Netcracker
An NEC Company

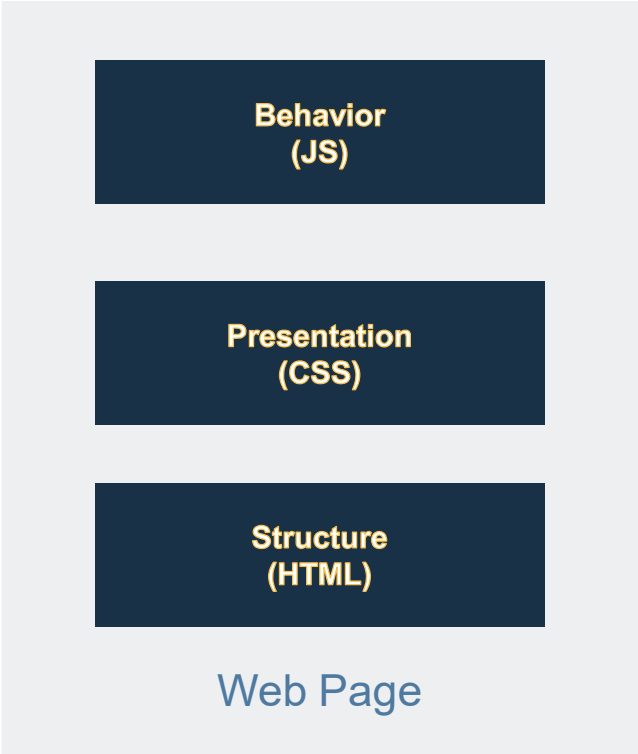
JavaScript (JS) - Basics

Introduction and Training

Introduction

- JavaScript (JS) is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions.
- While it is most well-known as the scripting language for Web pages, many non-browser environments - Node.js, Apache CouchDB and Adobe Acrobat.
- JavaScript is a prototype-based, multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles.
- Java script allows you to detect just about anything that takes place inside your web browser and you can learn to write code based upon these user interactions.

JS in Web Design



HTML is the markup language that we use to structure and give meaning to our web content, for example defining paragraphs, headings, and data tables, or embedding images and videos in the page.

CSS is a language of style rules that we use to apply styling to our HTML content, for example setting background colors and fonts, and laying out our content in multiple columns.

JavaScript is a scripting language that enables you to create dynamically updating content, control multimedia, animate images, and pretty much everything else.

JS_Intro.html

Basics of JS – Part 1

- Comments
- Declaring variables and functions
- Data Types –
  1. Primary Data-types
     a) String
     b) Number
     c) Boolean
  2. Composite Data-types
     a) Object
     b) Array
     c) Date
  3. Special Data-types
     a) null
     b) undefined

Basics of JS – Part 2

- Conditional Statements
- Loops
- Operator Precedence –
  - [] ( ) Refinement and invocation
  - delete new typeof + - ! Unary operators
  - * / % Multiplication, division, modulo
  - + - Addition/concatenation, subtraction
  - >= <= > < Inequality
  - === !== Equality
  - && Logical and
  - || Logical or
  - ?: Ternary

Basics of JS – Part 3

- Composite Data Types
  1. Arrays
  2. Objects
  3. Dates

jsArray.js
jsObject.js

Scopes and Closures

Lexical Scope

- Set of well-defined set of rules for storing variables in some location, and for finding those variables at a later time is called *Scope*.
- Consider the following expression

```
var a = 2;
```

1. On Encountering `var a`, Compiler asks Scope to see if `a` exists for this particular scope collection. If so compile ignores the declaration and moves on. Otherwise, Compiler asks Scope to declare a new variable called `a` for that scope collection.
2. Compiler produces code to Engine to later handle `a = 2` assignment. The code Engine asks Scope if there is a variable called `a` accessible in current scope collection. If so Engine uses it or looks elsewhere. Finally if it is unable to find it, it throws out an Error.

- **Nested Scope**: We mentioned Scope is a set of rules for looking up variables, but there is usually more than one Scope to consider. If a variable is not found in the current Scope collection continue looking until found or outermost/global scope has been reached.

- Scope for any variables is defined during lexing process. Thus the name Lexical Scope.
- In simpler words lexical scope is based on where variables and blocks of scope are authored at write time and thus freezed by the time lexer processes your code.

NestedScope.js

Scopes and Closures

Shadowing

- Scope look-ups stop once they find the first match. Same variable name can be specified at multiple layers of nested scope aka shadowing. (inner variable shadows outer variable). Regardless of shadowing, scope lookup always starts at innermost scope being executed at that time, and works its way outwards until first match.

- *Note: Global variables are automatic properties of global object (window for browsers) so it is possible to reference a global variable as a property reference to the global object.

- non-global shadow variables cannot be accessed. No matter where the function is invoked from even how it is invoked, its lexical scope is only defined by where the function was declared.

Shadow,js

Scopes and Closures

Function vs. Block Scope

- Declared vs. Undeclared variables
- Variables declared with a `**var**` keyword are functional Scope.
- `**let**` keyword attaches the variable declaration to the scope of whatever block (commonly a { .. } pair) it's contained in.
- Variables created by `**const**` keyword are block-scoped variable, but whose value is fixed (constant). Any attempt to change that value at a later time results in an error.

DeclaredVsUnDeclaredVar.js
functionalScope.js
blockScope.js

Scopes and Closures

Hoisting

- All declarations, both variables and functions, are processed first, before any part of your code is executed.
- When you see `var a = 2;`, you probably think of that as one statement. But JavaScript actually thinks of it as two statements: `var a;` and `a = 2;.` The first statement, the declaration, is processed during the compilation phase. The second statement, the assignment, is left in place for the execution phase.
- variable and function declarations are "moved" from where they appear in the flow of the code to the top of the code. This gives rise to the name "Hoisting".
- Both function declarations and variable declarations are hoisted, functions are hoisted first, and then variables.
- While multiple/duplicate var declarations are effectively ignored, subsequent function declarations do override previous ones.

Hoisting.js

Scopes and Closures

Closure and Modules

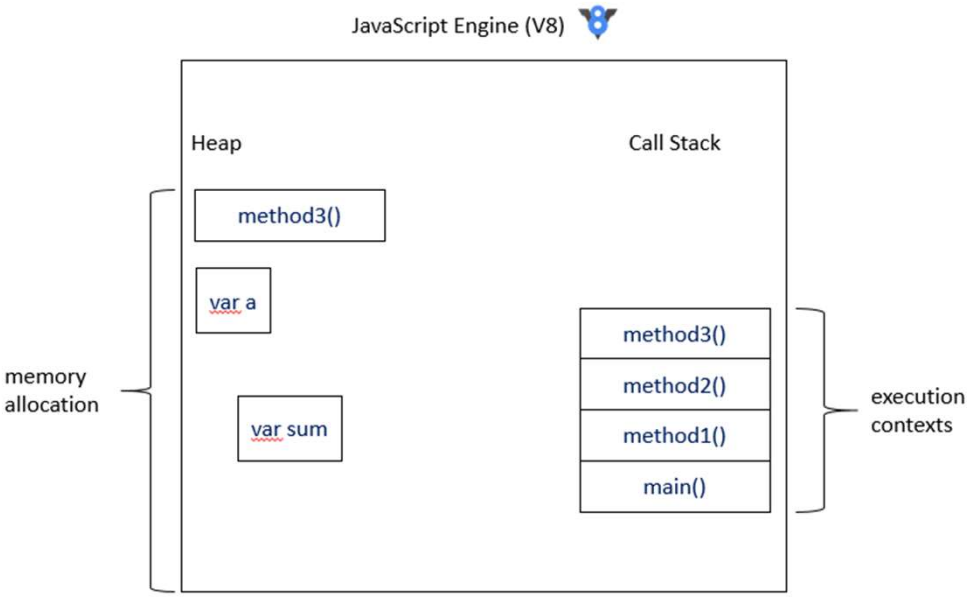
- Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

- Modules leverages the power of closures
  1. There must be an outer enclosing function, and it must be invoked at least once (each time creates a new module instance).
  2. The enclosing function must return back at least one inner function, so that this inner function has closure over the private scope, and can access and/or modify that private state.

Clouser1.js
Modules1.js

Event Loop

JS Engine/ Runtime



- JS Browser consists of heap, call stack, event loop, a callback queue and some other helper APIs.
- JS Engine consists of heap and call stack.
- Lets take a closure look at Java-script Engine. The most popular JS engine is Chrome's V8 Engine. It looks something like this.

Event Loop

Blocking

- What Happens when things are slow?
- Sync Http request problem.
- Async callbacks & call stack

This Keyword in Java Script

Common misconceptions of *this*

This is not represented by any of the following:

1. Itself
2. Its Scope

this is not an author-time binding but a runtime binding. It is contextual based on the conditions of the function's invocation. this binding has nothing to do with where a function is declared, but has instead everything to do with the manner in which the function is called.

this1.js
this2.js

This Keyword in Java Script

Default Binding

- Applies to standalone function invocation.
- default catch-all rule when none of the other rules apply.

*Note :If ***strict mode*** is in effect, the global object is not eligible for the default binding, so the this is instead set to *undefined*

defaultBinding.js

This Keyword in Java Script

Implicit Binding

- This rule applies if the function call has a owning object or a containing object.
- In this case this keyword points to that object.

This Keyword in Java Script

Explicit Binding

- You will use this type of binding if you want to force a function call to use a particular object as a reference to *this* keyword.
- All functions in JS have prototype methods *call(…)* and *apply(…)* . They both take, as their first parameter, an object to use for the *this.*
- Since you are directly stating what you want the *this* to be, we call it “explicit binding”.

explicitBinding.js

This Keyword in Java Script

new Binding

- In JS, constructors are just functions that happen to be called with the *new* operator in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions with the *new* keyword added before calling them.
- Adding *new* in front of it, and that makes that function call a constructor call. In JS there is no such thing as "constructor functions", but rather construction calls of functions.
- When a function is invoked with new in front of it, otherwise known as a constructor call, the following things are done automatically:
    1. a brand new object is created (aka, constructed) out of thin air
    2. the newly constructed object is [[Prototype]]-linked (Read Object Prototype)
    3. the newly constructed object is set as the this binding for that function call
    4. unless the function returns its own alternate object, the new-invoked function call will automatically return the newly constructed object.

newBinding.js

Objects in JavaScript

Declarative (literal) form

```
var myObj = {
    key: value
    // ...
};
```

Constructed form

```
var myObj = new Object();
  myObj.key = value;
```

*Note: It's extremely uncommon to use the "constructed form" for creating objects as just shown. You would pretty much always want to use the literal syntax form.

Copying Objects in JavaScript

- Shallow copy – use Object.assign({}, obj);

- Deep copy –
  - In case of JSON safe objects (ones that don't have circular reference) use
    JSON.parse(JSON.stringify(obj))

  - In case of Circular JS Objects you will need to programmatically iterate through the object keys and clone individual properties. Doing a deep copy will ultimately lead you to go over an infinite loop.

clone.js
objectAssign.js

Object Property Descriptors

```
Object.getOwnPropertyDescriptor( myObject, "a" );
    // {
    //    value: 2,
    //    writable: true,
    //    enumerable: true,
    //    configurable: true
    // }
```

- the property descriptor (called a "data descriptor" since it's only for holding a data value) for our normal object property a is much more than just its value of 2. It includes 3 other characteristics: writable, enumerable, and configurable.

dataDescriptor.js

Object Property Descriptors

Name	Description
Writable	The ability for you to change the value of a property is controlled by writable
Configurable	As long as a property is currently configurable, we can modify its descriptor definition, using the same defineProperty(..) utility.
Enumerable	this characteristic controls if a property will show up in certain object-property enumerations, such as the for..in loop.
Getters & Setters	The default [[Put]] and [[Get]] operations for objects completely control how values are set to existing or new properties, or retrieved from existing properties, respectively.

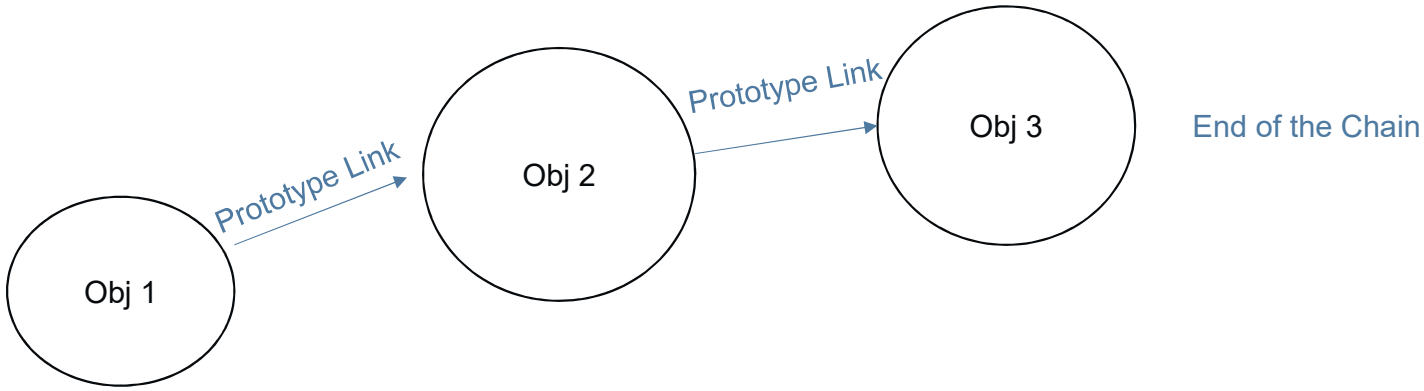
Prototypes

- Objects in JavaScript have an internal property, denoted in the specification as [[Prototype]], which is simply a reference to another object. Almost all objects are given a non-null value for this property, at the time of their creation.
- This is the primary way Objects in JavaScript inherit structure and behavior from other objects
- If a property/method is not found on the current Object. JS Engine will parse through the prototype link of the object until it finds the property/method or the chain ends

proto1.js

Prototypes

- The top-end of every normal [[Prototype]] chain is the built-in Object.prototype. This object includes a variety of common utilities used all over JS, because all normal (built-in, not host-specific extension) objects in JavaScript "descend from" (aka, have at the top of their [[Prototype]] chain) the Object.prototype object.



Prototypal Inheritance

- In JavaScript, we don't make *copies* from one object ("class") to another ("instance"). We make *links* between objects. aka - prototypal inheritance.

- these relationships are not *copies* like traditional Inheritance but delegation links.

proto2.js