Netcracker
An NEC Company

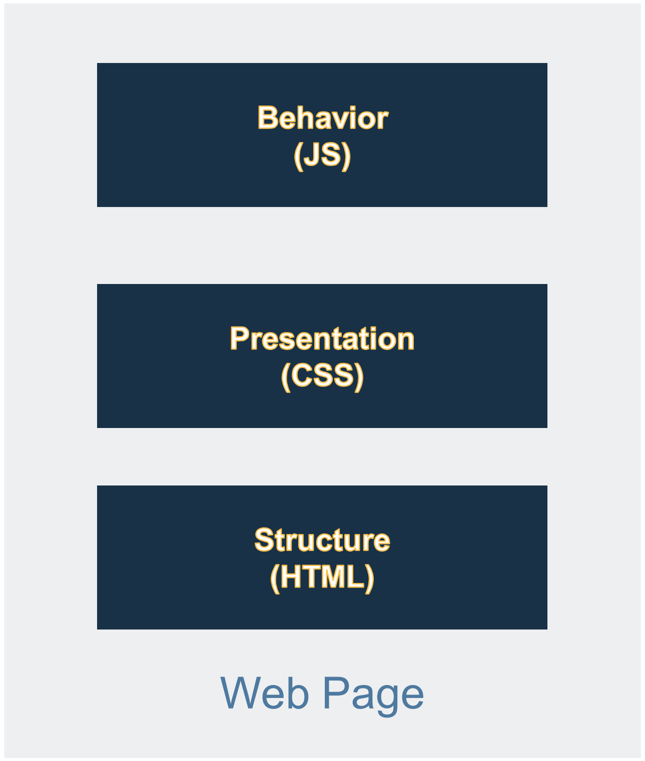
JavaScript (JS) - Basics

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 ot 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