Department of Computer Science
University of California, Los Angeles

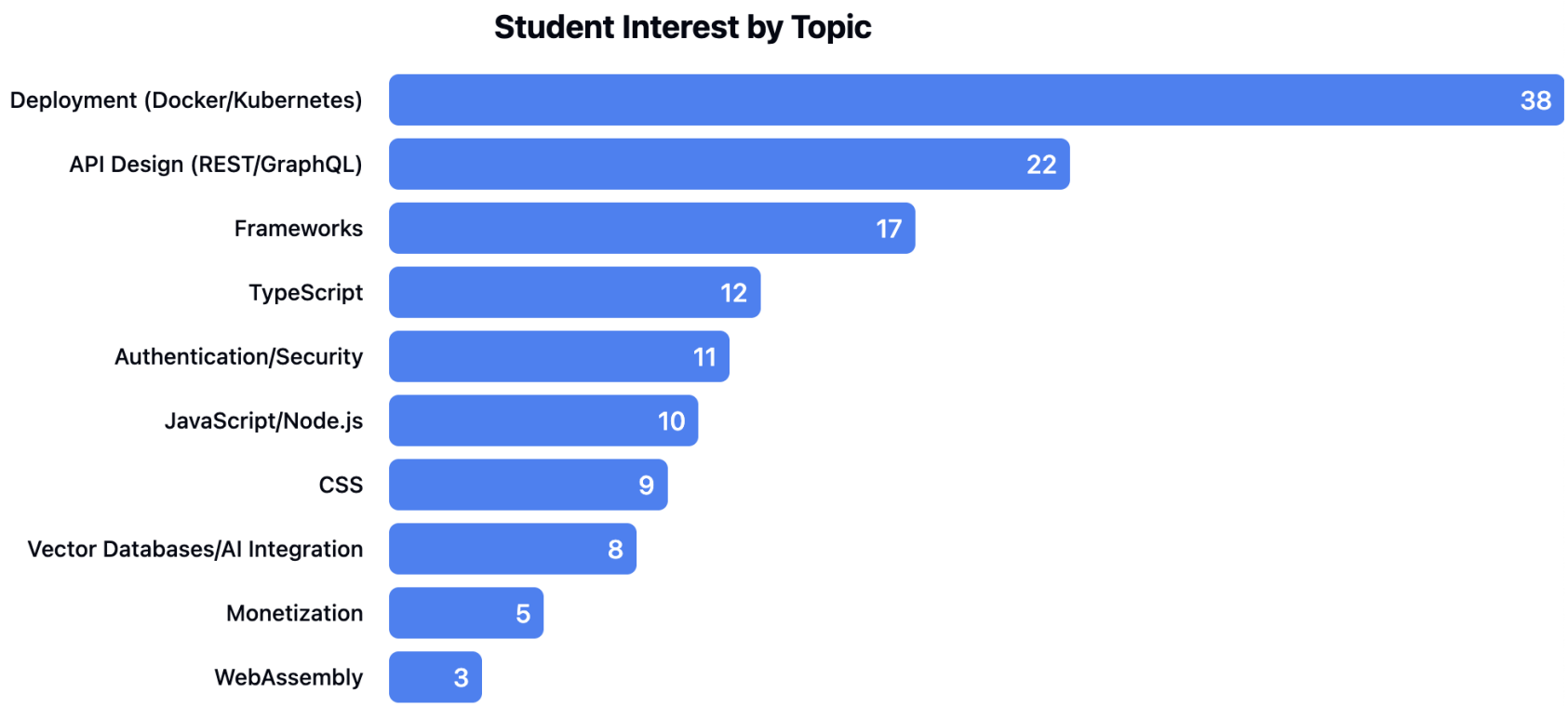
Computer Science 144: Web Applications

Spring 2025
Prof. Ryan Rosario

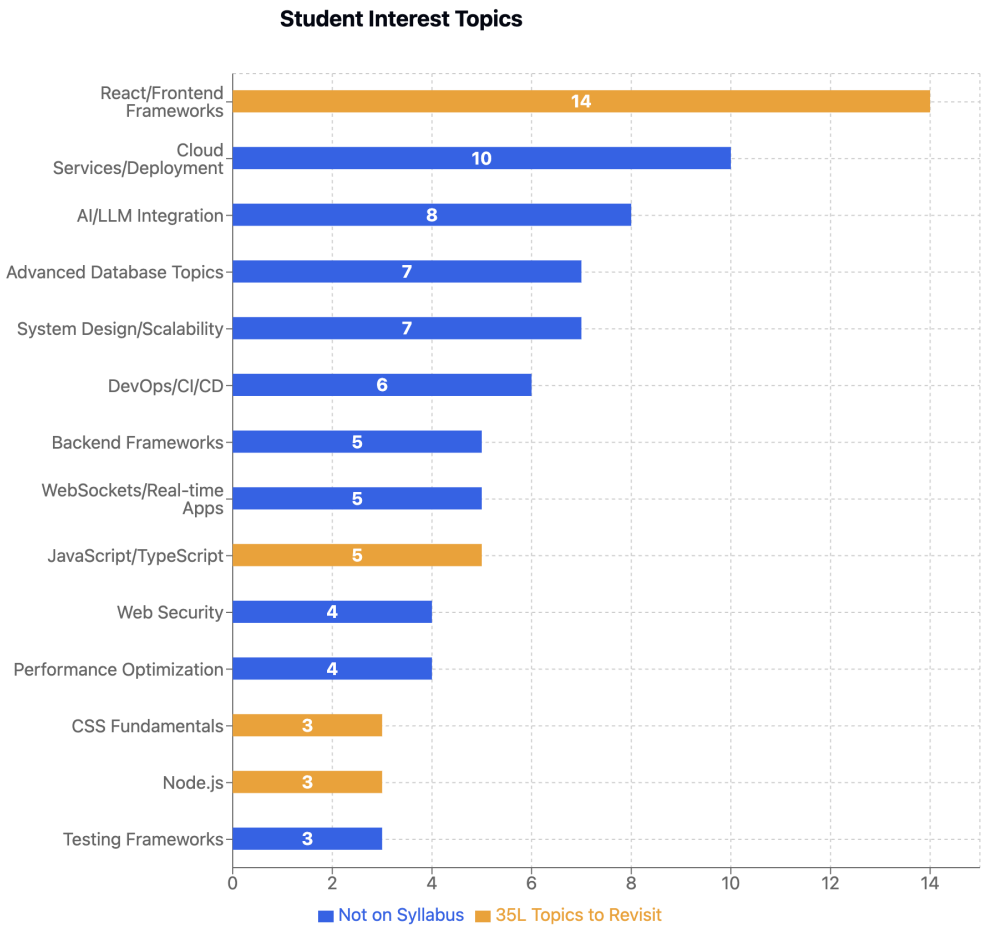
Lecture 5: April 14, 2025

Outline

Thank you for your participation in the student survey in the quiz. It helps me immensely. Interest in syllabus topics:



In terms of topics that are not on the syllabus but were mentioned.



Technical issues with quiz:

1. Lag
2. Unable to access Google Drive or Docs. I am looking into this. Please print your notes.
3. MDN apparently was restricted.
4. Getting kicked out of quiz or crashing the laptop (rare, but I've seen it each time)
5. Most students were able to access the quiz.

JavaScript

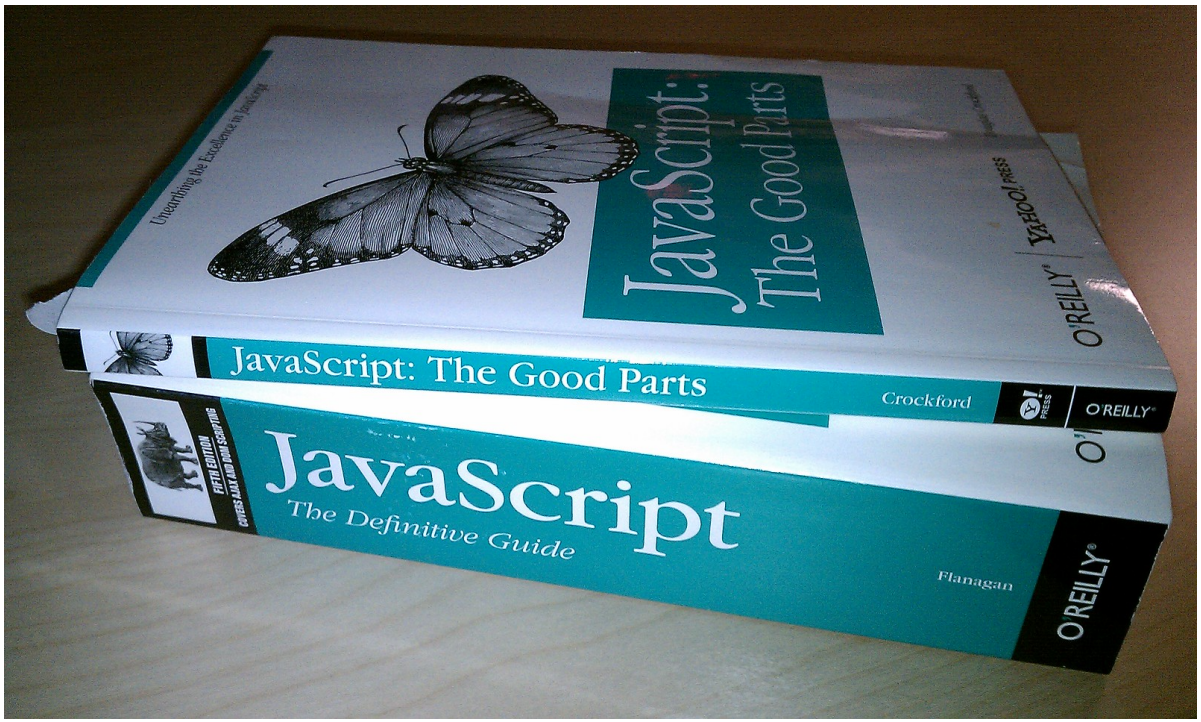
The goal today is not to go through all of the syntax of JavaScript — you have references for it.

The goal is to discuss important differences from other languages and important concepts.

JavaScript (contd.)

JavaScript has had a messy past. At one point, this was a very popular book.

JavaScript (contd.)



It was very thin. . .

What is JavaScript

JavaScript is sometimes called the "programming language of the web."

It is the final puzzle piece of the *basic* modern web stack with HTML and CSS.

What is JavaScript (contd.)

Some features of JavaScript:

1. high-level
2. usually just-in-time compiled (similar to "interpreted")
3. weakly and dynamically typed
4. prototype-based object-oriented
5. is multi-paradigm
   - event-driven
   - functional
   - imperative
6. Based on the ECMAScript standard

What is JavaScript (contd.)

JavaScript is a dynamic weakly typed language, so it sometimes does bizarre things.

1. **Dynamic:** type *checking* is performed during execution not during compilation (there may not be any compilation at all).
2. **Weak:** types are usually implicitly converted without the user knowing.

What is JavaScript (contd.)

JavaScript was developed as a client-side scripting language, with each web browser providing an engine for execution.

JavaScript can be run outside of the browser. The most popular non-browser runtime is Node.js.

MongoDB also relies on JavaScript as its DDL and DML.

What is JavaScript (contd.)

The name JavaScript was a marketing ploy due to the popularity of Java. Java and JavaScript were not intended to have much in common.

Brief History of JavaScript and ECMAScript

Javascript was originally very limited (client-side) and extremely slow. Open-source developments started to materialize making JavaScript more useful (and more fun?).

Jesse James Garrett develops AJAX, backed by JavaScript. This spawned libraries like jQuery.

Brief History of JavaScript and ECMAScript (contd.)

Google releases Chrome with the V8 JavaScript engine, which was much faster than others due to its use of just-in-time (JIT) compilation.

Brief History of JavaScript and ECMAScript (contd.)

In 2009, Node.js, by Ryan Dahl, makes its debut. JavaScript could now be executed outside the browser.

Brief History of JavaScript and ECMAScript (contd.)

Work continued feverishly for several years. Node caused a surge in the use of JavaScript and interest in its future.

ECMAScript 6 was released in 2015.

It added several new features including class and module notation. This transformed JavaScript from a scripting language to a serious general purpose programming language.

Brief History of JavaScript (2015-today)

Since ECMAScript 6 ("ECMAScript 2015"), new versions are released each year using the year as the identifier (e.g. there is no ECMAScript 7).

The most recent version is ECMAScript 2024 with 2025 released in June.

JavaScript vs. ECMAScript

ECMAScript is the name of the standard and many purists refer to ECMAScript or "ES".

JavaScript is based on ECMAScript. Regardless, most just call it JavaScript.

JavaScript is actually a trademark or Sun/Oracle.

Our Discussion of JavaScript

Our discussion will mostly use ECMAScript 2015 (ES6) to 2022:

- ES6 is much cleaner than previous versions.
- ES5 and earlier had a lot of flaws. Many still exists for backwards compatibility. — `use strict`
- Many resources use ES5 and a lot of ES5 code still exists.

I cannot stress this enough. Be careful that the resources you are using are **recent** and based on **ES6**. This is not always obvious.

JavaScript

For lecture, we will mostly focus on differences between JavaScript and other languages, particularly Java and C(++), and sometimes Python.

As CS majors, you can use resources presented at the end of the slides to learn more about specific syntax.

Unfortunately, there are a lot of differences to discuss.

We will have a project and some discussion exercises about JavaScript where we can dive more into syntax.

Where to Put Your JavaScript

External: Good for reuse.

```
<html>
<head>
    <script type="text/javascript" src="myscript.js"></script>
</head>
```

Inline: Either in <head> or right before closing the <body>.

```
<script>
    // JavaScript code
    document.getElementById("demo").innerText = "Hello, World!";
</script>
```

Print Statements

If you need to print any values from JavaScript, use:

```
console.log(...)
```

Variable Declaration and comments.

To declare a variable and constant use:

```
x = 10;  // This creates a globally scoped variable.

let y = 10;    // standard variable assignment

// To *redeclare* a variable
y = 20;  // must be already defined, else it's global

const z = 10;  // declare constant. Can't be reassigned or re-declared.
```

What about `var`? **Don't use it!** It was popular before ES6. We will discuss later.

Basic JavaScript Syntax

Arithmethic operators are the same:

- `+`, `-`, `*`, `/`, `%`
- Note that division (/) is not integer.
- Related operators: ++ and – (pre/post), +=, -=, *= etc.

Bitwise operators are also the same:

- `~`, `&`, `|`, `^`, ...

Logical operators are the same:

- `!`, `&&`, `||`
- Note that how statements are evaluated is different though (dynamic and weak typing).

Basic JavaScript Syntax (contd.)

Much syntax is similar to Java/C:

```
if (condition1) {
    statement;
} else if (condition2) {
    // not elif or elsif
    statement;
} else {
    statement;
}
```

Basic JavaScript Syntax (contd.)

A loop and array interlude.

```
let fruits = ["banana", "pitaya", "pineapple", "coconut"];
fruits.type = "tropical";

// very slight difference in for
// const also works
for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}

for (const key in fruits) {
    console.log(key);
}

for (const fruit of fruits) {
    console.log(fruit);
}
```

What do you notice?

Basic JavaScript Syntax (contd.)

Arrays work similar to in other languages, but it is an **object** not a distinct type.

Another difference with other languages is that arrays can have properties *in addition* to the values in the array. We saw this earlier with `fruits`.

```
let primes = new Array(2, 3, 5, 7, 11);
primes.countable = true;

let fib = [1, 1, 2, 3, 5, 8];
console.log(fib.length);

let knapsack = [1, "yarn", false, , ["feather", 42]];
console.log(knapsack.length);
```

Get/set using integers using `[]`. For array methods, see here.

Basic JavaScript Syntax (contd.)

By the way, some places to practice:

- JavaScript Playground
- JSFiddle
- NodeJS Command Line

Basic JavaScript Syntax (contd.)

There are two concepts of equality: == and ===. For two values:

- `x == y` is true if and only if the coerced values of `x` and `y` are the same.
  - `"5" == 5` is true.
- `x === y` is true if and only if the values `x`, `y` **have the same value and the same type**.
  - `"5" === 5` is false
  - `5 === 5` is true
  - `5.0 === 5` is true. Why?

Why? JavaScript does not have different number types.

Basic JavaScript Syntax (contd.)

Also:

- JavaScript is **case sensitive** but HTML is not, and this can cause confusion.
- Every line should end in a semicolon (;).
- JavaScript bindings (e.g. variables) can have letters, numbers, underscores and $. Typically written in `camelCase`.
- Use // to declare a one-line comment and /* */ for multi-line just like in Java/C.

Functions

Function declaration has changed over versions.

```
// declared with a statement
// can be declared anywhere, hoisted to the "top"

function square(x) {
    // if no return value -> undefined
    return x * x;
}
```

There are no parameter types or return types.

Functions (contd.)

Functions can also be used as values (first-class):

```
// Store them
const mathFunctions = {
    add: (a, b) => a + b,
    subtract: (a, b) => a - b,
    multiply: (a, b) => a * b,
    quartic: (a) => square(a) * square(a)
}

// Pass functions to other functions
function applyOperation(a, b, op) {
    return op(a, b);
}
```

Functions (contd.)

We also create a function factory. This is a common pattern in JavaScript and example of a **closure**.

```
function createMultiplier(factor) {
    return function(x) {
        return x * factor;
    }
}
const double = createMultiplier(2);
const triple = createMultiplier(3);
console.log(double(5));
console.log(triple(5));
```

Functions (contd.)

ES6 also added Arrow Functions to make it possible to write small functions without overhead (syntactic sugar). They are usually passed to functions:

```
// Two parameters. Example using a return and ()
const add = (x, y) => { return x + y; };

const square1 = (x) => { return x * x; };

// Don't need () for single argument. Don't need return, it's implied.
const square2 = x => x * x;

// No parameters
const horn = () => { console.log("Beep! Beep!"); };
```

Functions (contd.)

Functions are objects according to the standard but if you query its type you get `function` not `object`.

Functions can also have properties.

Functions can also be nested as in other languages.

Functions (contd.)

Other important facts about functions:

- Function definitions are hoisted. They can be used before definition, if in the same scope.
- Can be called with fewer arguments than in the definition. Missing parameters are `undefined`.
- All functions return a value, even if no `return` is provided. It's `undefined`.

Differences in Data Types

Types in JavaScript are either primitive or objects. Primitive types are immutable.

- `number` (no integer, float etc.)
- `bigint` (the only exception)
- `boolean`
- `string`
- `null`
- `undefined`

For primitive types, the binding (variable name) simply points to a value and reassigning simply changes the pointer, not the value. (Similar to Python).

Differences in Data Types (contd.)

Since JavaScript is dynamic and weak, we should check the types of variables frequently as they may change.

```
let weather = "sunny";
typeof weather;          // returns string
```

This isn't always straightforward though.

Numbers, Booleans and Strings

Numbers, booleans and strings are important but a bit of a mess.

- Represented as **64-bit floating point** — there were no native integer types
- `NaN` (0/0) and `Infinity` (n/0)
- `bigint` (64 bit integer) added in 2020 as primitive type.
  - It is NOT a number type.
  - Add `n` to the back of a number: `42n`.
  - Cannot mix with `number`. Can't do `1 + 1n`.
  - Some values are falsy:
    - `0, -0, 0n, NaN, "", null, undefined`
    - `[], {}` are not.
- Strings are immutable. `'hello'`, double quotes `"hello"` or backticks `` `Hello` ``. Many string functions

Numbers, Booleans and Strings (contd.)

Use backticks for templated strings.

```
const name = "Ryan";
const temp = 21;  // temp in celsius
let greeting = `Hello, ${name}. The temperature is ` +
    `${Math.floor((9/5) * temp + 32)} degrees F.`
```

You must escape special characters: `,$,{,}.

`null` and `undefined`

`undefined` is the value of a(n):

1. uninitialized variable
2. function parameter that was not passed to a function
3. return value from a function if nothing is returned

`null` is more for a missing value.

`null` and `undefined` are often used interchangeable but they are different:

- `null == undefined` is true
- `null === undefined` is false
- `typeof null` is `object`
- `typeof undefined` is `undefined`

`null` and `undefined` (contd.)

Point: Use `===` and `!==` instead of `==` and `!=`.

This website has a nice visual about how equality works.

Type Coercion

- **Booleans:** `&&`, `||` convert the left hand side to `boolean` if possible.
  - `null || "user"` → "user"
  - `"Agnes" || "user"` → "Agnes"
  - `"false" || "user"` → "false". Why?
  - What about `"true" || "user"` → "true"
- **Strings and Numbers**
  - **With +:** `"5" + 2 = "52"`
  - **Other Arithmetic Operators:** Try to convert to number, else `NaN`. `"5" * 3 = "15"`, `true + 2 = 3`, `2 * "hello" = NaN`
  - **A Mix?** Order of operations dictates behavior.

Type Coercion (contd.)

It's better to explicitly type cast:

1. **Number, boolean to string:** `String()`
2. **String to number:** `Number()`, `parseInt()`, `parseFloat()`
3. **Number to boolean:** `Boolean()`.
4. **Object x to string:** `x.toString()` but there are caveats.

This site contains a nice table showing how values are coerced.

Objects

Anything that is not a **primitive** type is an object. Objects are arbitrary collections of properties. They can be nested too:

```
let cs144 = {
    instructor: "Rosario, R.R.",
    prerequisites: ["CS143"],
    meet_time: {
        "start time": "18:00",
        "end time": "19:50"
    }
    room: "MS4000A",
    ta: ["Agrawal, V.", "Theeranantachai, S."]
}
```

Can get/set instructor name with `cs144.instructor` or `cs144["instructor"]`. What about `start time`?

JavaScript Object Notation (JSON)

We can serialize many objects using JSON, and construct objects from JSON:

```
// An inside threat at Jamba Juice wants to steal a recipe and
// send it to Shake Shack.
const pb_mood = {
    name: "Peanut Butter Moo'd",
    ingredients: ["peanut butter", "banana"],
    calories: 7000
};

// Serialize the object.
const pb_mood_recipe = JSON.stringify(pb_mood);

// Shake Shack receives this message as in the demo.
const pb_mood_ss = JSON.parse(pb_mood_recipe);  // pb_mood_recipe is JSON
```

Values like `undefined`, functions etc. are ignored or converted to `null`.

JavaScript Object Notation (JSON) (contd.)

Objects in JavaScript are not JSON, but they can be **serialized** to storage as JSON, which looks a lot like the object itself:

```
JSON.stringify(cs144);
```

- Property names must be in double quotes.
- Strings must be enclosed in double quotes (not single).
- Values cannot be functions or `undefined`.

Objects can be reconstructed on other machines, or stored, by passing around JSON (serialization).

Equality with Objects

Equality works differently with objects.

For two bindings ("variables") `x` and `y`, `x == y`, `x === y` are only true if and only if both bindings point to the same object:

- `let o1 = { id: 1 }; let o2 = o1; o1 === o2` is `true`.
- But `let o3 = { id: 1 }; o3 === o1` returns `false`.
- There is no deep comparison operator.

Strict Mode

JavaScript, because of the mess before ES6, lets us do a lot of stupid things that may make you say "WTF?". Strict mode will warn us if we are doing something legal, but not recommended.

```
function canYouFindTheProblem() {
    "use strict";
    for (counter = 0; counter < 10; counter++) {
        console.log(counter);
    }
}
canYouFindTheProblem();
```

What happens?

Variable Scope

A block is defined as a syntactic unit enclosed in `{}`:

- Variables declared outside of **any** block are *globally scoped*.
- A variable **defined** without `let` is also *globally scoped* regardless where it is defined. **This is strongly discouraged.**
- A variable declared with `let` inside a block has *block level scope*. Everything in that block can access it.
- Variables in nested functions follow *lexical scope*. Visibility is determined by code order, not function call order.
- But what about `var`? We are getting to it, I promise.

```
// Global scope
let ucla = "bruins";
var usc = "trojans";
stanford = "cardinal";

function scopeDemo() {
  // Function scope
  console.log(ucla);
  console.log(usc);
  console.log(stanford);

  // Redefining variables
  ucla = "bruWins";
  var usc = "fail";

  // Creating new variables
  berkeley = "bears";
  let ucsd = "boring";
  var ucsb = "party";

  // Nested function demonstrating
  // lexical scoping
  function inner() {
    console.log(ucla);
    console.log(ucsd);
    let ucla = "nested";
    console.log(ucla);
  }

  inner();
  console.log(ucla);
  console.log(usc);
}

scopeDemo();

// After function execution
console.log(ucla);
console.log(usc);
console.log(berkeley);
console.log(ucsd);
```

Variable Scope

DO NOT use `var` in new code. It is function scoped and can lead to unexpected behavior.

Code Quality Checkers

When writing Javascript code:

1. You should use code quality checkers like `jslint` (web), `jshint` (web) or `eslint`.
2. You should prettify (web) your code so that it is readable.

More about Objects

A bit different from other languages, objects can be created without defining a class.

Object = Data + Method(s)

Data consist of properties that may be primitive or objects.

```
let pb_mood = {
  name: "Peanut Butter Moo'd",
  ingredients: ["peanut butter", "banana"],
  calories: 7000
}
pb_mood.make_diet = function(cals) { this.calories -= cals };
// note we could instead do pb_mood.calories -= cals
```

Into to Class (ES6)

A modern class syntax was introduced in 2015 with ES6.

```
class Animal {

  #habitat;
  #food;

  constructor(habitat) {
    this.#habitat = habitat;
  }

  get habitat() {
    return this.#habitat;
  }

  set habitat(newHabitat) {
    if (typeof newHabitat === "string" &&
        newHabitat.length > 0) {
      this.#habitat = newHabitat;
    } else {
      console.error("Invalid habitat");
    }
  }

  about() {
    return `habitat: ${this.habitat}`;
  }

  // Static method
  static isAnimal(obj) {
    return obj instanceof Animal;
  }
}
```

```
let cow = new Animal("pasture");
cow.habitat;
cow.habitat = 'barn';
cow.about();
Animal.isAnimal(cow);
```

Intro to Class Inheritance (ES6)

Of course, animals are hierarchical.

```
class Mammal extends Animal {
  constructor(habitat, species, sound) {
    super(habitat);
    this.species = species;
    this.sound = sound;
  }
  about() {
    return `${sound} I am a ${species}. I live in the ${habitat}.`;
  }
    // home is missing. We use the base class' method.
}
let c = new Animal("pasture", "cow", "moo");
c.home();
c.about();
```

A *Very* Quick Note on Prototypes

Objects have a *prototype*, which is another object that is used as a fallback when we query for a property that does not exist. We search for the property in each prototype in a hierarchy:

What is `Mammal`'s prototype?

What is `Animal`'s prototype?

A *Very* Quick Note on Prototypes (contd.)

Methods live on the **prototype** of the class, not the class itself.

Class

```
1  class Dog {
2    constructor(name) {
3      this.name = name;
4    }
5
6    bark() {
7      return `${this.name} ` +
8        `says woof!`;
9    }
10 }
11
12 const rover = new Dog("Rover");
13 rover.bark();  // OK
```

Prototype

```
1  function Dog(name) {
2    this.name = name;
3  }
4
5  Dog.prototype.bark = function() {
6    return `${this.name} says woof!`;
7  };
8
9  const rover = new Dog("Rover");
10 rover.bark();  // OK
```

Why? All instances of `Dog` share the same `bark` method. More memory efficient. All instances dynamically "receive" the method.

`this` Is Confusing

The `this` keyword is used in C++ and Java. In Javascript it causes a lot of confusion. `this` can be used:

We need to cover this. Please do not hate me.

This may save you a lot of pain.

`this` Is Confusing (contd.)

The Golden Rule

`this` is determined by how a function is called —— not how or where it's defined!

this Is Confusing (contd.)

In Methods (Object Context)

```
const dog = {
  name: "Rover",
  bark() {
    console.log(this.name);
  }
};
dog.bark();
```

Point: In the call `dog.bark()`, `this` refers to what is left of the dot.

this Is Confusing (contd.)

In Standalone Functions (Global Context)

```
function speak() {
  console.log(this);
}
speak();
```

Point: Don't do this. Here `this` is the global object (non-strict) or `undefined (strict)`.

this Is Confusing (contd.)

In Event Handlers

```
document.body.addEventListener("click", function() {
  console.log(this); // the element clicked (body)
});
```

Point: this is the element that the event listener is attached to (body in this case)

this Is Confusing (contd.)

In Arrow Functions

```
const dog = {
  name: "Rover",
  bark: () => {
    console.log(this.name);
  }
};
dog.bark();
```

Point: The arrow function does not bind `this`. It is bound to the enclosing scope, whatever that is. Here, `dog` is not a scope, it's a data structure. The global scope is most likely the enclosing scope.

this Is Confusing (contd.)

In Classes

```
class Dog {
  constructor(name) {
    this.name = name;
  }
  bark() {
    console.log(this.name);
  }
}
const rover = new Dog("Rover");
rover.bark();
```

Point: this is the instance of the class just like in C++ and Java.

Newer Features of ES

Rest Parameters

```
function greet(greeting, ...names) {
  const nameList = names.join(", ");
  console.log(`${greeting}, ${nameList}!`);
}
greet("Aloha!", "Alice", "Bob", "Charlie");
```

Spread

```
function aloha(...names) {
  const nameStr = names.join(", ");
  console.log(`Aloha, ${nameStr}!`);
}
const names = ['Alice', 'Bob', 'Carol', 'David'];
aloha(...names);
```

Array Destructuring

```
let colors = ['red', 'green', 'blue'];
let [r, g, b] = colors;
```

Object Destructuring

```
let instructor = {
  name: 'Bruin', dept: 'CS', age: 20
};
let { name, dept, age } = instructor;
```

Nullish Coalescing

```
let student = {
  name: "Joe Bruin",
  major: null
};
console.log(student.major ?? "Undeclared");
```

Optional Chaining

```
let catalog = {
  cs111: ["Eggert", "Reiher", "Kumar"],
  cs118: ["Lu", "Zhang", "Varghese"]
};
console.log(catalog.cs112?.instructors);
```

References

- Eloquent JavaScript
  - Concise, short chapters. To the point.
- Javascript: The Definitive Guide by David Flanagan
  - Canonical reference. Strongly recommended if you will be coding in Javascript a lot. Long winded.
- ECMAScript: ECMA 262
  - The oracle of truth.
  - Very boring read.
  - Browsers sometimes fall behind.
- JSON: ECMA 404

JavaScript Execution Model

As we discussed earlier, JavaScript used to be painfully slow. It was a frontend language **only**.

With the advent of JavaScript runtimes like Chrome's V8, JavaScript is now used on the backend ("full-stack").

Speed plus asyncronous processing allowed Node to flourish.

JavaScript Execution Model (contd.)

Each browser has a runtime. Node uses V8.

Nitro

SpiderMonkey

V8

JavaScript Execution Model (contd.)

JavaScript is:

1. **Single-threaded** just as with the browser — avoids race conditions
2. **Meant to be non-blocking** — no waiting for I/O
3. **Event-driven** — uses callbacks

JavaScript Execution Model (contd.)

So this means that JavaScript can only do one thing at a time, right?

Not quite. Standard JavaScript is not **parallel** — it is **concurrent**.

JavaScript can only execute one task at a time, but it can switch between tasks rapidly. This creates the illusion of concurrency, even though it's single-threaded. The mechanism that enables this is the **event loop**, which coordinates the scheduling of asynchronous tasks without blocking the main thread.

The Call Stack

The stack is where JavaScript keeps track of function calls.

1. When a function is called, we push it to the top of the stack.
2. When the function returns, we pop it off the stack.
3. Only one thing at a time can occur on the stack, thus single-threaded.

The Call Stack (contd.)

A function can be **synchronous** or **asynchronous**.

1. A **synchronous** function is executed immediately and blocks the call stack until it completes. These are usually quick.
   - Normal function calls: `foo()`, `bar(5)`.
   - Functions that initiate an asynchronous **callback** (e.g. `setTimeout`) are also synchronous
2. An **asynchronous** function is executed in the "background" and does not block the call stack. These are usually slow.
   - A callback is a function that executes after the asynchronous function completes.
   - This asynchronous processing function is **not** pushed to the stack.

The Call Stack (contd.)

An example of callbacks is the `setTimeout` function which is like an alarm clock. Chef can only do one thin at a time. Right now he is chopping steak, but he is also baking a cake. He can't tend to both at the same time.

He sets a timer, a callback. **When he finishes chopping the steak**, he checks if the timer has gone off. If it has, he takes the cake out of the oven.

The Call Stack (contd.)

JavaScript async is not about doing things in parallel. It's about deferring work until JavaScript is ready to handle it.

What do we do with this asynchronous processing? It, and the callback, are handed off to the **browser runtime not the JS runtime.** The browser may:

1. execute HTTP requests in background threads
2. execute disk I/O in use background threads
3. execute timers in background threads

But your JavaScript only runs on the **main thread**. When the processing is complete, the callback is added to a queue.

The Call Stack (contd.)

Callback functions are added to the queue as the processing by the browser runtime is compleed. **They stay there.**

They stay there until the call stack is empty. — no more processing.

How do we know when that is?

The Event Loop

So far, we have function calls on the stack which execute fully (but quickly), and longer running tasks that are handed off to the **browser runtime** and are processed asynchronously. Once complete, the **callback** is pushed to the back of a queue.

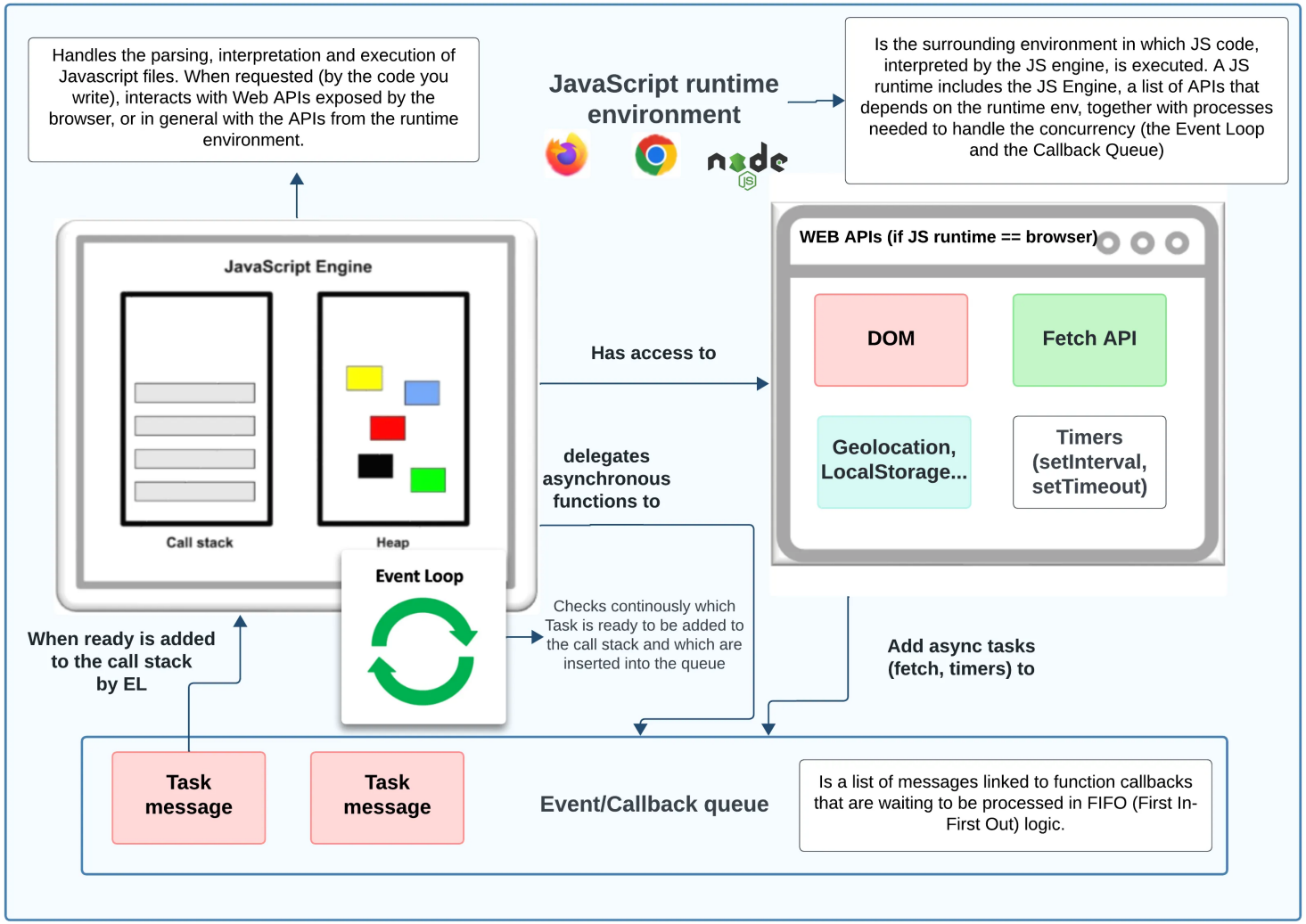
The event loop is infinitely running and checks if the stack is empty. If it is, it takes the first callback from the queue and pushes it to the stack.

Is JavaScript Really Non-Blocking?

JavaScript is non-blocking by design, not by nature. It only stays non-blocking if you avoid long-running synchronous code and use async APIs correctly.

JavaScript is only non-blocking if you let it be.

Execution Model



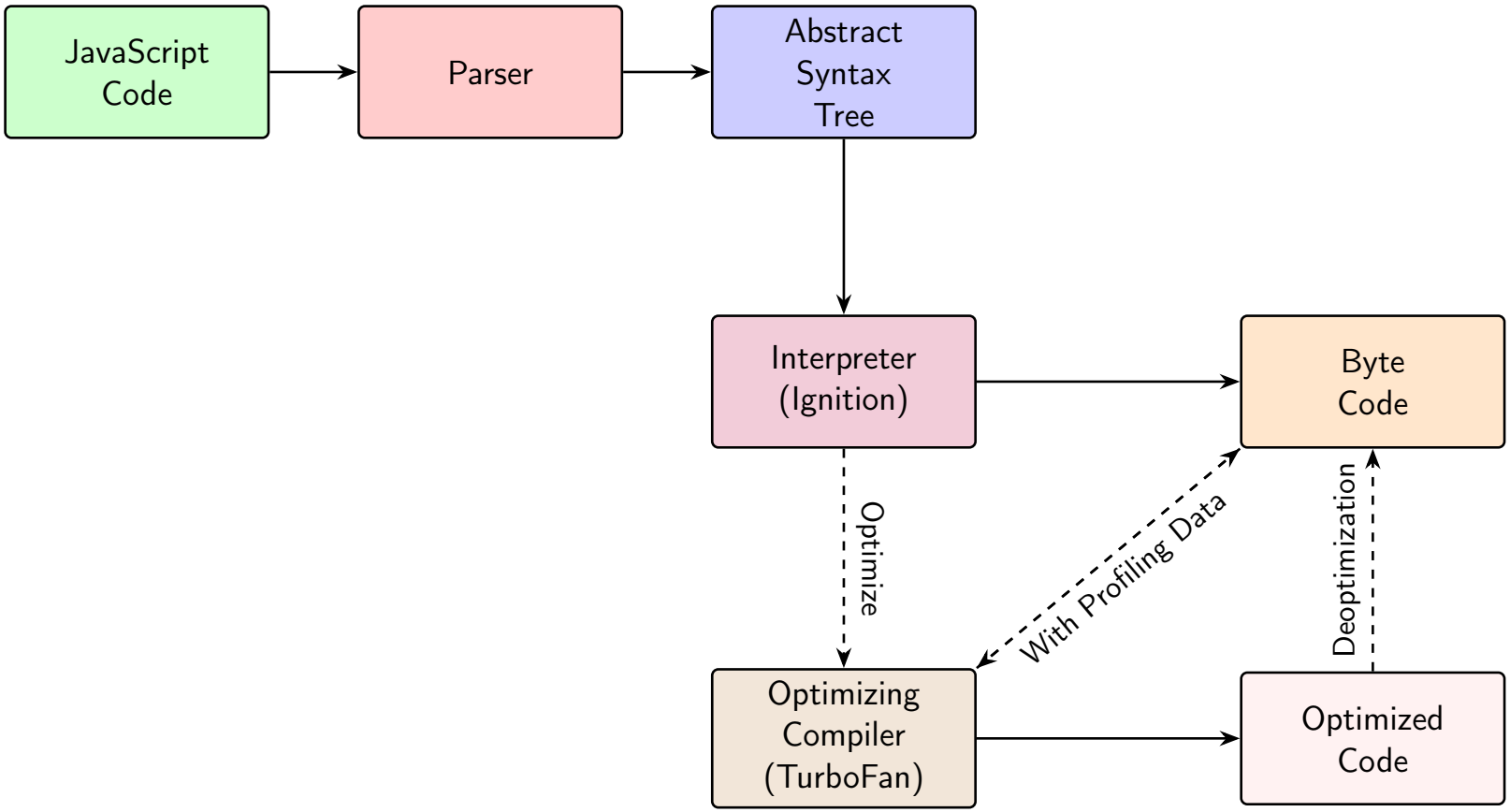
Source: Luca Di Molfetta

JavaScript Compilation Model

Even though JavaScript has historically been an interpreted language, modern JS is **compiled**, but at runtime.

This is called Just-In-Time (JIT) compilation.

Internals


JavaScript Code
Parser
Abstract Syntax Tree
Interpreter (Ignition)
Byte Code
Optimize
With Profiling Data
Deoptimization
Optimizing Compiler (TurboFan)
Optimized Code

V8 Analogy

V8 is named after a V8 engine — a powerful 8-cylinder engine commonly found in high-performance cars.

The components are named after parts, sort of.

Parsing and Abstract Syntax Tree

First, JS is parsed into an Abstract Syntax Tree (AST) and syntax is validated.

ByteCode Intepretation: Ignition

Ignition is JavaScripts *warm-up* system. It takes the AST and converts it into bytecode.

This bytecode is run by a lightweight stack-based interpreter. Quick to generate, slow to execute:

- It begins to execute code quickly.
- It collects profiling information (e.g. takes samples of executions)
- Decides whether or not to send the bytecode to TurboFan, the optimizing compiler.

Optimizing Compilation: TurboFan

TurboFan is JavaScript's racecar engine. It is a *hot* compiler that takes the bytecode and generates optimized machine code.

Hot code is code that is executed frequently.

- Ignition compiles to bytecode and starts interpreting
- Functions that run repeatedly are marked as hot
- TurboFan compiles those hot functions to optimized native code
- That optimized code is executed instead of bytecode

Optimizing Compilation: TurboFan (contd.)

What does optimized code mean? TurboFan:

- Inlines functions
- Removes unused variables
- Optimizes loops
- Performs constant folding (precomputing values)
- Eliminates type checks where stable types are observed

It uses type feedback gathered during Ignition's interpretation to specialize the machine code.

Optimizing Compilation: TurboFan (contd.)

Through these actions, TurboFan is making certain assumptions. When the assumptions are violated, the code is deoptimized and sent back to Ignition, or Ignition's bytecode is used. Deoptimization is a slow process.

This is an iterative process.

```
function double(x) {
    return x + x;
}

double(2);      // TurboFan sees `x` is a number
double(4);      // Still safe
double("hi");   // Assumption broken
```

TurboFan is like a pit crew that tunes your racecar engine to the track conditions.

Optimizing Compilation: TurboFan (contd.)

References

- JavaScript Execution Model
- JavaScript Execution Contexts and Event Loop
- Ignition and TurboFan Compiler Pipeline