you that). In C++ you have to think low (I mean low-level), go deeper, you have to understand [at least some] low-level details. As a matter of fact, you have to understand those details just to understand how your code works or could have worked (and then understand more in years to come and still be confused about how difficult it is to master such a language). Server-side JavaScript, Node.js, is really perfect for backend, actually, my experience with Node.js so far was just a REST API development (those nicely structured URLs with corresponding handlers, if this, then send that, else send this, mostly

routine stuff). Node.js took my heart by its asynchronicity, and the part in me, the part who screamed about the greatness of C++ was always whispering those magic words, “but how? How does it work?”.

As a fan of C++ (that doesn’t mean I know it well enough), I loved the idea that Node.js is based on **libuv** (written in C) and **V8** (written in C++, well, for the most part). So it’s kind of easy to dig. Easy peasy, right? Easy.. peasy..

We gonna investigate (sort of) the JavaScript V8 engine.

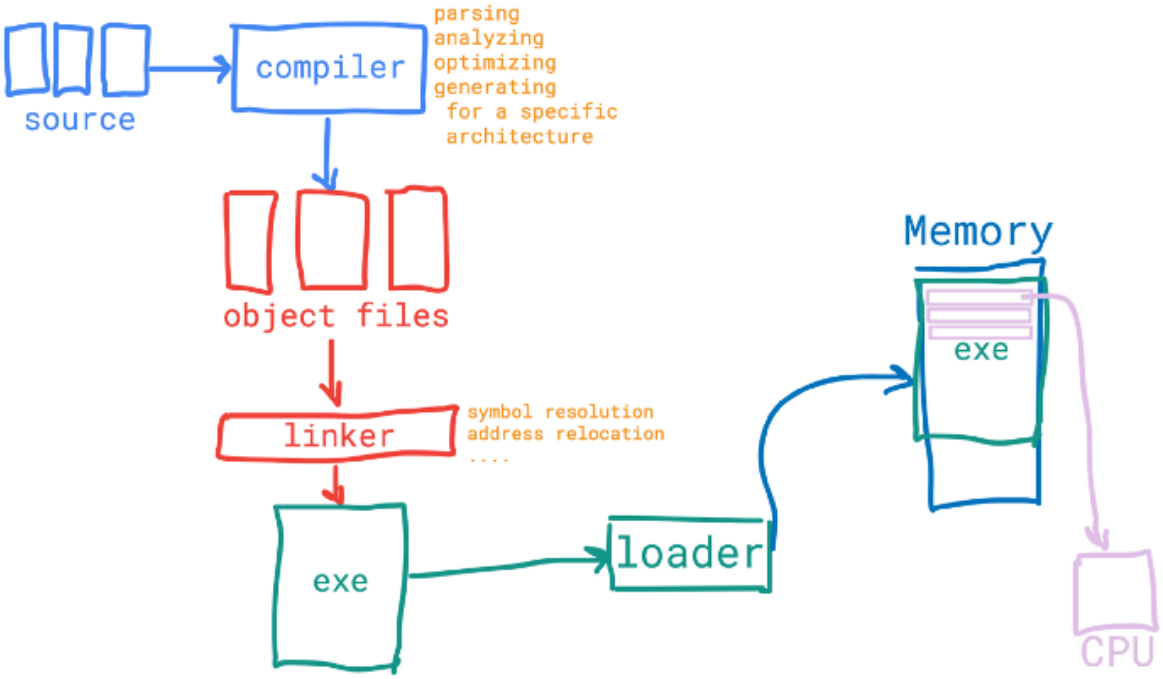
> V8 is open source JavaScript execution engine, which provides a high performance execution environment for JS code.

There are really tons of JS engines, I just like the idea of investigating a product produced by Google engineers. It's kind of a brand name, you see, the engine, it’s not V8, it’s Google’s V8. And why are we gonna dive deeper? Well, besides the fact that it’s really interesting to dig some proof of concept technology, it also will help us to write better code.

First of all, we need to compare the C++ compilation

process with the Node.js interpretation process (from now on I will use JS and Node.js interchangeably). At first, I thought of JS as a “simple” scripting language with a single interpreter doing a routine translation job.

In C++ we got source code, the compiler compiles source code units into object files. Sure there is a pre-processing phase when a special program called preprocessor (I just think of it as a part of the compiler) makes the source code “ready” for compiling. So the source is being compiled into object code. Object code units. Lots of them. (Suppose we have a big enough project).



AoT Compiling and executing (simple abstraction)

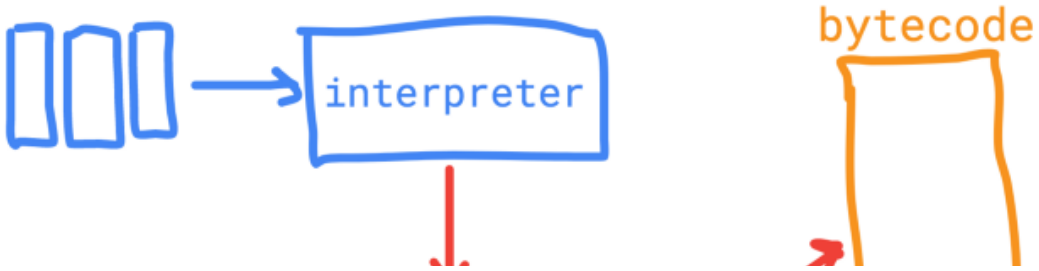
This process is pretty complicated, those who want to

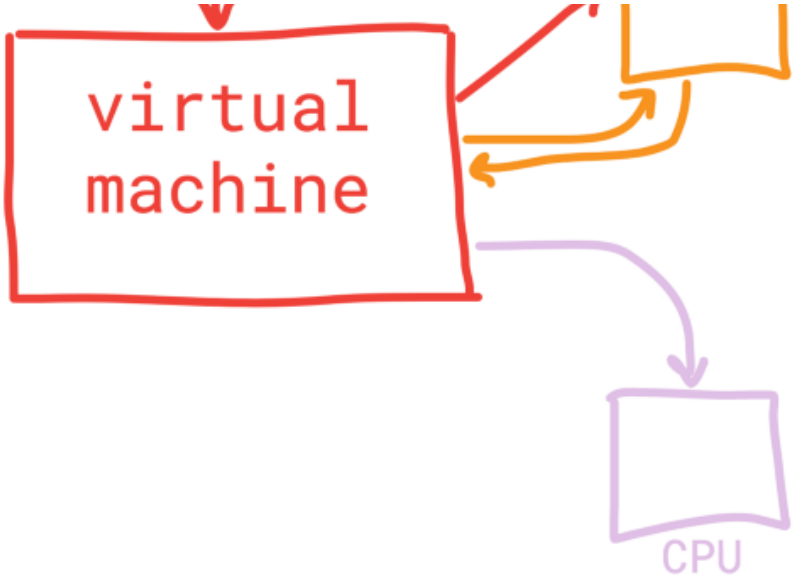
dive deeper into compilation/translation technologies/techniques, it is suggested to read the Dragon book (still I haven't read it, I didn't even start, though I've read the Contents section, the About section, the Acknowledgement section, well, as usual with all my books). During the compiling process, the compiler generates object code as its output, some pretty optimized object code (yes, the machine code). Some developers insist that the compiler optimizes and produces better code than the developer itself. I can't argue with that.

So there goes the **linker**, which is another tool meant to combine those object files, doing some address relocations, symbol resolution, etc. (it's easy to find more info on linkers, really advised to do so). The meaning of the linker is really important, it glues the generated object code units into one big file, **the executable.** The file could be run as a program. Program. Someone must run the program. Not the user, the user is just initiating the start by double-clicking on the corresponding icon. There is another tool, the **loader**. The loader does what its name says, just like Hurry hurries, the Loader loads. The contents of the executable file must be loaded into the virtual

memory by the operating system (loader is the part of the OS). So the loader just copies our executable file contents into the virtual machine, sets the corresponding instruction pointer registers of the CPU to the starting routine (main function?), and voila, the program begins to run. (The most abstract description of a hard process). So no lookbacks, the code is already compiled, the code is fully machine code, generated for the particular machine architecture. No interpretation, the CPU just executes the instructions (machine code instructions) one by one (well, suppose we have a straight CPU). That's why people started to name this compilation AoT compilation. Ahead of Time (AoT), like "there is no need to compile it now, but, yep, you did it ahead of time". The opposite of this is for sure the popular JIT compilation. Just in Time (JiT), like "compile it when you need it". It's like some Nike slogan, Just Compile It!

So here's how the Just in Time compilation (aka interpretation-compilation, or you might say intercompilation) works in my point of view.





Just in Time compilation

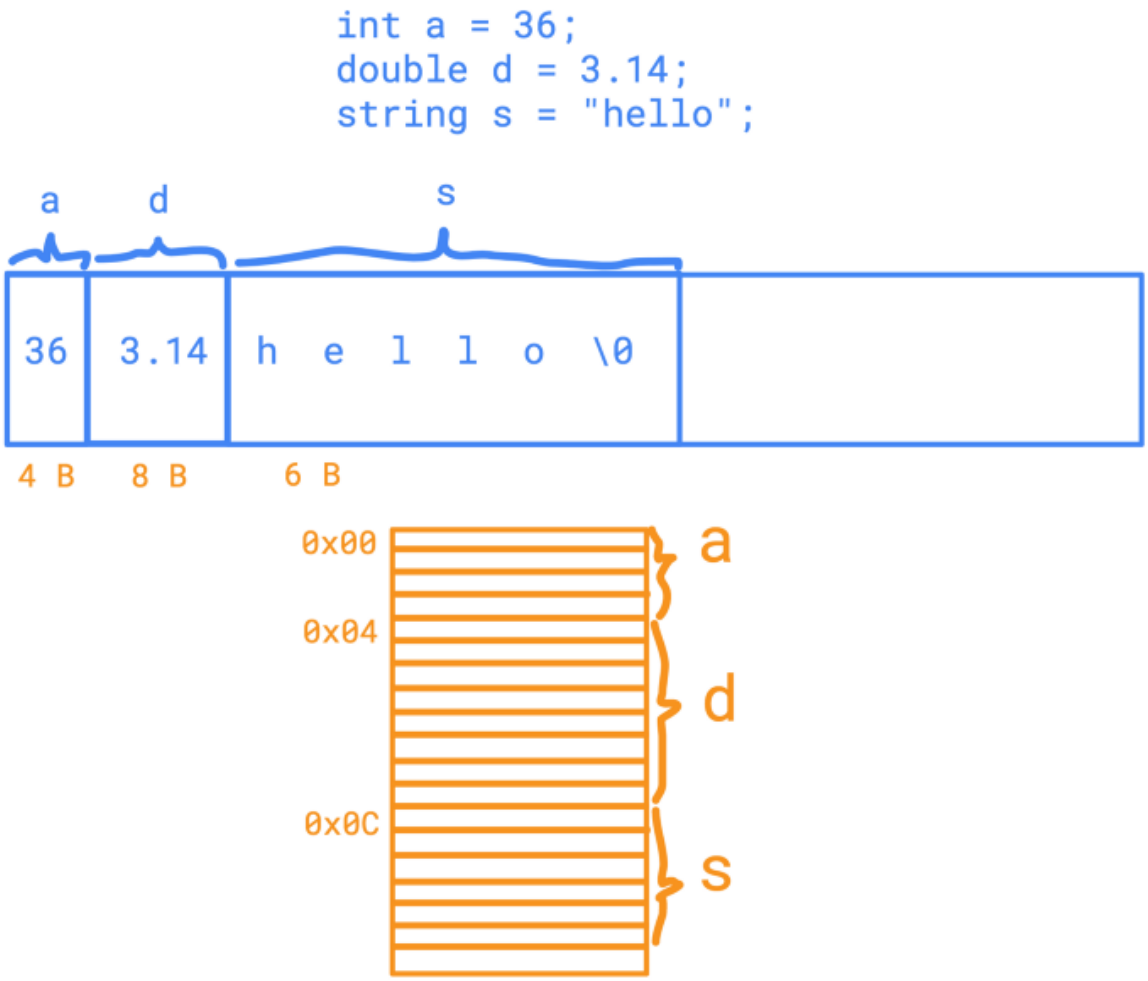
Again, we have a source code (no preprocessing here, no C# allowed), and the source code is being interpreted by the interpreter. Remember Hurry? Hurry hurried, the interpreter interprets. An interpreter is a tool just meant to help the Virtual Machine. A virtual machine is a program that is being installed on a user's computer and supposed to run the code we write. In this particular case the “user’s computer” is our backend server, and the server runs our Node.js code. So the interpreter does not generate a machine code. It doesn’t need to know the different architectures/machines/platforms (Intel, AMD, Windows, Linux, …). The interpreter just needs to know about the virtual machine, well the virtual machine has to know about the architectures/machines/platforms. So

the interpreter generates “code”, some “intermediate code” understandable by the virtual machine, and the virtual machine must generate the actual machine code to run the program. I thought it was just translating a higher-level code into a lower-level code line by line and executing each of it. Turns out, the interpreter and the virtual machine do some really tough stuff. Why so? Why not compile directly into a machine code? That’s a long story, a long and another story. The short answer is “the platforms”. There are so many…

Just cutting to the chase, interpreting is bad, compiling is good. Mm, more practically, interpreting is slow, and the execution of a compiled program is faster. But there are many other “bad” things about JavaScript particularly. First of all, the weak typing. C++ is a strictly typed language, JavaScript is weakly typed. What does that mean? That means we can be free.

Strict typing is good, at least for the compiler itself. The compiler must strictly know the corresponding types of variables. Why? Because the compiler has to generate code. It has to place those variables in the right places, allocate fixed-size boxes for them, and place them as optimized as possible. The compiler can’t afford to place a 4-byte integer into a 20-byte box just because it

could be transformed into some string.



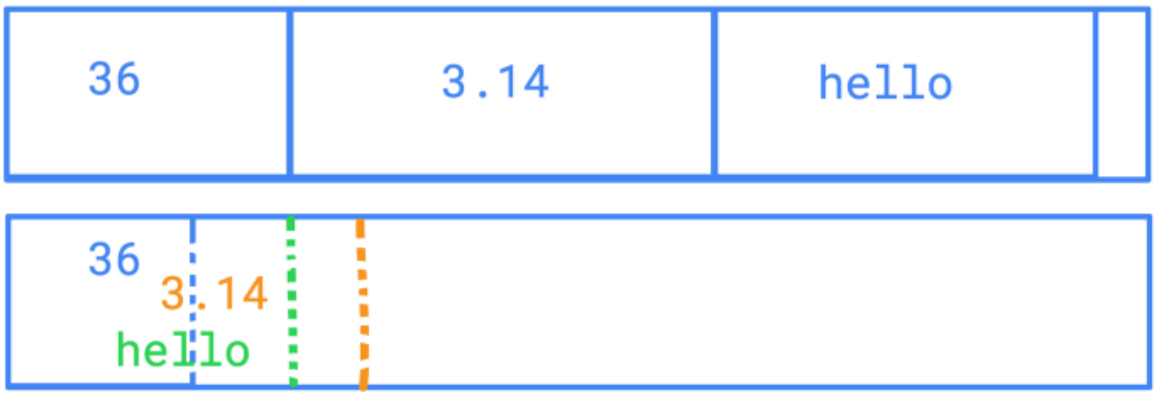
Strict typing helps compiler to generate correct code

The compiler places variables exactly where they “should be”.

Instead of this, you can write something like this in JavaScript without any feeling of incompetence (a little).

```
var some = "this is a string"; // declaring a string
some = 36; // change the type on the fly
some = 3.14;
```

```
var a = 36;
a = 3.14;
a = "hello";
```



Weak typing slows down the execution

How should the interpreter interpret the variable ‘a’ above (sometimes I use interpreter, virtual machine, and execution engine interchangeably)? Should it place “a” in a large (just in case) box? Should store hidden variables for each of the “a”s types? Anything the interpreter (or virtual machine) will do to support these “language quirks”, will slow down the execution of the code.

There is more coming…

What about the objects that can “change” their properties dynamically, on the fly? See below.

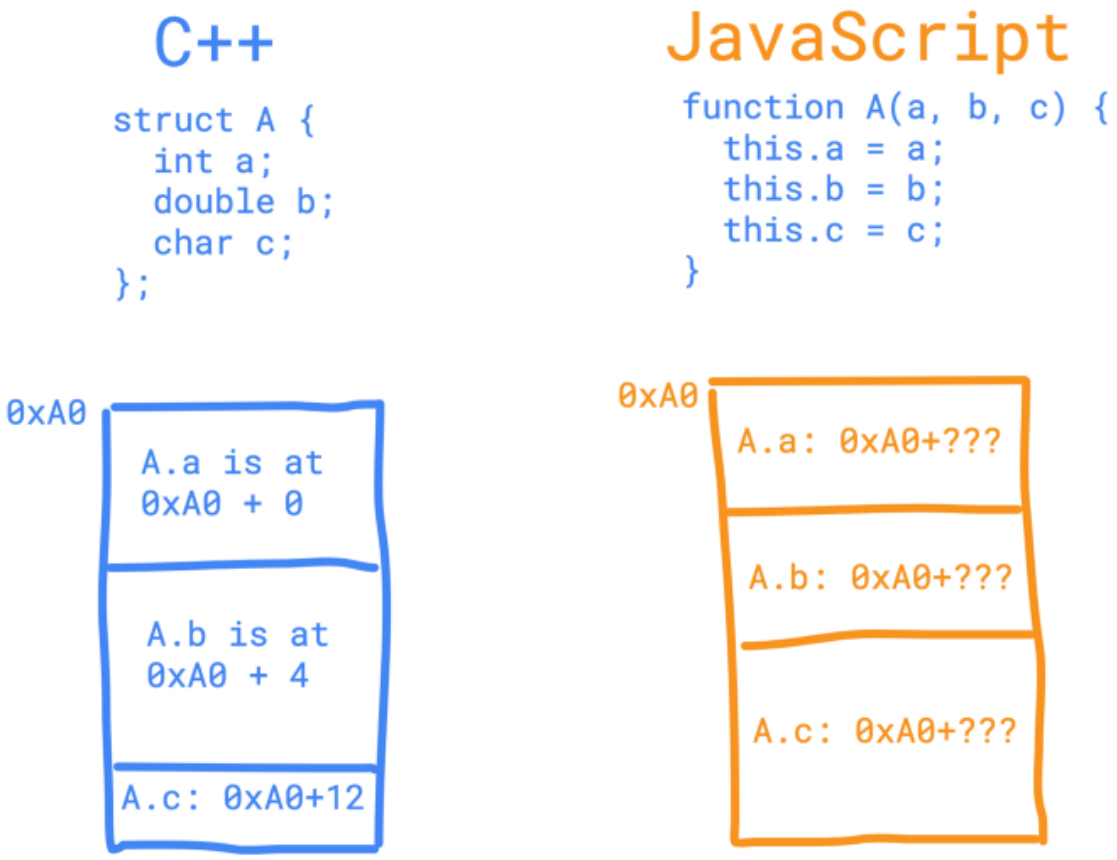
```
function A(a, b, c) {
```

```
    this.a = a;
    this.b = b;
    this.c = c;
}

let obj = new A();
let obj2 = new A(1, 2, 3);
let obj3 = new A("str", 3.14, 20);
obj.b = 45;// change on the fly
// add on the fly
obj3.new_prop = "some str";
delete obj2.b; // remove on the fly
```

Stress testing the virtual machine

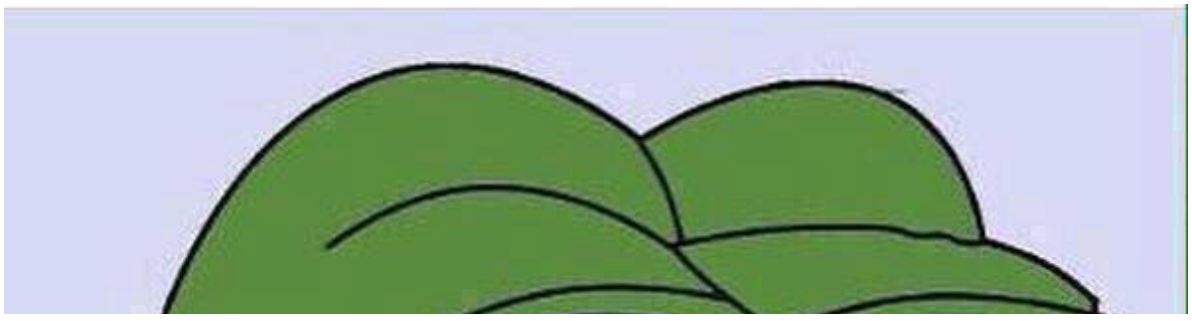
All we care about is the object layout. See, the C++ compiler generates objects smoothly laid out, while JavaScript's "execution engine" struggles on the thinking, how to "frakin" do that?



Object memory layout

I call these illustrations “pseudo-illustrations” as sometimes they contain some invalid but not significant details. In the illustration above we see that in C++, A’s instance is laid out “smoothly”, “a” property takes 4 bytes (just an assumption, like I said, pseudo-illustration) and starts from the address 0xA0. The object itself (A’s instance) starts at address 0xA0, so the first property, “a”, is placed at the address 0xA0 + the corresponding offset (0 in this case, as the “a” property, is the very first property of the instance). So as “a” takes 4 bytes, the next (2nd) property “b” starts at 0xA0 + 4 (the size taken by “a” property). “c” starts at 0xA0 + 12 (plus the size used by “a” and “b”). There is an aligning factor we omitted (not a major case in this context).

So now look at the right side of the same illustration. How should JS place an instance of A and its properties? How should it guess the type, what if they would change? What should it do? Where shall it go?



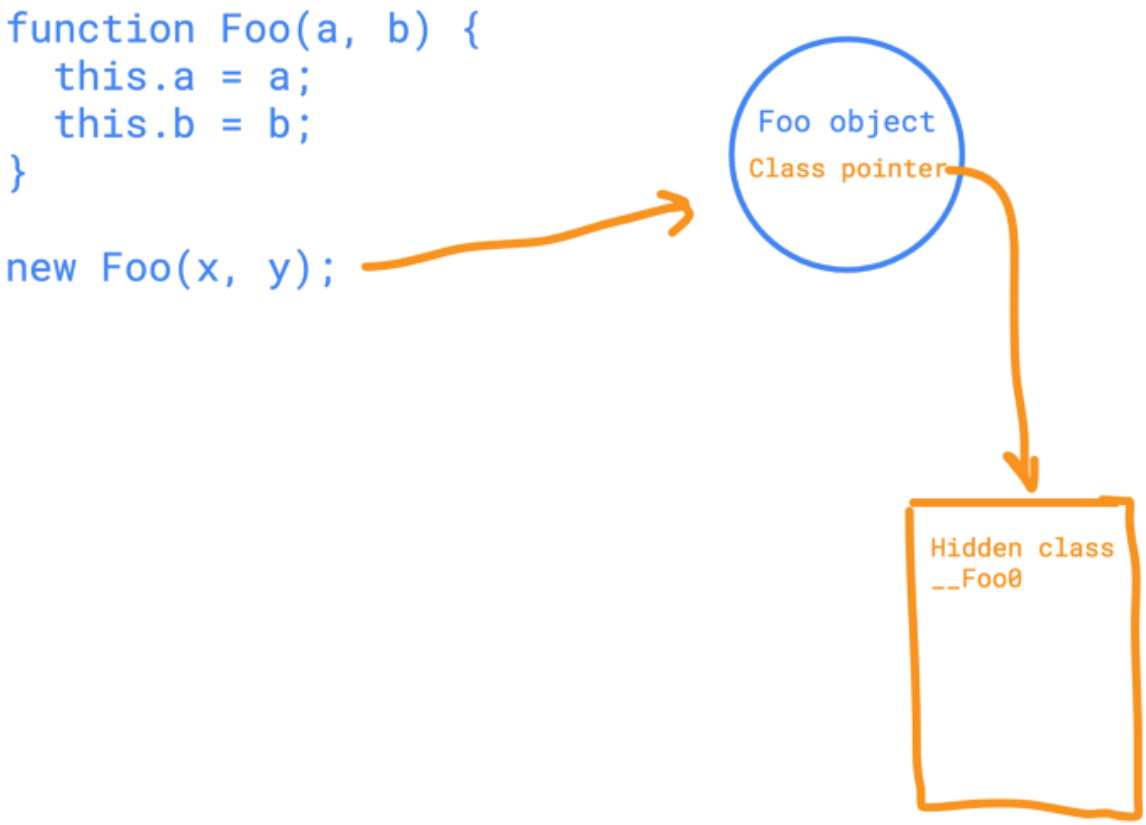
V8, trying to execute your code as fast as it can

Finally, we got to the point, Hidden Classes. V8 introduced the idea of the hidden classes just to make the code execution faster. Faster than properties' "dictionary lookup". It's done by generating hidden classes for each object instance you create plus some more. For instance, when we have code like this.

```
function Foo(a, b) {
  this.a = a;
  this.b = b;
}
```

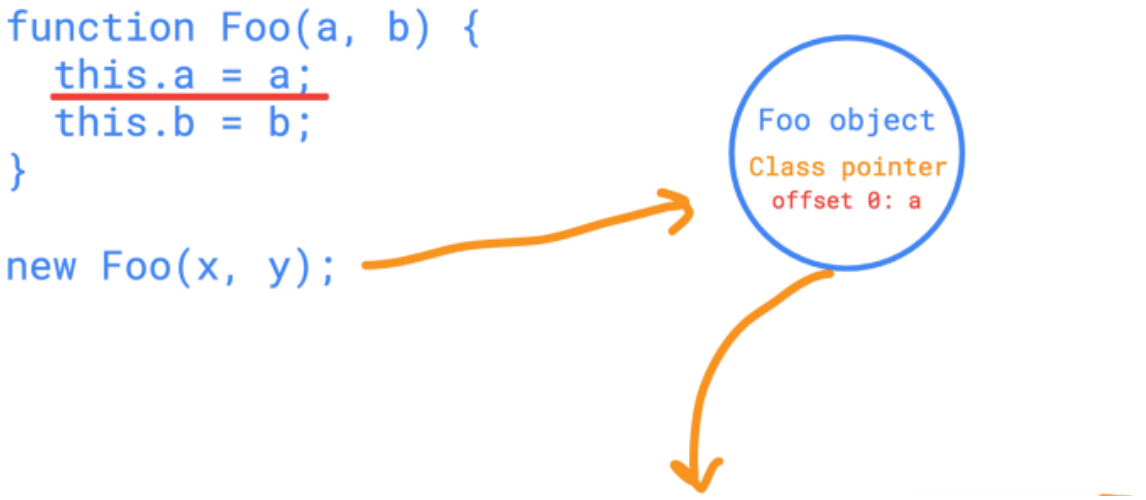
We see a "struct" with two properties, while V8 sees

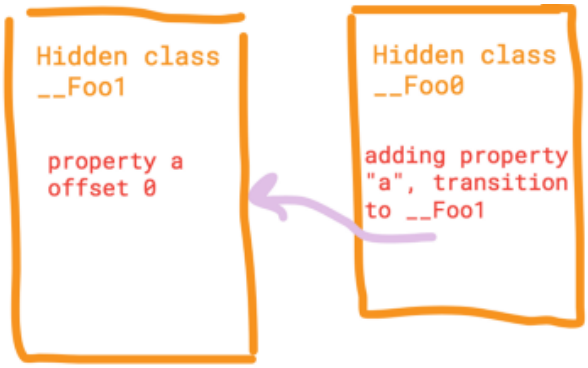
pain in the…class. That's because to execute `new Foo(x, y);` V8 has to create several hidden classes.



V8 creates an empty hidden class for the object

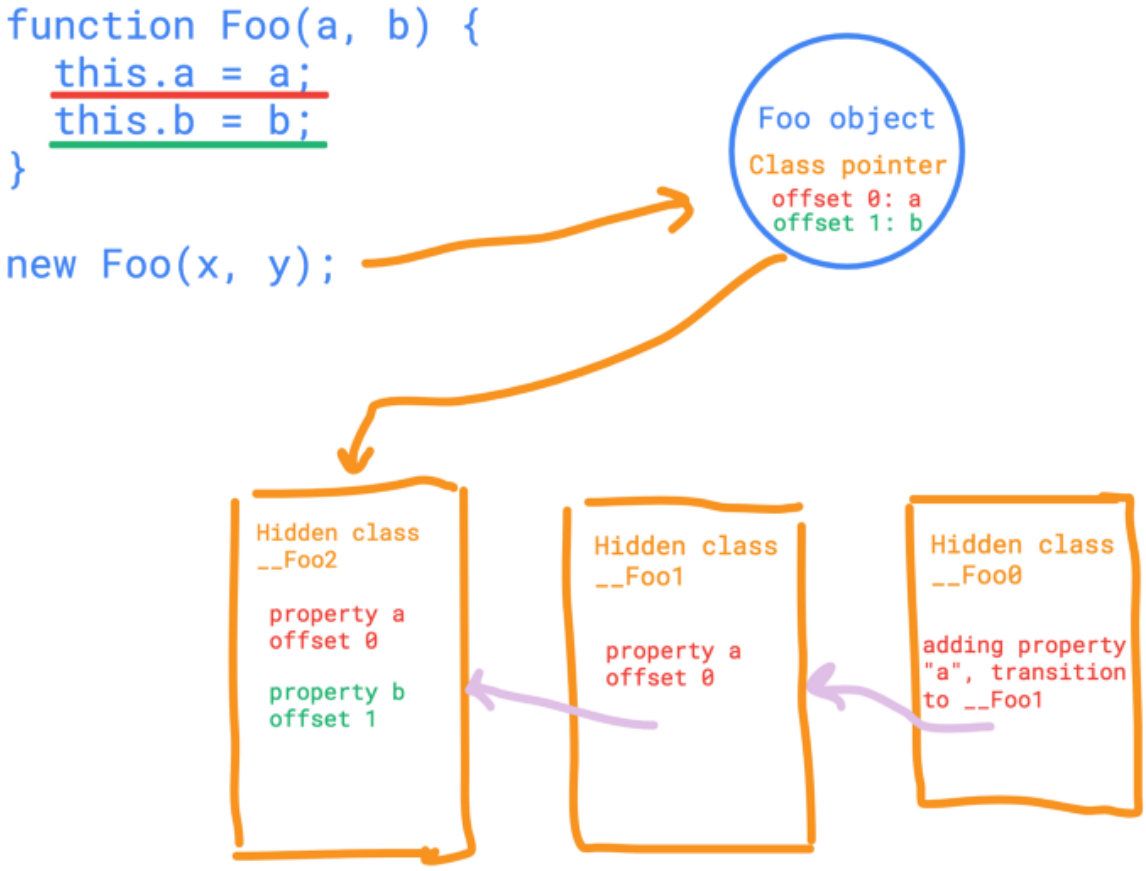
When executing the first line of the Foo, V8 makes transition from hidden class __Foo0 to hidden class __Foo1.





Execution of the first line adds the first property into the object, and makes a hidden class transition

The same logic applies to the execution of the second line (`this.b = b;`).



Adding the second property makes another transition, to __Foo2

The concept is okay. When V8 executes the `new`

`Foo(x, y)` it creates a hidden class. An empty hidden class. Something like this in C++.

```
struct __Foo0 {};
```

When V8 executes the first line, e.g. (creating) assigning the property “a”, it generates another hidden class and makes the so-called “transition” to a “newer version” of the __Foo0 hidden class.

```
struct __Foo1 {
  Type a; //
};
```

I intentionally left the Type of the “a” property unspecified (you might guess that based on the actual type another hidden class would be generated, tough times for V8, huh?).

Finally, the execution of the last line produces another, 3rd hidden class, something like this.

```
struct __Foo2 {
  Type a; // I have offset 0, I'm the first property, yay
  Type b; // I have offset 1, I'm the second and unamused
};
```

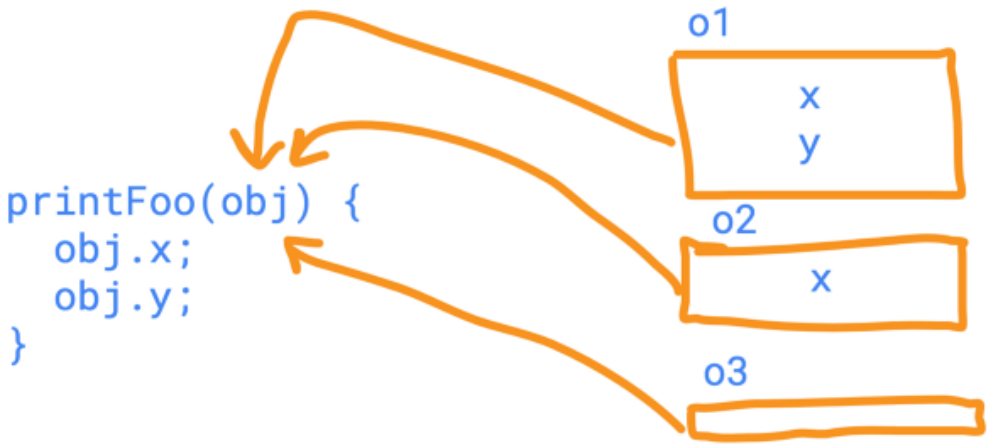
Why is this done so? Just because it’s faster than a simple dictionary lookup. V8 can treat our JS objects

like C++ objects, with proper smooth memory layouts, with corresponding offsets for each property. This might add some overhead on the code size but indeed makes the code run faster. At least faster than before.

Hidden classes are also helpful in Inline Caching, another “trick” implemented in V8 (invented somewhere else). I’m unable to provide more info on Inline Caches, a topic I’ve just started to dig into, but here’s a sample code, where the hot function with corresponding Inline Caching applied would make some sense.

```
let o1 = new Foo(x, y);
let o2 = new Foo(x);
let o3 = new Foo();

printFoo(o1);
printFoo(o2);
printFoo(o3);
// printFoo considered as a "hot" function
```

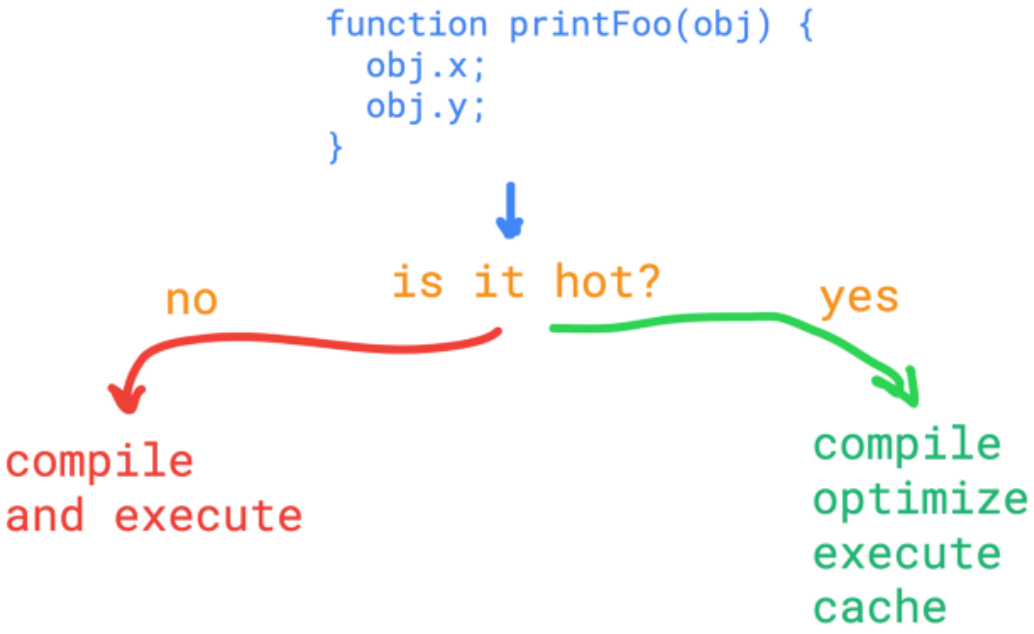


Hot function optimization, Inline Caching super-abstract intro

We have `printFoo` function, taking an object of type Foo discussed before. So this code above calls printFoo 3 times. With the same “declared” type of object, but as we already can see, with different hidden classes. The trick here’s that V8 optimizes function calls based on the input argument’s “kind”. When a function gets called several times (say, more than 2), then V8 starts to think about marking it as a “hot” function. “Hot” functions should be optimized as soon as it really possible, mostly because if the function is called many times, and if the function makes some load, then the load will be “applied” as many times as the function is called. Sounds simple. So by optimizing these “hot” functions V8 tries to reduce the code execution time. The actual optimization goes by profiling the function call and gathering required data on the “kind” of the passed argument. We pass `obj` , then we access both `x` and `y` properties of `obj` . So if the hidden classes of all passed objects are the same, V8 caches the call to the corresponding memory location of properties `x` and `y` (I might say something wrong here, be alert). In the code above we passed objects of different hidden classes. So if we would do something like this:

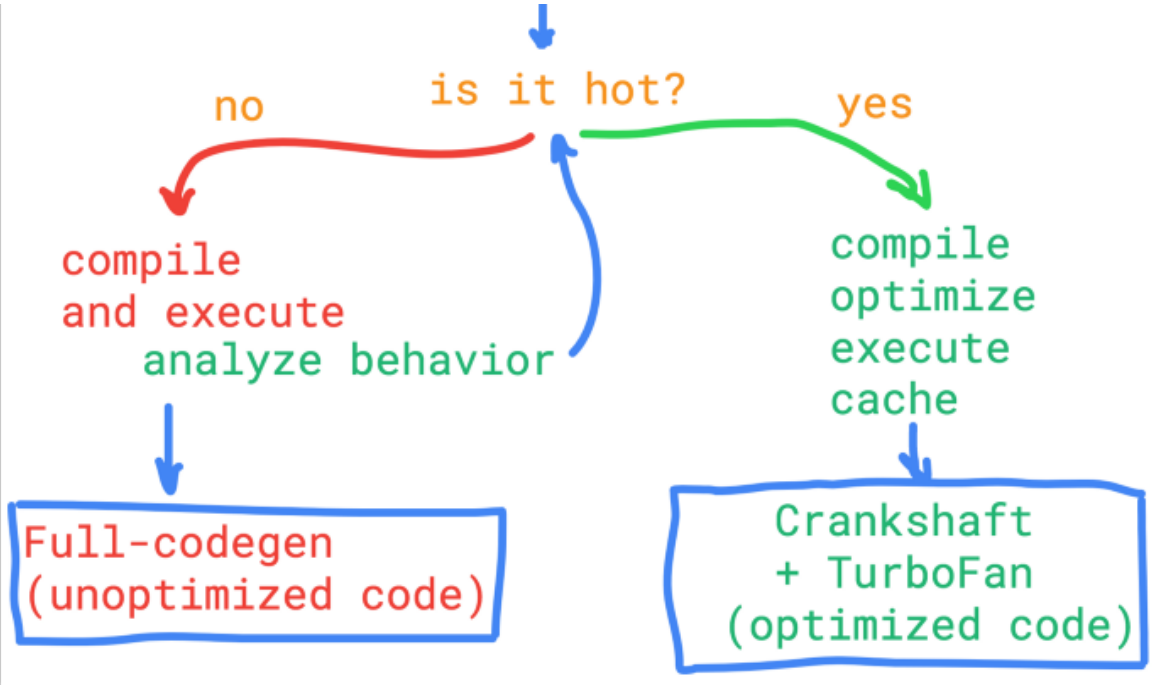
```
printFoo({a: 1, b: 2});
printFoo({a: 2, b: 3});
printFoo({a: 44, b: 55});
```

then these calls would be successfully optimized by V8. (this is some really artificial example, however it illuminates the idea).



The previous illustration shows the simple idea, but how does the “is it hot?” check work? Turns out V8 makes some behavior analysis when a function is being called.

```
function printFoo(obj) {
  obj.x;
  obj.y;
}
```



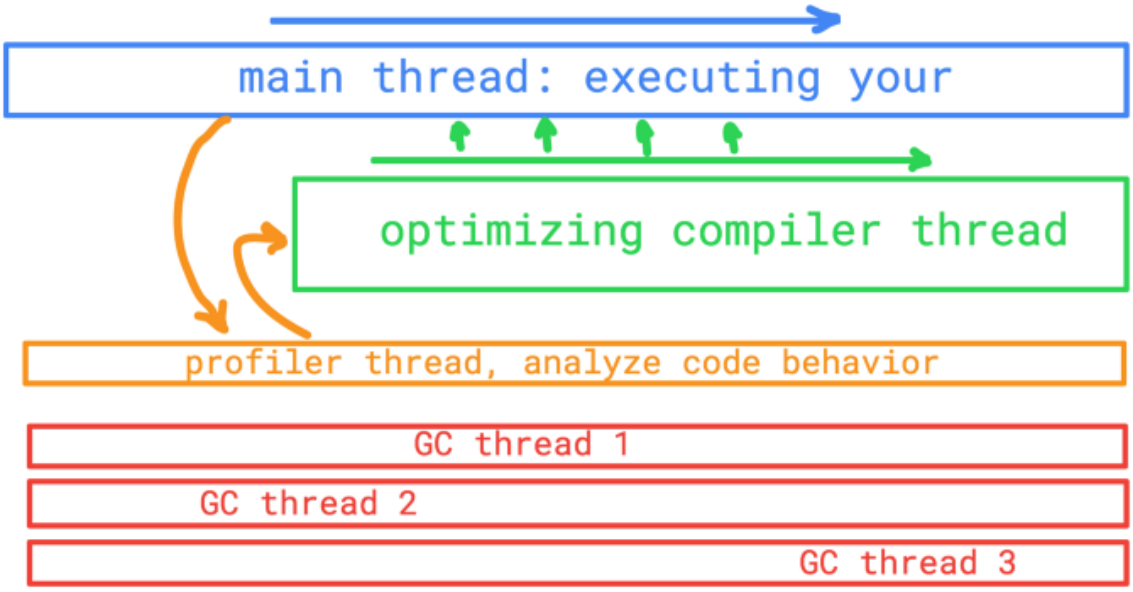
Optimized vs Unoptimized code execution

For each suspicious function call, some profiling data is being updated on that particular function. Before the next call “is it hot?” check is done. So if the function is hot, then the code execution switches to the optimization course. Optimization is done by another compiler, it generates an optimized machine code and updates the corresponding “execution context” with the optimized code.

We see in the illustration above that there are different names for two different compilers. Full-codegen for unoptimized machine code compiler and Crankshaft+TurboFan as optimized compiler. Actually, it was just Crankshaft, and as I got it from V8’s blog posts, Crankshaft wasn’t good enough for scaling. And

it is being replaced by TurboFan and already or sooner (not sure yet) Crankshaft would be completely removed from V8. And then comes Ignition. Ignition is the V8's interpreter, intended to replace the Full-codegen unoptimized compiler.

So far we got this double compiler idea. V8 generates some unoptimized machine code from our source JS and executes it. During this execution, V8 analyzes code behavior and collects some metadata. Then based on this metadata, V8 generates an optimized machine code making this optimized version as current running version. All of these mentioned couldn't be done on a single thread. V8 uses different threads to manage this huge work.

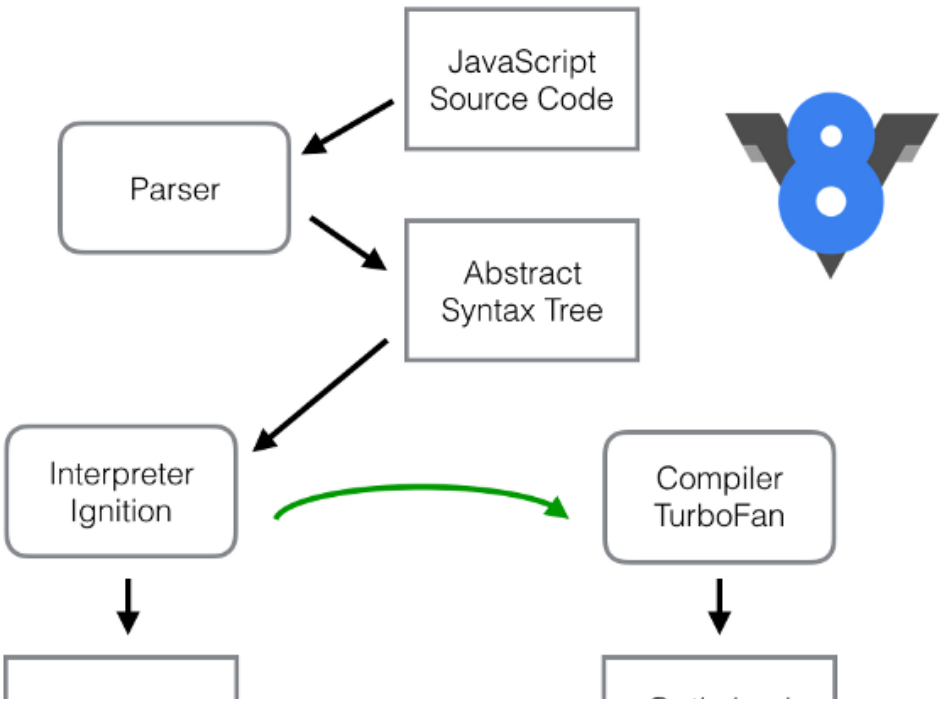


Different threads for code execution

There is the main thread, used to run our code as soon

as possible. And by “as soon as possible” I mean it takes time to parse the code, it takes time to generate corresponding machine code, and then the main thread starts to run it. Besides the main thread, there is another thread marked in the illustration as “optimizing compiler thread”. I put a “profiler thread” in the illustration just to make the picture complete. And there are some Garbage Collector threads just to keep all threads in a single illustration. So as the illustration shows, the main thread parses and runs our code. During this run, the optimizing compiler works to generate optimized machine code and replace “hot” parts of the code.

So, the full picture looks like this (I grabbed it from Franziska Hinkelmann’s blog post about V8 bytecode).



V8 compiler pipeline

As shown in the image above, the parser parses JS code into Abstract Syntax Tree, which is then being interpreted by Ignition (the interpreter). Ignition produces bytecode able to be executed instruction by instruction (compiling to unoptimized machine code). During the code, execution TurboFan works hard on the source code (working shoulder to shoulder with Ignition).