LESSON 5
JAVASCRIPT FOR MODERN WEB APPS

Actions Supported by All the Laws of Nature

Maharishi University of Management -Fairfield, Iowa © 2023

Key JavaScript Concepts

- Key JavaScript Concepts
- JavaScript Syntax
- Program Logic
- Advanced JavaScript Syntax

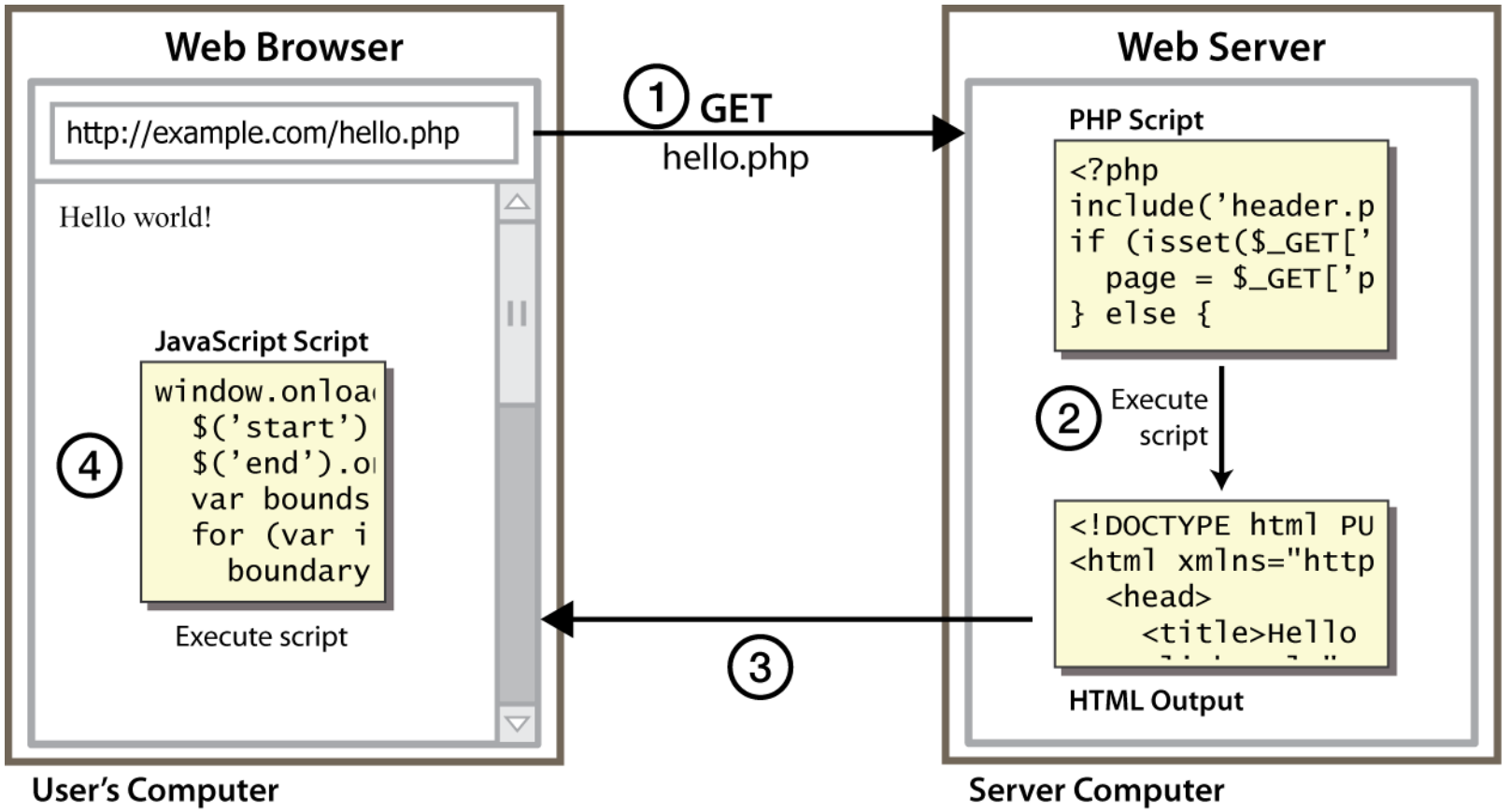
Main Point Preview

- JavaScript is a loosely typed language. It has types, but does no compile time type checking. Programmers must be cautious of automatic type conversions, including conversions to Boolean types. It has a flexible and powerful array type as well as distinct types of null and undefined.

- **Science of Consciousness:** To be an effective JavaScript programmer one needs to understand the principles and details of the language. If our awareness is established in the source of all the laws of nature then our actions will spontaneously be in accord with the laws of nature for a particular environment.

Client-side Scripting

- client-side script: code runs in browser *after* page is sent back from server
  - often this code manipulates the page or responds to user actions



Why use client-side programming?

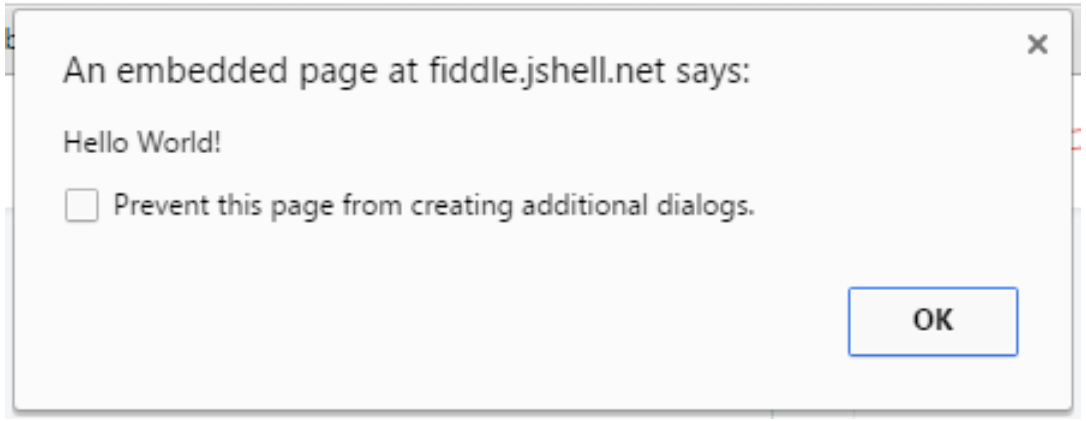
- client-side scripting (JavaScript) benefits:
  - usability: can modify a page without having to post back to the server (faster UI)
  - efficiency: can make small, quick changes to page without waiting for server
  - event-driven: can respond to user actions like clicks and key presses

What is JavaScript?

- a lightweight programming language ("scripting language")
- used to make web pages interactive
  - insert dynamic text into HTML (ex: user name)
  - react to events (ex: page load user click)
  - get information about a user's computer (ex: browser type)
  - perform calculations on user's computer (ex: form validation)
- a web standard (but not supported identically by all browsers)
- NOT related to Java other than by name and some syntactic similarities

A JavaScript statement: alert

```
alert("Hello World!");
```



- A JS command that pop up a dialog box with a message

Block-scoped variable (ES6)

- ```
  let numArr= [];
  for (let i = 0; i <= 3; i++) {
   numArr[i] = i * 2;
  }

  console.log(numArr[0]);
  console.log(numArr[1]);
  console.log(numArr[2]);
  ```
  ```

Variables and types

```
var name = expression;
let name = expression; (ES6)
const name = expression; (ES6)

var age = 32;
let weight = 127.4;
const clientName = "Connie Client";
```

- variables are declared with the `var/let/const` keyword (case sensitive)
- types are not specified, but JS does have types ("loosely typed")
  - Number, Boolean, String, Null, Undefined, Symbol, Object
  - can find out a variable's type by calling typeof

Constants (ES6)

- also known as "immutable variables"
  - cannot be re-assigned new content.
- only makes the variable itself immutable, not its assigned content
  - object properties can be altered
  - array elements can be altered

- ```
  const PI = 3.1415926;
  pi = 3.14; //error
  const point = {x:1, y: 10};
  point.y = 20; //ok
  ```
  ```

Number Type

```
let enrollment = 99;
let medianGrade = 2.8;
let credits = 5 + 4 + (2 * 3);
```

- integers and real numbers are the same type (no int vs. double)
- same operators: + - * / % ++ -- = += -= *= /= %=
- similar precedence to Java
- many operators auto-convert types: "2" * 3 is 6
  - What is “2” + 3 ?

Logical Operators

- >, <, >=, <=, &&, ||, ! ==, !=, ===, *!*==
- most logical operators automatically convert types:
  - 5 < "7" is true
  - 42 == 42.0 is true
  - "5.0" == 5 is true
- === and !== are **strict** equality tests; checks both type and value
  - "5.0" === 5 is false
- Always use **strict** equality

Comments (same as Java)

```
// single-line comment
/* multi-line comment */
```

- identical to Java's comment syntax
- recall: 4 comment syntaxes
  - HTML: <!-- *comment* -->
  - CSS/JS/PHP: /* *comment* */
  - Java/JS/PHP: // *comment*
  - Python: # *comment*

Boolean Type

```
let iLikeWebApps = true;
let ieIsGood = "IE6" > 0; // false
if ("web dev is great") { /* true */ }
if (0) { /* false */ }
```

- any value can be used as a Boolean
  - "falsey" values: false, 0, 0.0, NaN, empty String(""), null, and undefined
  - "truthy" values: anything else, include objects
- !! Idiom – gives boolean value of any variable
  - const x=5;
  - console.log(!x);
  - console.log(x);
  - console.log(!!x);

String Type

```
let s = "Connie Client";
let fName = s.substring(0, s.indexOf(" ")); // "Connie"
let len = s.length; // 13
let s2 = 'Melvin Merchant'; // can use "" or ' '
```

- methods: charAt, charCodeAt, fromCharCode, indexOf, lastIndexOf, replace, split, substring, toLowerCase, toUpperCase
  - charAt returns a one-letter String (there is no char type)
- length property (not a method as in Java)
- concatenation with + : 1 + 1 is 2, but "1" + 1 is "11"

Special values: null and undefined

```
let ned = null;
const benson = 9;
let caroline;
// at this point in the code,
// ned is null
// benson's 9
// caroline is undefined
```

- `undefined` : has been declared, but no value assigned
  - e.g., `caroline var` above
  - vars that are "hoisted" to beginning of a function
- `null`: exists, and was specifically assigned an value of `null`
- reference error when try to evaluate a variable that has not been declared
  - reference error different from undefined
  - undefined means declared, but no value assigned

Creating objects via object literal

```
const name = {
  'fieldName': value,
  ...
  'fieldName': value
};
const pt = {
  'x': 4,
  'y': 3
};
alert(pt.x + ", " + pt.y);
```

- in JavaScript, you can create a new object without creating a class
- the above is like a Point object; it has fields named x and y
- the object does not belong to any class; it is the only one of its kind, a singleton
  - typeof(pt) === "object"

JavaScript objects

- objects in JavaScript are like associative arrays
- the keys can be any string
- you do not need quotes if the key is a valid JavaScript identifier
- values can be anything, including functions
- you can add keys dynamically using associative array or the . syntax
- object properties that have functions as their value are called ‘methods’

```
const x = {
  'a': 97,
  'b': 98,
  'c': 99,
  'd': 199,
  'mult': function(a, b)
  {
       return a * b;
  }
};
```

if/else statement (same as Java)

```
if (condition) {
  statements;
} else if (condition) {
  statements;
} else {
  statements;
}
```

- Identical structure to Java's if/else statement
- JavaScript allows almost anything as a *condition*

for loop (same as Java)

```
for (initialization; condition; update){
 statements;
}

let sum = 0;
for (let i = 0; i < 100; i++) {
 sum = sum + i;
}

const s1 = "hello";
let s2 = "";
for (let i = 0; i < s.length; i++) {
 s2 += s1[i] + s1[i];
} // s2 stores "hheellllloo"
```

while loops (same as Java)

```
while (condition) {
  statements;
}

do {
  statements;
} while (condition);
```

- break and continue keywords also behave as in Java

Function Declaration

```
function name() {
 statement ;
 statement ;
 ...
 statement ;
}

function square(number) {
 return number*number;
}
```

- declarations are "hoisted" (vs function expressions) – see Lecture07
  - They can be declared anywhere in a file, and used before the declaration.

Function Expressions

- Can be Anonymous function
  - Widely used in JS with event handlers

```
const square = function(number) { return number * number };
const x = square(4) // x gets the value 16
```

- Can also have a name to be used inside the function to refer to itself //NFE (Named Function Expression)

```
const factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1)
};
console.log(factorial(3));
```

- Basically, a function expression is same syntax as a declaration, just used where an expression is expected

Anonymous functions

- JavaScript allows you to declare anonymous functions
- Can be stored as a variable, attached as an event handler, etc.
- Keeping unnecessary names out of namespace for performance and safety

```
window.onload = function(){
  alert("Hello World!");
}
```

Arrow functions (ES6)

- Arrow functions can be a shorthand for an anonymous function in callbacks

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
          // equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter:
(singleParam) => { statements }
singleParam => { statements }

// A function with no parameters requires parentheses:
() => { statements }
```

Arrow Functions Example

```
function multiply(num1, num2) {
 return num1 * num2;
}
var output = multiply(5, 5);
console.log("output: " + output);

var multiply2 = (num1, num2) => num1 * num2;
var output2 = multiply2(5, 5);
console.log("output2: " + output2);
```

Arrays

```
let name = []; // empty array
let name = [value, value, ..., value]; // pre-filled
name[index] = value; // store element

let ducks = ["Huey", "Dewey", "Louie"];
let stooges = []; // stooges.length is 0
stooges[0] = "Larry"; // stooges.length is 1
stooges[1] = "Moe"; // stooges.length is 2
stooges[4] = "Curly"; // stooges.length is 5
stooges[4] = "Shemp"; // stooges.length is 5
```

- two ways to initialize an array – see example for another two ways
- length property (grows as needed when elements are added)

Array methods

```
let a = ["Stef", "Jason"]; // Stef, Jason
a.push("Brian"); // Stef, Jason, Brian
a.unshift("Kelly"); // Kelly, Stef, Jason, Brian
a.pop(); // Kelly, Stef, Jason
a.shift(); // Stef, Jason
a.sort(); // Jason, Stef
```

- array serves as many data structures: list, queue, stack, ...
- methods: concat, join, pop, push, reverse, shift, slice, sort, splice, toString, unshift
  - push and pop add / remove from back
  - unshift and shift add / remove from front
  - shift and pop return the element that is removed

Arrow Functions Example – filter

Returns an Array containing all the array elements that pass the test. If no elements pass the test it returns an empty array.

```
const a = [
 "Hydrogen",
 "Helium",
 "Lithium",
 "Beryllium"
];
const a2 = a.filter(function(s) {
            return s.length > 7 });
const a3 = a.filter( s => s.length > 7 );

const a4 = a.find( s => s.length > 7 );
const a5 = a.findIndex( s => s.length > 7 );
```

Arrow Functions Example - map

```
var a = [
 "Hydrogen",
 "Helium",
 "Lithium",
 "Beryllium"
];
var a2 = a.map(function(s) { return s.length });
console.log("a2: " + a2);

var a3 = a.map(s => s.length);
console.log("a3: " + a3);
```

Arrow Functions Example - reduce

- executes a **reducer** function (that you provide) on each member of the array resulting in a single output value.
- returned value is assigned to the accumulator, whose value is remembered across each iteration throughout the array and ultimately becomes the final, single resulting value.
- initialValue: optional value to use as the first argument to the first call of the callback. If no initial value is supplied, the first element in the array will be used.

```
function calc(multiplier, base, ...numbers) {
 var temp = numbers.reduce((accum, num) => accum + num, base);
 return multiplier * temp;
}
var total = calc(2, 6, 10, 9, 8);
console.log("total: " + total);
```

Main Point

- JavaScript is a loosely typed language. It has types, but does no compile time type checking. Programmers must be cautious of automatic type conversions, including conversions to Boolean types. It has a flexible and powerful array type as well as distinct types of null and undefined.

- **Science of Consciousness:** To be an effective JavaScript programmer one needs to understand the principles and details of the language. If our awareness is established in the source of all the laws of nature then our actions will spontaneously be in accord with the laws of nature for a particular environment.

Main Point Preview

Functional programming methods map, filter, reduce make code more understandable and error free by automating details of general-purpose looping mechanisms, indicating their intent by their name, and not changing the state of the original array.

Science of Consciousness: Growth of consciousness makes behavior simpler and more error free because intentions that arise from deep levels are spontaneously in accord with and supported by all the laws of nature.

Good things to know

- No function overloading in JavaScript
  - Extra arguments ignored and missing ones ignored
- Arguments object and Rest (ES6) parameters
- Arrow functions(ES6)
  - Syntactic sugar for anonymous functions (ala Java lambdas)
- Functional programming via map, filter, reduce

Function Signature

- If a function is called with missing arguments(less than declared), the missing values are set to : `undefined`

```
function f(x) {
 console.log("x: " + x);
}
f();
f(1);
f(2, 3);
```

No overloading!

```
function log() {
 console.log("No Arguments");
}
function log(x) {
 console.log("1 Argument: " + x);
}
function log(x, y) {
 console.log("2 Arguments: " + x + ", " + y);
}
log();
log(5);
log(6, 7);
```

- Why? Functions are objects!

`arguments` Object

The **arguments** object is an Array-like object corresponding to the arguments passed to a function.

```
function findMax() {
 var i;
 var max = -Infinity;
 for (i = 0; i < arguments.length; i++) {
   if (arguments[i] > max) {
   max = arguments[i];
   }
 }
 return max;
}
var max1 = findMax(1, 123, 500, 115, 66, 88);
var max2 = findMax(3, 6, 8);
```

Rest parameters (ES6)

rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function

```
function sum(x, y, ...more) {
 //"more" is array of all extra passed params
 let total = x + y;
 if (more.length > 0) {
   for (let i = 0; i < more.length; i++) {
   total += more[i];
   }
 }
 console.log("Total: " + total);
 return total;
}
sum(5, 5, 5);
sum(6, 6, 6, 6, 6);
```

Spread operator (ES6)

The same … notation can be used to unpack iterable elements (array, string, object) rather than pack extra arguments into a function parameter.

```
var a, b, c, d, e;
a = [1,2,3];
b = "dog";
c = [42, "cat"];

// Using the concat method.
d = a.concat(b, c); // [1, 2, 3, “dog”, 42, “cat”]

// Using the spread operator.
e = [...a, b, ...c]; // [1, 2, 3, “dog”, 42, “cat”]
```

Semicolon ;

- Semicolons are (technically) 'optional'
  - JS implicitly adds them to our code if it makes the parser happy
  - "semicolon insertion"
  - "bad part" of JavaScript
  - 'seemed like a good idea at the time' …
  - in certain cases that can cause problems
    - return, var, break, throw, …
  - Best practice to explicitly include them
  - Include barces at the end of line versus new line
    - K&R (Kernighan and Ritchie) versus OTBS (one true brace style) styles

```
function a() {
   return {
        a: 1
 };
}
function b(){
 return  //semicolon gets
 inserted here
 {
  a: 1
  };
}
console.log(a()); //object
console.lo(b());
 //undefined
```

What is destructuring assignment?

- Special syntax that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
const numbers = [10, 20];
let [a, b] = numbers;
console.log(a);
console.log(b);
```

- Benefits:
  - 'Syntactic sugar' to replace the following:
    - `let a = numbers[0];`
    - `let b = numbers[1];`
  - syntax sugar for calling for..of over the value to the right of = and assigning the values.

Destructuring assignment

- Unwanted elements of the array can also be thrown away via an extra comma:

```
const [first, , third] = ["foo", "bar", "baz"];
console.log(first);
console.log(third);
```

- Can use any “assignables” at the left side.

```
let user = {};
[user.name, user.surname] = "John Smith".split(' ');
console.log(user); //{ name: 'John', surname: 'Smith' }
```

Destructuring objects

- Destructuring on objects lets you bind variables to different properties of an object.
  - Order does not matter

```
let options = {
    title: "Menu",
    width: 100,
    height: 200
};
let { title, width, height } = options;
alert(title); // Menu
alert(width); // 100
alert(height); // 200
```

Destructure property to another name

- to assign a property to a variable with another name, set it using a colon

```
// { sourceProperty: targetVariable }
let { width: w, height: h, title } = options;

// width -> w
// height -> h
// title -> title

alert(title); // Menu
alert(w); // 100
alert(h); // 200
```

export

- In JavaScript, the import and export statements are used to define and manage module dependencies, enabling modular code organization and reuse.
- Named Exports:
  - ◦ To export one or more variables, functions, or objects from a module, you can use the export keyword followed by the name of the item you want to export.

```
Math.js

export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

  - ◦ You can also export multiple items in a single statement.

```
Math.js

const add = (a, b) => a + b;
const subtract = (a, b) => a - b;
export { add, subtract };
```

Import

- Named Imports:
  - ◦ To import specific items (variables, functions, objects) from another module, use the import statement followed by the item names and the from keyword.

```
import { add, subtract } from './math.js';
const result1 = add(5, 3);
const result2 = subtract(8, 2);
```

- Default Imports:
  - ◦ When importing a default export, you can use any name you prefer for the imported item.

```
import greeting from './utility.js';
const message = greeting('Alice');
```

Default Exports

- You can export a default value (typically a single function, class, or object) from a module using the `export default` syntax.

utility.js

```
const greet = (name) => `Hello, ${name}!`;
export default greet;
```

Main Point

Functional programming methods map, filter, reduce make code more understandable and error free by automating details of general-purpose looping mechanisms, indicating their intent by their name, and not changing the state of the original array.

Science of Consciousness: Growth of consciousness makes behavior simpler and more error free because intentions that arise from deep levels are spontaneously in accord with and supported by all the laws of nature.