CSS222 WEB PROGRAMMING

LECTURE 05 – JAVASCRIPT #1

Dr. Wittawin Susutti
wittawin.sus@kmutt.ac.th

Outline

- Type
- Number
- Text
- Boolean
- Null and undefined
- Equality
- Variable
- Conditional logic
- Loop

What we've learned so far

- We've learned how to build web pages that:
  - Structured via semantic HTML
  - Look the way we want them to via CSS
  - Display differently on different screen sizes

- But we don't know how build web pages that ***do*** anything:
  - Get user input
  - Save user input
  - Show and hide elements when the user interacts with the page
  - etc.

JavaScript

- JavaScript is a programming language.
- It is currently the only programming language that your browser can execute natively.
- Therefore, if you want to make your web pages do stuff, you must use JavaScript: There are no other options.

Types

- JS **variables** do not have types, but the **values** do.
- There are six primitive types (mdn):
  - Boolean : true and false
  - Number : everything is a double (no integers)
  - String: in 'single' or "double-quotes"
  - Symbol
  - Null: a value meaning "this has no value"
  - Undefined: the value of a variable with no value assigned
- There are also Object types, including Array, Date, String, etc.
- `typeof` operator returns a string indicating the type of the operand's value.

Numbers

- The 64-bit floating- point format defined by the IEEE 754 standard, can represent numbers as large as $\pm 1.7976931348623157 \times 10^{308}$ and as small as $\pm 5 \times 10^{-324}$.
- When a number appears directly in a JavaScript program, it's called a ***numeric literal***.

- **Integer Literals**
  - A base-10 integer is written as a sequence of digits.
  - A hexadecimal literal begins with 0x or 0X, followed by a string of hexadecimal digits.
  - In ES6 and later, you can also express integers in binary (base 2) or octal (base 8) using the prefixes 0b and 0o (or 0B and 0O) instead of 0x .

```
0xff              // => 255: (15*16 + 15)
0xBADCAFE         // => 195939070

0b10101           // => 21: (1*16 + 0*8 + 1*4 + 0*2 + 1*1)
0o377             // => 255: (3*64 + 7*8 + 7*1)
```

Numbers

- **Floating-Point Literals**
  - Floating-point literals may also be represented using exponential notation: a real number followed by the letter e (or E), followed by an optional plus or minus sign, followed by an integer exponent.

```
3.14
2345.6789
.333333333333333333
6.02e23 // 6.02 x 10^23
1.4738223E-32 // 1.4738223 x 10^-32
```

- **Separators in Numeric Literals**
  - You can use underscores within numeric literals to break long literals up into chunks that are easier to read

```
let billion = 1_000_000_000; // Underscore as a thousands separator.
let bytes = 0x89_AB_CD_EF; // As a bytes separator.
let bits = 0b0001_1101_0111 // As a nibble separator.
let fraction = 0.123_456_789; // Works in the fractional parts, too.
```

Arithmetic in JavaScript

- Include **+**, **-**, *****, **/**, **%** and ****** for exponentiation
- JavaScript supports more complex mathematical operations through a set of functions and constants defined as properties of the Math object

```
Math.pow(2,53)      // => 9007199254740992: 2 to the power 53
Math.round(.6)      // => 1.0: round to the nearest integer
Math.celi(.6)       // => 1.0: round up to an integer
Math.floor(.6)      // => 0.0: round down to an integer
Math.abs(-5)        // => 5: absolute value
Math.max(x,y,z)     // Return the largest argument
Math.min(x,y,x)     // Return the smallest argument
Math.random()       // Pseudo-random number x where 0 <= x < 1
Math.PI()           // π: circumference of a circle / diameter
Math.E()            // e: The base of the natural logarithm
Math.sqrt(3)        // => 3**0.5: the square root of 3
Math.pow(3, 1/3)    // => 3**(1/3): the cube root of 3
Math.imul(2,3)      // => 6: optimized multiplication of
                       32-bit integers
Math.clz32(0xf)     // => 28: number of leading zero bits in
                       a 32-bit integer
Math.trunc(3.9)     // => 3: convert to an integer by
                       truncating fractional part
Math.fround(x)      // Round to nearest 32-bit float number
Math.sinh(x)        // Hyperbolic sine.
                       Also, Math.cosh(), Math.tanh()
Math.asinh()        // Hyperbolic arcsine.
                       Also, Math.acosh(), Math.atanh()
```

```
Math.sin(0)             // Trigonometry: also Math.cos,
                           Math.atan, etc.
Math.log(10)            // Natural logarithm of 10
Math.log(100)/Math.LN10 // Base 10 logarithm of 100
Math.log(512)/Math.LN2  // Base 2 logarithm of 512
Math.exp(3)             // Math.E cubed
Math.cbrt(27)           // => 3: cube root
Math.hypot(3,4)         // => 5: square root of sum of squares
                           of all arguments
Math.log10(100)         // => 2: Base-10 logarithm
Math.log2(1024)         // => 10: Base-2 logarithm
Math.log1p(x)           // Natural log of (1+x); accurate for
                           very small x
Math.expm1(x)           // Math.exp(x)-1; the inverse of
                           Math.log1p()
Math.sign(x)            // -1, 0 or 1 for arguments <, == or
                           > 0
```

Arithmetic in JavaScript

- Arithmetic in JavaScript **does not raise errors** in cases of overflow, underflow, or division by zero.
- When the result of overflow, the result is a special infinity value, ***Infinity***.
- JavaScript returns 0 for underflow.
- **Division by zero is not an error in JavaScript**: it returns infinity or negative infinity.
- However: zero divided by zero does not have a well- defined value, and the result of this operation is the special not-a-number value, **NaN**.

```
Infinity                    // A positive number too
                               big to represent
Number.POSITIVE_INFINITY    // Same value
1/0                         // => Infinity
Number.MAX_VALUE * 2        // Infinity; overflow
-Infinity                   // A negative number too
                               big to represent
Number.NEGATIVE_INFINITY    // The same value
-1/0                        // => -Infinity
-Number.MAX_VALUE * 2       // => -Infinity
NaN                         // The not-a-number value
Number.NaN                  // The same value, written
                               another way
0/0                         // => NaN
Infinity/Infinity           // => NaN
Number.MIN_VALUE/2          // => 0: underflow
-Number.MIN_VALUE/2         // => 0: negative zero
-1/Infinity                 // -> -0: also, negative zero
-0
```

```
Number.parseInt()           // Same as the global parseInt()
                               function
Number.parseFloat()         // Same as the global parseFloat()
                               function
Number.isNaN(x)             // Is x the NaN value?
Number.isFinite(x)          // Is x a number and finite?
Number.isInteger(x)         // Is x an integer?
Number.isSafeInteger(x)     // Is x an integer
                               -(2**53) < x < 2**53?
Number.MIN_SAFE_INTEGER     // => -(2**53 -1)
Number.MAX_SAFE_INTEGER     // => 2**53 -1\
Number.EPSILON              // => 2**-52: smallest difference
                               between numbers
```

Binary Floating-Point and Rounding Errors

- There are infinitely many real numbers, but only a finite number of them can be represented exactly by the JavaScript floating-point format.
- This means that when you're working with real numbers in JavaScript, the representation of the number will often be **an approximation of the actual number**.
- The IEEE-754 floating-point representation used by JavaScript (and just about every other modern programming language) is a binary representation.
- **BigInt** - 64-bit integers

```
1234n                       // A not-so-big BigInt literal
0b111111n                   // A binary BigInt
0o7777n                     // An octal BigInt
0x8000000000000000n         // => 2n**63n: A 64-bit integer
1000n + 2000n               // => 3000n
3000n - 2000n               // => 1000n
2000n * 3000n               // => 6000000n
3000n / 997n                // => 3n: the quotient is 3
3000n % 997n                // => 9n: and the remainder is 9
(2n ** 131071n) - 1n        // A Mersenne prime with 39457
                               decimal digits
```

```
BigInt(Number.MAX_SAFE_INTEGER)      // => 9007199254740991n
let string = "1" + "0".repeat(100); // 1 followed by 100 zeros.
BigInt(string)                       // => 10n**100n:
1 < 2n                               // => true
2 > 1n                               // => true
0 == 0n                              // => true
0 === 0n                             // => false: the === checks
                                        for type equality as well
```

Text

- The JavaScript type for representing text is the ***string***.
- The *length* of a string is the number of 16-bit values it contains.
- UTF-16 encoding of the Unicode character set
- Zero-based indexing
- The *empty string* is the string of length 0.
- Most string-manipulation methods defined by JavaScript operate on 16-bit values, not characters.
- Strings are ***iterable***, and if you use the for/of loop or ... operator with a string, it will iterate the actual characters of the string, not the 16-bit values.

String Literals

- Simply enclose the characters of the string within a matched pair of single or double quotes or backticks (' or " or `).
- Double-quote characters and backticks may be contained within strings delimited by single-quote characters, and similarly for strings delimited by double quotes and backticks.

```
""  // The empty string: it has zero characters
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"π ratio of the circumference of a circle to its radius"
`"She said 'hi'", he said.`

// A string representing 2 lines written on one line
'two\nlines'

// A one-line string written on 3 lines:

"one\
long\
line"

// A two-line string written on two lines:

`The newline character at the end of this line
is included literally in this string`
```

Sequence	Character represented
\0	The NUL character (\u0000)
\b	Backspace (\u0008)
\t	Horizontal tab (\u0009)
\n	New line (\u000A)
\v	Vertical tab (\u000B)
\f	Form seed (\u000C)
\r	Carriage return (\u000D)
\"	Double quote (\u0022)
\'	Apostrophe or single quote (\u0027)
\\	Backslash (\u005c)
\xnn	The unicode character specified by the two hexadecimal digits nn
\unnnn	The unicode character specified by the four hexadecimal digits nnnn
\u{n}	The unicode character specified by the codepoint n, where n is one to six hexadecimal digits between 0 and 10FFFF (ES6)

Working with Strings

```
let msg = "Hello, " + "world"     // Produces the string "Hello, world"
let greeting = "Welcome to my blog," + " " + name;
let s = "Hello, world";           // Start with some text.
s.length                          // 12
s[0]                              // => H
s[s.length-1]                     // => d

// Obtaining portions of a strings
s.substring(1,4)                  // "ell": the 2nd, 3rd and 4th characters.
s.slice(1,4)                      // "ell": same thing
s.slice(-3)                       // "rld": last 3 characters
s.split(", ")                     // ["Hello", "world"]: split at delimiter string

// Searching a string
s.indexOf("l")                    // => 2: position of first letter l
s.indexOf("l", 3)                 // => 3: position of first letter "l" at or after 3
s.indexOf("zz")                   // => -1: s does not in substring "zz"
s.lastIndexOf("l")                // => 10: position of last letter l

// Boolean searching functions in ES6 and later
s.startWith("Hell")               // => true: the string starts with these
s.endWith("!")                    // => false: s does not end with that
s.includes("or")                  // => true: s includes substring "or"
```

Working with Strings

```
// Creating modified versions of a string
s.replace("llo", "ya")       // => "Heya, world"
s.toLowerCase()              // => "hello, world"
s.toUpperCase()              // => "HELLO, WORLD"
s.normalize()                // Unicode NFC normalization: ES
s.normalize("NFD")           // NFD normalization. Also "NFKC", "NFKD"

//Inspecting individual (16-bit) characters of a string
s.charAt(0)                  // => "H": the first character
s.charAt(s.length-1)         // => "d" the last character
s.charCodeAt(0)              // => 72: 16-bit number at the specified position
s.codePointAt(0)             // => 72: ES6, works for code points > 16 bits

//  String padding functions in ES2017
x.padStart(3)                // => "  x": add spaces on the left to a length of 3
x.padEnd(3)                  // => "x  ": add spaces on the right to a length of 3
x.padStart(3, "*")           // => "**x": add stars on the left to a length of 3
x.padEnd(3, "-")             // => "x--": add dashes on the right to a length of 3

// Spaces trimming functions. trim() is ES5; others ES2019
" test ".trim()              // => "test": remove spaces at start and end
" test ".trimStart()         // => "test ": remove spaces on left. Also, trimLeft
" test ".trimEnd()           // =>  " test": remove spaces at right. Also, trimRight

// Miscellaneous string methods
s.contact("!")               // => "Hello, world!": just use + operator instead
"<>".repeat(5)               // =>  "<><><><><>": concatenate n copies. ES6
```

Template Literals

- In ES6 and later, string literals can be delimited with backticks:

```
let name = "Bill";
let greeting = `Hello ${name}.`;  // greeting == "Hello Bill."
```

- Everything between the **`${ and the matching }`** is interpreted as a JavaScript expression.

Boolean Values

- Any JavaScript value can be converted to a boolean value.
- The following values convert to, and therefore work like, false.

```
undefined
null
0
-0
NaN
"" // the empty string
```

- All other values, including all objects (and arrays) convert to, and work like, true.

Null and Undefined

- Both indicate **absence of a value.** What's the difference?
- **Null**
  - Using the `typeof` operator on null returns the string “object”, indicating that null can be thought of as a special object value that indicates “no object”.
  - It can be used to indicate “**no value**” for numbers and strings as well as objects.
- **Undefined**
  - The value of variables that have not been initialized and the value you get when you query the value of an object property or array element that does not exist.
  - If you apply the `typeof` operator to the undefined value, it returns “undefined”
- null and undefined can often be used interchangeably.
- The equality operator == considers them to be equal. (not by ===).

Type Conversions

- JavaScript is very flexible about the types of values it requires.

```
10 + "objects"      // => "10 objects": Number 10 converts to a string
"7" * "4"           // => 28: both strings converts to numbers
let n = 1 - "x"     // n == NaN; String x can't convert to a number
n + "objects"       // "NaN objects": NaN converts to string "NaN"
```

Value	to String	to Number	to Boolean
`undefined`	`"undefined"`	`NaN`	`false`
`null`	`"null"`	`0`	`false`
`true`	`"true"`	`1`	
`false`	`"false"`	`0`	
`""` (empty string)		`0`	**`false`**
`"1.2"` (nonempty, numeric)		`1.2`	`true`
`"one"` (nonempty, non-numeric)		`NaN`	`true`
`0`	`"0"`		**`false`**
`-0`	`"0"`		**`false`**

Value	to String	to Number	to Boolean
1 (finite, non-zero)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		**`false`**
`[]` (empty array)	`""`	`0`	`true`
`[9]` (one numeric element)	`"9"`	`9`	`true`
`['a']` (any other array)	*use join() method*	`NaN`	`true`

Equality

- **JavaScript's == and `!=` are basically broken** they do an implicit type conversion before the comparison.
  - `'' == '0'`
  - `'' == 0`
  - `0 == '0'`
  - `NaN == NaN`
  - `[''] == ''`
  - `true == 1`
  - `false == 0`
  - `false == undefined`
  - `false == null`
  - `null == undefined`
- **Always use === and `!==` and don't use == or `!=`**

Adding behaviors to browser

- **`alert()` -** instructs the browser to display a dialog with an optional message, and to wait until the user dismisses the dialog.

```
alert(message)
Example
        window.alert("Hello world!");
        alert("Hello world!");
```

- **`prompt()`** - instructs the browser to display a dialog with an optional message prompting the user to input some text, and to wait until the user either submits the text or cancels the dialog.

```
prompt(message)
prompt(message, defaultValue)
Example
        let sign = window.prompt("Are you feeling lucky");
        const aNumber = Number(window.prompt("Type a number", ""));
```

Adding behaviors to browser

- **`confirm()` -** shows a modal window with a question and two buttons: OK and Cancel. The result is true if OK is pressed and false otherwise.

`confirm(question)`

Example

```
let isPass = confirm("Are you pass the exam?");
alert( isPass ); // true if OK is pressed
```

`console.log`

- You can print log messages in JavaScript by calling `console.log()`:

```
console.log('Hello, world!');
```

Variable Declaration and Assignment

- In modern JavaScript (ES6 and later), variables are declared with the **let** keyword
- To declare a constant instead of a variable, use **const** instead of let.
- const works just like let except that you must initialize the constant when you declare it
- It is a common (but not universal) convention to declare constants using names with all capital letters such as H0 or HTTP_NOT_FOUND as a way to distinguish them from variables.
- let and const are *block scoped*

```
let message = "hello";
let i = 0, j = 0, k = 0;
let x = 2, y = x*x      // Initializers can use previously declared variables

const H0 = 74;          // Hubble constant (km/s/Mpc)
const C = 299792.458;   // speed of light in vacuum (km/s)
const AU = 1.496E8      // Astronomical Unit: distance to the sun (km)
```

What's a "block"?

- In the context of programming languages, a **block** is a group of statements, usually surrounded by curly braces.
- Also known as a **compound statement**
- Has **absolutely nothing** to do with the HTML/CSS notion of "block", i.e., block elements

What's a "block"?

For example, the precise definition of an if-statement might look like:

```
if(PM>2.5)
      {
      console.log('Hello, world!');
      console.log('Today is a good day.');
       }
```

Understanding `let` and `var`

```
function printMessage(message, times)
{
   for (let i = 0; i < times; i++)
   {
     console.log(message);
   }
   console.log('Value of i is ' + i);
}
printMessage('hello', 3);
```

```
function printMessage(message,
times) {
   for (var i = 0; i < times; i++)
   {
      console.log(message);
   }
   console.log('Value of i is ' +
i);
}
printMessage('hello', 3);
```

Understanding `let` and `var`

```
let x = 10;
if (x > 0)
{
   x = 10+1;
   let y = 10;
}
console.log(x);
console.log(y);
```

```
var x = 10;
if (x > 0)
{
   x = 10+1;
   var y = 10;
}
console.log(x);
console.log(y);
```

```
function  meaningless()
{
   var x = 10;
   if (x>0) {
     var y = 10;
   }
   console.log('y is ' + y);
}
meaningless();
console.log('y is '+y);
```

Understanding `const`

```
const y = 10;
y = 0;                  // error!
y++;                    // error!
const list = [1, 2, 3];
list.push(4);           // OK
```

Variable Naming Rules

- Variable names can include letters, digits, $, and _.
- The **first character cannot be a digit**.
- JS is Case Sensitivity
- Examples of valid names

```
let userName;
let test123;
```

- Common style: **camelCase** (e.g., myVeryLongName).
- $and _ are allowed, same as letters.

```
let $ = 1;
let _ = 2;
```

Coding styles or Code code conventions

Style	Example	Typical Usage	Why / Notes
camelCase	userName, cartTotal	Variables, functions, object properties (JavaScript, TypeScript)	Widely used, concise, readable
PascalCase	UserProfile, ShoppingCart	Class names, constructors, React components	Distinguishes classes/objects from normal variables
snake_case	user_name, cart_total	Variables & functions in Python, database column names	Easy to read, common in Python & SQL
UPPER_CASE	MAX_LIMIT, API_KEY	Constants, environment variables	Signals immutability, stands out clearly
kebab-case	user-name, shopping-cart	CSS classes, HTML attributes, filenames	Clean for markup/styles, **not valid in JS variables**

Variables best practices

- Use `const` whenever possible.
- If you need a variable to be reassignable, use `let`.
- Don't use `var`.
  - You will see a ton of example code on the internet with var since const and let are relatively new.
  - However, const and let are well-supported, so there's no reason not to use them.
- This is also what the Google and AirBnB JavaScript Style Guides recommend.
- Uppercase Constants - Used for **hard-coded values.**

```
const COLOR_RED = "#F00";

const COLOR_ORANGE = "#FF7F00";
```

- Avoid reusing variables for different purposes.
- Extra variables improve readability and debugging.

Conditional Logic

if / else if / else

- Used for branching based on conditions.

```
let age = 20;

if (age < 18) {
  console.log("Underage");
} else if (age < 65) {
  console.log("Adult");
} else {
  console.log("Senior");
}
```

Conditional Logic

switch

- Cleaner alternative when checking **many possible values** of one variable.

```
let color = "green";

switch (color) {
  case "red":
    console.log("Stop");
    break;
  case "yellow":
    console.log("Caution");
    break;
  case "green":
    console.log("Go");
    break;
  default:
    console.log("Unknown color");
}
```

Conditional Logic

Ternary Operator (? :)

- Shorthand for **simple conditions**.

```
let age = 20;
let access = (age >= 18) ? "Allowed" : "Denied";
console.log(access); // Allowed
```

Loop

for loop

- Best when the number of iterations is **known in advance**.

```
for (let i = 0; i < 5; i++) {
  console.log("Iteration:", i);
}
```

while loop

- Runs **while the condition is true**.

```
let i = 0;
while (i < 5) {
  console.log("Count:", i);
  i++;
}
```

do..while loop

- Runs the block **at least once**, then checks the condition.

```
let i = 0;
do {
  console.log("Number:", i);
  i++;
} while (i < 5);
```