Intro To Javascript

Coder Academy

Day 1

- History of Javascript - From Browser to server
- What is node?
- Javascript vs Ruby
- JS in depth
  - Syntax Parser
  - Execution Context
  - Lexical Environment
  - Outer Environment
  - Hoisting
  - Scope Chain
  - this, null and undefined

History of Javascript

- JavaScript was created in 1995 by Brendan Eich while he was an engineer at Netscape
- It was originally going to be called LiveScript, but renamed based on popularity of Sun MicroSystem's Java
- Netscape submitted JavaScript to Ecma International, hence the name ECMAScript, 6th major editions was published in 2015 - JS version ES6
- ES6 vs ES5
- JS is not limited to browser anymore - frontend, backend, desktop apps, mobile apps, IOT
- JS has no concept of input or output - runs as a scripting language in the host environment and it is up to host environment to provide ways to communicate with outside world
- Host environment includes - browser, server side environment such Node.js
- Eg : prompt in browser = process.stdin.read() in server!

What is Javascript?

- JavaScript is a multi-paradigm, dynamic language, with types and operators standard built in objects and methods.

- JavaScript supports both object oriented programming and functional programming.

- In Js functions are objects, hence allowing functions to be passed around like any other objects, hence making functions first class citizens.

What is Node?

- Node is a server side javascript
- It has two sections:

  1. Acts as a backend similar to irb : Js code can be run in the terminal

     - node can also use the global objects in terminal, in the browser global object is window, in terminal if you jump into node and type global, you can access various parameters related to your computer
     - >process.env.USER // prints out your computer name

  2. Has inbuilt function to start server

```
var http = require("http");

http.createServer(function(request, response){
  response.end('Hello World \n');
}).listen(8000);

console.log(`server running at http://127.0.0.1:8000/ or localhost:8000:
${process.env.USER} `);
```

Js primitive types

Number

String

Boolean

Symbol

Object - Function, Array

Null - Lack of existence, set by the developer

Undefined - Lack of existence set by the browser

Type Coercion

while using operators: == operator coerces one of the operators if not same datatype

Hence use === which is strict equality and !== strict inequality

```
> Number(true)
< 1
> Number(false)
< 0
> Number(undefined)
< NaN
> Number(null)
< 0
> String(1)
< "1"
> String(true)
< "true"
> String(undefined)
< "undefined"
> "3" == 3
< true
> "3" === 3
< false
>
```

```
> false == 0
< true
> true == 1
< true
> '1' == 1
< true
> null == 0
< false
> '1' === 1
< false
> null < 1
< true
> "" == 0
< true
> " " == 0
< true
> "" === 0
< false
> null === 0
< false
>
```

Js - Objects

- Js Objects can be defined like a hash in ruby -> called **object literal** which is a key value pair

  let user = {name: “nandini” , age: 10};

- Object can be declared using **object constructor**

  let user = new Object();

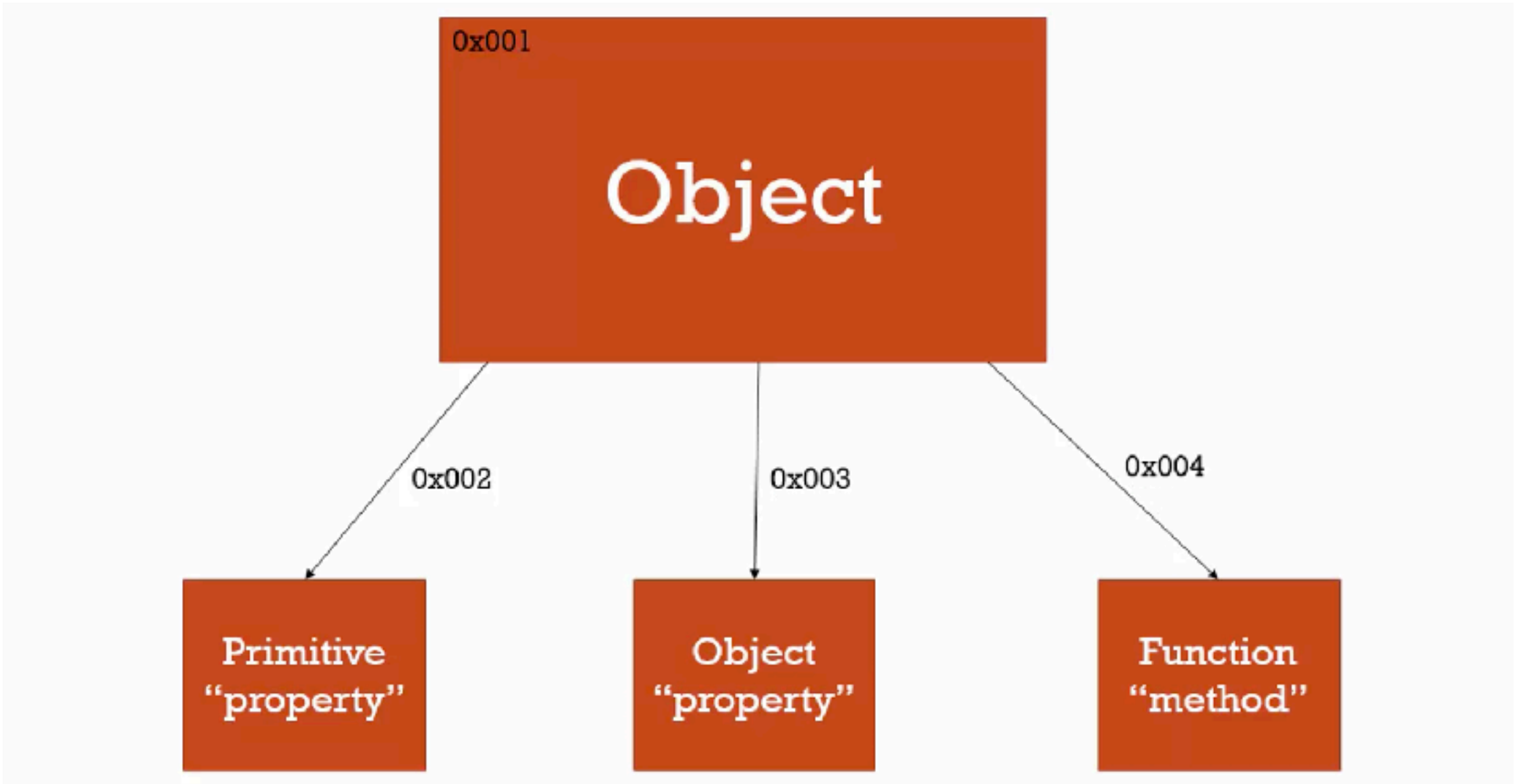
- However airbnb style guide recommends usage of object literal syntax over object constructor syntax.

- Square bracket and dot (.) notation
  - user.hasName = true;
  - User.has name = true; // throws an error, as symbols cannot have spaces hence cannot use dot notation
  - user[“has name”] = true; // works

- Object has several method constructors: click here

Js - Objects

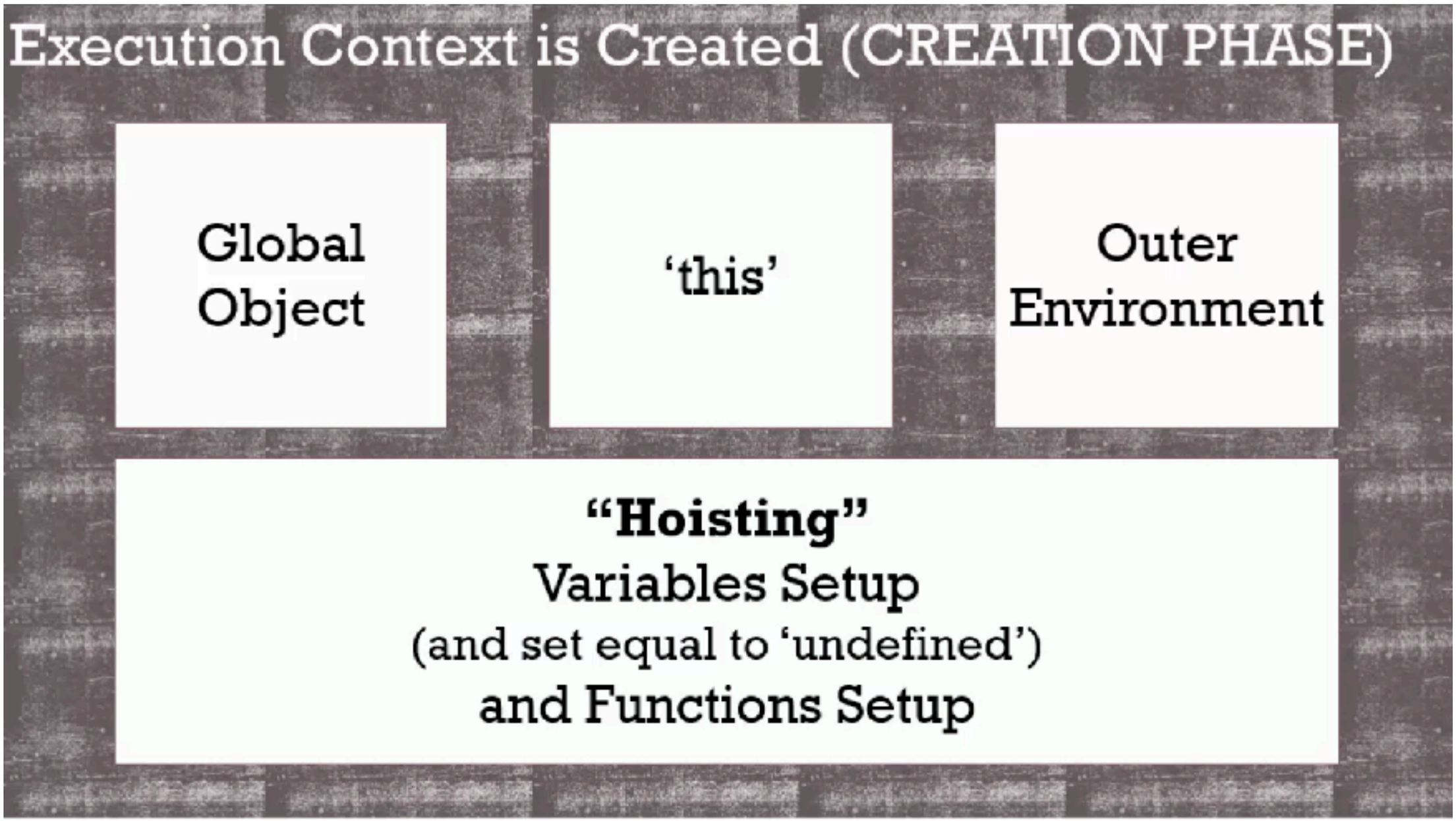
Object can have its value to be a primitive property, another object(key value pair) or even a function



Object Allow NameSpacing

```
1  var greet = 'Hello!';
2  var greet = 'Hola!';
3
4  console.log(greet);
5
6  var english = {};
7  var spanish = {};
8
9  english.greet = 'Hello!';
10 spanish.greet = 'Hola!';
```

Execution context



`this` concept in js

- `**this**` is a special variable created upon the creation of execution context and at the global level point to the global object - which is a **window object** in case of browser.
- Inside an object `**this**` variable refers to the object it resides in.
- ‘this’ keyword object in global functions refer to window object,
- ‘this’ keyword in methods(functions) in an object literal refers to corresponding object,

Js - Symbol

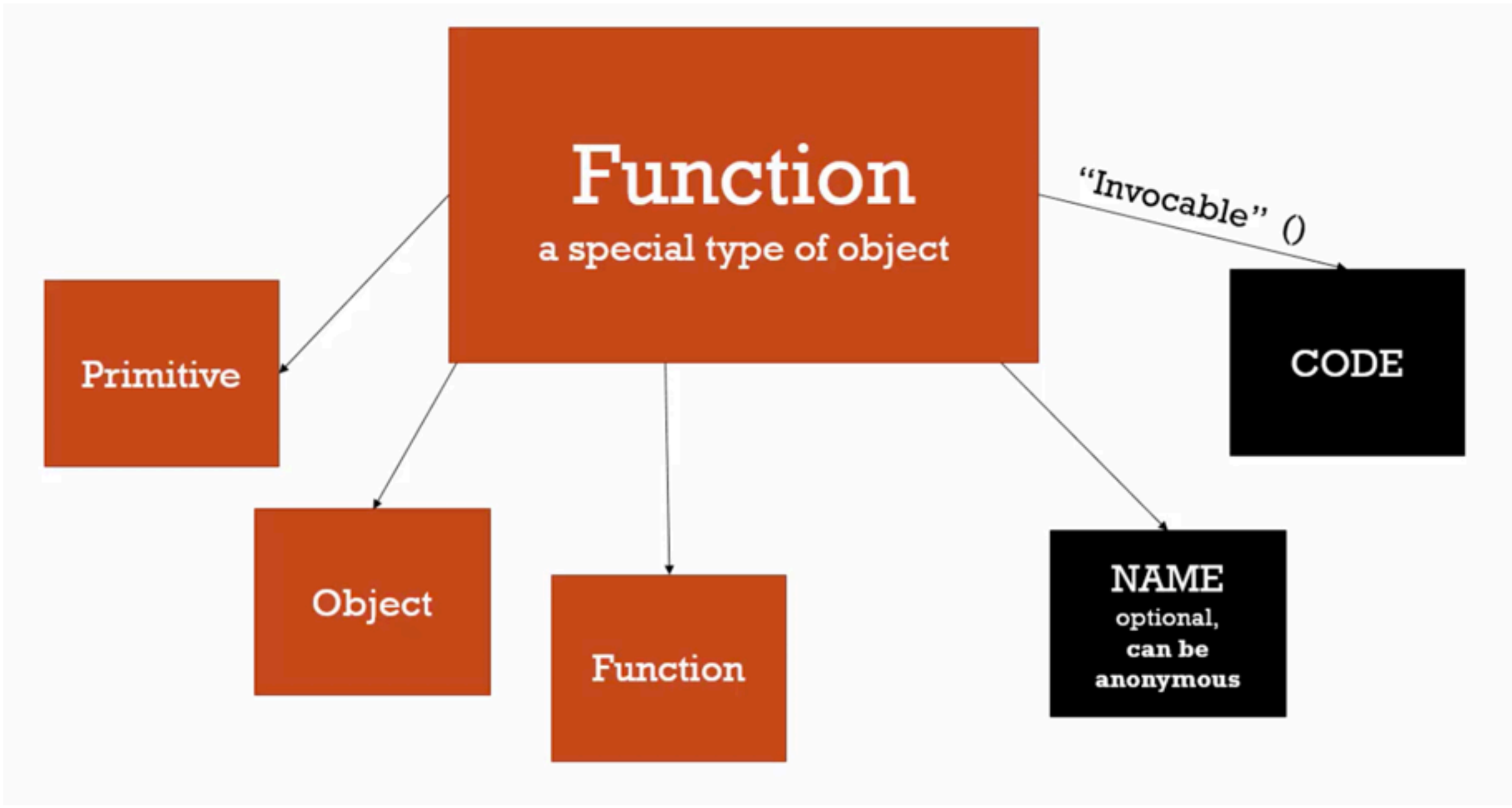
- Object property key may either be string type or of symbol type, cannot be a boolean or number.
- Symbols are guaranteed to be unique:
- https://javascript.info/symbol
- As Symbols are unique they reduce the risk of overwriting each other unlike strings

Js - Functions

- In javascript functions are objects

- Functions are first class citizens and can be used like any other object including
  - Assigned to a variable
  - Passed as an argument
  - Created on the fly
  - Returned from another function
  - Can also be part of an array

Functions are objects with an invocable code



Types of Functions

- Named function
- Anonymous function
- Arrow Function
- Immediately invoked function expression

Function Statement vs Function expression

- If the first token in the statement while parsing is a function then it is a function statement

    - Function statements are hoisted during syntax parsing

- If a function is assigned to a variable, it becomes a function expression

    - Function expressions are not hoisted, hence can be invoked only after declaration

- IIFE can be created and invoked on the fly, it is still and expression.

Function Prototype

- When a function is created in JavaScript, JavaScript engine adds a special property to the function called **prototype**

- This prototype property is an object (called as **prototype object**) which has a constructor (could be considered like an initialize in ruby Class) property by default

- The protoype object can be updated with some properties, which can be **accessed** by **all the objects created from this function**

- Note : In Js **instance method**

  can be created from function

```
function Human(firstName, lastName) {
  this.firstName = firstName,
  this.lastName = lastName,
  this.fullName = function() {
    return this.firstName + " " + this.lastName;
  }
}
var person = new Human("nandini", "nayak");
```

  Note : Both Js functions and classes contain prototype object

  Note: instance methods can access prototype properties but cannot update, only class or a function can update.

Execution Stack

```
function b() {
}

function a() {
      b();
}

a();
```

Js - Loops

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration

- for statement

- do..while statement

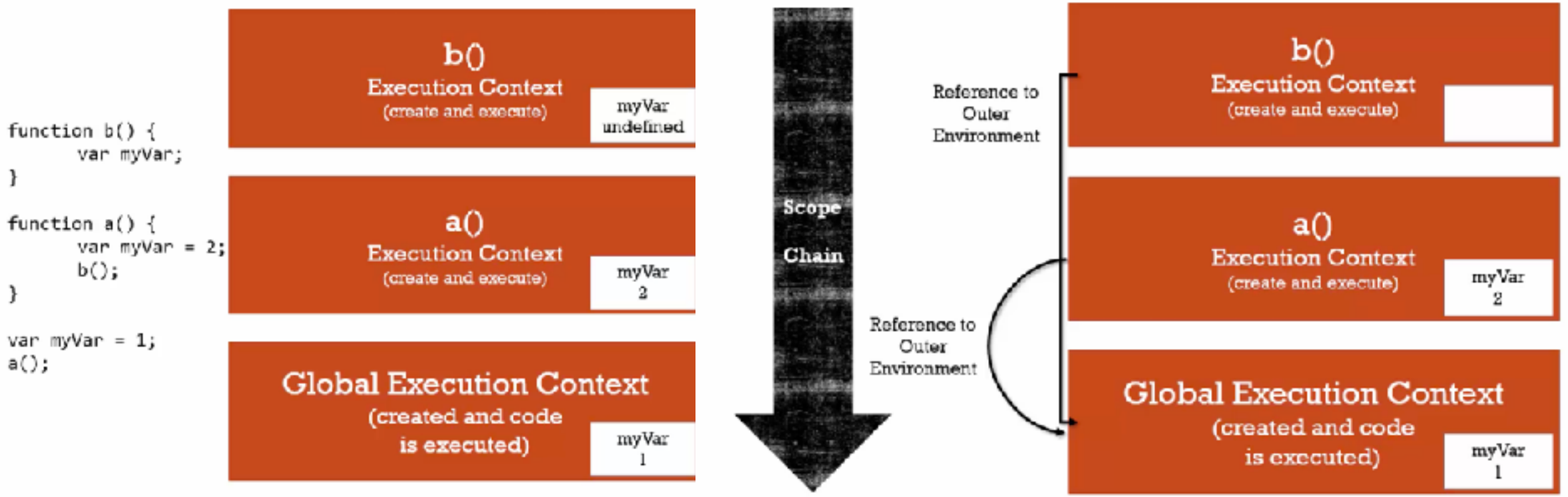
- while statement

- for..in statement. [iterates over property names]

- for .. of statement [iterates over property values] (equivalent to array.each in ruby)

Scope Chain

- Every function in the execution stack always has access to the variables in the outer environment(global).



Js - Class, Constructors, Prototypes

- Javascript Classes introduced in ECMAScript 2015, are primarily syntactic sugar over existing prototype based inheritance.

- The class syntax **does not** introduce a new object oriented model to javascript

- Important difference between class declaration and function is, functions are hoisted, but classes are not. Hence class can be used only after its creation.

- Just like functions, a Class can be defined as a class **declaration** and **expression**

- Main difference between class and function is the **static method**, which allows calling class methods directly without an instance

Summary

- **Constructor** method are like initialize in ruby, for creating and initialising an object created from class or a function

- Prototype are special objects in function and classes which allows us to add or update properties of the function/class, hence allowing all the instance methods to access newly added properties

- Major difference between class and functions are hoisting and availability of **static methods** in class

JSON

JSON is a great way of sending data over the internet. Huge amount of bandwidth is wasted in tag format, where you have opening and closing tags with same name.

```
1  var objectLiteral = {
2      firstname: 'Mary',
3      isAProgrammer: true
4  }
5
6  console.log(objectLiteral);
7
8  <object>
9      <firstname>Mary</firstname>
10     <isAProgrammer>true</isAProgrammer>
11 </object>
```

JSON vs ObjectLiteral

In JSON property has to be wrapped in quotes: unlike object literal syntax

```
1  var objectLiteral = {
2      firstname: 'Mary',
3      isAProgrammer: true
4  }
5
6  console.log(objectLiteral);
7
8  {
9      "firstname": "Mary",
10     "isAProgrammer": true
11 }
```

JSON methods

- JSON.parse() - converts string to Json

- JSON.stringify() - converts Json to string

Class - Mixins and extends

Callbacks

- In programming languages like c, ruby there is the expectation that whatever happens on line 1 will finish before the code on line 2 starts running and so on down the file, but Js is different.

- Callback functions are the functions that allows you to perform other tasks while waiting for a time-consuming task to be completed.

- Time consuming tasks could be like loading pictures, downloading files etc.

- Callback behaviour makes Js faster:

- Eg: A task similar to gets.chomp in ruby, which halts further execution unless an input is provided, on the contrary continues execution in case of Js.

```
// action similar to gets.chomp
console.log("What is your name.");
var name;
process.stdin.on('readable', function() {
    name = process.stdin.read();
    if (name !== null) {
        console.log(name);
        console.log(`Hello ${name} How are you`);
        console.log("Hello " + name + " How are you");
         // used to exit from the code
        // process.exit();
    }
});

// this could be a good example of callback function notice how the following
code continues, while waiting for the name to be entered
console.log("something else happens while waiting for the name to be entered,
due to callback function, hence not slowing up the process");

// also note how the timeout function  allows us to enter name, while waiting
for 3 seconds
setTimeout(function(){
   console.log("Hello after 3 seconds");
 }, 3000);
```

Asynchronous Non-Blocking

- Is Js synchronous (one task at a time) or Asynchronous(multiple tasks at a time) ??
    - **Js Engine is synchronous**
- If it is synchronous?? How does it handle external event such as click function?
    - Js handles async events(external trigger such as a click event) through a **event loop**, which executes **only after Js code is executed**
        - http://latentflip.com/loupe/?code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXIoKSB7CiAgICAgICAgY29uc29sZS5sb2coJ1lvdSBjbGlja2VkIHRoZSBidXR0b24hJyk7CiAgICAgIH0sIDAsICdmaWMjAwMCk7Cn0pOwoKY29uc29sZS5sb2coIkhpISIpOwoKc2V0VGltZW91dChmdW5jdGlvbiB0aW1lb3V0KCkgewogICAgY29uc29sZS5sb2coIkNsaWNrIHRoZSBidXR0b24hIik7Cn0sIDUwMDApOwoKY29uc29sZS5sb2coIldlbGNvbWUgdG8gbG91cGUuIik7!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D

Async event handling

```
<html>
    <head></head>
    <body>
        <script>
        // long running normal function
        function waitThreeSeconds() {
            var ms = 3000 + new Date().getTime();
            while (new Date() < ms){}
            console.log('normal timer function done: click event cannot be
            responded during a long running js function ');
        }
        function clickHandler() {
            console.log('click event!');
        }
        // listen for the click event
        document.addEventListener('click', clickHandler);
        waitThreeSeconds();  // normal function
        console.log('finished execution');

        // but if a wait timer is scheduled in a callback then click event gets
        executed while waiting for the timer callback to finish
        setTimeout(function timeout() {
          console.log("callback timer function done: click event can be
          responded while waiting for the callback to be completed");
        }, 3000);
        // run this code and click while long running function is executing:
        notice, click event gets registered only after finishing the js code.
        hence long runnning function cannot be interrupted, while events occur.
        // However a long running event created as a callback can be
        interrupted by event loop functions
        // This is how js deals with async events, anything happening ouside of
        JE in browser is stored in Event que in JE
        </script>
    </body>
</html>
```

Event loop with a restaurant analogy

- https://www.youtube.com/watch?v=s9Zy8ISjxIw

Execution Stack
Priority 1

Event loop(keeps track of external events such as click and callback functions)
Priority 2

Callback hell

- http://callbackhell.com/

- **Don’t nest functions**, give them names and place them at top level of the program

- Handle **every single error** in every one of your callbacks

    - With callbacks the most popular way to handle errors is to reserve first argument of callback for an error

- Create reusable functions and place them in a module

Promise

- The promise object is used for deferred and asynchronous computation to handle callback hell via error handling

- It represents an operation that hasn’t completed yet, but is expected in the future

- It has 3 states

    - Pending - Promise starts out in pending state, where one can retrieve a value promise is holding by calling **then** method.

    - Fulfilled - If successfully retrieved moves to fulfilled state

    - Rejected - if failure occurs, will be in rejected state.

Promise resolve-reject/then-catch

- A function that returns a promise object runs resolve method if successful or a reject method upon failure.

- Resolve method is executed by then method

- Reject method is executed by catch method

- **Chaining** - .then() isn't the end of the story, you can chain then’s together to transform values or run additional async actions one after another.

A promise returning function

```
function waitUpToThreeSeconds(seconds){
  return new Promise((reject,resolve) => {
      console.log(`I will wait for 3 seconds, you asked me to wait for
      ${seconds} seconds`)

    setTimeout(
      () => {
          reject('3 seconds are up you asked me to wait to long')
      },
      3000
    )

    setTimeout(
      () => {
        resolve (`${seconds}seconds are finished!`)
      },
      seconds*1000
    )

  })
}
waitUpToThreeSeconds(2).then(successMessage => {
  console.log(successMessage)
})
.catch(errorMessage => {
  console.log(errorMessage)
})

waitUpToThreeSeconds(4).then(successMessage => {
  console.log(successMessage)
})
.catch(errorMessage => {
  console.log(errorMessage)
})
```

DOM

- JavaScript is a language that the browser reads and does stuff with. But the DOM is where that stuff happens.

- The Document Object Model (DOM) is an application programming interface (API) for valid HTML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

- DOM is a Dynamic HTML code that exists in the dev tool.

- In the browser the object, JS have access to manipulate html content is DOM (document)

Events

- https://developer.mozilla.org/en-US/docs/Web/API/EventListener

- A very useful function - addEventListener accepts two arguments - 1) **Event type** 2) callback function

- addEvenListener accepts a **variety of event types** ranging from **click**-**scroll**-**keypress**-**mouse hover** etc.

- Note: For **forms** the event is **submit** : equivalent to enter!

Web APIs

- https://developer.mozilla.org/en-US/docs/Web/API

- Note: WebApIs for Js works only on browser not on node.js.

- Node.js have there own set of APIs

- https://nodejs.org/api/