Introduction to JavaScript

CIS 1962 (Winter 2026)
January 15th, 2026

Penn Engineering

Lesson Plan

Class Introduction

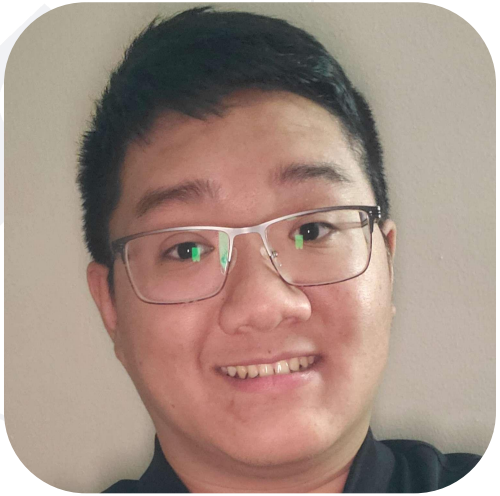
4	Staff Introductions
6	Class Logistics
9	AI Policy

Introduction to JavaScript

12	What is JavaScript?
19	Hello World and Data Types
28	Operators
36	Variables
46	Functions
52	Collections

Class Introduction

Staff



Instructor:
Voravich Silapachairueng
(he/him)

- Alumni of the MCIT program at UPenn, currently pursuing a career in teaching and software engineering.
- **Office hours**: Tuesday, 5 - 7 PM at Levine 501 (Bump Space)
- **Email**: voravich@seas.upenn.edu

Staff

- Co-instructor of CIS 1962, teaching the other section (202)
- Junior studying NETS
- **Office hours**: TBD (Check class website soon!)
- **Email**: esinx@seas.upenn.edu

TA:

Eunsoo Shin

(he/him)

Prerequisites

CIS 1200, or equivalent programming experience

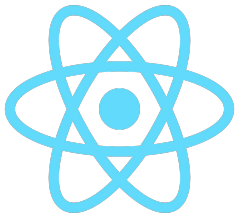
The class pace will be brisk for basic programming topics such as variables, functions, and control flow, with the assumption that you already have experience using them in other programming languages (like Java and Python).

This class will focus on the JavaScript implementation of those topics, and how you apply them to a web development context.

What will this class be about?

JavaScript Fundamentals:

The core concepts of JavaScript that make it unique



NEXT.JS

Modern JS Frameworks and Tools

Exploring various libraries and frameworks that power modern web applications

Foundations of Web Development:

Learning how JavaScript interacts with HTML and CSS to create dynamic web pages

Class Policies

Grade Breakdown:

Attendance	5%
Homework	60%
Final Project	35%

- Attendance taken through online polling (Slido)
  - Attendance is not mandatory, but lectures will not be recorded
- Homeworks are graded based on **correctness** (passing tests), and **style** (running through eslint) based on a posted style guide
  - Up to 72 hours late = -10%
  - Resubmission period at the end of the semester for -20%
- **Extensions** will be handled through email/Ed, feel free to reach out if you have extenuating circumstances or accommodations!

AI Policy

- No AI use...
- But we cannot stop you from using it, so use responsibly!
- What you can use AI for:
  - Conceptual questions
  - An alternative "search engine"
  - Teach you concepts/syntax not taught in class
- What you shouldn't use AI for:
  - Code generation
  - Copying the instructions and asking it to give you an answer

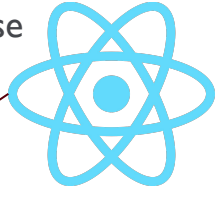
Any questions?

Introduction to JavaScript

What is JavaScript?

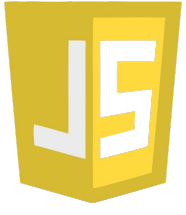
- JavaScript is a **high-level**, **interpreted** programming language
- It is used for dynamic client-side scripting on web pages
  - Controlling the content of web pages
  - Retrieving from servers
  - Storing to databases
  - Building and deploying web interfaces
  - … among other things!

What websites do you visit often?



~40% of professionals use report using React.js

JavaScript



98.9% of websites use JavaScript!

Penn Engineering

https://w3techs.com/technologies/details/cp-javascript
https://survey.stackoverflow.co/2023/#technology-web-frameworks

Key Features of JavaScript

- Runs natively in all browsers

- Single-Threaded Event Loop

```
console.log('Hi!');

setTimeout(() => {
    console.log('Execute
immediately.');
}, 0);

console.log('Bye!');
```

output:

```
Hi!
Bye!
Execute immediately
```

Key Features of JavaScript

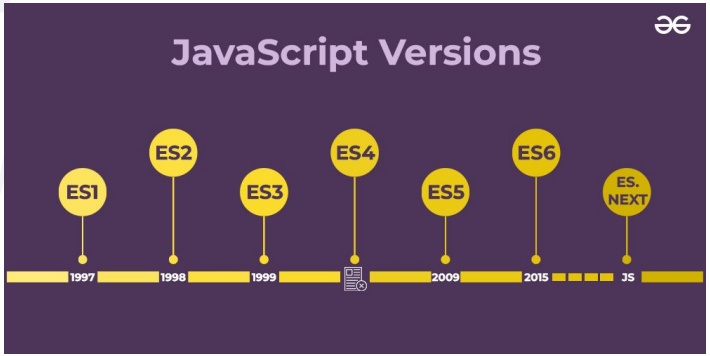
- Loosely-Typed, Aggressive Type Coercion

```
console.log("5" * "2") // 10
console.log(false + null) // 0

var empty;

if (empty) {
     console.log(“falsy!”);
}
```

- Event-driven programming


```
btn.addEventListener('click',
function() {
  btn.style.backgroundColor = 'red';
});
```

Where is JS Today?



The most recent version if JS is ECMAScript 2025 (ES16)

Future versions of JavaScript are released yearly, often referred to as ESNext.

For the most part, you can keep using ES6!

JavaScript: An Interpreted Language

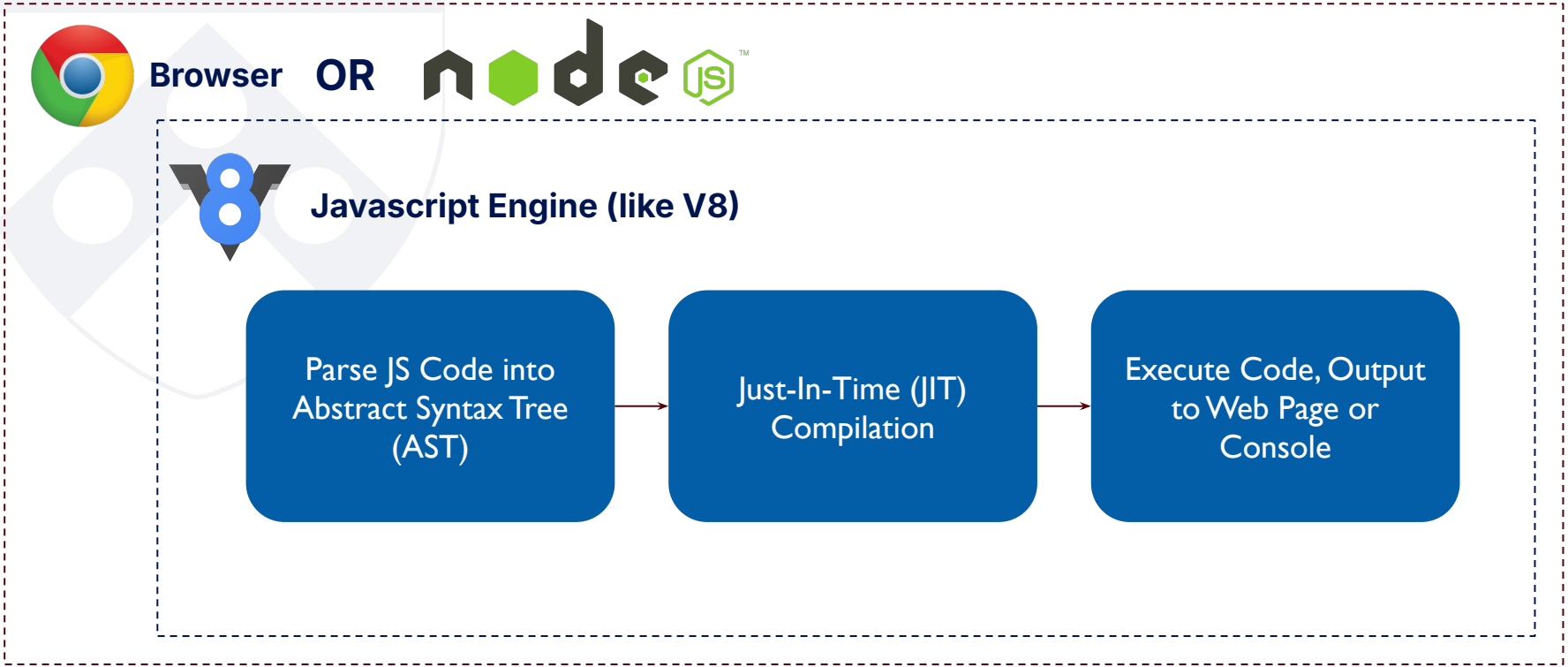
How do we run a .js file?

It must be parsed and interpreted by a **JavaScript engine.**

Unlike purely interpreted languages (like Ruby or Python), JavaScript makes use of a **Just-In-Time (JIT) Compiler** to both compile and interpret source code.

This is done inside a browser environment or Node.js.

Interpreting JavaScript



Hello World & Data Types

How do we print out items in JavaScript and work with various types of data?

hello_world.js

JavaScript files use the extension **.js**

You can print to the console using **console.log()**

Ending statements in semicolons (;) is **optional**, but good style.

Comments can be declared with **//**

```
// hello_world.js
console.log("hello world!");
```

Run the line of code within a browser, or if using Node for the file:

```
node hello_world.js
```

Data Types

```
let n; // undefined
n = null; // null
n = 10; // number
n = “ailurus fulgens”; // string
n = true // boolean
n = {
    name: “UPenn”,
    state: “Pennsylvania”,
    ivy_league: true,
}; // object
n = [ 1, 2, “hello”, false]; // array
```

Type Coercion

JavaScript is **loosely typed** and **flexible**.

The language will attempt to convert types into sensible ones with they are mixed.

`console.log("5" + 1)`	"51"
`console.log("5" - 1)`	4
`console.log(0 + true)`	1
`console.log(0 + false)`	0
`console.log("a" + 2 - 4 + true)`	NaN

Booleans: Truthy and Falsy

JavaScript popularized the terms "**truthy**" and "**falsy**" due to its very aggressive type coercion.

Values that are **falsy**: 0, "", null, undefined, NaN, false

All other values are **truthy**.

```
let password = ""

if (!password) {
    console.log("password empty!")
}
```

```
if (nonexistant) {
    console.log("this variable
doesn't exist yet!")
}
```

Numbers

Numbers in JavaScript are stored as double precision floating point numbers, maxing at $2^{53} - 1$.

Numbers beyond $2^{53} - 1$ require the **BigInt** type. **(append n)**

```
let big = 12345678901234567890123456789012345678890n;
```

Numbers have precision up to 15-17 decimal places. Floating point arithmetic can result in errors due to binary representations.

```
console.log(0.1 + 0.2 === 0.3) // false!
```

Strings

Strings can be created with single quotes (‘’), double quotes (“”), or backticks (``, often called **template literals**)

```
let string1 = ‘hello world’
let string2 = “hello world”
let string3 = `hello world`
```

Strings are **immutable**; you cannot change individual characters within a string after declaration.

```
let string = “hello world”
string[0] = “y”
console.log(string) // Still hello world
```

Strings: Template Literals

Introduced in ES6, template literals allow for **multi-line strings** and **embedding expressions** (like variable values) into strings.

```
let string = `change da world

my final message.

Goodbye`
```

```
let name = "Voravich"
console.log(`Hello, ${name}!`);
```

```
let x = 9, y = 8;
console.log(`Sum: ${x + y}`);
```

```
let isMember = false;
console.log(`Access: ${isMember ?
"Granted" : "Denied"}`);
```

Operators

What operators are available in JavaScript and how do we use them?

Operators: Arithmetic

Operator	Name	Example
+	Addition	`let x = 4 + 7 // x = 11`
-	Subtraction	`let x = 11 - 4 // x = 7`
*	Multiplication	`let x = 4 * 7 // x = 28`
/	Division	`let x = 28 / 7 // x = 4`
%	Modulus	`let x = 30 % 7 // x = 2`
++	Increment	`let x = 2; x++; // x = 3`
--	Decrement	`let x = 2; x--; // x = 1`

Operators: Assignment

Operator	Example
=	`let x = 5`
+=	`let x = 5; x += 5 // x = 10`
-=	`let x = 10; x -= 5 // x = 5`
*=	`let x = 5; x *= 6 // x = 30`
/=	`let x = 30; x /= 5 // x = 6`
%=	`let x = 31; x %= 5 // x = 1`

Operators: Logical

Operator	Name	Example

!	NOT	`!true // false`
		`!false // true`
		`!(true || false) // false`

&&	AND	`true && true // true`
		`true && false // false`
		`false && false // false`

||	OR	`true || true // true`
		`true || false // true`
		`false || false // false`

Operators: Comparison

Operator	Name	Example
==	Loose Equality	`2 == "2" // true`
!=	Loose Inequality	`2 != "2" // false`
===	Strict Equality	`2 === "2" // false`
!==	Strict Inequality	`2 !== "2" // true`
>	Greater Than	`6 > 3 // true`
<	Less Than	`5 < 10 // true`
>=	Greater Than or Equal To	`50 >= 50 // true`
<=	Less Than or Equal To	`40 <= 100 // true`

Loose Vs. Strict Equality/Inequality

The behavior of **loose equality/inequality** (==, !=) is another consequence of JavaScript type coercion!

== or != between a number and a string will attempt to coerce both types to match each other, giving unintended behavior.

Use **strict equality/inequality**(=== , !==) in cases where it matters where numbers and strings are used together and should not be strictly equal.

Loose Vs. Strict Equality/Inequality

```
let input = ""

if (input == 0) {
    console.log("input is 0");
} else {
    console.log("input is NOT 0");
}
```

output

```
input is 0
```

"" is coerced to be 0!

```
let input = ""

if (input === 0) {
    console.log("input is 0");
} else {
    console.log("input is NOT 0");
}
```

output

```
input is NOT 0
```

Strict Equality fixes this behavior.

5-Minute Break!

Variables

How do you declare a variable for use in JavaScript?

Variables

Variables in JavaScript are used to store data values that can be referenced and manipulated in a program.

JavaScript variables does not use explicit type declarations - variables are **dynamically typed**.

```
let value;
value = 157;
console.log(typeof value); // number

value = “hello world”;
console.log(typeof value); // string
```

Variables in Javascript

Variables in Javascript can be declared in 4 ways:

undeclared

```
x = 1
```

var

```
var x = 1
```

let

```
let x = 1
```

const

```
const x = 1
```

Undeclared Variables

Variables not declared using var, let, or const are undeclared.

They, by default, become **global variables**, which may clash with other declared variables.

```
function test() {
    x = 1
}

test()
console.log(x) // prints 1
```

"use strict"

The literal expression `"use strict"`, introduced in ES5, **prevents the use of undeclared variables.**

```
"use strict"
x = 1 // ReferenceError: x is not defined
```

Using "var"

Before 2015 (ES6), using var to declare variables was the standard in Javascript

```
function test() {
    var x = 9
    console.log(x) // prints 9
}
```

A variable declared with var is **function-scoped**

Problems with var

In modern Javascript, using var is often **too permissive** and causes unintentional behavior:

```
function test() {
    if (true) {
        var x = 9
    }
    console.log(x) // prints 9
}
```

Because of function-scoping, variables persist beyond internal blocks of code

```
function test2() {
    var x = 0
    var x = 9
    console.log(x) // prints 9
}
```

Variables declared with var can be redeclared

let and const

ES6 introduced the statements let and const, which are improvements from using var

Block-scoped:

variables exist only in the {} block they are defined in

```
function test() {
    if (true) {
        let x = 9
    }
    console.log(x) // ReferenceError
}
```

Cannot be redeclared:

Redeclaration causes an error

```
function test2() {
    let x = 2
    let x = 3 // SyntaxError
}
```

Differences between let and const

let and const mostly differ by their mutability:

const is **immutable**:

you cannot reassign a const variable, though it can be changed

```
function test() {
    const arr = [1, 4, 2]
    arr.push(8) // allowed
    arr = [5, 7] //  TypeError
}
```

let is **mutable**:

You can reassign items declared with let

```
function test2() {
    let count = 0
    for (let i = 0; i < 10; i++) {
        count++;
    }
    console.log(count)
}
```

Best practices for variable declaration

- By default, you should use **const** if intend not to re-assign the variable in the current scope.
- If you intend to reassign, use **let**.
- Some legacy code may require you to utilize **var**, but these cases are rare.
- Avoid leaving undeclared variables, use strict mode if applicable.

Functions

How do you write functions in JavaScript?

What are some unique ways they can be used?

Functions

Functions are reusable blocks of code that perform tasks.

Functions in JavaScript are **first-class objects**. They can be:

- Assigned as variables
- Stored in data structures (arrays, objects)
- Passed as arguments
- Returned from other functions

Higher-order functions

Declaring Functions

```
function add(a, b) {
    return a + b;
}
```

Function Declaration (hoisted)

```
const add = function(a, b) {
    return a + b;
}
```

Function Expression/ Anonymous Function

```
const add = (a, b) => {
    return a + b;
}
```

Arrow Function

Calling Functions and Arguments

You can call a named function using its name followed by the defined number of arguments in the declaration or expression.

```
add(34, 56)
```

Arguments of primitive data types (number, string, boolean, null, undefined) are **pass by value**.

- Changes to the parameter inside the function do not affect the original value

Arguments of objects, arrays, and functions are partially **pass by reference**.

- Changes to the object inside the function affect the original object, but reassignments do not affect the original object.

Argument Examples

```
function double(arg) {
    arg = arg * 2
}

let x = 20;
double(x)
console.log(x) // prints 20
```

Primitives: Pass By Value

```
function editName(file, newName) {
    file.fileName = newName;
}

let newFile = {
    fileName: ""
}
editName(newFile, "hello_world.js")
console.log(newFile.fileName) //
prints hello_world.js
```

Non-primitives: Pass By Reference

Functions Hoisting

Functions declared using function declaration are **hoisted**.

This means within the scope where the function is defined, the **function declaration is moved to the top**.

You can call them before you declare them in the code!

Function Declaration

```
add(34, 56) // Allowed

function add(a, b) {
    return a + b;
}
```

Function Expression

```
add(34, 56) // Error: add is not a
function

const add = function(a, b) {
    return a + b;
}
```

Higher Order Functions: Intro

Functions can be specified as arguments for other functions.

This allows you to use that argument to call them from within the other function.

```
function compute(a, b, operation) {
  return operation(a, b);
}

function multiply(x, y) {
  return x * y;
}

console.log(compute(5,6,multiply))
```

Collections

What are Collections?

Collections are data structures that allow you to group values together, allowing for more data organization and easier iteration.

There are 4 primary collections in JavaScript:

- Arrays
- Objects
- Maps
- Sets

Arrays

JavaScript **Arrays** is a 0-indexed collection of values that is:

Ordered

Arrays have a defined order that can be accessed using an index starting at 0 and ending at length - 1.

```
let arr = [1, 2, 3]
console.log(arr[0]) // 1
```

Dynamically-Sized

Arrays can grow or shrink in size by certain built-in methods.

```
let arr = [1, 2, 3]
arr.push(4) // 1, 2, 3, 4
arr.pop() // 1, 2, 3
arr.pop() // 1, 2
```

Mixed Type

Arrays can contain a mixture of types, as they do not have a defined singular type mandated of all elements.

```
let arr = [1, "2", true,
{name:"v",
role:"teacher"}]
```

Arrays and JavaScript Built-In Methods

JavaScript has built-in methods, functions that are properties of objects, for certain items, including all the collections.

Methods are called using dot notation (object.method()), and can take arguments.

For example, some array methods include:

- `.push(value)`, `.pop()`, `.filter(callback)`, `join()`

Array Methods: Access

Arrays are 0-indexed, and can be accessed using [].

Accessing items outside the existing indices of the array will return `undefined`.

```
let arr = [“a”, “b” , “c”, d”]

console.log(arr[0]) // a
console.log(arr[1]) // b
console.log(arr[2]) // c
console.log(arr[3]) // d
console.log(arr[4]) // undefined
console.log(arr[-1]) // undefined
```

Array Methods: Length and Setting

You can set the item at a certain index of an array using the index, like so:

```
arr[index] = desiredItem
// For instance:
let arr = [1, 2, 3]
arr[5] = 20
console.log(arr) // [ 1, 2, 3, <2 empty items>, 20 ]
```

This can increase the length of the array, as seen above

You can verify the length of the array using `array.length`:

```
let arr = [1, 2, 3]
console.log(arr.length) // 3
```

Array Methods: Insertion/Deletion

`push(value)`: insert an element **or** element**s** to the **end**, returns new length

`unshift(value)`: insert an element **or** element**s** to the **front**, returns new length

`pop()`: removes **and** returns the last element

`shift()`: removes **and** returns the first element

```
let arr = [“a”, “b” , “c”, d”]

console.log(arr.push(“e”)) // arr = [a, b, c, d, e], returns 5
console.log(arr.push(“f”, “g”)) // arr = [a, b, c, d, e, f, g], returns 7
console.log(arr.pop()) // arr = [a, b, c, d, e, f], returns g
console.log(arr.shift()) // arr = [b, c, d, e, f], returns a
console.log(arr.unshift(“a”)) // arr = [a, b, c, d, e, f], returns 6
```

Array Methods: Finding Elements

`indexOf(value)`: Find the index of a certain value, `-1` if it doesn't exist

`includes(value)`: Check if and array contains a certain value

```
let arr = [1, 2, 3]

console.log(arr.indexOf(2)) // 1
console.log(arr.indexOf(4)) // -1
console.log(arr.includes(4)) // false
```

Array Methods: Slicing and Splicing

`slice(start, end)`: Returns a **new** array from some starting index to some ending index. If no ending index is specified, slices from the starting index to the last index. **Does not modify the original array.**

`splice(start, deleteCount, item(s))`: **Modifies the original array** by adding, removing, and replacing elements in place. **Returns removed elements.**

```
let arr = [1, 2, 3, 4, 5]

let slicedArr = arr.slice(2)
console.log(slicedArr) // [3, 4, 5]

arr.splice(2, 0, 6) // at index 2, insert item 6
console.log(arr) // [1, 2, 6, 3, 4, 5]

arr.splice(3, 3, 7, 8, 9) // at index 3, replace
the next 3 elements with 7, 8, and 9
console.log(arr) // [1, 2, 6, 7, 8, 9]
```

Arrays and Shallow Copies

Any array copying and methods that use copies of arrays create only **shallow copies.**

This means that only the **top-level references** are copied to the array, **nested references** to objects will still reference the original

```
let original = [1, 2, { a: 5 }];
let copy = original.slice();

copy[0] = 99;  // Only affects the copy
copy[2].a = 42; // Affects both

console.log(original) // [1, 2, { a: 42 }]
console.log(copy) // [99, 2, { a: 42 }]
```

Array Methods: Callbacks

There are certain array methods that take in **callback functions** as arguments.

We will discuss callback functions and higher order functions later.

These methods are often used with functions with arrow notation to define their behavior concisely.

```
let arr = [1, 2, 3, 4]

// map
let mapped = arr.map((x) => x * x)
console.log(mapped) // [1, 4, 9, 16]

// filter
let even = arr.filter((x) => x % 2 == 0)
console.log(even) // [2, 4]
```

Objects

Objects are data structures that store data in a collection of **key-value pairs**, or **properties**. They are similar to dictionaries in Python or HashMaps in Java.

Objects are the basis of **JSON** (JavaScript Object Notation), used widely in web APIs and data storage.

Objects are typically **unordered**, though many engines will preserve insertion order for keys.

Objects: Keys and Values

KEY : **VALUE**

- **Data Type**: Must be a **string** or **symbol**, all other types are coerced into strings
- **Data Type: Any,** can store primitives and other objects and arrays

Creating Objects

There are 3 typical ways of making objects:

Object Literal

```
let animal = {
     name: "red panda",
     formal: "ailurus
fulgens",
     endangered: true,
     population: 15000
};
```

new Object()

```
let animal =
     new Object();
animal.name =
     "red panda",
animal.formal =
     "ailurus fulgens",
animal.endangered = true,
animal.population = 15000
```

Constructors (ES6)

```
class Animal {
       constructor(name, formal,
           endangered, population)  {
               this.name = name
               this.formal = formal
               this.endangered =
           endangered
               this.population =
           population
       }
}
let redPanda = new Animal("red panda",
"ailurus fulgens", true, 15000)
```

Object Properties

Properties of an object can accessed, added, and deleted using either **dot** notation or **bracket** notation.

```
let account = {}
account.username = "Klein"
account.password = "*********"

console.log("Resetting password
for " + account.username)

delete account.password
```

Dot Notation

```
let account = {}
account["username"] = "Klein"
account["password"] = "*********"

let key = "username"

console.log(account[key]) // Klein
```

Bracket Notation

Object Properties: Methods

`Object.keys(object)`: list out the keys of an object

`Object.values(object)`: list out the values of an object

`Object.entries(object)`: list out the entries of key/value pairs of an object

```
let account = {}
account.username = "Klein"
account.password = "*********"

console.log(Object.keys(account)) // ['username', 'password']
console.log(Object.values(account)) // [ 'Klein', '*********' ]
console.log(Object.entries(account))
// [ [ 'username', 'Klein' ], [ 'password', '*********' ] ]
```

Functions in Object Properties

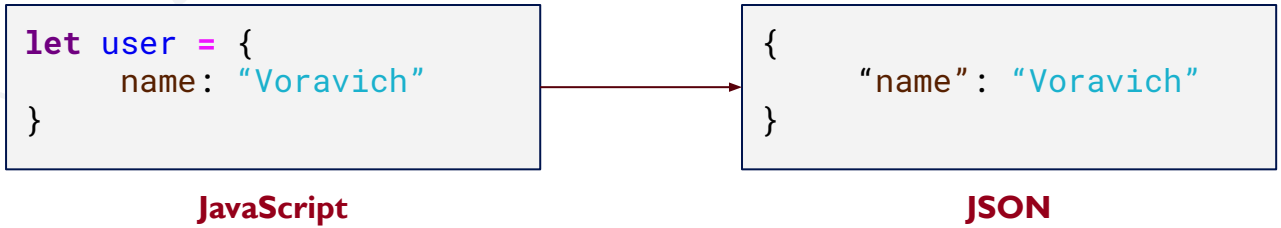
Because functions are first class objects, they can be stored as properties within objects.

```
let calculator = {
     add: function(x, y) { return x + y; },
     subtract: function(x, y) { return x + y; },
     multiply(x, y) { return x * y },
     divide(x, y) { return x / y },
     power: (x, y) => { return x ** y },
}

console.log(calculator.add(2, 3)) // 5
console.log(calculator.multiply(2, 3)) // 6
console.log(calculator.power(2, 3)) // 8

calculator.mod = function(x, y) {
     return x % y;
}
console.log(calculator.mod(10, 3)) // 1
```

Objects and JSON

JSON, an often-used format for data storage, looks like a JavaScript object literal, but is **entirely a string.**

Any keys are always enclosed in double quotes (“”):



JSON: Serialization and Parsing

The process of transferring from object to JSON is called **serialization**, and can be achieved with `JSON.stringify()`.

The opposite process, transferring JSON to a JS object, is called **parsing or unserialization**, done through `JSON.parse()`.