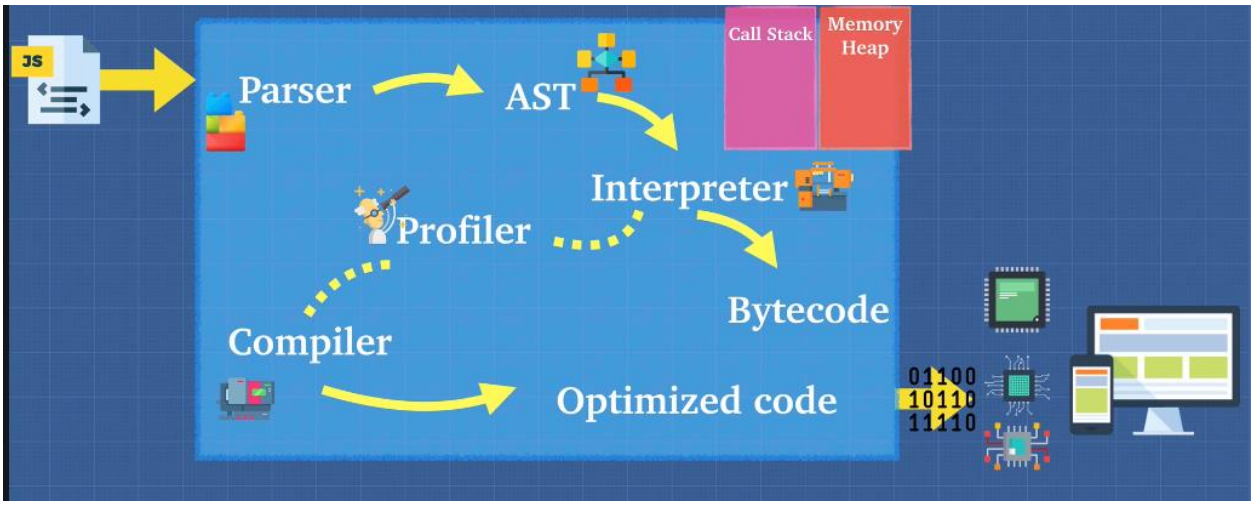
Advanced JavaScript Concepts:

6. Exercise: JavaScript Engine

- **Who do you think** created the very first JavaScript engine: first person to create the very first JavaScript engine was its creator Brandon Eich. while working at Netscape which became the first commercially available browser.
- **He created the early version** of what became to be known as Spider Monkey which is what Firefox still
- **Uses right now as their JavaScript engine** and Brendan Eich was the very first person who created the language JavaScript to implement this engine so that you're able to run JavaScript on a browser that

7. Inside the Engine:



- **we give it a JavaScript** file and first is does something called lexical analysis [with the help of Parser] which breaks the code into something called tokens to identify their meaning.
- **So that we know what** the code is trying to do, and these tokens are formed into an AST that is an abstract syntax tree. it gets formed into this tree like structure called abstract syntax tree.
- **Astexplorer.net** → this is a fun little tool online that you can use to demo
- In AST form it goes through something called an interpreter profiler compiler and we get some sort of code that is Machine Code.
- **Above one is an example of one of the JS**, we can create our own JS engine keeping ECMA scripts standards in mind.

8. Exercise: JS Engine for all

- **ECMA script tells people** hey here's the standard and how you should do things in JavaScript and how it should work.
- **ECMA script is the governing** body of JavaScript that essentially decides how the language should be standardized.
- **It tells engine creators** This is how JavaScript should work. But internally how you build the engine is up to you as long as it conforms to the standards.

9. Interpreters and Compilers:

10. Babel and TypeScript:

- **Babel** is a JavaScript compiler that takes your modern JS code and returns browser compatible JS (older JS code).
- **Typescript is a superset** of JavaScript that compiles down to JavaScript.
- **Both do exactly what compilers do**: Take one language and convert into a different one.

11. Inside the V8 Engine:

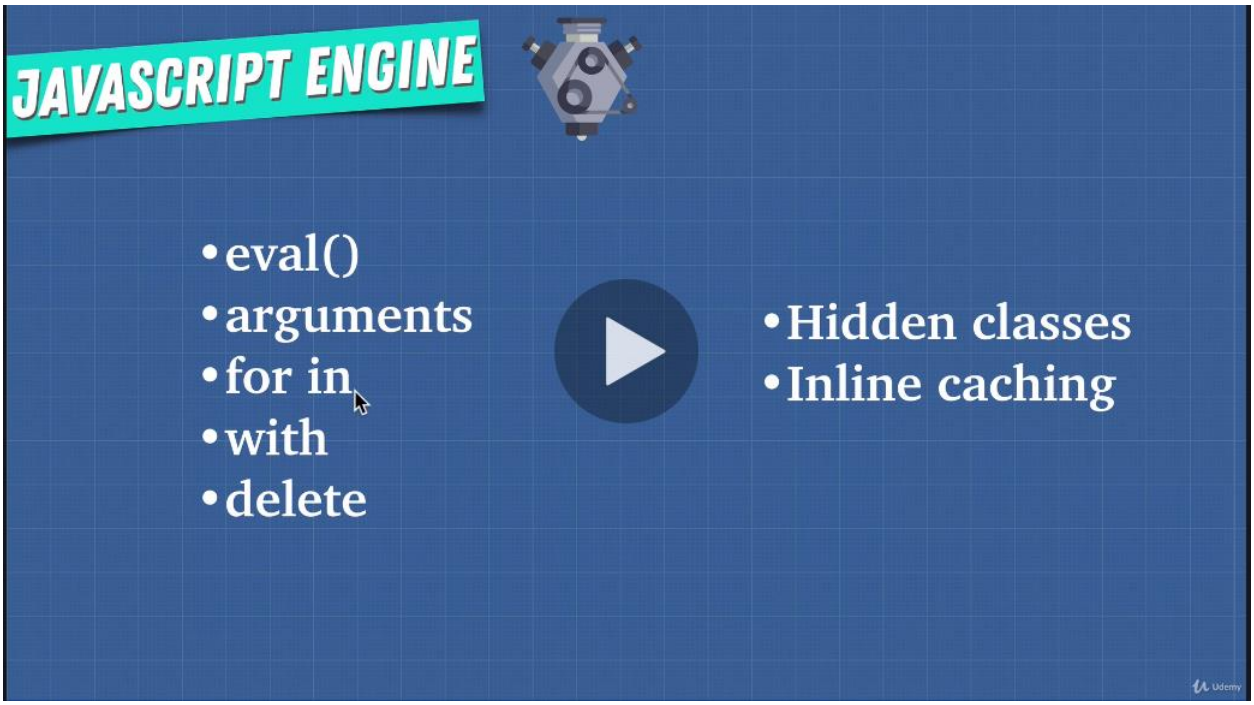
- **Compiler obviously takes a little bit longer to get up and running** but the code is going to eventually run faster.
- **Interpreter, that is fast to get up** and running but unfortunately doesn't do any optimizations.
- **Is there a way that we can** get the best of both worlds and this is what some engineers came up with in late 2000 and if we use Google as our example with the **V8 engine what if we combine the best of both worlds what if instead of using the compiler and interpreter we combine these two and create** something called a JIT compiler or just in time compiler.
- **This is exactly what browser started doing**, browsers started mixing compilers specifically these JIT compilers for just in time compilation to make the engines faster.
- **So, let's see how V8 engine does this**:
- Interpreter allows us to run the code right away. And the compiler and profiler allow us to optimize this code as we're running.
- **Pending some part……**

12. Comparing Other Language:

- **Initially when JavaScript first came** out you had JavaScript engine such as Spider Monkey created by Brendan that interpreted JavaScript to byte code and that engine was able to run inside of our browsers to tell our computers what to do.
- **But things have evolved now**. We don't have just interpreters. We also use compilers to optimize this code.
- **So, this is a common misconception**. When somebody says JavaScript is an interpreted language yes there is some truth to it.
- **But it depends on the implementation**. You can make an implementation of JavaScript of the JavaScript engine that perhaps only compiles and
- **So, the next time somebody tells** you hey JavaScript that's an interpreted language. You can sound super smart and nitpicky by saying well not technically. Technically it depends on the implementation.

13. Writing Optimized Code:

- In order to help the JavaScript engine, we should be careful while using below JS features:



- Functions that call literally the **eval ()** function which comes with JavaScript can be very problematic

- **aguments** Something that comes with every function. There are many ways that we can use this keyword by JavaScript but there are many ways that makes compilers more optimizable.
- **So, we must be careful with using arguments** and I'll show you later how we can use parameter De-structuring to avoid using arguments.
- **“for in” loop** for example when looping over objects can also be problematic sometimes. So, I like to use object keys and iterate over the object keys that way
- “**With**” statement which you don't see very often in the world anymore is another problematic one.
- And, the **delete** key word in JavaScript can be problematic.
- But there's two main things I want to talk about here that are the main reasons that these things in JavaScript can make our code less optimized.

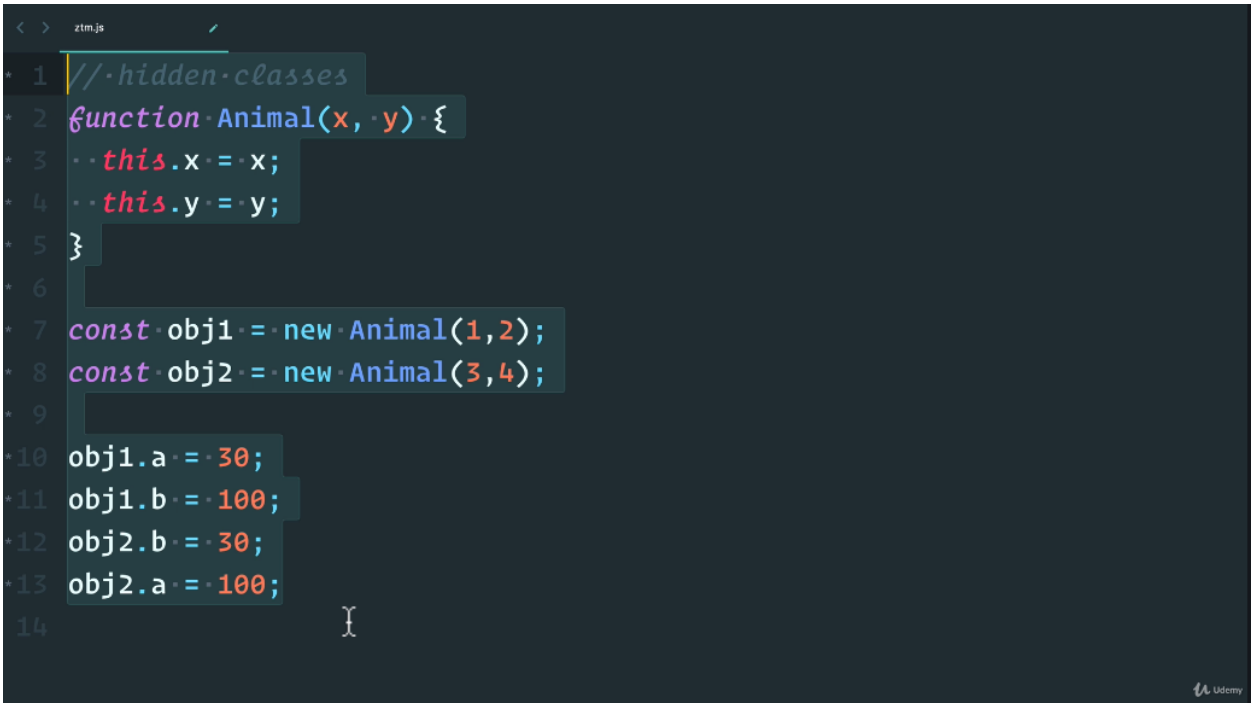
- **Inline Caching:**

- **Now all we do here is say** find user and give it the user data object now inline caching does something interesting due to inline caching done by the compiler for code that executes the same method repeatedly.
- **That is let's say the find user** method or function was being called multiple times. Well the compiler can optimize this so that whenever it's looking for the user data which has first name and last name it can use something called inline

caching where instead of looking up this object every time finding the key that is first name and last name and then the values it will cache or inline cache so that find user just becomes this piece of text.

```
'found Johnson Junior'
```

- So, if we call find user over and over and over, we will just replace that with found Johnson Junior because at the end of the day that's all that the function is doing.
- **Hidden Classes:**



- **That code is going to make the compiler run** slower or de optimize the code and in that is something called Hidden classes.
- **You want to try and instantiate your object** properties in the same order so that hidden classes which is what the compiler uses underneath the hood to say oh this animal class has objects obj1 an obj2 have the same hidden class that is they have the same properties
- **but as soon as you start introducing things** in different orders it's going to get confused and say they don't have a shared hidden class they're two separate things and internally that's going to slow things down.

- **so, one thing you want to do** in a situation like this is to assign all properties of an object in its constructor over here or add things in the same order
- **It isn't something that you need** to worry about too much. it's going to help us write fast efficient code take help of below resources to learn more.
  https://github.com/petkaantonov/bluebird/wiki/Optimization-killers#3-managing-arguments
  https://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html

14. Web Assembly:

- https://webassembly.org/
- **Why don't all programs just use machine code ahead of time. We compile our code let's say JavaScript and then we just give our machine code, so they don't have to worry about all this compiler interpretation and git compiling??**
- You see if JavaScript were compiling then either compilation would have to be super-fast because remember our JavaScript files get sent from the server to the browser.
- **So the compiling has to happen on the browser** or the competing browsers that is Firefox, Microsoft Edge and chrome browsers would have to agree on some binary executable format or standard that can understand this machine code because at the end of the day these browsers are the ones that are executing the code.
- **So, compiling the code ahead of time or** even just compiling the code on the browser was not feasible at all because back in the day that was slow. But also having all the browsers agree on an executable format to run JavaScript
- **Web Assembly**: Even now browsers have different ways of doing things and there's just no real standard, but luckily enough things might change in the future.
- **We now have something called Web assembly** and something that you should really keep an eye on because this could be a game changer in the future.
- **We have the standard binary executable format** called Web assembly and this is what we didn't have in 1995, we didn't have the competing browsers agreeing on this format where we can compile code all the way down to web assembly. This executable format so that it runs really fast on the browser instead of having to go through that entire JavaScript engine process.

- **If you're curious what it is that's the big advantage of it**, → We might not have to do all these interpreters, compilers and all the steps that JavaScript requires to run the code on the browser.

15. Call Stack and Memory Heap:

- **JavaScript engine** does a lot of work for us. But the biggest thing is reading our code and executing it.
- **Two most important things in this step are**: One we need a place to store and write information that is to store our variables our objects our data of our apps, another one is a place to actually run and keep track of what's happening line by line on our code. While we use call stack and memory heap for that.
- We need the memory heap as a place to store and write information because at the end of the day all programs are just read and write operations. That way we have a place to allocate memory use memory and release memory.
- **And second with the call stack**: We need a place to keep track of where we are in the code So with that, we can run the code in order
- The memory heap is where the memory allocation happens, and the call sack is where the engine keeps track of where your code is and its execution.
- **Memory heap** is simply a free store is a large region in memory that the JavaScript engine provides for us which can be used to store any type of arbitrary data in an unordered fashion there's no order to this memory it just allows us to use variables to point to different storage areas.
- Simple variables can usually be stored on the stack and complex data structures like objects, arrays and functions are stored in memory heaps.

16. Stack Overflow:

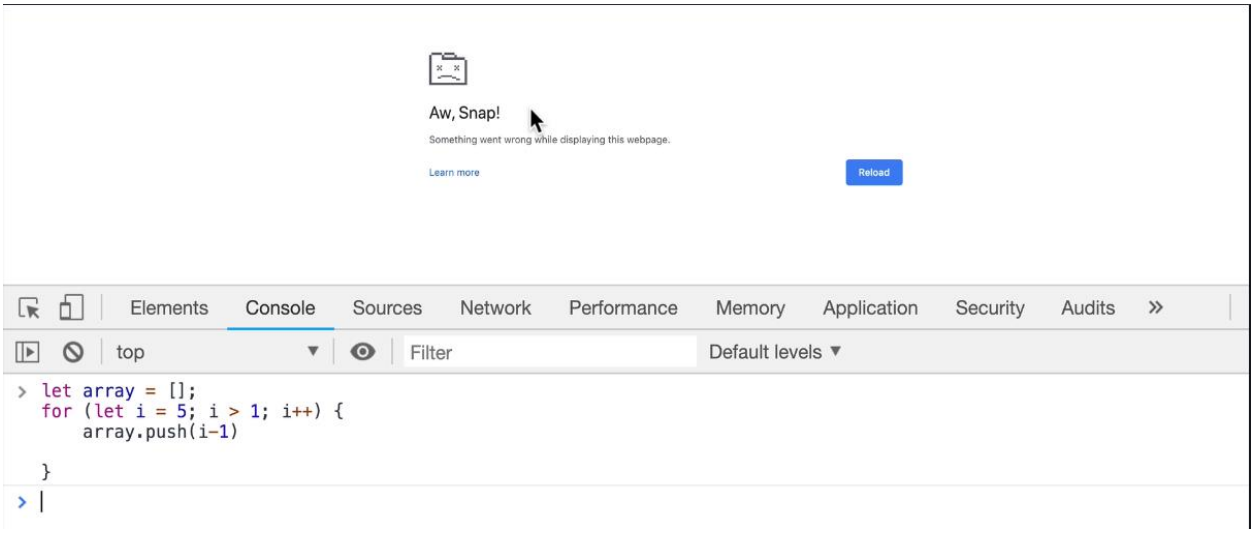
17. Garbage Collection:

- **JavaScript is a garbage collected language** that means when JavaScript allocates memory let's say within a function, we create an object and that object gets stored somewhere in our memory heap automatically. With JavaScript when we finish calling the function and let's say we don't need that object anymore. It's going to clean it up for us.
- **So, JavaScript automatically frees up this memory** that we no longer use and will collect our garbage.

- **In garbage collected languages like JavaScript** the garbage collector free memory on the heap and prevents what we call memory leaks.

18. Memory Leaks:

- https://developers.soundcloud.com/blog/garbage-collection-in-redux-applications
- https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setInterval



- Above code is creating memory leak resulting in crash.

- One reasons that we don't want to have too many global variables such as **var a =1**
- Well I've just added variable a as a variable on our global scope But if hypothetically I just keep adding variables to my environment while we're adding more and more pieces of memory and if these are objects and deeply nested objects you can see the memory being used will be more and more and more.
- **Another one is event listeners**. so, if I have an event listener that's going to listen on the click event and it's going to do some function that just call it onclick.
- **Now this is one of the most common** ways to leak memory and that is you add these event listeners and you never remove them when you don't need them so that you keep adding keep adding keep adding event listeners. And because they're just there in the background you forget about them.
- **And next thing you know you create a memory leak** this happens a lot especially if you go back and forth between single page applications where you're not removing the event listeners of the page and as a user goes back and forth back and forth, the memory keeps increasing more and more as more event listeners are added.
- **Another common one** is simply using something like **setInterval** where you have the **setInterval function** that does something and inside of this you start referencing objects and these objects will never be collected by the garbage collector because of this **setInterval** unless we clear it and stop it is going to keep running and running and running.
- **So, memory leaks are something that we must be careful of**, so one example is SoundCloud.

19. Single Threaded:

- **JavaScript is a single threaded programming language**. What does that mean.
- Being a single threaded means that only one set of instructions is executed at a time. It's not doing multiple things.
- And the biggest way to check that a language is single threaded is well it has only **one call stack**. This one call stack allows us to run code one at a time.
- We're never running functions in parallel as you saw the stack keeps growing as we push new functions on the stack and then we pop them one at a time.
- A good way to think about it is like this **single threaded means I'm the JavaScript engine and I'm eating with one hand putting food in my mouth** that is the

function and then use that same hand to grab the next food when I'm done chewing my food.

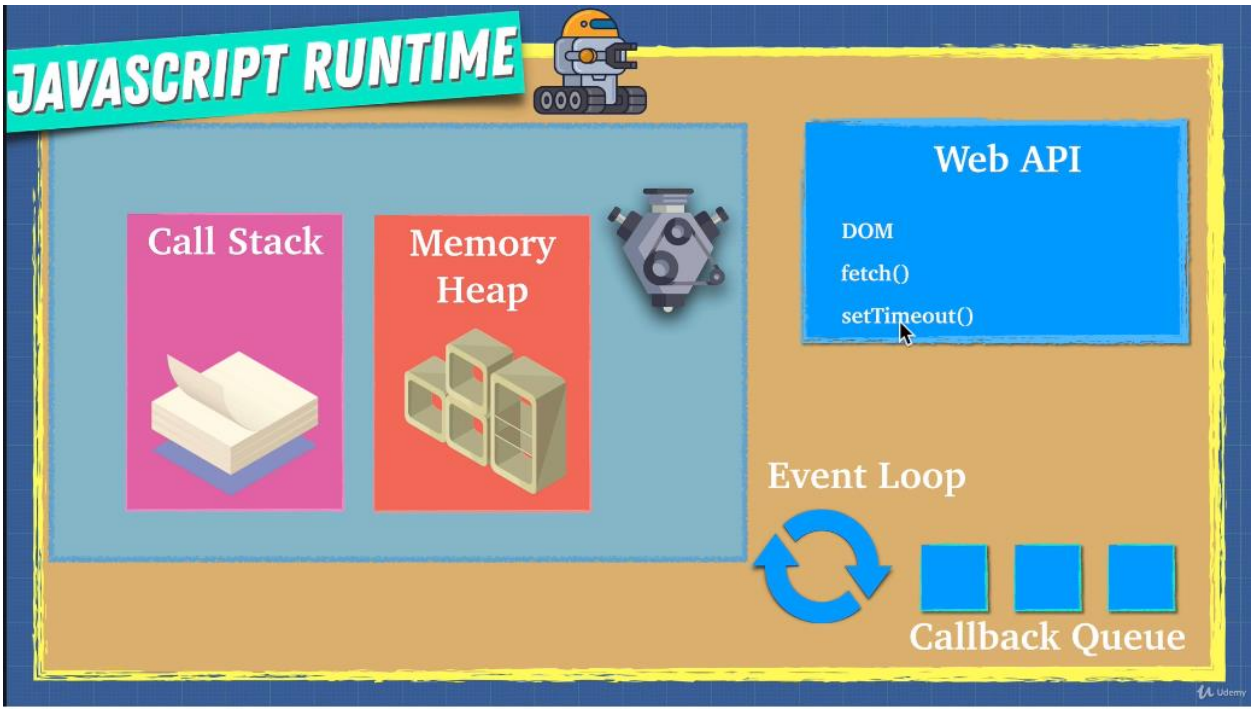
- **I can't put any more things** in my mouth until I'm done chewing and I also can't grab anymore food until I finish chewing. It's single threaded. One thing at a time. And because of this JavaScript is synchronous.

20. Exercise: Issue with Single Thread:

- **What problems do you see with synchronous code?**
- Well with single threaded synchronous code that JavaScript engine runs it's going to make it difficult. If we have long running tasks what does that mean.
- **Imagine we are running Twitter and Twitter is an application** that we can tweet we can check other people's tweets We can look at followers our tweets our notifications our messages.
- **Imagine if they had a function that let's say a big loop that takes five seconds** to complete. When that happens if we just use the JavaScript engine Well, I wouldn't be able to click. I wouldn't be able to scroll. I wouldn't be able to do anything.
- **A classic example of this is the alert function** if I run alert function my JavaScript is running right now, and I can’t scroll. I can't click anywhere.
- The Web site is frozen in time until I take some sort of action and this mimics a long running JavaScript and alert function mimics a long running JavaScript.
- On the call stack right now, we have something like a function that's running and until that call stack is empty, I can't really do anything. That's terrible.
- **Why would anybody use JavaScript? I mean it's just going to slow our pages down.**
- Well here's the thing I haven't been completely honest with you. That's because when we talk about JavaScript most of the time, you're never just using the JavaScript engine which is synchronous.
- We need to introduce I had the idea of asynchronous code and in the next couple of videos we're going to talk about what's happening behind the scenes where it's not just the JavaScript engine that's running our code.
- **We have something called the JavaScript runtime** and as we'll see no JS will also implement something similar because for us to write code that we can use in this modern day we need something beyond just the JavaScript engine.

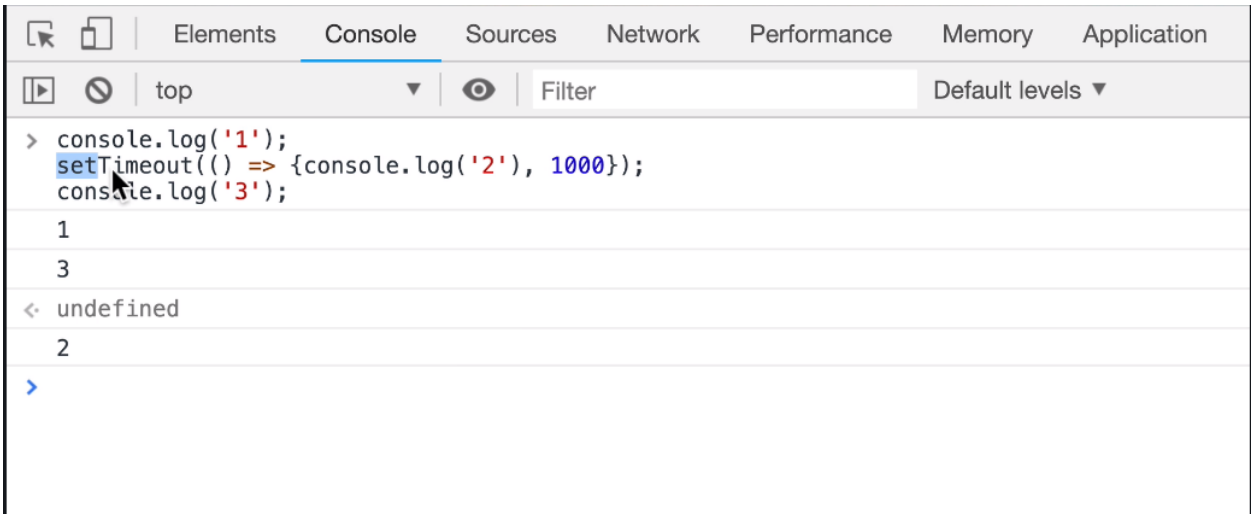
21. JavaScript Runtime:

- **In case of JS Runtime the web browser is working in the background** while the synchronous JavaScript code is running and it's using something called the web API or the web browser API to communicate and let the J.S. engine know Hey I'm back with some data some work that you told me to do in the background.
- Web API comes with the browser Chrome, Microsoft, Edge, safari, Firefox.
- **All of them have their JavaScript engine implementation** and all of them have a JavaScript runtime that provide a web API these web APIs are applications which can do a variety of things like send HTTP requests, listen to Dom events maybe click events on a Dom, delay execution using something like setTimeout, we can use setInterval, can even be used for caching or database storage on the browser.



- If we go to the console and I look at the window object Well this is what the browser provides. This is the web API that we can use.
- **If I scroll down here, I can see different** things that the browser provides for us that our JavaScript engine can use. For example, if I scroll down to fetch, fetch that is the function to make HTTP calls.
- **I keep scrolling down and let's** look at something fun like IndexedDB. That's a little database that we can use on our browser. If I go to application here, I see that I have some storage and indexedDB that the browser provides for us.

- If I keep scrolling down, I also see something familiar to us such as setTimeout. SetTimeout and setInterval all of them are provided by the browser. They're not native to JavaScript.
- **So, remember that the browsers are helping us create rich Web applications** so that users aren't just sitting around waiting for our JavaScript to execute anything that can be offloaded. They'll take care of that for us in the background because you could imagine if the browser had to use the same JavaScript thread for execution of these features it's going to take a long time.
- So, browsers underneath the hood use low level programming languages like C++ to perform these operations in the background. And these API's are called Web API because these API's are provided by a web browser.
- **These web API's are what we call asynchronous that means you can instruct these API to do something in the background and return data once it's done.** Meanwhile we can just continue working on our JavaScript call stack and execute functions.
- **Working of JS Runtime:**
- We have items on the call stack and as soon as something comes up like setTimeout that's not part of JavaScript. It is part of the web API.
- The call stack is going to say oh I have something that's not for me. Here it's for the web browser for the web API. So, it's going to say hey web API I don't know what to do with this. You take care of it.
- **The web API is going** to say Oh I know what setTimeout is. Let me take care of that and do that in the background.
- **Now with set time out you usually** want something to happen afterwards. So once the Web API is done working on it maybe it's fetching some data from a server it's going to say all right I'm done with this work here is the data and here is perhaps a callback or what I want you to do with that data.
- And then that event loop is going to say as soon as the call stack is free it is going to say hey, I have something for you here. Would you like to be added onto the call stack? And if the stack is empty it's going to push this callback onto the stack.
- **Code Example:**



- The web API is going to take the setTimeout it's going to start a timer that is going to run for one second and once that one second is over, it’s going to push the callback.
- Once it's done running for one second and the callback in this case is console log 2. It's like it's calling us back saying hey we're ready for you. One second has passed. Can you console log 2.
- The console log 2 goes to the callback queue and says I'm the first person that's done can you console log.
- **Event loop is a loop that's constantly running that saying hey is the call stack empty is the call stack.** The event loop only runs once the call stack is empty and the entire JavaScript file has been read in our case console log 3.
- No matter how fast the web API response adds the console log 2 to the callback queue. Then event loop start putting anything back into the call stack until the call stack is empty and we've run through the entire JS file at least once.
- So now in case after console log 3 gets printed and popped off the call stack the event loop is going to tick and say all right can you console log 2 and that's why we get this weird pattern now.
- But in the meantime, our JavaScript engine doesn't care what's happening in Web API it's just going to keep working on the call stack.

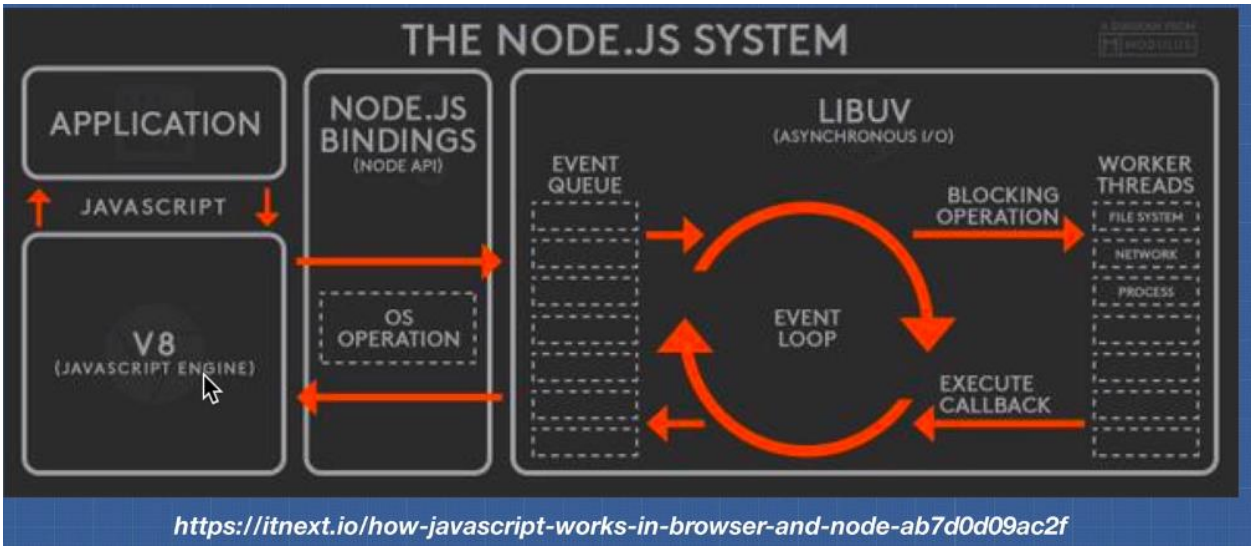
- **Interview Question:**

```
> console.log('1');
  setTimeout(() => {console.log('2'), 0});
  console.log('3');
  1
  3
<· undefined
  2
> |
```

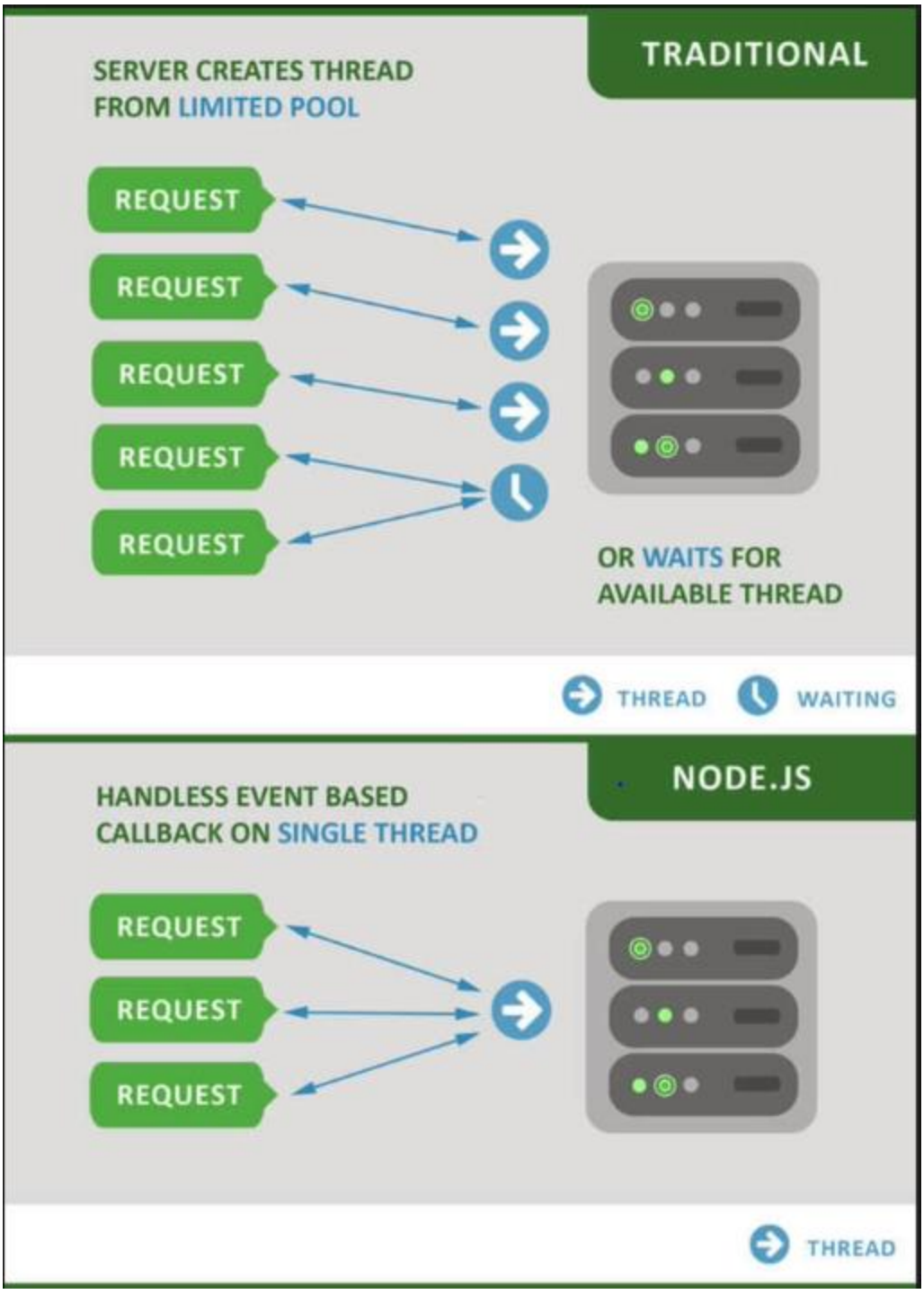
- The reason that happens is that no matter how fast this setTimeout timer happens it still gets sent to the web API still get sent to the callback queue and the event loop still needs to check those two checks Hey is the stack empty and has the entire file been run.
- In our case, no we are still waiting on console log 3 run. And only then will push our callback function our console log 2 to our call stack.
- How cool is that and using this method we're able to have this power of asynchronous code that is instead of being limited to one call stack and one memory heap whenever we get tasks that can be asynchronous that take a long time possibly like modifying Dom or making HTTPP requests. Well in that case we can just send that off to the browser the browser can just run that in the background and whenever we're ready it can just use its callback to an event loop to notify us.
- **JS Runtime Playground:**
- http://latentflip.com/loupe/?code=ZnVuY3Rpb24gcHJpbnRIZWxsbygpIHsNCiAgICBjb25zb2xlLmxvZygnSGVsbG8gZnJvbSBiYXonKTsNCn0NCg0KZnVuY3Rpb24gYmF6KCkgew0KICAgIHNldFRpbWVvdXQocHJpbnRIZWxsbywgMzAwMCk7DQp9DQoNCmZ1bmN0aW9uIGJhcigpIHsNCiAgICBiYXooKTsNCn0NCg0KZnVuY3Rpb24gZm9vKCkgew0KICAgIGJhcigpOw0KfQ0KDQpmb28oKTs%3D!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D

22. Node.js:

- **It's a JavaScript runtime built on Chrome's V8 JavaScript engine.**
- It is a C++ program you can think of it as node.exe. it's an executable, a C++ program that provides this runtime for us and if we look at the node.js runtime you see that it looks quite similar with our browser-based runtime.



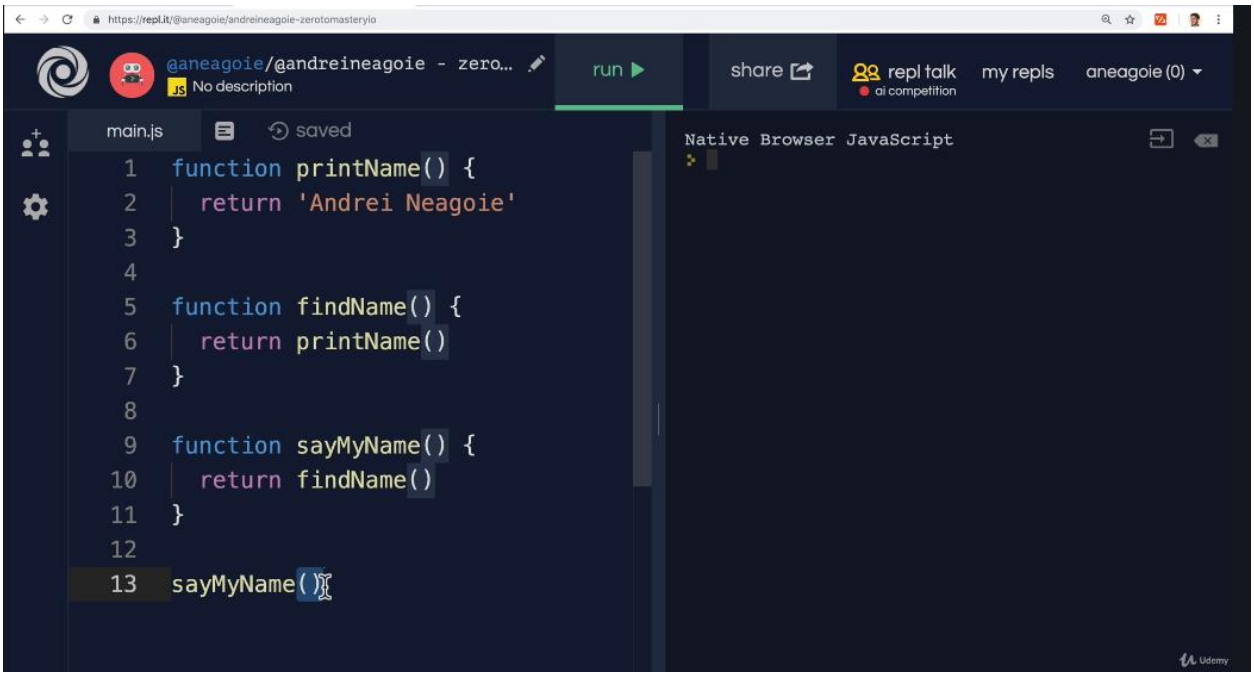
- When it comes to node.js **There are a few differences**. It does a little bit more than our web browser runtime.
- In the case of a browser we're limited to what we can do in the background right. The browser isn't going to allow us to do much on the person's computer. You can't really access for example the file system of the user. I mean that would be a huge security flaw. It just runs on the browser or on the tab that we're currently on in what we call a sandbox environment.
- **But in node we can pretty much do most of the things in the background. We can access file systems. We can do all sorts of things.**
- You see no Node.js uses the Google V8 engine to understand the JavaScript but it uses this **LIBUV library which works along to create this event loop to extend** what we can do in the background.
- Instead of window node has something called global and that's its global API. Which also happens to have setTimeout, setInterval.
- **That is why node is set to be a server-side platform based on asynchronous IO that is non-blocking**. It means that it uses JavaScript but outside of the browser, but it creates this entire runtime environment.
- **This runtime that runs outside of the browser and it allows us to have the same model of a single threaded model, but any asynchronous tasks can be non-blocking that is they can be passed on to what we call worker threads in the background to do the work for us and then get sent back through the callback queue and event loop onto the stack.**



- Traditionally we would create a thread for each request something like in PHP would work like above and the thread pool that is however much the server can work is maxed out.

- Well the requests would just have to wait for the other ones to be available again. **Node.js changed this mentality and said** hey we only have this one single thread but as soon as you give me a thread, I'm just going to pass it off to my asynchronous runtime to take care of it. And this simplified things a lot.
- Now there's pros and cons to each method but I hope now you see the power of the runtime.

27. Execution Context:

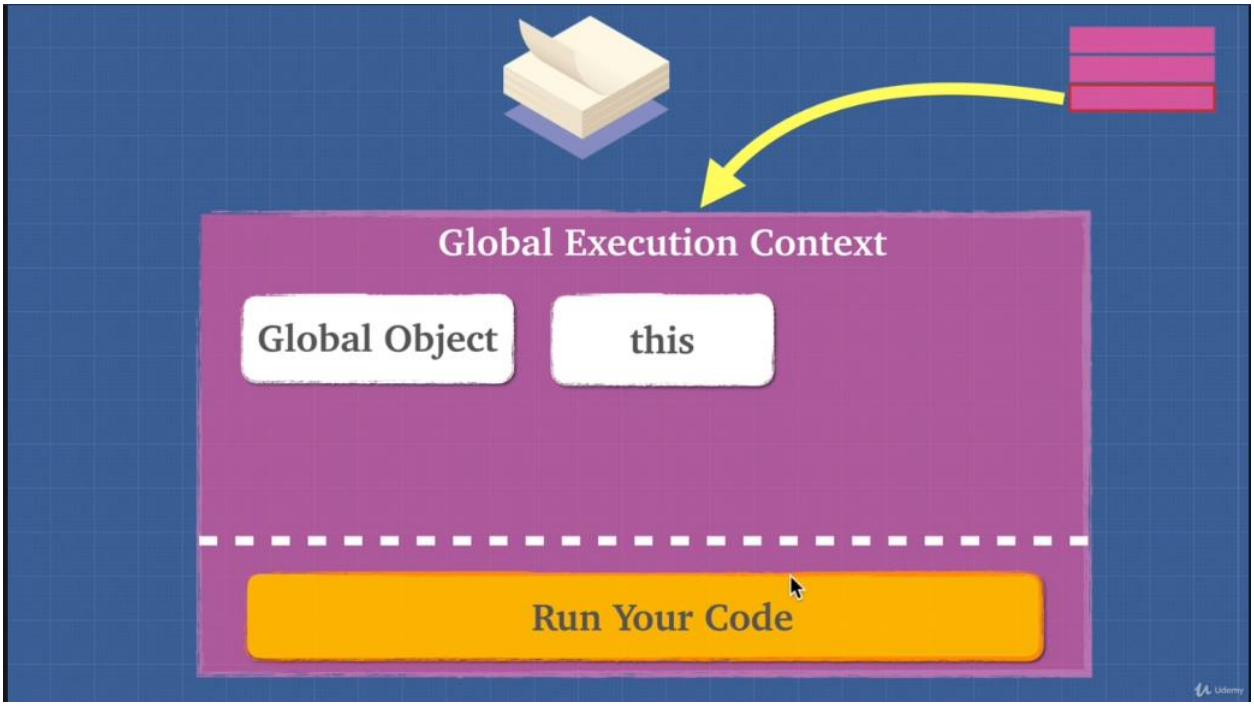


- **Well when the JavaScript engine sees a function call it run the function and create an execution context**. So, the first thing it does is create an execution context sayMyName and add this execution context onto the stack.
- **And then it tries to run this function** and sees that this function is calling another function. So, I'm going to create a new execution context because I'm going to see another function call. That's called findMyName. Another item that gets pushed onto the stack and then that function. Once again, the JavaScript engine is going to see oh here's another function call printName. Let's run this function and then this function after it's created its own execution context is going to return the string.

- **As below call stack diagram**:

- Diagram that we just saw above isn't 100 percent accurate because the base execution context that runs is called the Global execution context.
- **Initially our JavaScript engine** is going to create a global execution context. It's underneath the hood but it's saying hey here's the JavaScript file for you. Just start reading it for me. And on top of that that's when we start adding sayMyName, Then findMyName then printName.
- **And then eventually** as these execution contexts get popped off the last thing that remains is the global execution context. And when the final line of our code runs, and we're done with the JavaScript engine this is going to get popped off the stack.

- Anytime we run code in JavaScript it's always going to **be part of an execution context**. That is, it's part of global or inside of some function that we call.
- But if we look at the global execution context remember that's the very first item on the stack. The first thing the JavaScript engine does is to create this global execution context and it gives you two things.
- **First thing is a global object**. And the other thing is that this keyword in JavaScript everybody's favorite topic right and that this keyword in JavaScript.
- These two things we get when the JavaScript engine starts up.



- When code is run on the JavaScript engine a global execution context is created.
- And when you run a function a new execution context is added, a function execution context and we start running our code until everything gets popped off the stack and all our code is run.

28. Lexical Environment:

- **Lexical environment is simply where you write something**. If compiler is doing lexical analysis all its saying is it's checking to see where the words were written and their location.
- And again, lexical means at compile time.
- In JavaScript every time we have a function it creates a new world for us inside of that function. We are shot up into that planet every time we add it onto the call

stack and inside of that planet, we can do different things have different information inside of them and they can communicate with each other in different ways.

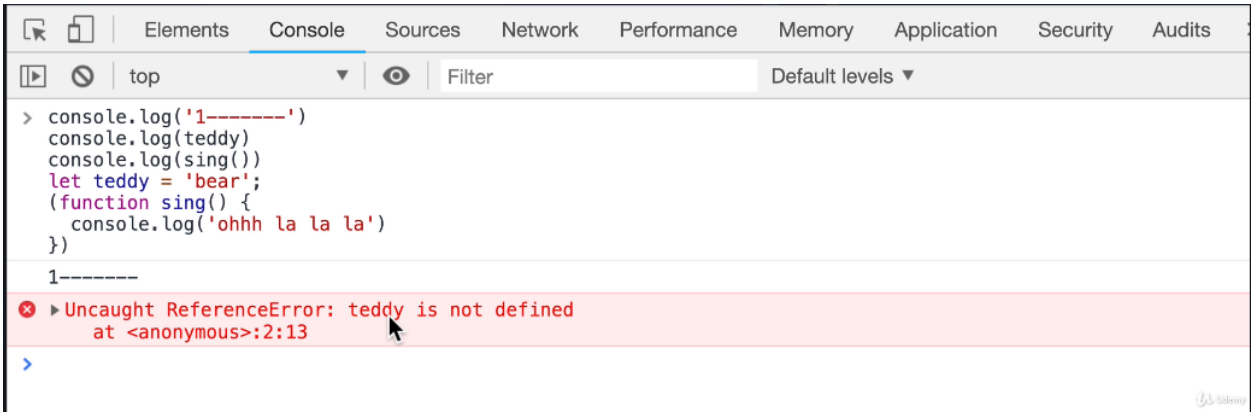
- **So, if I told you execution context tells** you which lexical environment is currently running. This execution context is going to tell me which lexical environment is currently running and that's how things work in JavaScript.
- **In JavaScript our lexical scope (available data + variable where the function was defined) determines our available variables, not where the function is called (dynamic scope).**
- **The very first lexical environment is the global lexical environment** where we write our code.
- **Lexical Scope: Arrow function usi ka scope lega jis scope me ye likha hua h. Isilye bolte hai ki arrow function ka lexical scope hota h**.

29. Hoisting:

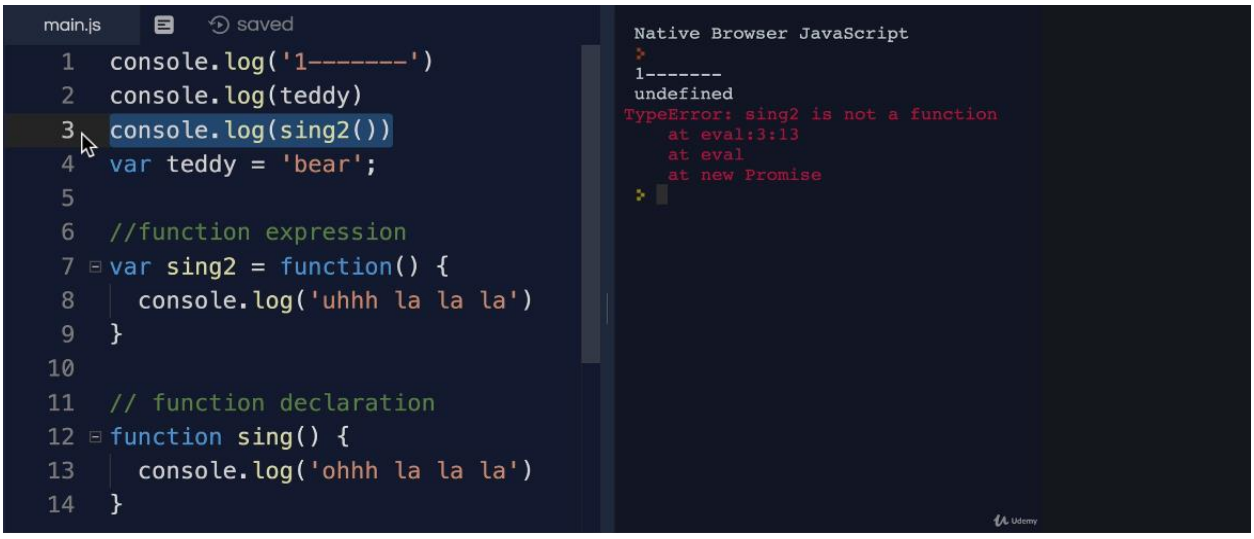
- **Hoisting is the behavior of moving the variables or function declarations to the top of their respective environments during compilation phase.**
- So, what happens underneath the hood is that during the creation phase the JavaScript engine is going to look through the code and as soon as it sees two things either the VAR keyword or the function keyword. It's going to say oh let me allocates memory because we are gonna use it.

- We get a reference error. Sing is not defined because it doesn't hoist because the first thing it sees wasn't a function keyword. This was not hoisted.

- **Remember this rule is only for var and function. So as soon as it sees let or const it will error out.**
- Compiler is not physically moving variable Teddy or function all the way up. It looks like it may but what it's doing is simply having one **pass through the code and saying reserve memory or assigned memory into this space.**



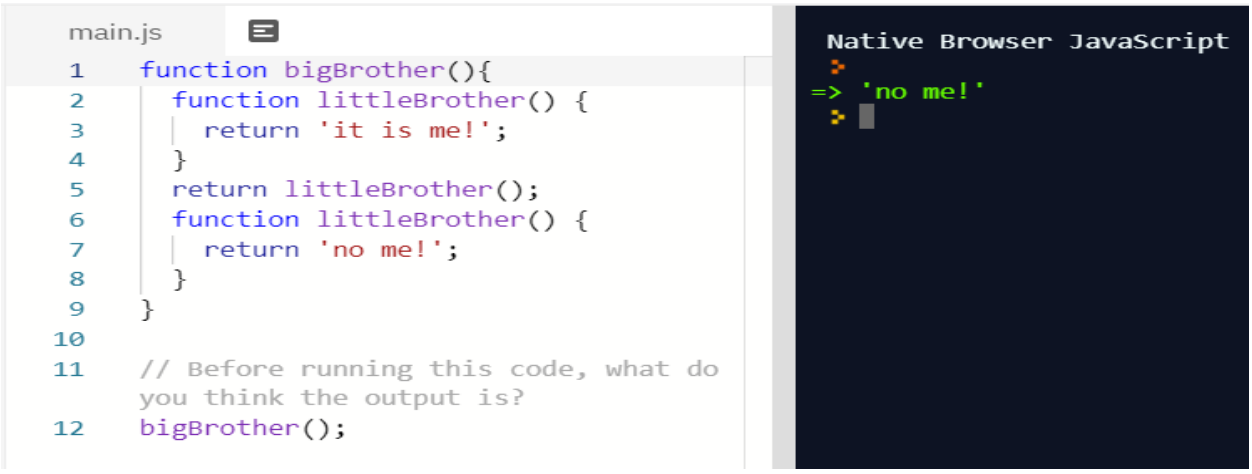
- During our global execution phase, we're simply saying Hey It looks like we're gonna have a few functions and a few variables. Can you allocate some memory for us here?
- Just get us ready for the program about to run. JavaScript engine during its execution phase is going to say can you grab this variable from memory
- **Example**: For Function Expression:



30. Hoisting Exercise: Function 'a' is also a variable like var variables.

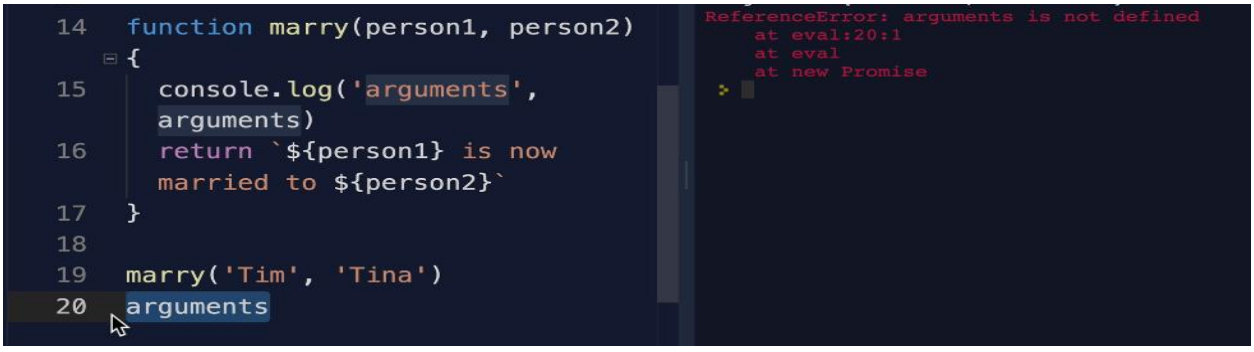
31. Hoisting Exercise: 2

32. Hoisting Exercise: 3

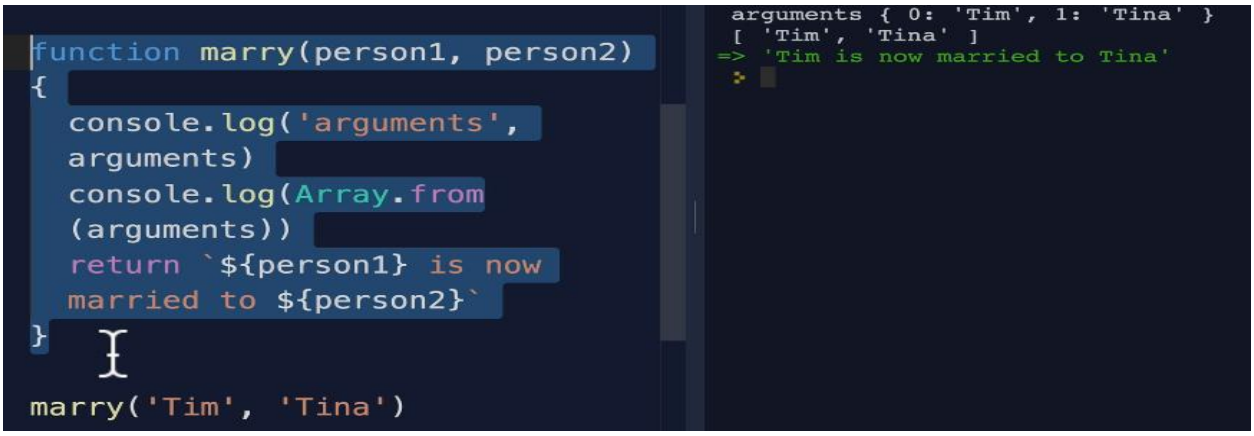


33. Function Invocation:

- 'arguments' is only available to us when we create a new execution context with function. It is not available for global scope.



34. arguments Keyword: We should try to avoid arguments using ES6.



```
function marry(person1, person2)
{
  console.log('arguments',
  arguments)
  console.log(Array.from
  (arguments))
  return `${person1} is now
  married to ${person2}`
}

marry('Tim', 'Tina')
```

```
arguments { 0: 'Tim', 1: 'Tina' }
[ 'Tim', 'Tina' ]
=> 'Tim is now married to Tina'
```

```
function marry2(...args) {
  console.log('arguments', args)
  console.log(Array.from
  (arguments))
  return `${args[0]} is now
  married to ${args[1]}`
}

marry2('Tim', 'Tina')
```

```
arguments [ 'Tim', 'Tina' ]
[ 'Tim', 'Tina' ]
=> 'Tim is now married to Tina'
```

35. Variable Environment:

36. Scope Chain: Closures

main.js — saved

```
1  function sayMyName() {
2    var a = 'a';
3    return function findName() {
4      var b = 'b';
5      console.log(c)
6      return function printName() {
7        var c = 'c';
8        return 'Andrei Neagoie'
9      }
10   }
11 }
12
13 sayMyName()
```

```
Native Browser JavaScript
=> [Function: findName]
```

main.js saved

```
1  function sayMyName() {
2    var a = 'a';
3    return function findName() {
4      var b = 'b';
5      return function printName() {
6        var c = 'c';
7        return 'Andrei Neagoie'
8      }
9    }
10 }
11
12 sayMyName()()
```

Native Browser JavaScript

```
=> [Function: printName]
```

main.js saved

```
1  function sayMyName() {
2    var a = 'a';
3    return function findName() {
4      var b = 'b';
5      return function printName() {
6        var c = 'c';
7        return 'Andrei Neagoie'
8      }
9    }
10 }
11
12 sayMyName()()()
```

Native Browser JavaScript

```
=> 'Andrei Neagoie'
```

main.js saved

```
1  function sayMyName() {
2    var a = 'a';
3    return function findName() {
4      var b = 'b';
5      return function printName() {
6        var c = 'c';
7        console.log(a)
8        return 'Andrei Neagoie'
9      }
10   }
11 }
12
13 sayMyName()()()
```

Native Browser JavaScript

```
a
=> 'Andrei Neagoie'
```

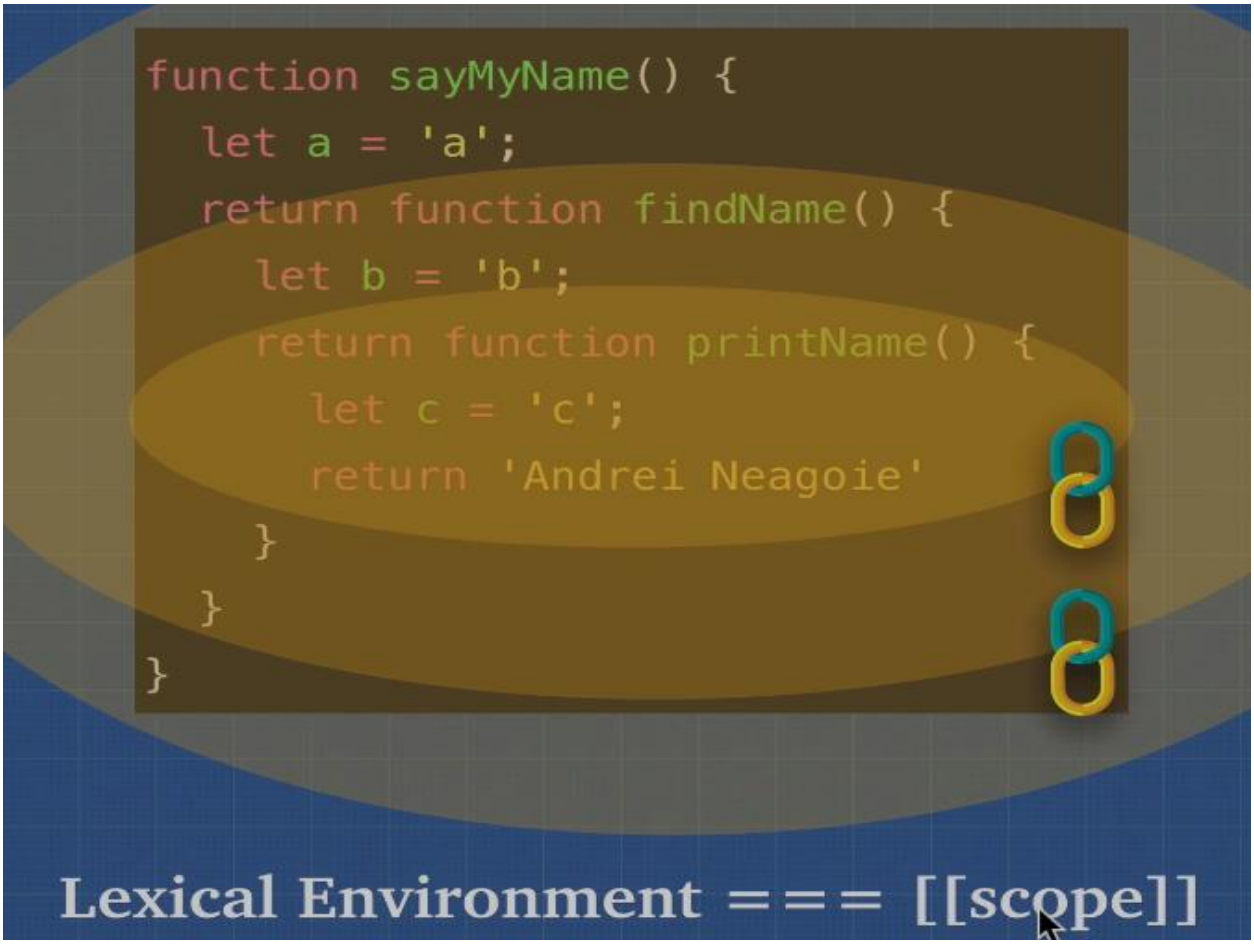
main.js saved

```
1  function sayMyName() {
2    var a = 'a';
3    return function findName() {
4      var b = 'b';
5      console.log(c)
6      return function printName() {
7        var c = 'c';
8        return 'Andrei Neagoie'
9      }
10   }
11 }
12
13 sayMyName()()()
```

Native Browser JavaScript

```
ReferenceError: c is not defined
    at findName:5:17
    at eval:13:12
    at eval
    at new Promise
```

37. Scope:

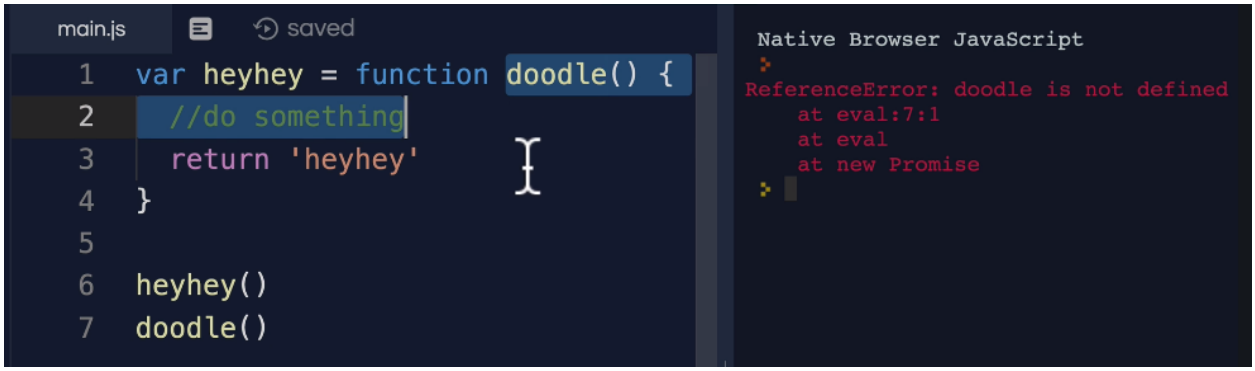


- If I create a function here let's just call it an if I go into the window object now and look for ‘a’ it is here Well because the scope of function is global.
- And as soon as we can't find what we're looking for we're going to look inside of **scopes([[Scopes]])** to our outer environment which in this case is the global type which is the window object.

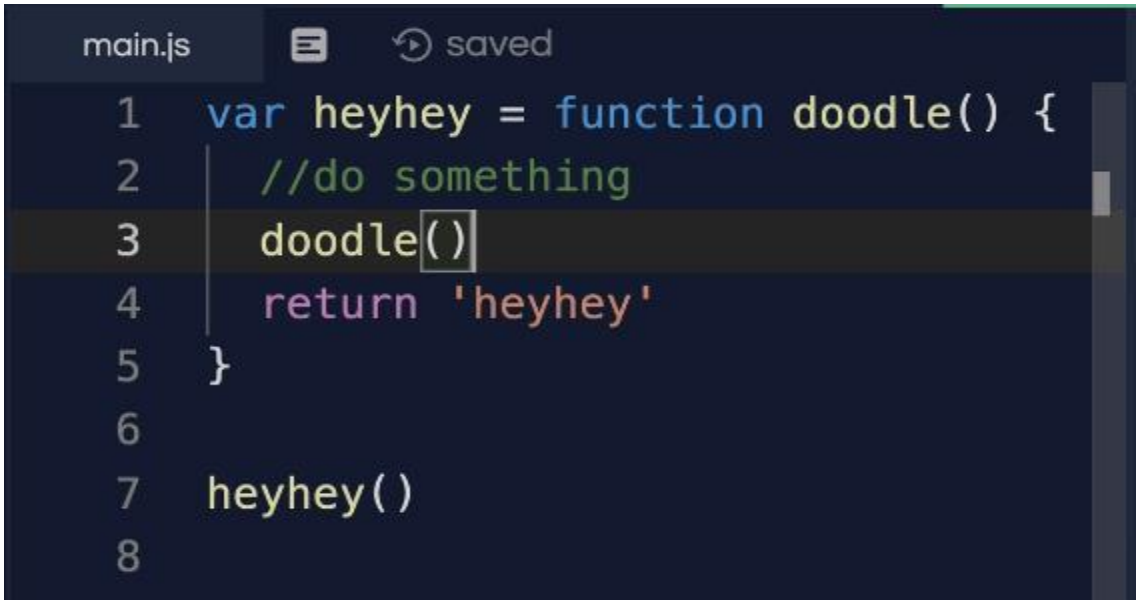
38. Exercise: JS is weird:

- **In above example** height will be declared in global scope.

- **In above example** ‘use strict’ avoids this forgiving behavior of JS.



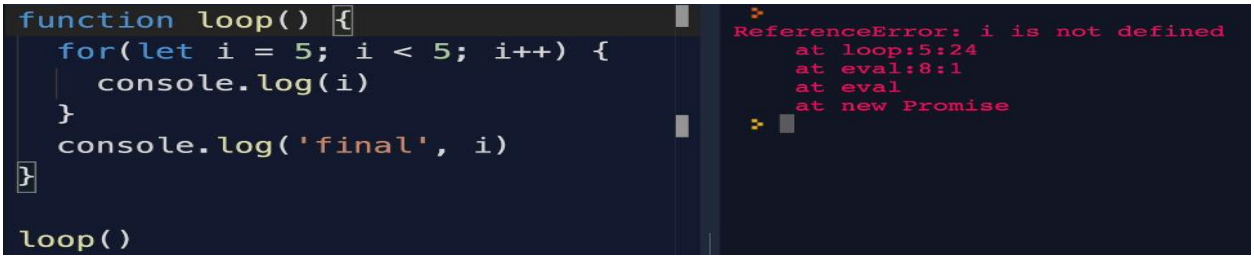
- This is because the doodle function is enclosed in its own scope doodle gets added to its own execution context variable environment very weird.
- But that's a little gotcha over here, we can't access it on the global scope. We can only access it as below:



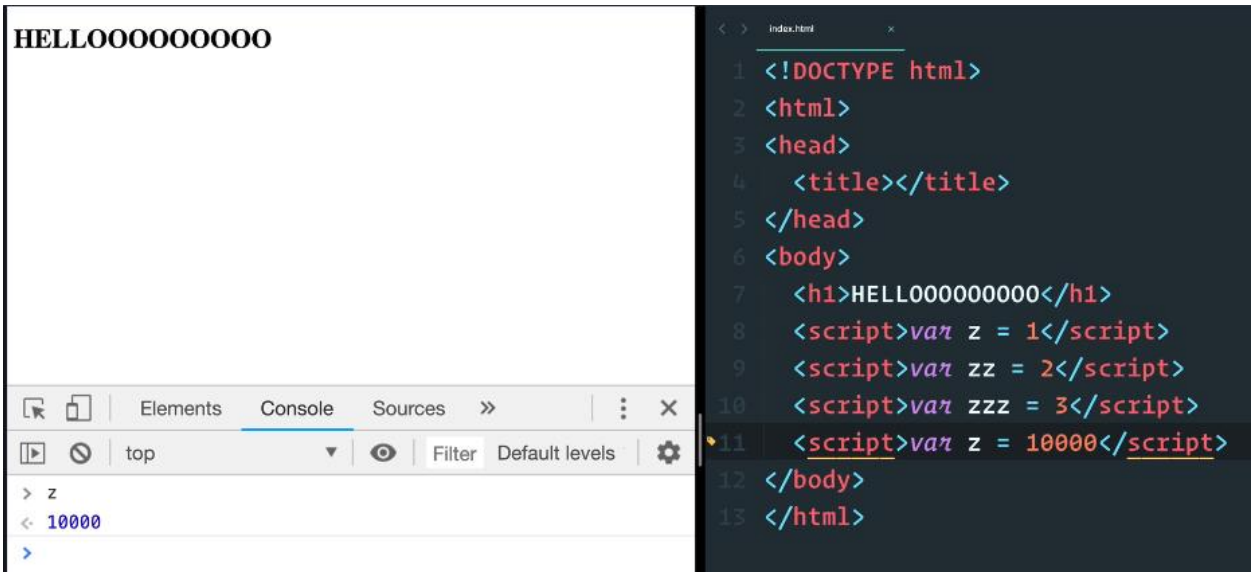
39. Function Scope vs Block Scope:

- Var is function scope and let, const are block scope.

40. Exercise Block Scope:



41. Global Variables:

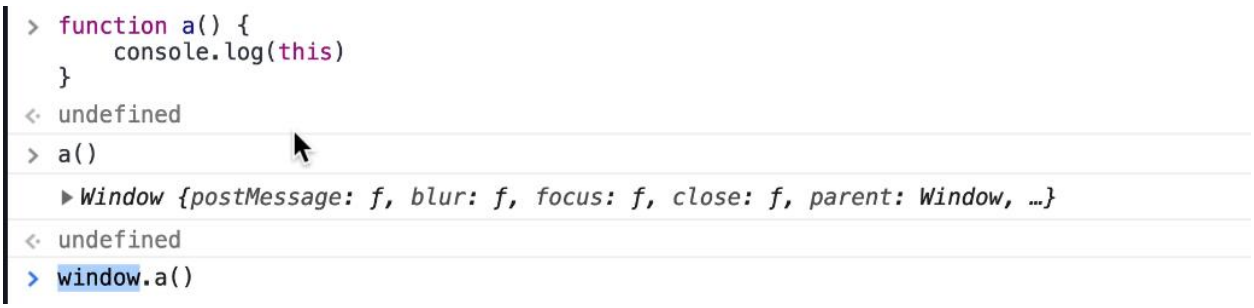


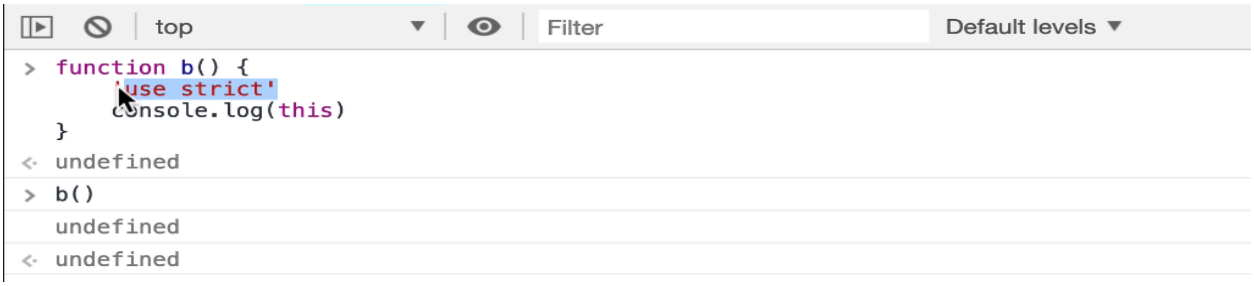
- We had a collision and the way this works is that in the HTML file all these script tags get combined essentially into one execution context.
- Everything is on the global execution context and they overwrite each other. If there's any duplicates so this creates a lot of possible bugs where we write different variables and maybe, we think we're safe here but somebody on another file overwrites one of our important variables.

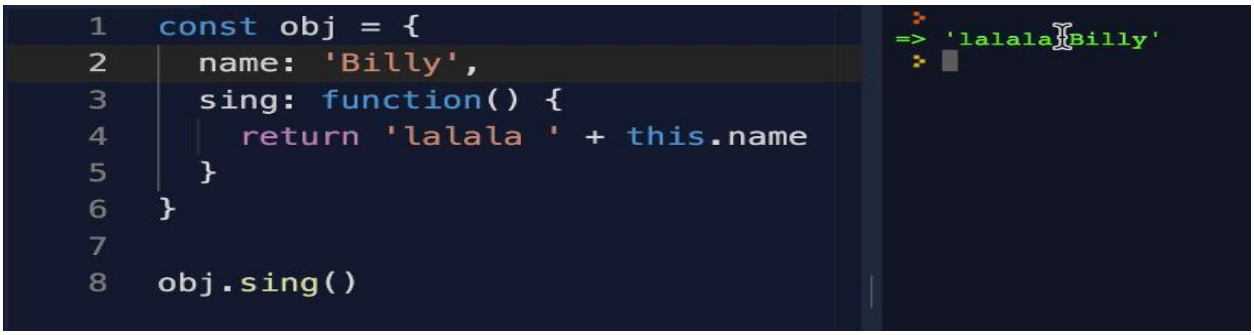
42. IIFE: To overcome problems of global variables.

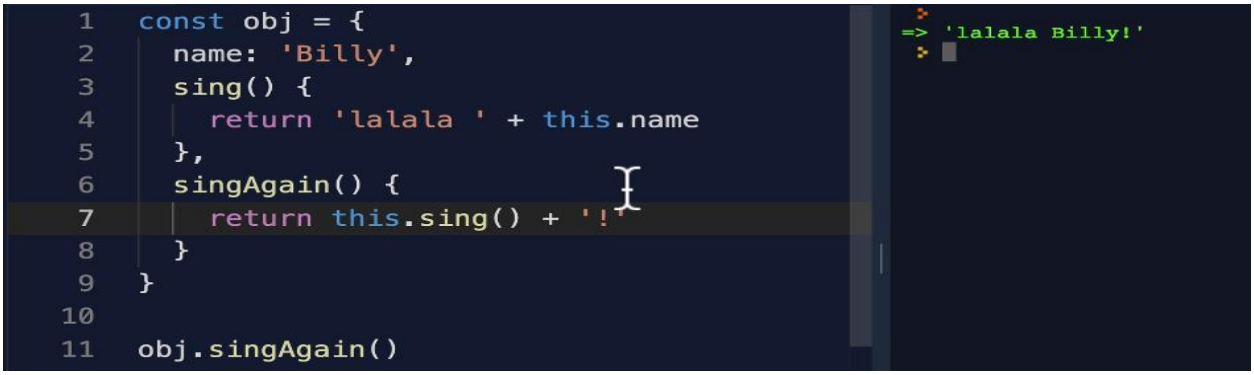
43. this keyword:

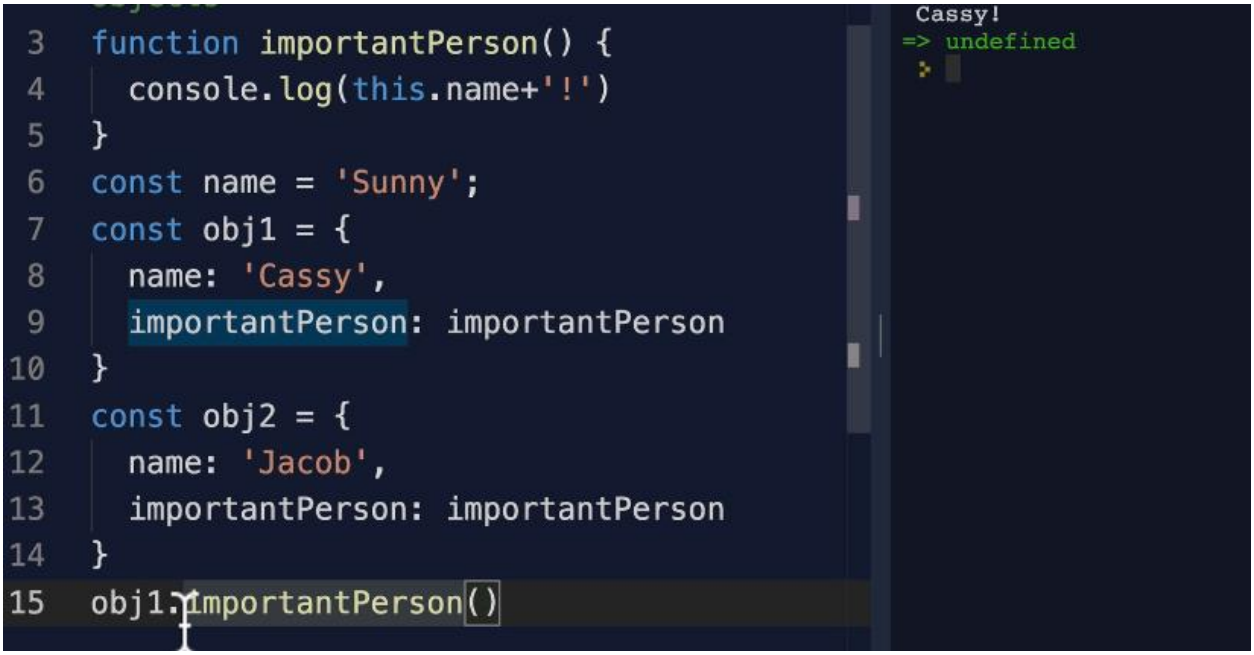
- This is the object that the function is a property of. Simply means that we have an object and this object has some function and inside of this function when we do something, we have access to the 'this' keyword. And 'this' keyword refers to the object that the function is a property of.



- Remember the definition. ‘this’ is the object that the function is a property of. That means we're calling window.a. Well the function ‘a’ is a property of the window object.
- One of the pitfalls with ‘this’ is that we unexpectedly have ‘this’ referred to the window object when we think it should be something else.
- 

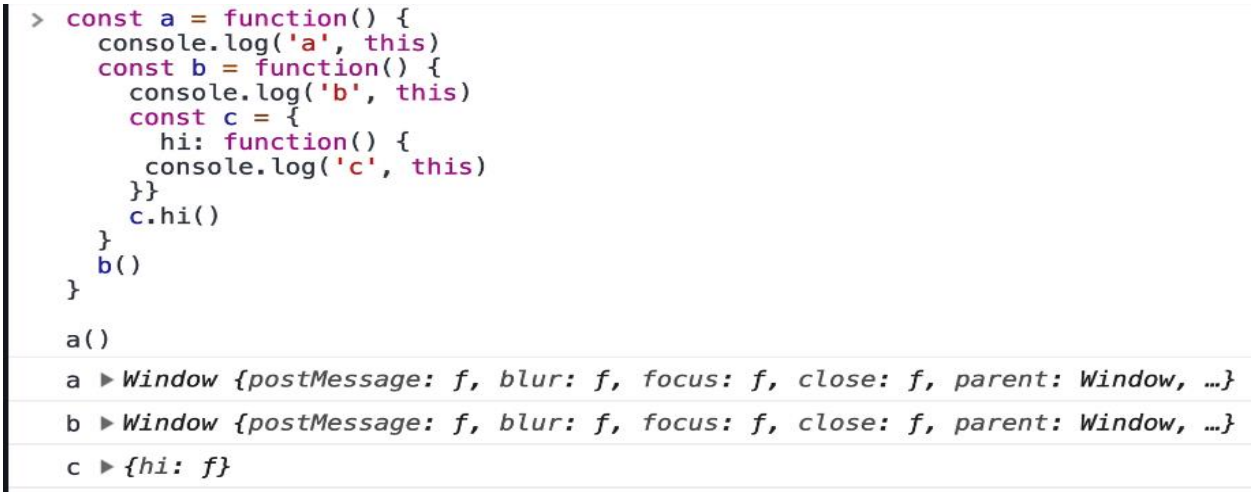
- 

- 

- 
function importantPerson() {
console.log(this.name+'!')
}
const name = 'Sunny';
const obj1 = {
name: 'Cassy',
importantPerson: importantPerson
}
const obj2 = {
name: 'Jacob',
importantPerson: importantPerson
}
importantPerson()
Sunny!
=> undefined
- 
function importantPerson() {
console.log(this.name+'!')
}
const name = 'Sunny';
const obj1 = {
name: 'Cassy',
importantPerson: importantPerson
}
const obj2 = {
name: 'Jacob',
importantPerson: importantPerson
}
obj1.importantPerson()
Cassy!
=> undefined

- 

44. Exercise: Dynamic Scope vs Lexical Scope:

- 

- Below image: anotherFunc runs on the global scope, to avoid global execution we can use arrow function.

```
> const obj = {
    name: 'Billy',
    sing() {
      console.log('a', this);
      var anotherFunc = function() {
        console.log('b', this)
      }
      anotherFunc()
    }
  }
<- undefined
> obj.sing()
  a ▶ {name: "Billy", sing: f}
  b ▶ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, …}
```

- Above example we are using arrow function which has the lexical scope.

```
> const obj = {
    name: 'Billy',
    sing() {
      console.log('a', this);
      var anotherFunc = function() {
        console.log('b', this)
      }
      return anotherFunc.bind(this)
    }
  }
<- undefined
> obj.sing()
  a ▶ {name: "Billy", sing: f}
<- f () {
       console.log('b', this)
     }
> obj.sing()()
  a ▶ {name: "Billy", sing: f}
  b ▶ {name: "Billy", sing: f}
```

- Store ‘this’ of outer function and pass it to the inner function.

```
> const obj = {
    name: 'Billy',
    sing: function() {
      console.log(this)
      var self = this;
      var anotherFunc = function() {
        console.log(self)
      }
      return anotherFunc
    }
  }

  obj.sing()()
  ▶ {name: "Billy", sing: f}
  ▶ {name: "Billy", sing: f}
```

45. call, apply, bind:

```
1  const wizard = {
2    name: 'Merlin',
3    health: 50,
4    heal() {
5      return this.health = 100;
6    }
7  }
8
9  const archer = {
10   name: 'Robin Hood',
11   health: 30
12 }
13 console.log('1', archer)
14 wizard.heal.call(archer)
15 console.log('2', archer)
```

```
1 { name: 'Robin Hood', health: 30 }
2 { name: 'Robin Hood', health: 100 }
=> undefined
```

- **With Parameter:**

```
const wizard = {
  name: 'Merlin',
  health: 50,
  heal(num1, num2) {
    return this.health += num1 + num2;
  }
}

const archer = {
  name: 'Robin Hood',
  health: 30
}
console.log('1', archer)
wizard.heal.call(archer, 50, 30)
console.log('2', archer)
```

```
1 { name: 'Robin Hood', hea
2 { name: 'Robin Hood', hea
=> undefined
```

- **Apply:** Same as Call but take this and array as parameters. Call take this and comma separated list of arguments.

main.js saved

```
const wizard = {
  name: 'Merlin',
  health: 50,
  heal(num1, num2) {
    return this.health += num1 + num2;
  }
}

const archer = {
  name: 'Robin Hood',
  health: 30
}
console.log('1', archer)
wizard.heal.apply(archer, [100, 30])
console.log('2', archer)
```

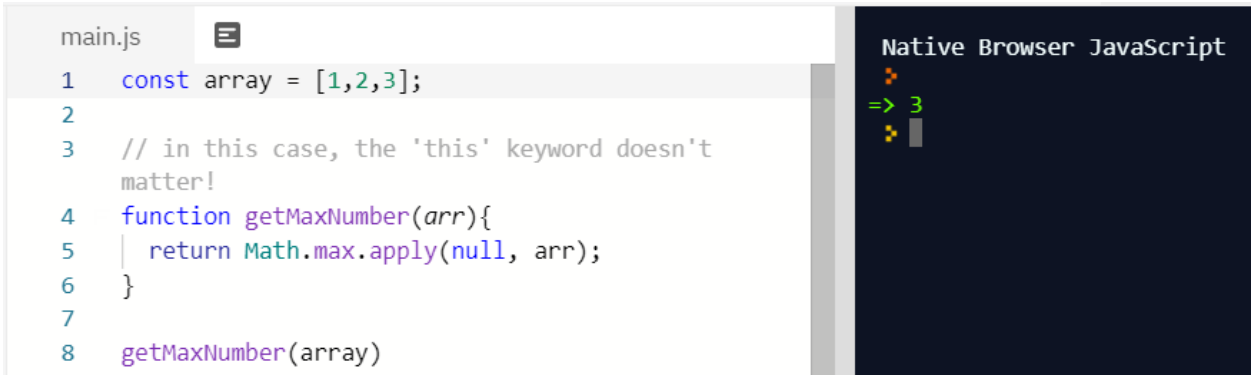
Native Browser JavaScript

```
1 { name: 'Robin Hood', health: 30 }
2 { name: 'Robin Hood', health: 160 }
=> undefined
```

- **'bind' allows us to store the 'this' keyword or function with changed 'this' for later use.**

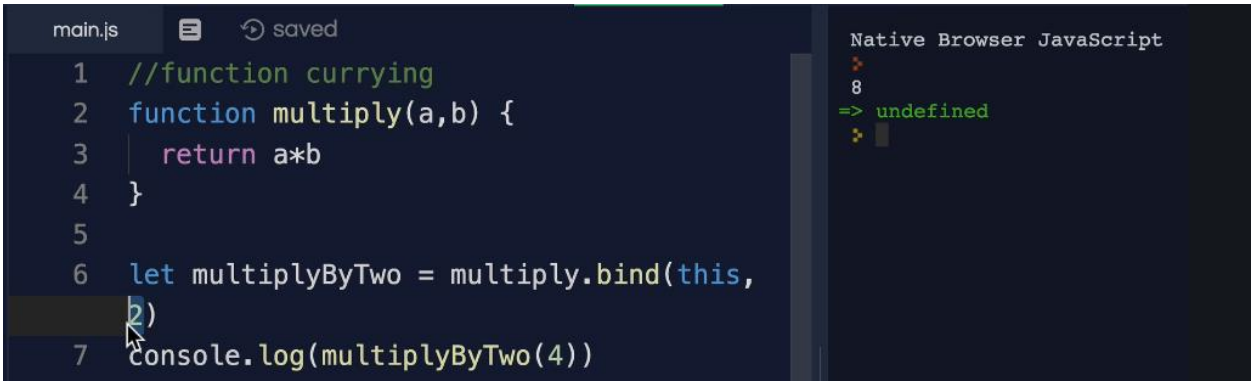
- call and apply are useful for borrowing methods from an object while bind is useful for us to call functions later on with a certain context or certain ‘this’ keyword.

46. Exercise: call, apply:

- 

47. bind and currying:

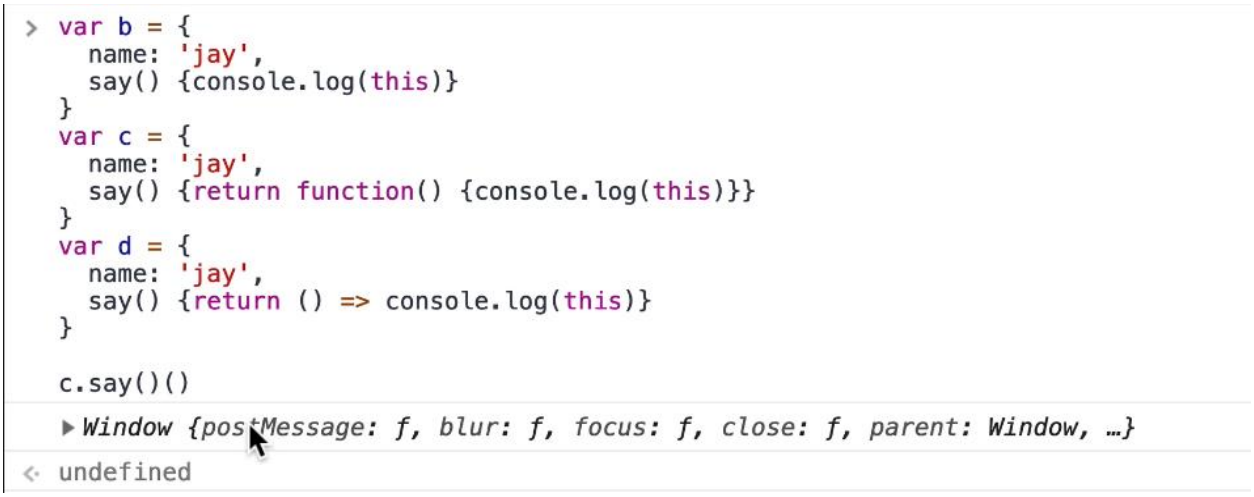
- There's one other useful trick that you can use with bind and it's called function currying. Now currying refers to well only partially giving a function a parameter. Will see later.

- 

- 

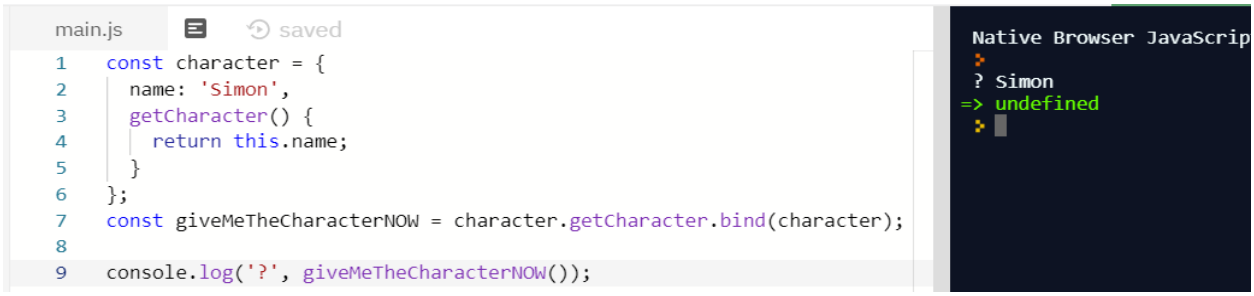
48. Exercise: this keyword:

- 

- 

- 

49. Exercise-2: this keyword:

- 

50. Context vs scope:

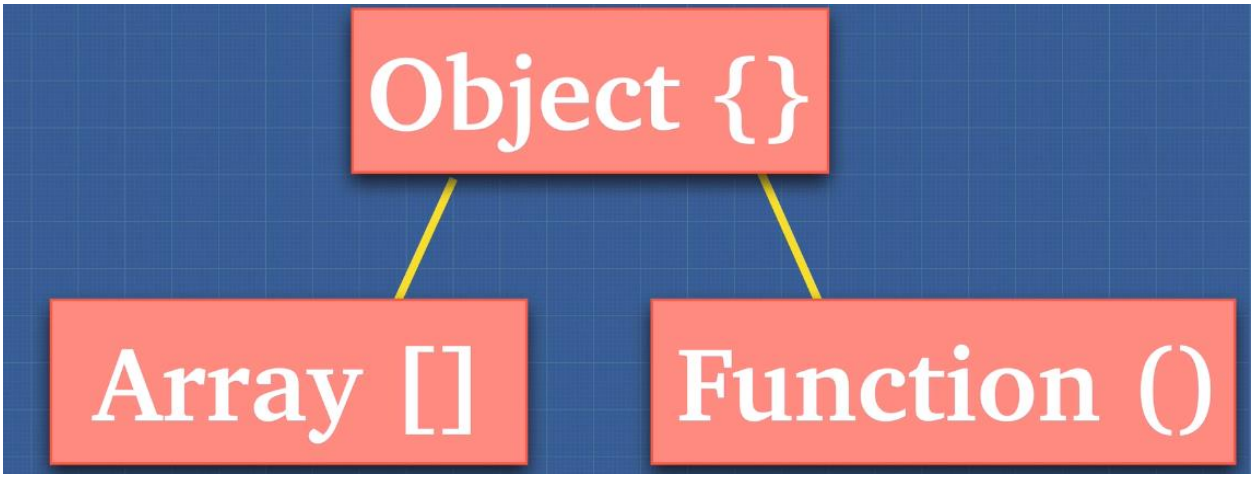
- Scope is a function-based thing. Scope means what is the variable access of a function when it is invoked.
- Context On the other hand is more about object-based context says what's the value of this keyword. Which is a reference to the object that owns that current executing code.
- Context is most often determined by how a function is invoked with the value of this keyword and scope refers to the visibility of variables.

53. JS types:

- ‘undefined’ is the absence of a definition. So, it's used as the default value when the JavaScript engine initializes our variables.

- JS engine initializes hoisted variables as ‘undefined’, only declared variables as ‘undefined’, functions return undefined when they don't return anything there's no return keyword in a function and if there's missing properties of an object.
- undefined simply means ‘absence of definition’
- null is an absence of value. There is no value there.
-

```
1   //Primitive
2   typeof 5
3   typeof true
4   typeof 'To be or not to be'
5   typeof undefined
6   typeof null
7   typeof Symbol('just me')
8
9   //Non-Primitive
10  typeof {}
11  typeof []
12  typeof function(){}
```

- Based on statement 12 There's clearly a type of function in JavaScript. Well technically no.
- I need you to trust me on something, that is arrays and functions are objects.
- 

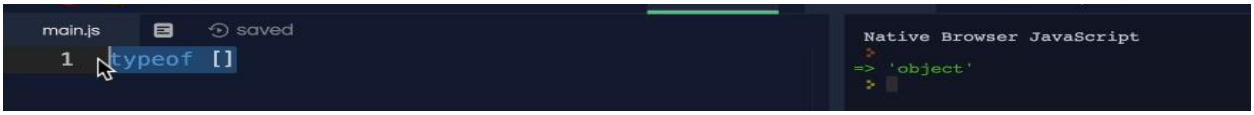
- Even though type of function gives us a ‘function’ type but underneath the hood a **function in JavaScript is just an object.**

-
```
function a() {
  return 5;
}

a.hi = 'hihihihih'

console.log(a.hi)
```
- We have two distinctions. Primitive and non-Primitive.
- **So, what is a primitive type**: It's a data that only represents a single value so that means this primitive five in memory the value is five. 5 is just 5 in memory.
- A variable of a primitive type directly contains the value of that type.
- **A non-primitive type** doesn't contain the actual value directly.
- Object doesn't contain the value here directly. Instead it has a reference like a pointer to somewhere in a memory where the object is held.
- Standard built in objects come with the language. It's part of the language. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects
- Many things that we interact with directly in JavaScript such just strings, numbers, Booleans which are primitive and not objects get a little bit complicated by the fact that these primitives have object wrappers like String or Number or Boolean.
- 

- ‘true’ is a primitive type. Why is it acting like an object by using dot notation and doing toString()? It silently creates a wrapper object around this
- Something like this when we try and attempt to access a property on a primitive. So, behind the scenes it's almost like it's wrapping this in Boolean() so that it has access to toString and then finally returns true. Like below:
- 

- 

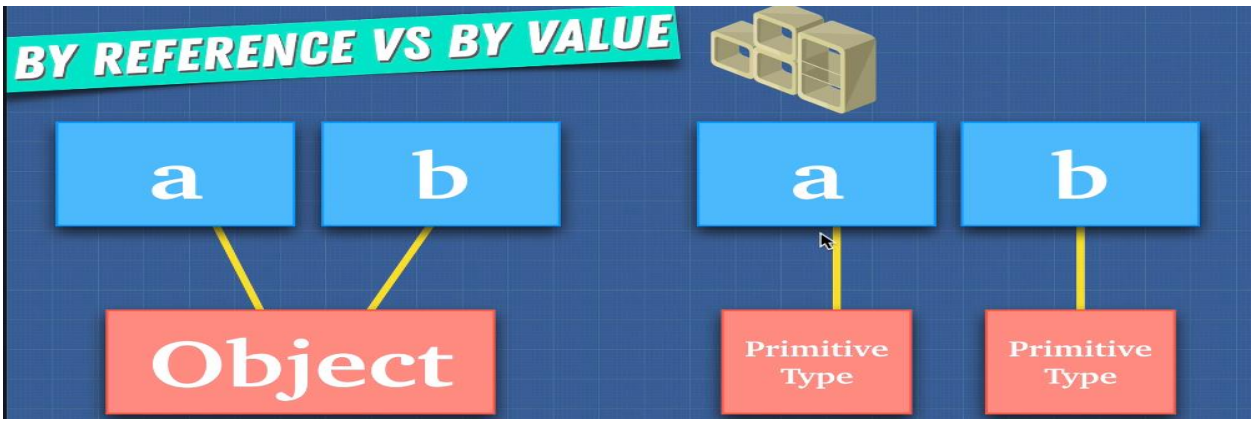
54. Array.isArray():

- Arrays are objects in JavaScript.
- 

- How to find weather it is an array ot not: Array.isArray()
- 

- 

55. Pass by Value vs Pass by Reference:

- 

- var a = 5; now ‘a’ has an address of where this primitive value 5 sits in memory.
- If var b=a; what happens: remember primitive types they're passed by value.
- pass by value simply means we copy the value and we create that value somewhere else in memory.

- Cloning the reference types: with Object.assign and spread operator.
-
```
let obj = {a: 'a', b: 'b', c: 'c'};
let clone = Object.assign({}, obj);
let clone2 = {...obj}

obj.c = 5;
console.log(obj)
console.log(clone)
console.log(clone2)
```
- Each object gets passed by reference. So, although we cloned the object, the initial object got cloned nested object did not clone. This is what we call a shallow clone it clones the first level.
-
```
b++;

let obj = {
  a: 'a',
  b: 'b',
  c: {
    deep: 'try and copy me'
  }
};
let clone = Object.assign({}, obj);
let clone2 = {...obj}

obj.c.deep = 'hahaha';
console.log(obj)
console.log(clone)
console.log(clone2)
```
```
{ a: 'a', b: 'b', c: { deep: 'hahaha' } }
{ a: 'a', b: 'b', c: { deep: 'hahaha' } }
{ a: 'a', b: 'b', c: { deep: 'hahaha' } }
=> undefined
```
- How can we do deep cloning:
-
```
let superClone = JSON.parse(JSON.stringify(obj))
```

- However, if you're doing a deep clone Well, we should be careful because above example can have some performance implications if this object was extremely deep or a massive object. It's going to take a long time to clone everything right.
- There's most likely some other ways that you should be doing things.

56. Exercise: Compare Objects

- https://stackoverflow.com/questions/1068834/object-comparison-in-javascript

57. Exercise: Pass by Reference:

-

main.js

```
const number = 100
const string = "Jay"
let obj1 = {
  value: "a"
}
let obj2 = {
  value: "b"
}
let obj3 = obj2;

function change(number, string, obj1, obj2) {
    number = number * 10;
    string = "Pete";
    obj1 = obj2;
    obj2.value = "c";
}

change(number, string, obj1, obj2);

//Guess the outputs here before you run the code:
console.log(number);
console.log(string);
console.log(obj1.value);
```

```
Native Browser JavaScript
100
Jay
a
=> undefined
```

58. Type Coercion:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness
- https://dorey.github.io/JavaScript-Equality-Table/
- https://www.ecma-international.org/ecma-262/5.1/#sec-11.9.3

- 

- This is because object that is works pretty much the same as the triple equals except for a few cases.

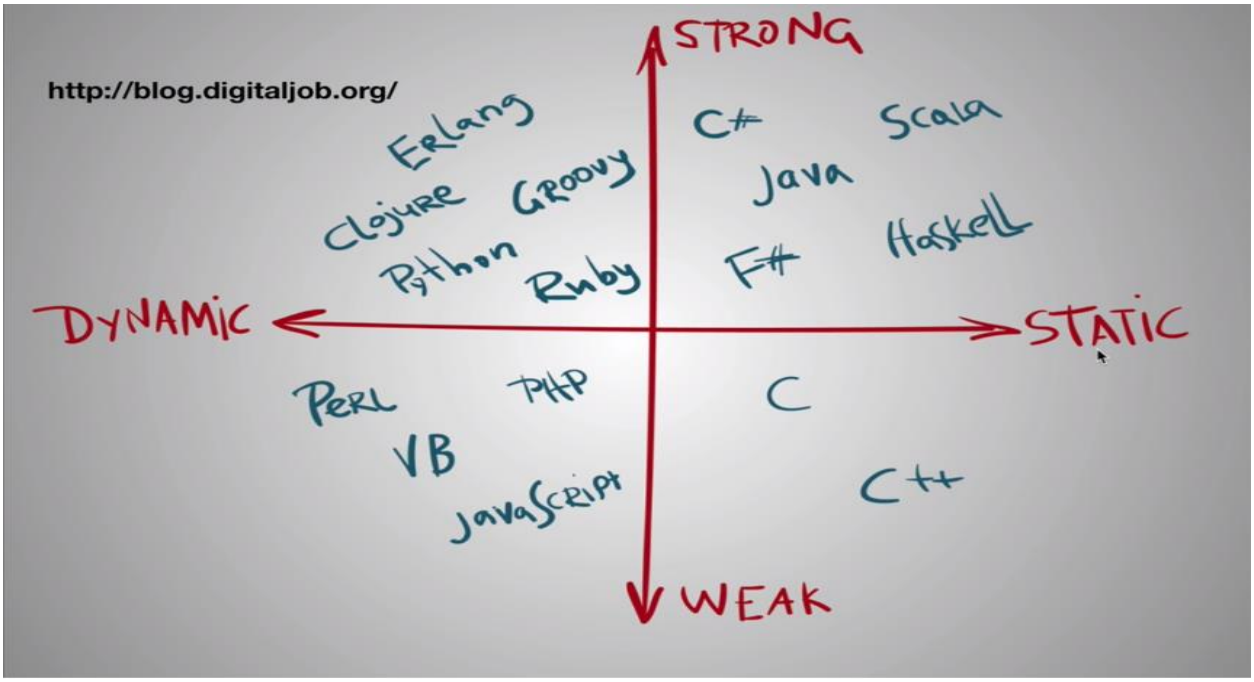
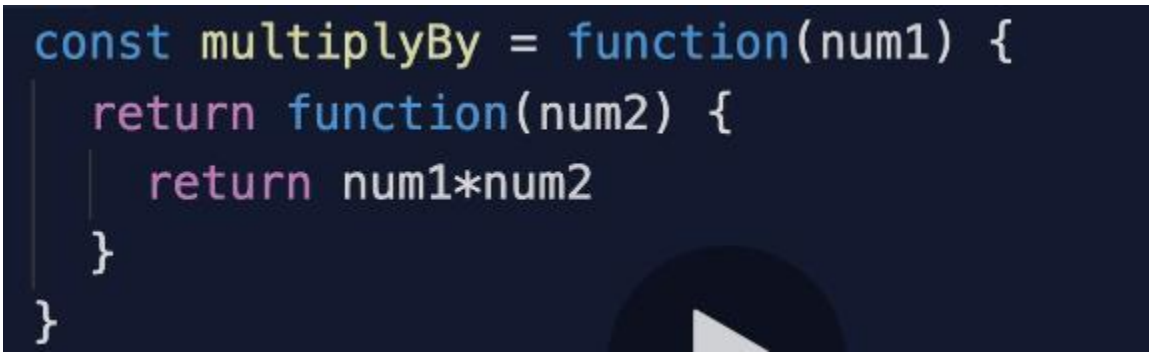
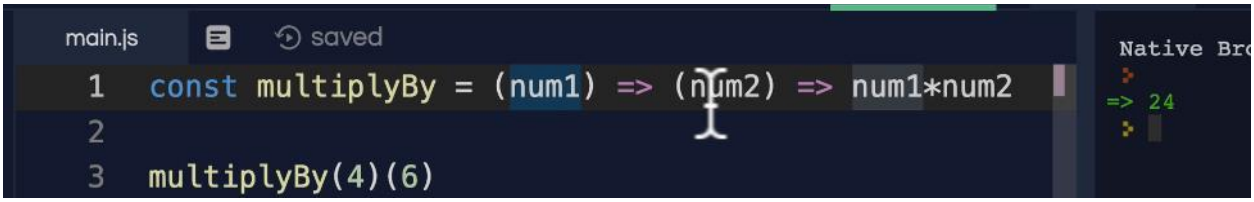
- 

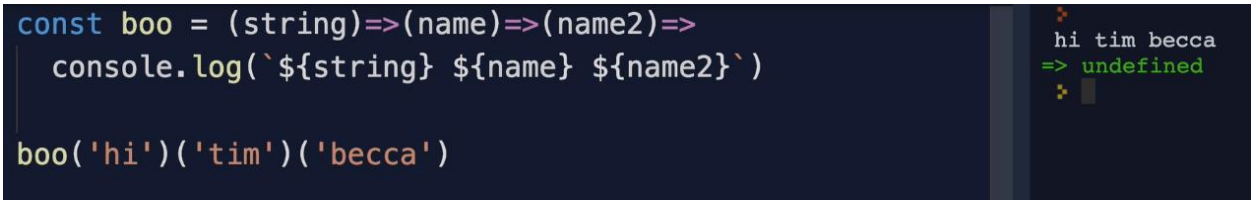
- 

59. Exercise: Type Coercion

- ```
  false == ""
  false == []
  false == {}
  "" == 0
  "" == []
  "" == {}
  0 == []
  0 == {}
  0 == null
  ```

61. JTS: Dynamic vs Static Typing

- **Typescript adds static typing** which adds an extra layer of security or an extra layer of type safety in the code.
- So, here's my outline of when to use static type checking like TypeScript. When your project grows larger and larger you already have tests written and has more developers join the team. You want the code to be self-documenting and also avoid bugs as new people touch the code.
- You also have the budget within the company to train new employees to learn this new language and expect that’s our development cycle. That is how fast we're able to write features and new code is gonna be slower because we're writing more code now that's your checklist.
- If that's the case, then we should add typescript. If not we shouldn't.
  ```

- 

63. JTS: Static Typing in JavaScript:

65. Functions are Objects:

68. Higher Order Function:

- Higher order functions are simply a function that can take a function as an argument or a function that returns another function

69. Exercise: Higher Order Functions

- 

- 

70. Closures:

- 
```
function a() {
  let grandpa = 'grandpa'
  return function b() {
    let father = 'father'
    return function c() {
      let son = 'son'
      return `${grandpa} > ${father} > ${son}`
    }
  }
}

a()()()
```
```
=> 'grandpa > father > son'
```

- ```
  //closures and higher order function
  function boo(string) {
    return function(name) {
      return function(name2) {
        console.log(`hi ${name2}`)
      }
    }
  }
  ```
- **Arrow function representation is below:**

- 
```
const boo = (string)=>(name)=>(name2)=>
  console.log(`${string} ${name} ${name2}`)

boo('hi')('tim')('becca')
```
```
hi tim becca
=> undefined
```

- That's because remember parameters are treated just like local variables that get stored in variable environments.
```

71. Exercise: Closures:

- Used Closures in below **example**:

```
//Exercise
function callMeMaybe() {
    const callMe = 'Hi! I am now here!';
    setTimeout(function() {
      console.log(callMe);
    }, 4000);
}

callMeMaybe();
```

```
=> undefined
Hi! I am now here!
```

- Used Closures in below **example**:

main.js saved

```
//Exercise
function callMeMaybe() {
  setTimeout(function() {
    console.log(callMe);
  }, 4000);
  const callMe = 'Hi! I am now here!';
}

callMeMaybe();
```

```
Native Browser JavaScript
=> undefined
Hi! I am now here!
```

74. Exercise: Closures 2

main.js

```
// Make it so that the initialize function can only be called once!
let view;
function initialize() {
    view = '🏠';
    console.log('view has been set!')
}

initialize();
initialize();
initialize();

console.log(view)
```

```
Native Browser JavaScript
```

75. Solution: Exercise 2

-
```
let view;
function initialize() {
  let called = 0;
  return function() {
    if (called > 0) {
      return
    } else {
      view = '';
      called = true;
      console.log('view has been set!')
    }

  }
}

const start = initialize();
start();
start();
start();
console.log(view)
```

```
view has been set!

=> undefined
```

76. Exercise Closures 3

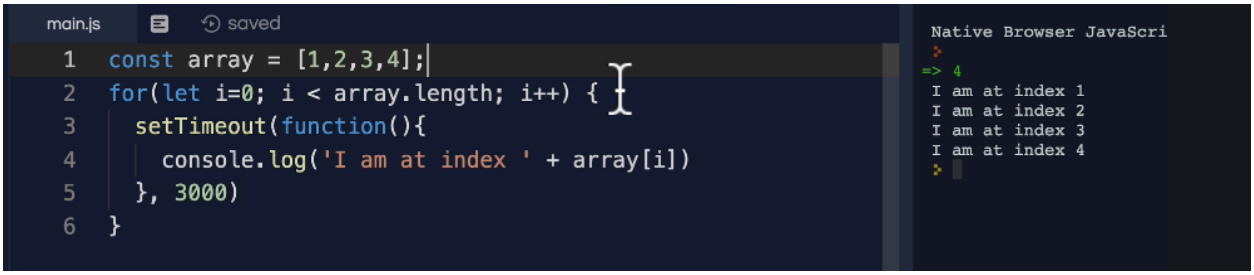
-
```
const array = [1,2,3,4];
for(var i=0; i < array.length; i++) {
  setTimeout(function(){
    console.log('I am at index ' + i)
  }, 3000)
}
```

```
=> 4
I am at index 4
I am at index 4
I am at index 4
I am at index 4
```

- Print 1, 2, 3, 4

77. Solution Closures 3:

- Now the easiest way to solve this is to change the VAR keyword to let.

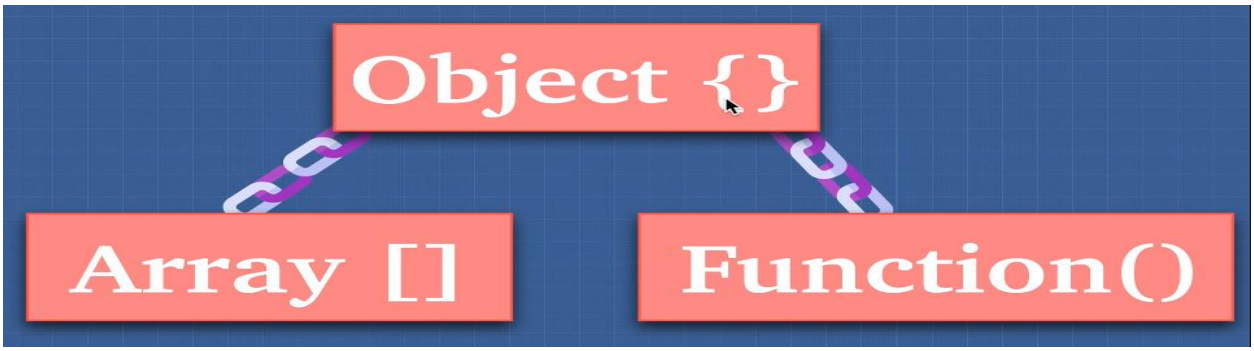


- **How it works:**

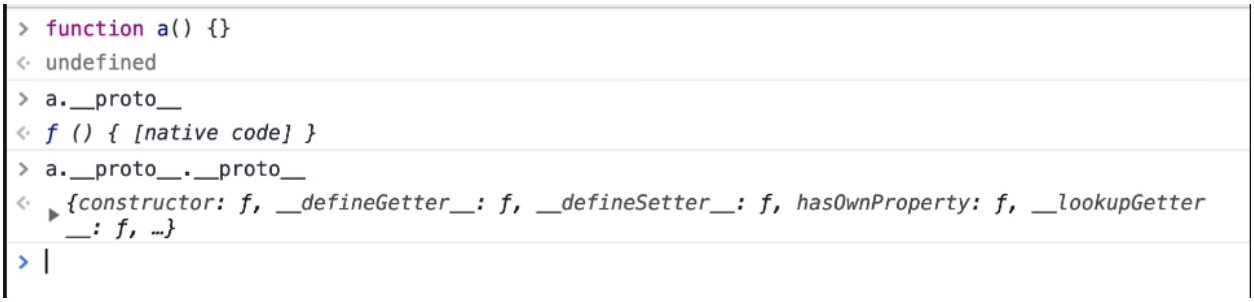
- Let allows us to use block scoping so that this block which is these curly brackets over here creates a scope for each I so that I is scoped within here because initially when we had variable I over here it was part of the global scope.

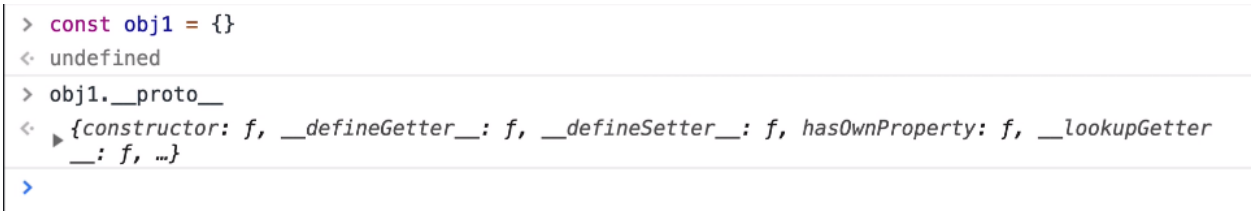
- Another way to solve: Using closures:

79. Prototypal Inheritance:

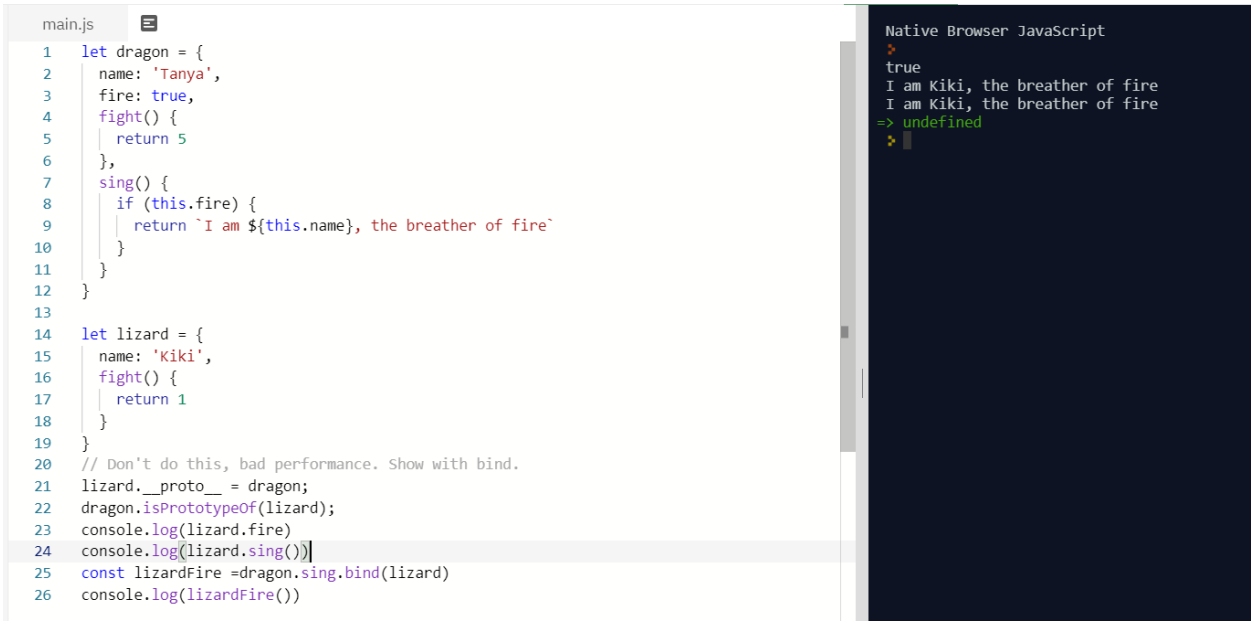


- Functions through the chain that we call prototypal inheritance functions get access to the methods and properties of object.

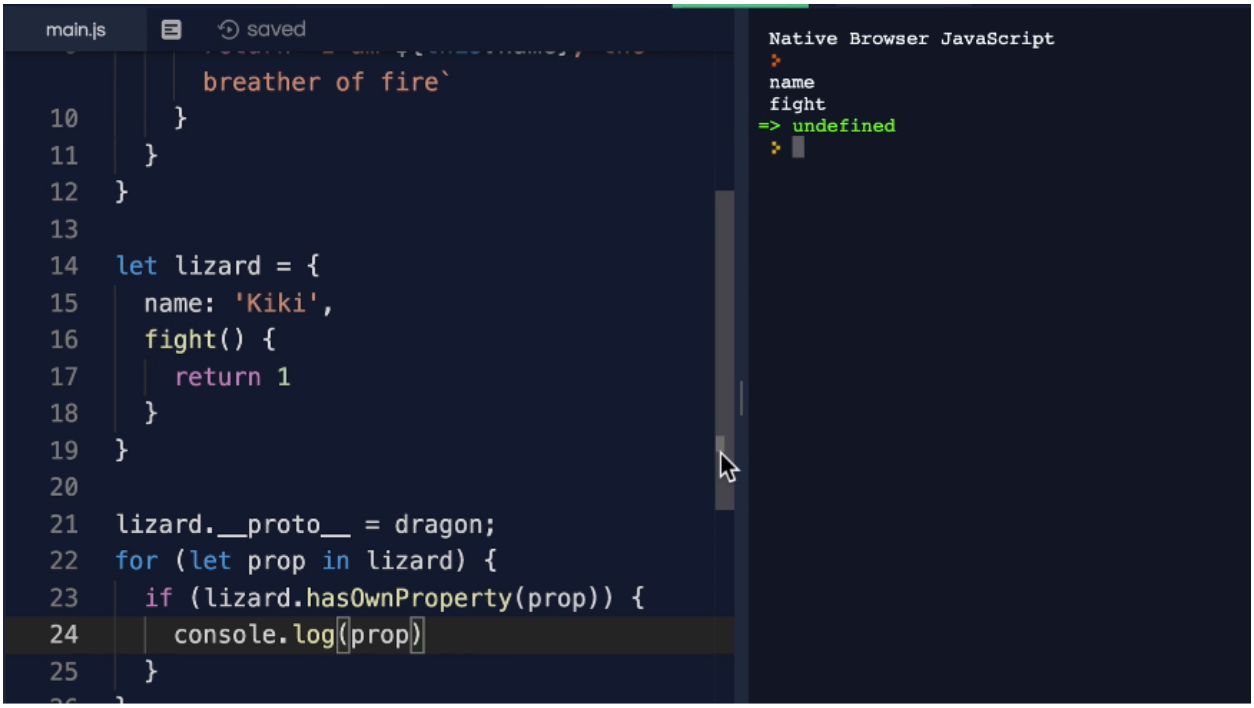


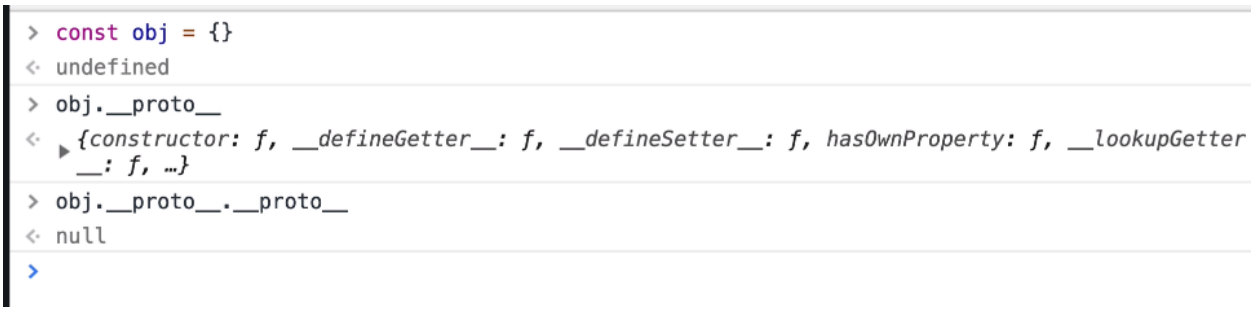


80. Prototypal Inheritance 2



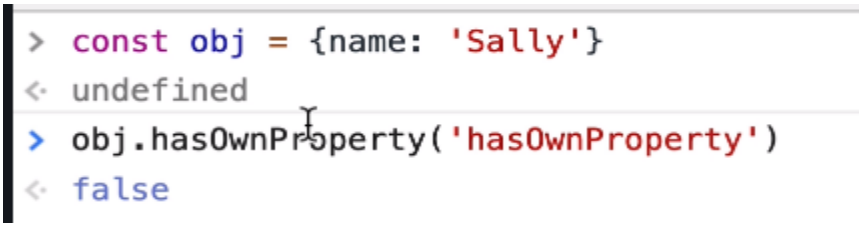
81. Prototypal Inheritance 3

- Printing only properties owned by lizard
- 

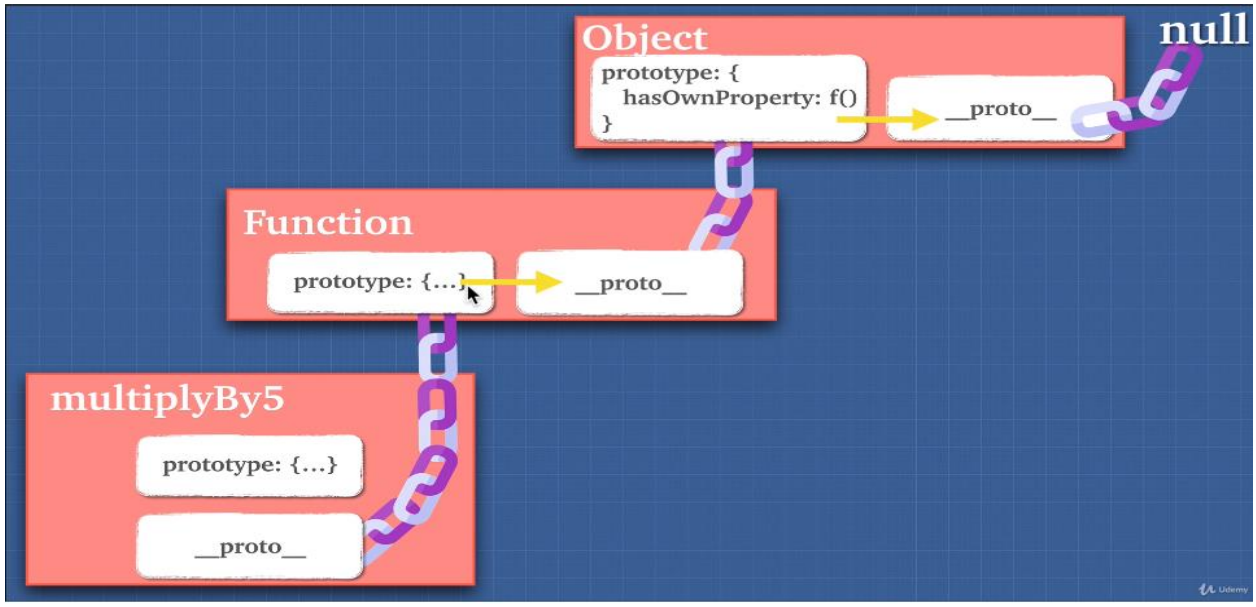
- 

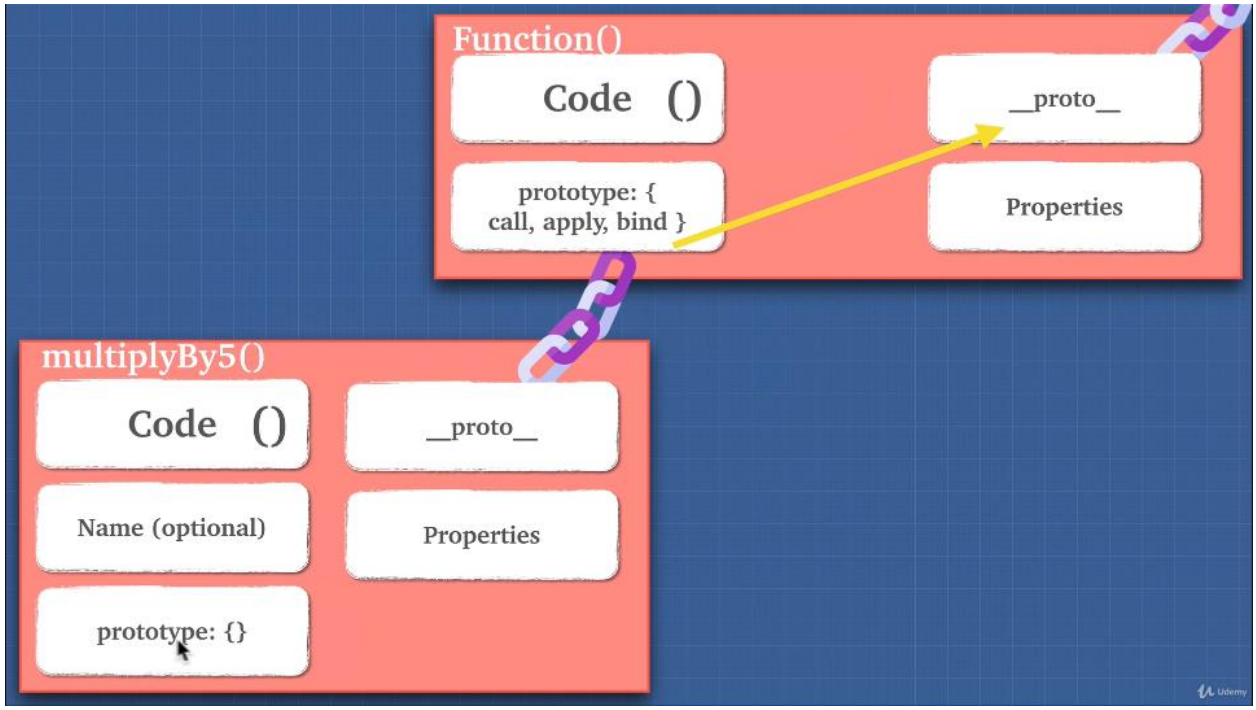
82. Prototypal Inheritance 4

- 

- 

```
> a.hasOwnProperty('name')
<- true
> a.hasOwnProperty('')
<- false
```





Packages

```
//Every Prototype chain links to a prototype object{}
function multiplyBy5(num) {
  return num*5
}

multiplyBy5.__proto__
Function.prototype
multiplyBy5.__proto__.__proto__
Object.prototype
multiplyBy5.__proto__.__proto__.__proto__
typeof Object
typeof {}
```

83. Prototypal Inheritance 5

main.js

```
// Create our own prototypes:
var human = {mortal: true}
var socrates = Object.create(human);
human.isPrototypeOf(socrates); // true
```

84. Prototypal Inheritance 6

```
> typeof Object.prototype
"object"
```

```
> String.prototype
<- String {"", constructor: ƒ, anchor: ƒ, big: ƒ, blink: ƒ, …}
    anchor: ƒ anchor()
    big: ƒ big()
    blink: ƒ blink()
    bold: ƒ bold()
    charAt: ƒ charAt()
    charCodeAt: ƒ charCodeAt()
    codePointAt: ƒ codePointAt()
    concat: ƒ concat()
    constructor: ƒ String()
```

```
> const obj = {}
<- undefined
> obj.prototype
<- undefined
> const arr = []
<- undefined
> arr.prototype
<- undefined
> 'string'.prototype
<- undefined
>
```

Only functions have the prototype property

- Using prototypes, we avoid repeating ourselves. We avoid adding the same code over and over and over and being inefficient with our memory

85. Exercise: Prototypal Inheritance

main.js

```
//Exercise - extend the functionality of a built in object

//#1
//Date object => to have new method .lastYear() which shows you last year 'YYYY' format.

new Date('1900-10-10').lastYear()
//'1899'


//#Bonus
// Mofify .map() to print '🗺' at the end of each item.
console.log([1,2,3].map())
//1🗺, 2🗺, 3🗺
```

86. Solution: Prototypal Inheritance

main.js

```
//Array.map() => to print '🗺'
Array.prototype.map = function()  { // what happens with arrow function here?
  arr = [];
  for (let i = 0; i < this.length; i++) {
    arr.push((this[i]+'🗺'));
  }
  return arr;
}
console.log([1,2,3].map())

//Date object => to have method .yesterday() which shows you yesterday's day in 'YYYY-MM-DD' format.
Date.prototype.lastYear = function(){
  return this.getFullYear() - 1;
}

new Date('1900-10-10').lastYear()
// don't worry if you didn't get this... we will expand on this later.
```

```
Native Browser JavaScript
[ '1🗺', '2🗺', '3🗺' ]
=> 1899
```

87. Exercise: Prototypal Inheritance with this

main.js

```
Function.prototype.bind = function(whoIsCallingMe){
  const self = this;
  return function(){
    return self.apply(whoIsCallingMe, arguments);
  };
}
```

92. OOP1: Factory Functions

- main.js

```
// factory function make/create
function createElf(name, weapon) {
  //we can also have closures here to hide properties from being changed.
  return {
    name: name,
    weapon: weapon,
    atack() {
      return 'atack with ' + weapon
    }
  }
}
const sam = createElf('Sam', 'bow');
const peter = createElf('Peter', 'bow');

sam.atack()
```

```
Native Browser JavaScript
=> 'atack with bow'
```

93. OOP2: Object.create()

- main.js

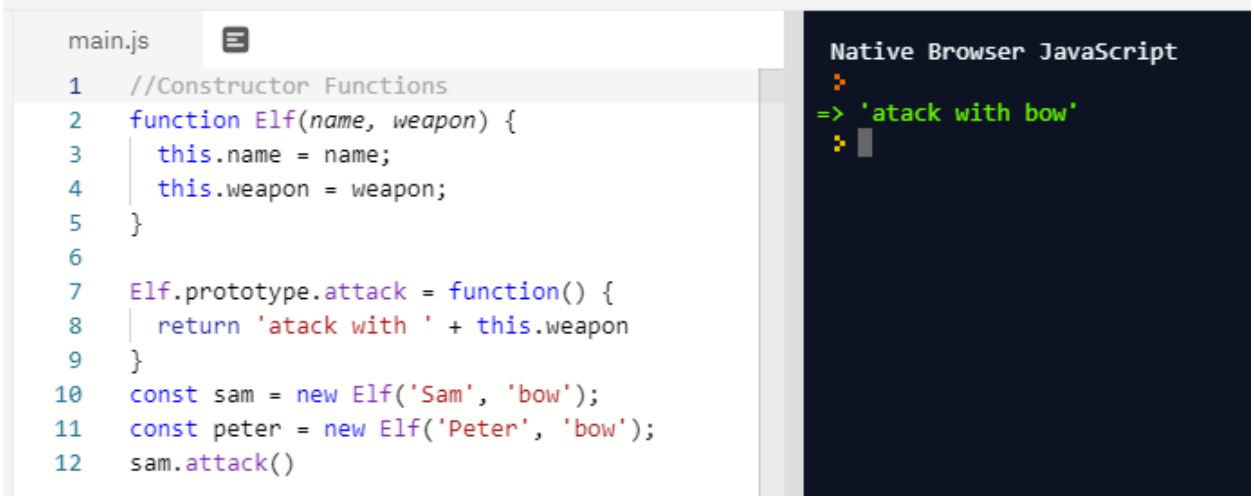
```
const elfFunctions = {
  attack: function() {
    return 'atack with ' + this.weapon
  }
}
function createElf(name, weapon) {
  //Object.create creates __proto__ link
  newElf = Object.create(elfFunctions)
  newElf.name = name;
  newElf.weapon = weapon
  return newElf
}


const sam = createElf('Sam', 'bow');
const peter = createElf('Peter', 'bow');
sam.attack()
```

```
Native Browser JavaScript
=> 'atack with bow'
```

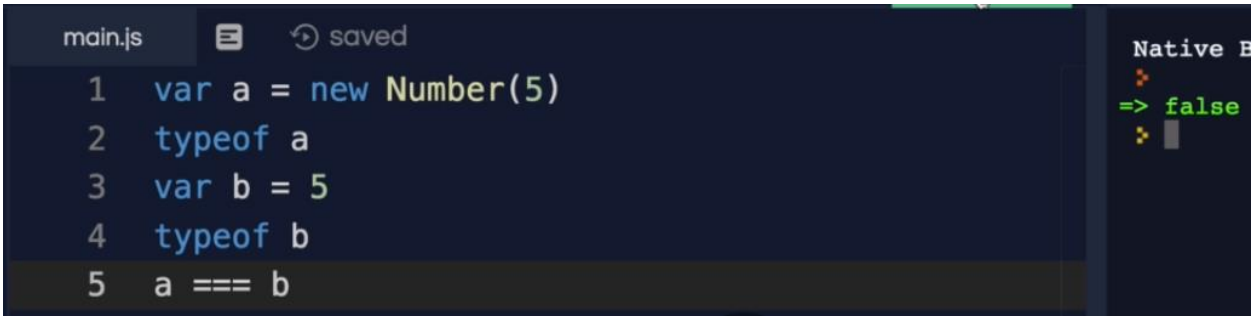
94. OOP3: Constructor Functions

-

```
const Elf1 = new Function('name', 'weapon',
`this.name = name;
  this.weapon = weapon;`)

const sarah = new Elf1('Sarah', 'fireworks')
sarah
```

- 

96. Funny Thing About JS...

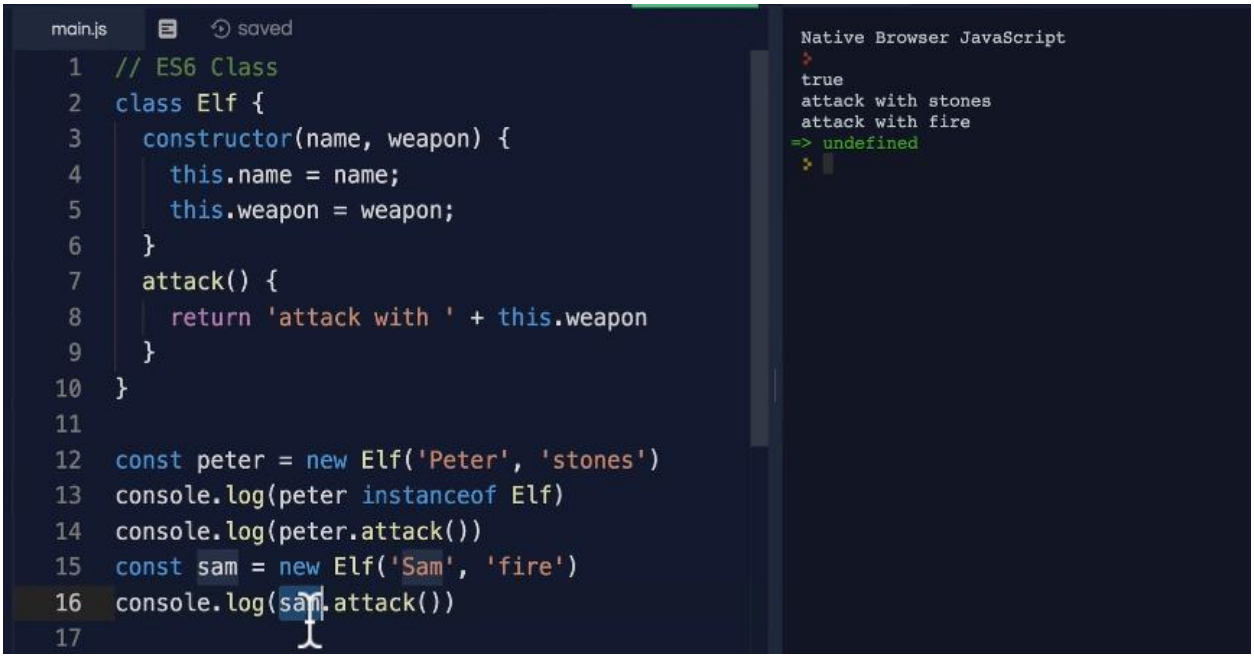
- If I do variable ‘a’ equal’s new number five. Well we know that JavaScript is going to create number five for us.
- But you see here how ‘a’ is a type of object because we've used a constructor function so it's not the same as we are saying variable ‘b’ equals to 5 which is a number.
- But if I do ‘a’ equal’s b and I run this I get ‘false’ because there's two different ways of constructing these things in JavaScript.
- 

- If I just do two equals, I get true because these types get worst.
- 

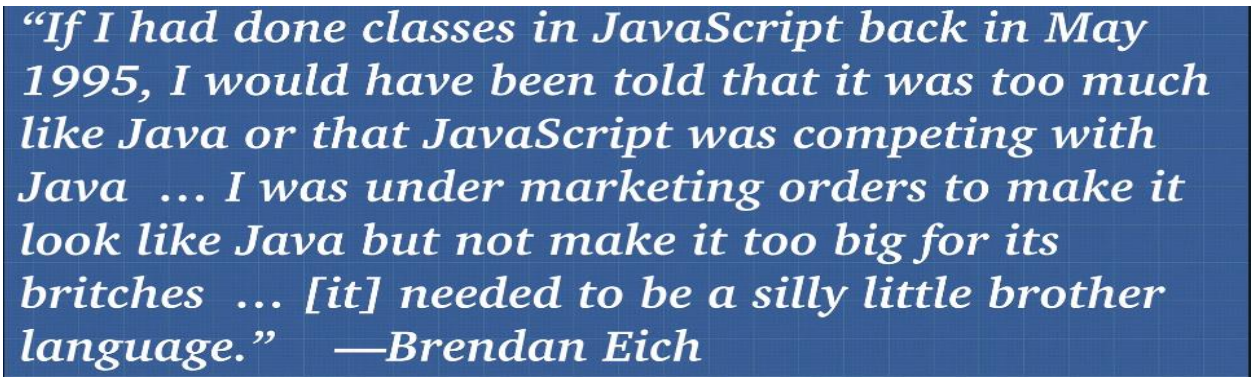
- In JavaScript everything is an object. Everything has a constructor function for it.
- I mean except for null and undefined we have constructor functions for everything so that we have methods that we can use.

97. OOP4: ES6 Classes:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes



- **Do we finally have classes we finally have object-oriented programming in JavaScript right? →**
- Well not really. You see this is what we call syntactic sugar underneath the hood in JavaScript. We're still using prototype all inheritance we're not using classes like classes work in other languages.
- Why JavaScript does not have classes:



- But at the same time, they wanted to create a competing language and for marketing purposes they couldn't make it the exact same.
- So, he had to be creative. So, he used prototype all inheritance which is quite different from how classes work in languages like Java and C++ in other languages.
- In JavaScript. Classes are still just objects. Everything in JavaScript is an object.
- **So, in an interview if you get asked hey does JavaScript have classes**.
- Well yes, they do **as syntactic sugar, but class keyword is still just prototype all inheritance. And some people call this pseudo classical inheritance because it's not real classical inheritance.**
- Looking at this I hope you see how we have gone from repetitive code that was harder and harder to maintain to something that's a little bit more organized.
- We've created this grouping of functionality so that it's easier to maintain and extend by the way.
- Question: In interview you may be asking yourself why to do we just add attack to the constructor. Why do we just put it outside over here?
- Answer: You see every time we use the new keyword and create or instantiate a class, the constructor function gets run and has a unique name and a unique weapon as parameter but attack method is shared by all instances of the class if we moved attack to the constructor, that's going to take up memory space.
- **Instead we just have one function in one location that all these instances can access**.

98. Object.create() vs Class:

- Now there is a big debate in the JavaScript community. People that love classes and people that never want to use them never want to see new keyword and they just want to avoid using ‘this’ keyword too much because they say it causes too much confusion and
- With Object.create We have everything we need to create these prototype chains without pretending like we have classes.
- However, most of the JavaScript community doesn't use object to create as much as the class syntax.

99. this - 4 Ways:

- main.js

```
1   // new binding
2   function Person(name, age) {
3     this.name = name;
4     this.age =age;
5     console.log(this);
6   }
7
8   const person1 = new Person('Xavier', 55)
9
10  //implicit binding
11  const person = {
12    name: 'Karen',
13    age: 40,
14    hi() {
15      console.log('hi' + this.name)
16    }
17  }
18
19  person.hi()
```

```
Native Browser JavaScript
Person { name: 'Xavier', age: 55 }
hiKaren
=> undefined
```

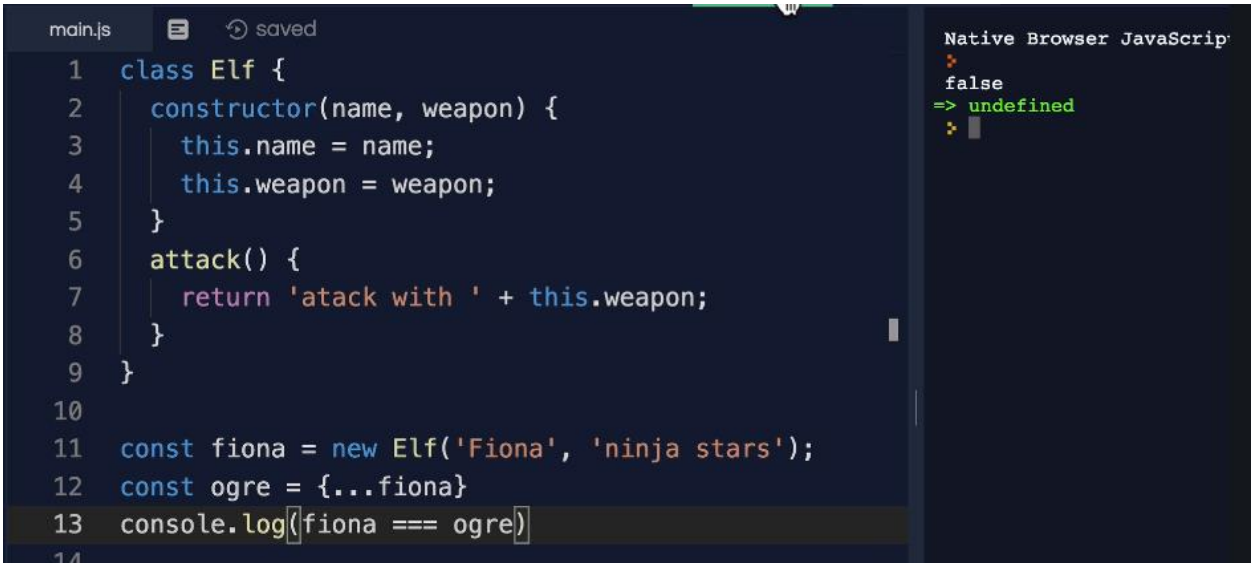
- main.js

```
1   //explicit binding
2   const person3 = {
3     name: 'Karen',
4     age: 40,
5     hi: function() {
6       console.log('hi ' + this.setTimeout)
7     }.bind(window)
8   }
9
10  person3.hi()
11
12  // arrow functions
13  const person4 = {
14    name: 'Karen',
15    age: 40,
16    hi: function() {
17      var inner = () => {
18        console.log('hi ' + this.name)
19      }
20      return inner()
21    }
22  }
23
24  person4.hi()
```

```
Native Browser JavaScript
hi function setTimeout() { [native code] }
hi Karen
=> undefined
```

- Unlike all the other ways where ‘this’ is dynamically scoped, arrow functions we can do lexical scoping that is wherever we write the function that's what it binds to.

100. Inheritance:

- 

- These objects (Fiona and ogre) are not referencing the same place in memory they're completely different things but I've also lost this chain this prototypal inheritance chain I can't even do ogre.attack.

- 

-

```
main.js   saved
    this.weapon = weapon;
  }
  attack() {
    return 'atack with ' + this.weapon;
  }
}

class Elf extends Character {
  constructor(name, weapon, type) {
    this.type = type
  }
}

const dolby = new Elf('Dolby', 'cloth', 'house');
dolby
```

```
Native Browser JavaScrip
ReferenceError: Must call
in derived class before a
returning from derived co
    at new Elf:13:5
    at eval:17:13
    at eval
    at new Promise
```

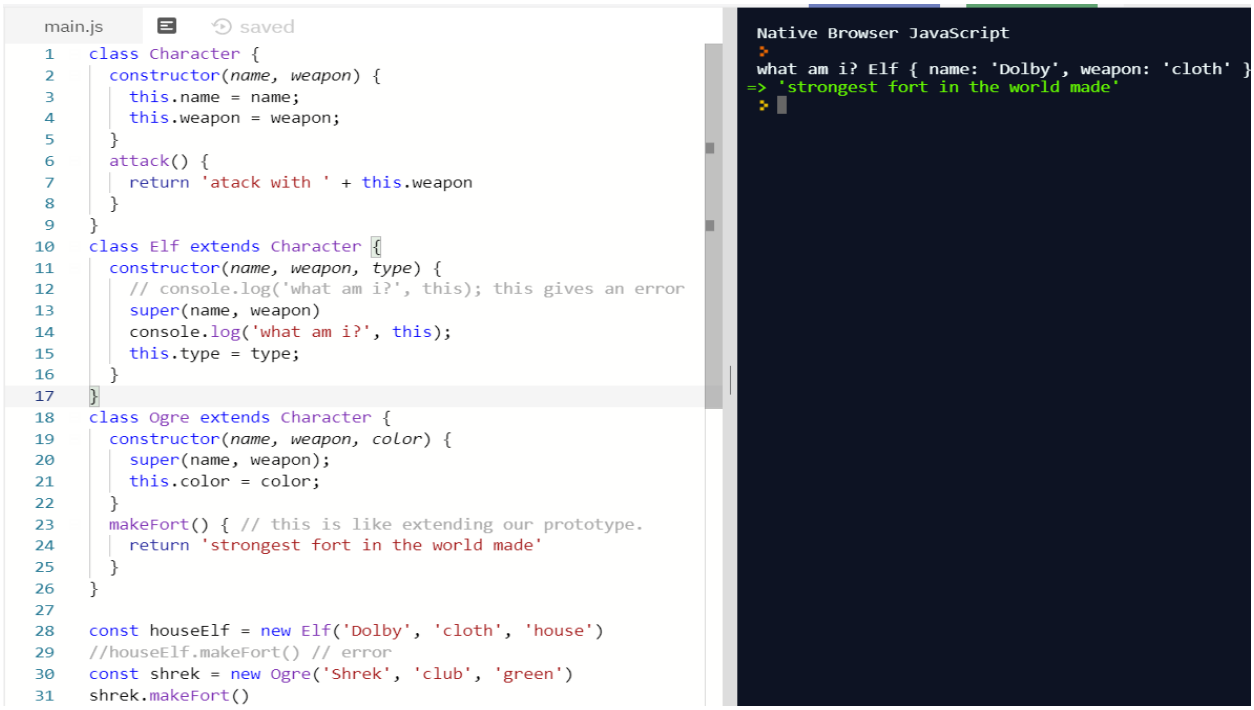
- **Error Message:**
-

```
Native Browser JavaScript
ReferenceError: Must call super constructor
in derived class before accessing 'this' or
returning from derived constructor
    at new Elf:13:5
    at eval:17:13
    at eval
    at new Promise
```

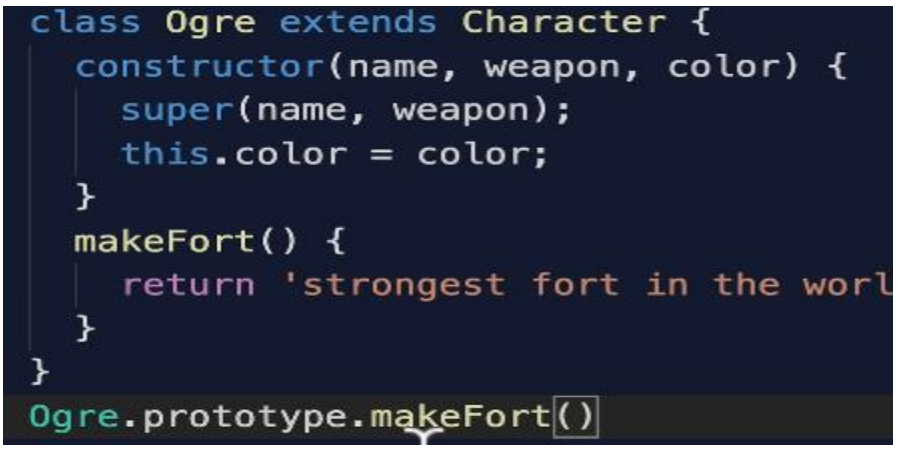
- We have a special keyword called ‘super’ to call superclass constructor.
-

```
class Character {
  constructor(name, weapon) {
    this.name = name;
    this.weapon = weapon;
  }
  attack() {
    return 'atack with ' + this.weapon;
  }
}

class Elf extends Character {
  constructor(name, weapon, type) {
    super(name, weapon);
    this.type = type
  }
}

const dolby = new Elf('Dolby', 'cloth', 'house');
dolby
```

```
=> Elf { name: 'Dolby', weapon
type: 'house' }
```

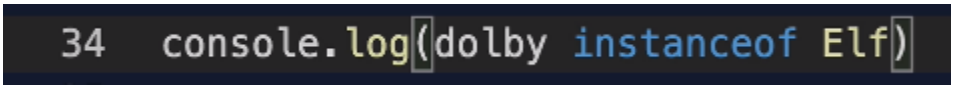
- When we do class, elf extends character it means hey extend and set the prototype that is the __proto__ to point to character. So, elf now has a prototype chain up to character.
- It's saying Hey anytime you run an instance of elf like Dobie and uses a property or a method that I don't have. Well then look up to character and tell me if character has.
- The constructor in elf is our own constructor just for the elf class. This is something that only gets run with an elf not with a character.
- And remember this keyword simply says who am I.
- 

101. Inheritance 2:

- 

- When we created makefort, what we did underneath the hood we extended the prototype.
- Now the cool part about this is a lot cleaner. and we're using some important object-oriented principles like classes and extending and creating subclasses and using the new keyword to create instances.
- We're also using underneath the hood the prototypal inheritance of JavaScript to make these inheritances.
- To test prototype chains between our objects and everything is linked properly. Let's do a few tests:
- **Test 1:**
- 

- **Test 2:**
- 

- **Test 3: Answer True**
- 

- **Test 4: Answer False**
- 

- **Test 5: Answer 5**
- 

- The keyword extends is inheriting something from a higher-class. Inheritance in JavaScript doesn't copy our functionality. It doesn't just simply copy whatever we have in character. Instead it simply links up the prototype chain.
- So, you're not creating copies and making things inefficient instead whenever it doesn't find something let's say on the ogre class it's going to look up to the ogre's superclass which is character.
- So, it's creating these efficient linking in JavaScript using prototypal inheritance.
- And I want to remind you unlike other class-based languages JavaScript is just objects its objects inheriting from objects.
- There are no technical classes in languages like Java on the other hand they have classes that are actual classes and classes inherit from classes.
- The interesting thing is that languages like Java and C++ copy objects. When we do something like extend instead of what we do with JavaScript which is that we link, and the objects are referenced. There's a bit of efficiency there in terms of memory.

102. Public vs Private:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Class_fields
- we're most likely the closest to doing something like this where we can have private state that is private data that can only be accessed inside of the class like this but private methods is something that we are still working towards again depending on when you watch the video.

105. Exercise: OOP and Polymorphism

- **Question**

main.js

```
class Character {
  constructor(name, weapon) {
    this.name = name;
    this.weapon = weapon;
  }
  attack() {
    return 'atack with ' + this.weapon
  }
}
//Polymorphism--
//Extend the Character class to have a Queen class. The output of the below code should be:
const victoria = new Queen('Victoria', 'army', 'hearts'); // create a new instace with the queen having (name, weapon, type). Type inlcudes: 'hearts', 'clubs', 'spades', 'diamonds'

victoria.attack() // will console.log the attack() method in Character class AND will return another string: 'I am the Victoria of hearts, now bow down to me! '
```

```
Native Browser JavaScript
ReferenceError: Queen is not defined
    at eval:12:16
    at eval
    at new Promise
```

- **Answer**

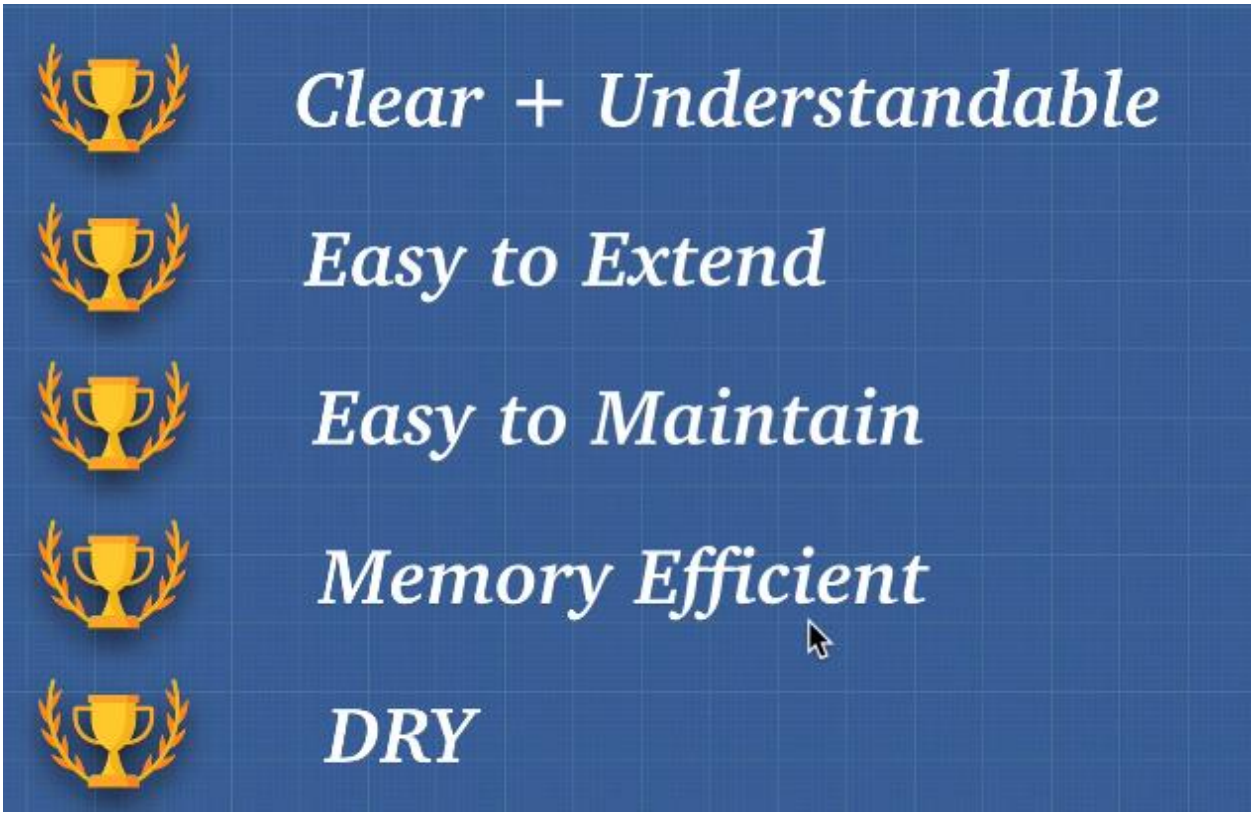
main.js

```
class Character {
  constructor(name, weapon) {
    this.name = name;
    this.weapon = weapon;
  }
  attack() {
    return 'atack with ' + this.weapon
  }
}

class Queen extends Character {
  constructor(name, weapon, kind) {
    super(name, weapon)
    this.kind = kind;
  }
  attack() {
    console.log(super.attack());
    return `I am the ${this.name} of ${this.kind}, now bow down to me! `
  }
}

const victoria = new Queen('Victoria', 'army', 'hearts');
victoria.attack()
```

```
Native Browser JavaScript
atack with army
=> 'I am the Victoria of hearts, now bow down to me! '
```

108. Functional Programming Introduction

- Functional programming has this idea as well of separating concerns but they also separate data and functions.
- There's data and then this data gets interacted with but we're not going to combine both data and functions as one piece or one object.

- Now there is no correct definition for what is and isn't functional but generally functional languages have an emphasis on simplicity where data and functions are concerned, we separate them up.
- The goals of functional programming are the exact same as that of object-oriented programming. It's the idea to make our code clear and understandable easy to extend easy to maintain.
- It's going to allow us to not repeat ourselves keeping our code dry and keeping our memory efficient by having these reusable functions that act on data.
- 

- Like pillars of oops in FP we have one pillar, if you want to break things down in functional programming it all comes down to this concept of pure functions.
- In all objects created in functional programming are immutable something that we'll talk about a little bit more. That means is once something is created it cannot be changed.
- We avoid things like shared state, and we adhere to this principle of pure functions.

109. Exercise: Amazon

main.js

```
// Amazon shopping
const user = {
  name: 'Kim',
  active: true,
  cart: [],
  purchases: []
}


//Implement a cart feature:
// 1. Add items to cart.
// 2. Add 3% tax to item in cart
// 3. Buy item: cart --> purchases
// 4. Empty cart

//Bonus:
// accept refunds.
// Track user history.
```

110. Pure Functions:

- Produces no Side effect
- Given the same input, will always return the same output.
-

main.js

```
1  //Side effects:
2  const array = [1,2,3];
3  function mutateArray(arr) {
4    arr.pop()
5  }
6  function mutateArray2(arr) {
7    arr.forEach(item => arr.push(1
8    ))
9  }
10 //The order of the function calls will matter.
11 mutateArray(array)
12 mutateArray2(array)
13 array
```

```
Native Browser JavaScript
=> [ 1, 2, 1, 1 ]
```

- But above function has side effect: side effects are doing the function modify anything outside of itself like here it modifies array which is from outside.

111. Pure Functions 2:

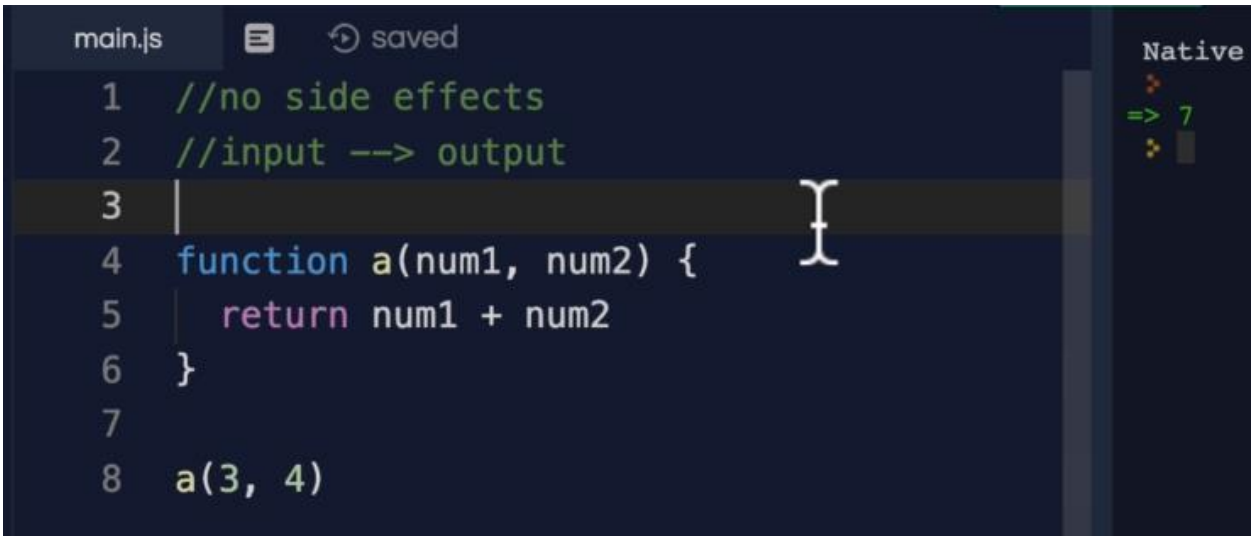
- Implementing no side effect: by changing in other array not on the same array.
- So, in removeLastItem method we created new array and concat out outside array with it.
- And in multiplyBy2 we used map method which will return a new array so no change in the outside array.
- map and concat methods can fix this issue of mutation
-

main.js saved

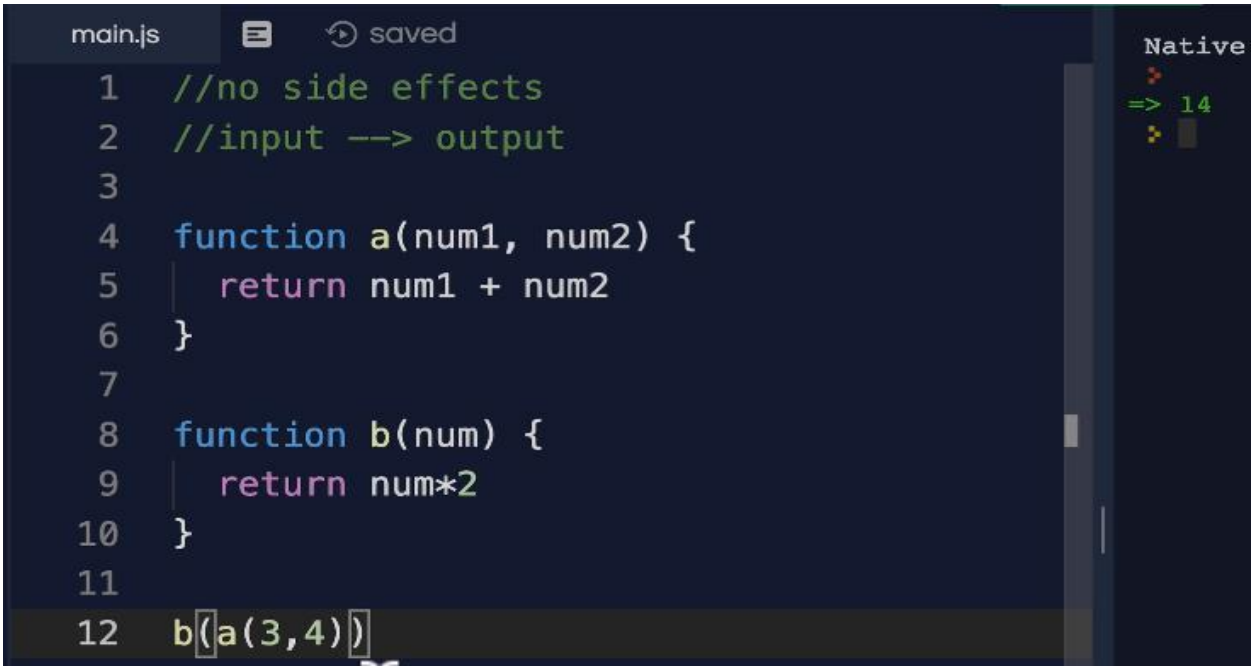
```
1  //no side effects
2  //input --> output
3
4  const array = [1,2,3]
5  function removeLastItem(arr) {
6    const newArray = [].concat(arr);
7    newArray.pop()
8    return newArray
9  }
10 function multiplyBy2(arr) {
11   return arr.map(item => item*2)
12 }
13 const array2 = removeLastItem(array);
14 const array3 = multiplyBy2(array);
15 console.log(array, array2, array3)
```

```
Native Browser JavaScript
[ 1, 2, 3 ] [ 1, 2 ] [ 2, 4, 6 ]
=> undefined
```

- Is this a pure function: NO?
- 

- That's a tricky one but console dialogue is well window specific. We're using the browser to log something to the browser. So, it's affecting the outside world.
- It's logging something to the output of the browser.
- It's modifying something outside of itself. so, this function has side effects.
- All right let's go to the next one. The input should always result in the same output.
- 

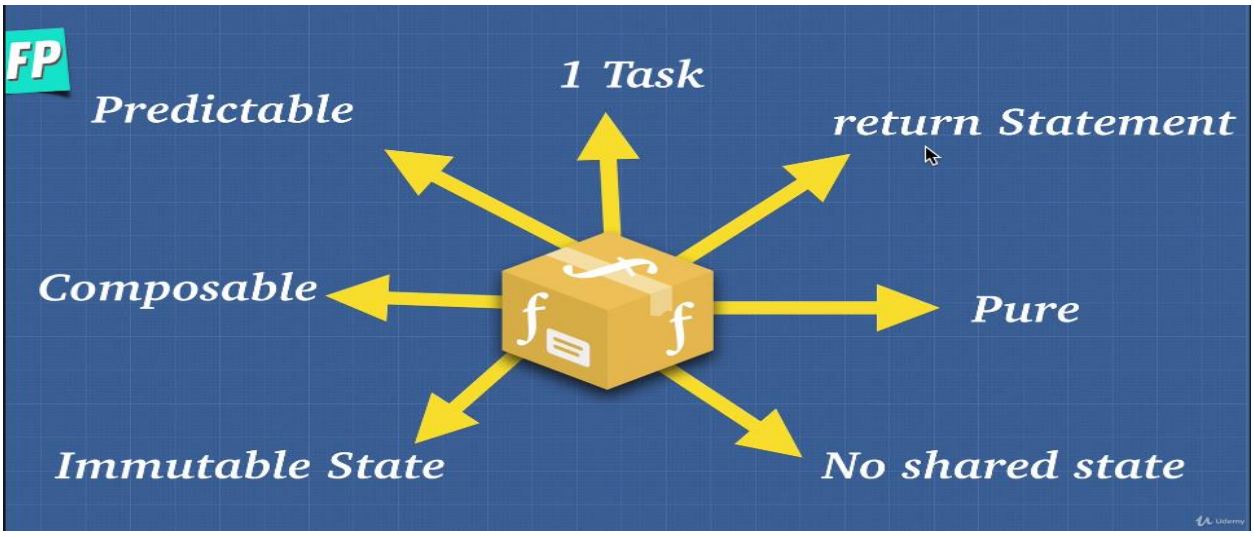
- So, let's do three and four clicks Run and I click many times and it's always 7 and this is what we call referential transparency.
- Referential transparency simply means if I completely change this function to the number 7 will affect any part of my program.

- 

- When we pass method a (3,4) in b method, we get 14.
- Referential Transparency says if we change a (3,4) to 7 in method b will it have any effect on the program. And no, it doesn't.
- Because of second rule no matter what my input if they're the same it's always going to give me the same output.
- And as a matter of fact, these functions also have no side effects right. They're not touching any of the outside world. They're only touching their own parameters and parameters.
- The idea with pure functions is that it makes functions very easy to test very easy to compose and it avoids a lot of bugs because we have no mutations no shared state.
- We have these predictable functions that minimize the bugs in our code.

112. Can Everything Be Pure?

- As a matter of fact, input output is a side effect that is communicating with the outside world in any way that is why they are not pure.
- The interesting thing is that with just pure functions which are just functions
- that do something on the inside and the outside world knows nothing about is philosophically.
- Well it doesn't do anything because a program cannot exist without side effects.

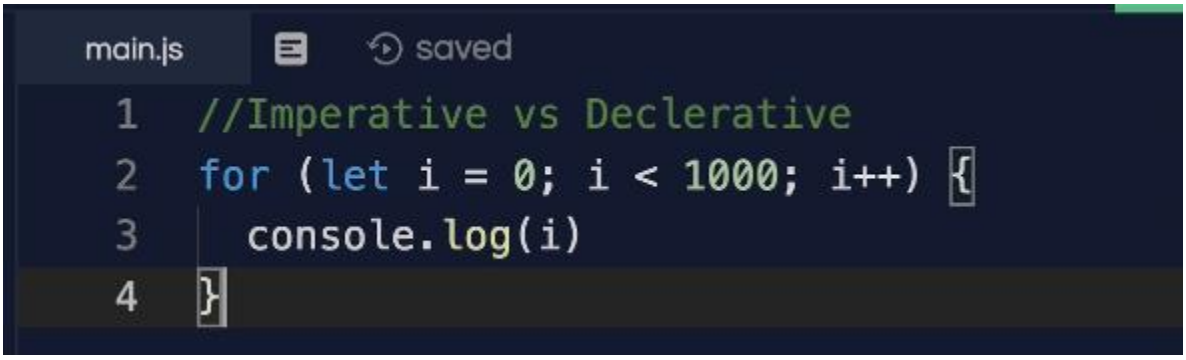
- We can't run this piece of code without having a side effect of interacting with the browser.
- We can't have Web sites with just pure functions. Browsers have to make fetch calls http calls to the outside world.
- We must interact with the Dom and manipulate what's on the Web site.
- The goal of functional programming is not to make everything pure functions.
- The goal is to minimize side effects. The idea is to organize your code with there is a specific part that has side effects but when you have a bug you know right away to go to that spot because that's where the side effects are happening.
- The rest of your code are just pure functions and because they're pure we don't have to worry about them as much.
- purity is more of a confidence level. It cannot be 100 percent.
- So, if there's one thing that you take away from this is that side effects and impurity is not necessarily bad, but the goal is to organize your code in a way so that you isolate these side effects.
- These database calls API calls input output to a certain location in your program in your code so that your code becomes predictable and easier to bug because at the end of the day we do have to have some sort of a global state to describe our application. That's unavoidable.
- The essence of functional programming is very simple. We want to build programs that are built with a bunch of very small very reusable predictable pure functions.
- How do we build the perfect function?
- 

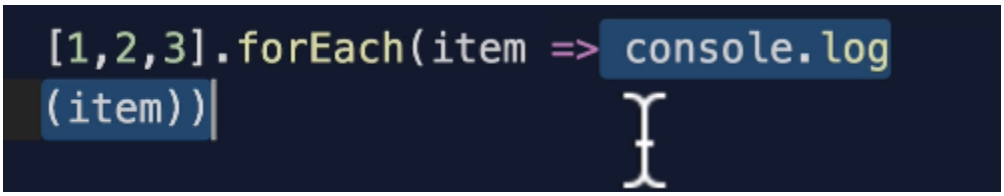
113. Idempotent:

- The idea of idempotence is a function that always returns or does what we expected to do.
- Now that sounds familiar with what we've just talked about pure functions but it's a little bit different because if I do console.log(5) here and I click run no matter how many times I click Run I always get 5.
- This function that console.log (5) to the outside world is still idempotent because multiple polls is still going to display the same text even though it's not pure because while it communicates with the outside world.
- Another thing that can be idempotent for example is deleting a user from a database.
- When we delete a user from a database and if I keep calling the function to delete that user, it's going to return the same result. It's going to return me that empty field where there's no more user.
- And things like idempotent, you see a lot in API is like http get requests. I can do an API call and I expect that that API call given some sort of parameter is always going to return the same results. Even though we are communicating with the outside world.

114. Imperative vs Declarative:

- Imperative code is code that tells the machine what to do and how to do it.
- Declarative code tells it what to do and what should happen. It doesn't tell the computer how to do things.
- computer is better at being imperative. That is, it needs to know how to do things.
- We on the other hand as humans are more declarative.

- If I tell my friend hey can you pass me that jug of water. Well I don't need to tell my friend hey can you walk over there with your right hand pick up the jug. Come towards me. Give it to me then release the jug into my hand. I don't have to tell it.
- Machine code is very imperative we say put the variable in this memory space and then take it out here and modify here.
- It's very descriptive of how to do things versus as we go higher and higher chain to something like a higher-level language. Well that becomes more declarative.
- We don't have to say hey this is where you should store the memory. We just declare variable with some sort of data, and we say what we need to get done but not how to do it. The computer takes care of that for us
- Another great example of imperative versus declarative is the idea of for loops.
- 

- Would you say that's imperative or declarative?
- This is imperative. We say declare variables zero and then loop for a thousand times and then also increment AI and console log. There's a lot of instruction here.
- How can you make this more declarative?
- 

- In above example I don't tell it to increment I by 1 and to loop through things. This is more declarative than previous one.
- Another classic example of this is jQuery. jQuery is a lot more imperative than what we have now on the front and frameworks like react, angular or view because with jQuery we told our Web site exactly what to do.

```
// Clicking away from the dropdown will collapse it.
$("html").click(function() {
  $(".dropdown").hide();
});
```

- Here in jQuery we tell Hey if the user clicks on the ACM L then grab the drop-down menu and then hide it. We tell it exactly what to do. But also, how to do it.
- But if you look at the react.js:

```
import React from 'react';

const Card = ({ name, email, id }) => {
  return (
    <div className='tc grow bg-light-green br3 pa3 ma2 dib bw2 shadow-5'>
      <img alt='robots' src={`https://robohash.org/${id}?200x200`} />
      <div>
        <h2>{name}</h2>
        <p>{email}</p>
      </div>
    </div>
  );
}

export default Card;
```

- Here if we look at card component for example react is fairly imperative, we have a function here that takes in some parameters and then returns a piece of HTML.
- We don't tell it what to display and how to do it just simply hey this is the data can you just display this.

```
<div className='tc'>
  <h1 className='f1'>RoboFriends</h1>
  <SearchBox searchChange={onSearchChange}/>
  <Scroll>
    { isPending ? <h1>Loading</h1> :
      <ErrorBoundry>
        <CardList robots={filteredRobots} />
      </ErrorBoundry>
    }
  </Scroll>
</div>
```

- See here I have an isPending and a loading and I'm simply saying Hey if isPending is the true. Then just do a loading screen otherwise just load up the Card List. I don't tell react how to do things. I just tell it hey this is what I want.
- This is more declarative why my teaching you this because functional programming helps us be more declarative by using functions and what we'll learn eventually which is composing functions we tell our programs What to do instead of how to do it.
- Declarative code is always going to end up either compiling down or being processed by something imperative like machine code right.
- In the case of something like react yes it abstracts away a lot of this complexity so that we as programmers don't have to do it but the react library itself and even functional languages like Lisp or Haskell eventually have to compile and do imperative things.
- But the idea is for us to go a step higher a level higher into declarative code so that it's easier to read and we can be more productive.

115. Immutability:

- https://medium.com/@dtinth/immutable-js-persistent-data-structures-and-structural-sharing-6d163fbd73d2
- The idea of immutability is not changing state instead making copies of the state and returning a new state every time.
- For Example:
- 

- Here clone function is pure, we're not doing anything here except cloning object.
- So, no matter how many times I call this function it's going to clone. It's a pure function.
- But afterwards I go ahead and do object.name and my program update the name to Nanna. Well this is mutating the state we're mutating data in our program.

- In functional programming. The idea of immutability is very important.
- If we want to change the name, we will create a function saying updateName that receives an object and clone it and make change in cloned object.

main.js

```
1   const obj = {name: 'Andrei'}
2   function clone(obj) {
3     return {...obj}; // this is pure
4   }
5
6   function updateName(obj) {
7     const obj2 = clone
8     (obj);
9     obj2.name = 'Nana'
10    return obj2
11  }
12
13  const updatedObj = updateName(obj)
14  console.log(obj, updatedObj)
```

```
Native Browser JavaScript
> 
{ name: 'Andrei' } { name: 'Nana' }
=> undefined
> 
```

-
-

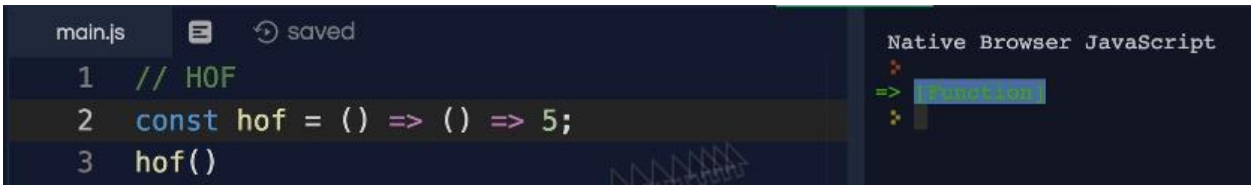
- **Structural Sharing:**

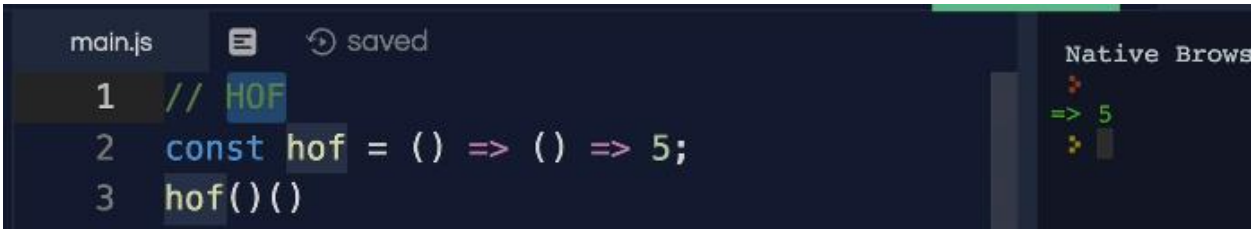
- Now you might be asking yourself for above program, this doesn't seem very memory efficient.
- I mean this is a silly example but if we're just copying things over and over every time, we want to make a change. Doesn't that just fill up our memory.
- Now this is a little bit beyond the scope of this course but there's something called structural sharing.
- The idea behind it is that when a new object or let's say an array or any sort of data structure is created, we don't copy everything. If it's a massive object or an array you can see that being very expensive.
- Instead of storing the entire copy underneath the hood what happens is that only the changes that were made to the state will be copied.
- But the things that don't change in memory are still there. And this is called structural sharing.

116. Higher Order Functions and Closures:

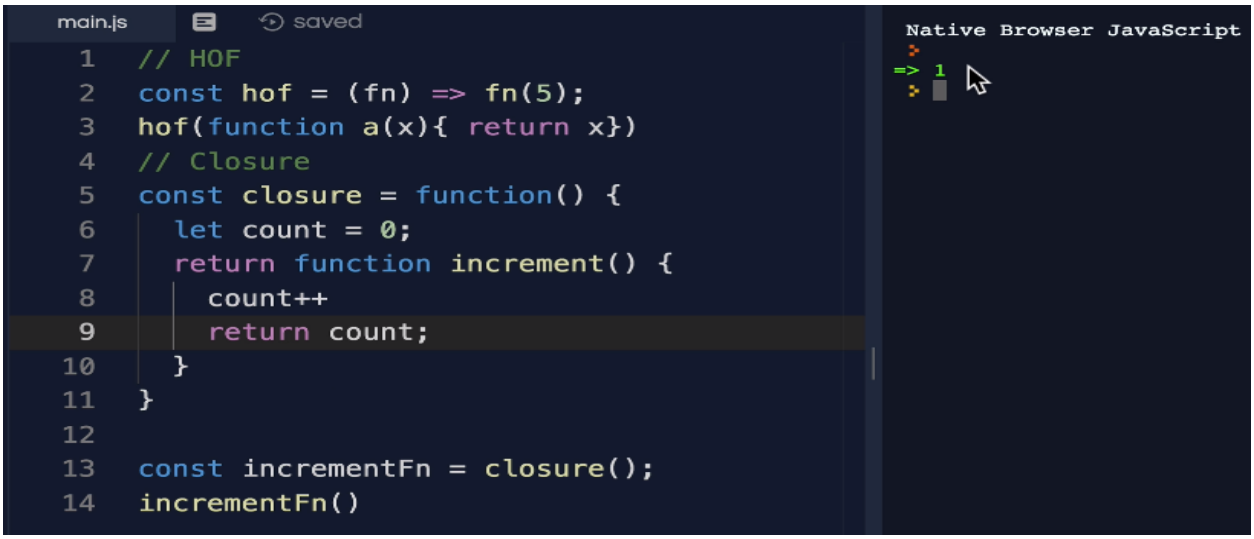
- Higher order functions both in mathematics as well as in computer science. simply means it's a function that does one of two things.
- It either takes one or more functions as arguments or returns a function as a result often called a callback.

- so, we can write a simple high order function such as:

- 

- You see that we get a function so it's a function that returns a function. If I call this again there you, I get 5. So that's a high order function.
- 

- **Below is also a higher order function:**
- 

- Closures in JavaScript are a mechanism for containing some sort of state and in JavaScript we create a closure whenever a function accesses a variable defined outside of the immediate function scope that is the scope of the parent.
- In closures also we used concept of higher order function:
- 

- If you have been following along with our functional programming concept one is that we're modifying state outside of our function.
- This increment function is touching state or data that belongs to another function. The closure function and when it comes to functional programming it doesn't mean we can't use closures.
- We can use closures and they're very powerful, but we have to be careful that closures only make a function impure.

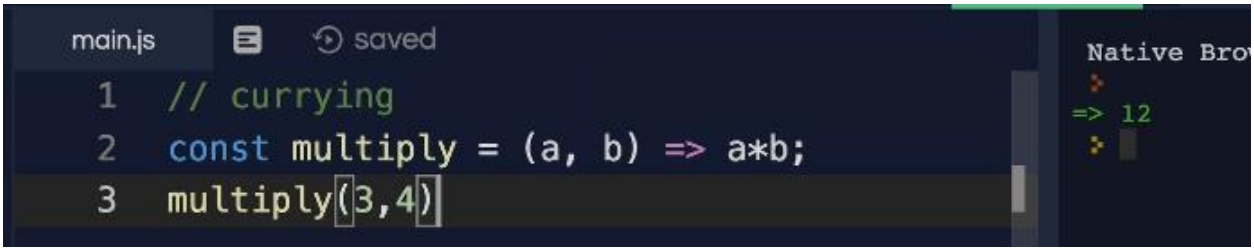
main.js saved

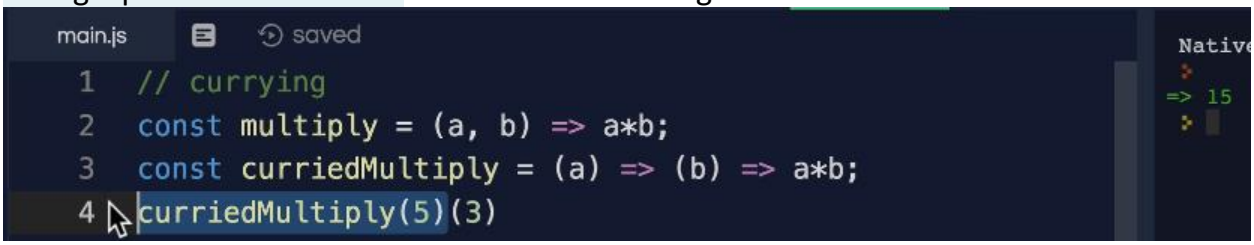
```
1   //HOF
2   const hof = (fn) => fn(5);
3   hof(function a(x){ return x})
4   //Closure
5   const closure = function() {
6     let count = 55;
7     return function getCounter() {
8       return count;
9     }
10  }
11
12  const getCounter = closure()
13  getCounter()
14  getCounter()
15  getCounter()
```

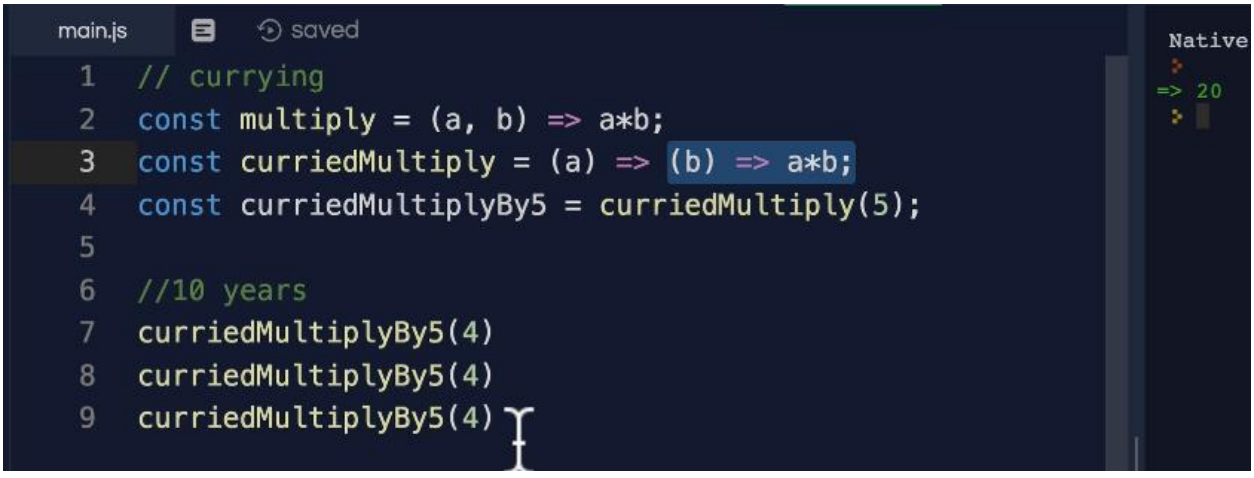
```
Native Brow
=> 55
```

- We're using closures here and although we're not modifying the state like we had before we still have access to data outside of ourselves.
- But if we don't modify it and mutate the data, we're still following the functional programming paradigm something that is nice with this is that we just created private variables. We can use closures to create data privacy which is very cool because now as a user I can't really modify the count.
- We're able to still have access to that variable but make sure that others don't modify it which is a very important tool and closures get used a lot in functional programming for this specific reason.

117. Currying:

- Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions each with a single argument.
- **Let's have a look at an example:**
- 

- **How can we use currying here?**
- Remember we want to change the function from taking multiple parameters to taking a parameter at a time we can do something like this.
- 

- So, I'm giving the function one parameter at a time but why is this useful?
- **I can now create multiple utility functions out of this**.
- 

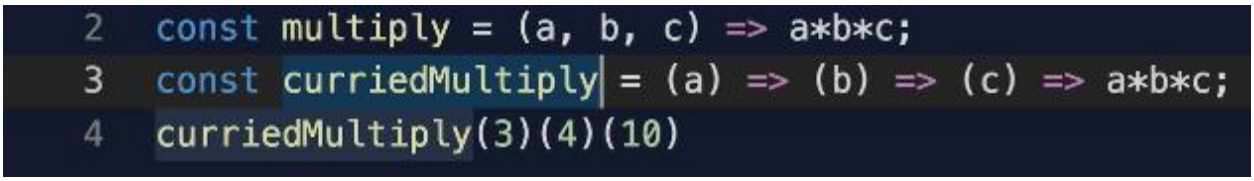
- **For example**, I can say curriedMultiplyByFive which is going to equal this curriedMultiply (5).

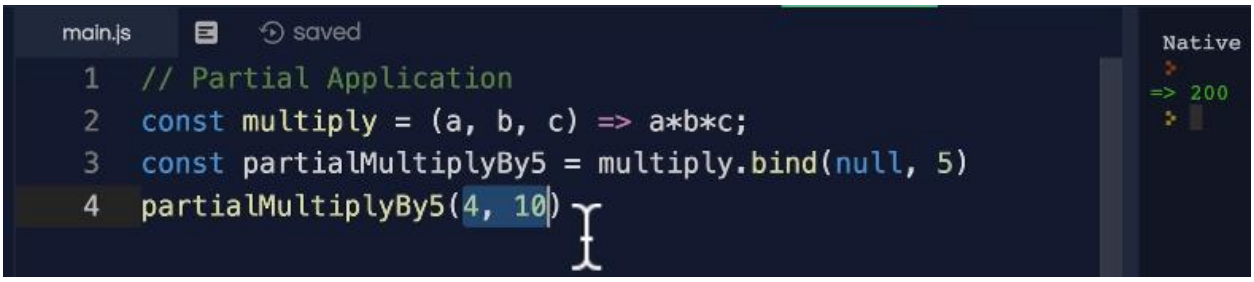
- So that now I've called this function once this function is going to remember this piece of data five for forever until we finish running the program.
- So that let's say 10 years from now we finally remember ohh we have this curriedMultiplyByFive function, we can use the curriedMultiplyByFive function and multiply anything that we want.

- Instead of running curriedMultiplyBy(5) function over and over I've run it once and now this curriedMultiplyBy(5) is there for us to use.
- So, if it's a function that gets called many times we only run this ((b) => a*b) part of the function.
- They're trying to save on memory or at least the amount that our computers have to work.

118. Partial Application:

- It is something similar yet slightly different to curry. Partial application is a way for us to well partially apply a function. It's a process of producing a function with a smaller number of parameters.
- Well it means taking a function applying some of its arguments into the function, so it remembers those parameters and then it uses closures to later be called with all the rest of the arguments.

- **Below is currying:**
- 

- Partial application says no what I want is to call the function once and then apply the rest of the arguments to it. So that means on the second call I expect all the arguments.
- 

- Main difference between currying and partial application is that partial application says on the second call I expect all the arguments, but currying says I expect one argument at a time.

122. Compose and Pipe

123. Arity

125. Solution: Amazon

127. Composition vs Inheritance

128. OOP vs FP

129. OOP vs FP 2