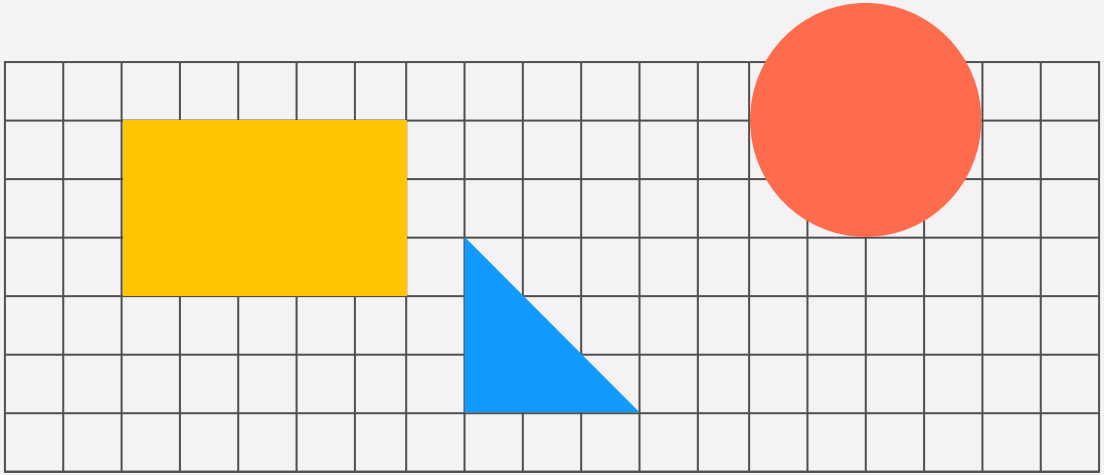
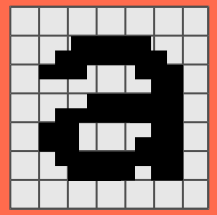
Beyond JavaScript

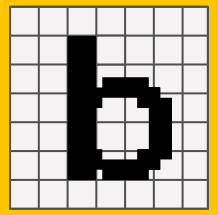
Ali Abrishami



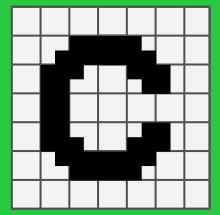
In This Lecture You Will...



Make console applications with JS



Learn how JS runs on servers



Solve the old JS typing issues

Limitations of JavaScript

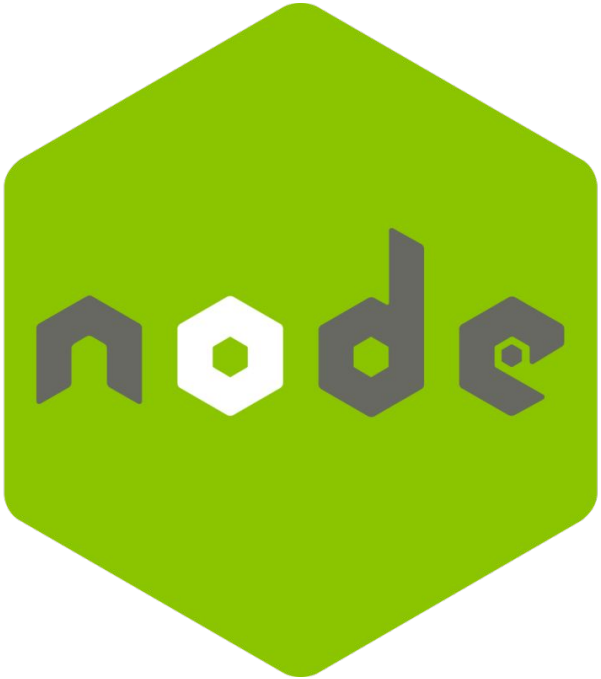
- Originally designed for simple browser interactions, not **full-scale applications** or **backend logic**.
- **Single-threaded** execution caused blocking issues during heavy computations or I/O operations.
- **Dynamic typing** led to unpredictable runtime errors and maintenance challenges in large projects.
- Early JavaScript lacked a proper **module system**, making code organization and reuse difficult.
- Limited tooling and poor scalability made building and maintaining complex applications hard.

Inception

- Developers wanted to reuse JavaScript skills beyond the browser, especially for **backend** logic.
- Web apps grew more dynamic, and client–server data exchange needed faster, unified handling.
- The rise of AJAX and JSON showed JS could efficiently handle **asynchronous** operations.
- **Engine improvements** like Google's V8 made JavaScript fast enough for server workloads.
- This shift inspired the idea of running JS outside the browser, starting the move toward **server-side** JavaScript.

Then Came Node.js

- Powered by Google's fast **V8 engine**.
- **Event-driven**, **non-blocking** I/O model.
- Same language for frontend and backend.
- Massive **npm ecosystem** for packages.
- Turned JavaScript into a **full platform**.



Node.js Features

- Built on **async/await** for cleaner, modern concurrency.
- Uses **ES** modules and import/export syntax natively.
- Embraces TypeScript, ESM, and modern tooling by default.
- Supports powerful frameworks like Express, NestJS, and Fastify.
- Integrates seamlessly with containers, cloud, and edge runtimes.

External Packages

- Open-source community began sharing **reusable** JS libraries.
- Developers could import external code instead of reinventing features.
- Enabled rapid development through **modular building blocks**.
- Popular libraries shaped best practices and common patterns.
- Also raised issues of trust, maintenance, and dependency overload.

NPM (Node Package Manager)

- Introduced a **central registry** to manage and distribute packages.
- Made installing and updating dependencies effortless.
- Standardized project setup with `package.json`.
- Fueled explosive growth of the JavaScript ecosystem.
- Later faced challenges with security, size, and version chaos.

NPM Commands

- `npm init`: Create a new project and generate a package.json file.
- `npm install <package>`: Add a dependency to your project.
- `npm uninstall <package>`: Remove a previously installed package.
- `npm update`: Update all dependencies to newer versions.
- `npm run <script>`: Execute custom scripts defined in package.json.

NPM Scripts

- Define custom commands in the scripts section of `package.json`.
- Run scripts with `npm run <script>` from the terminal.
- Can **automate tasks** like building, testing, or starting servers.
- Supports chaining commands and using environment variables.
- Widely used to **standardize project workflows** across teams.

Bundling for Browsers

- Combines multiple JS files into **a single output** for browsers.
- Reduces HTTP requests and improves load performance.
- Handles module resolution, imports, and exports automatically.
- Often includes **optimizations** like minification and tree-shaking.
- Popular tools: Webpack, Rollup, and Parcel.

Popular Packages

- **Express**: minimalist web framework for building APIs and servers.
- **React**: library for building interactive UIs with components.
- **Lodash**: utility functions for arrays, objects, and more.
- **Axios**: promise-based HTTP client for API requests.
- **Socket.io**: real-time bidirectional communication for web apps.

Popular Packages (cont.)

- **Moment.js / Day.js**: date and time manipulation utilities.
- **Jest**: testing framework for JavaScript and Node.js.
- **Three.js**: 3D graphics and animations in the browser.
- **Chalk**: style and colorize terminal output.
- **Electron**: build cross-platform desktop apps with web technologies.

TypeScript

JS Typing Issues

- JavaScript is **dynamically typed**, so variables can change type at runtime.
- **Type coercion** can lead to unexpected results (e.g., `"5" + 2 → "52"`).
- Harder to maintain large codebases due to hidden type bugs.
- Lacks **compile-time checks**, so many errors appear only during execution.
- Inconsistent behavior across engines and versions can introduce subtle bugs.

TypeScript

- Created by Microsoft in 2012.
- Anders Hejlsberg, lead architect of C# and Delphi, led its design.
- Developed to add **static typing and tooling** to JavaScript for large-scale projects.
- Designed to be a **gradual, optional superset**. developers can adopt it incrementally.
- Quickly gained popularity for enterprise applications and complex web apps.
- You write TypeScript → it becomes JavaScript → it runs anywhere JS runs.

Variables in TypeScript

- Use `let` and `const` like in JavaScript, but with optional type annotations.
- Example: `let age: number = 25;` ensures age is always a number.
- `const` creates immutable bindings, preventing reassignment.
- Supports union types: `let id: number | string`; can hold either type.
- TypeScript **infers types automatically** if not explicitly declared.

Interfaces

- Define the **shape of an object**: what **properties** and **types** it must have.
- Can be implemented by classes to enforce structure.
- Supports optional and readonly properties for flexibility and safety.
- Can be extended to create more complex types from simpler ones.

Functions

- Functions have **typed parameters** and return values for safety.
- Can include optional and default parameters.
- Supports rest parameters for variable-length arguments.
- Arrow functions provide concise syntax with type inference.

weee.ts

```ts
function createUser(
  name: string,
  age: number,
  ...hobbies: string[]
): { name: string; age: number; hobbies: string[] } {
  return {
    name,
    age,
    hobbies,
  };
}

const user = createUser("Alice", 30, "Reading", "Cycling");
console.log(user);
```

Classes

- Support object-oriented programming with constructors, **properties**, and methods.
- Can implement interfaces to enforce structure.
- Supports inheritance with `extends` and method overriding.
- **Access modifiers** (`public`, `private`, `protected`) control property and method visibility.

weee.ts

```ts
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, ${this.name}`);
  }
}
```

Just like JS, but better!

Generics

- Allow creating reusable, type-safe components or functions.
- Can be applied to functions, classes, and interfaces.
- Provide flexibility while still enforcing type safety.
- Prevents code duplication by working with any data type without losing type checking.

weee.ts

```ts
function identity<T>(value: T): T
{
  return value;
}

let num = identity<number>(42);
let str = identity<string>("Hello");
```

You remember generic from Java, right?

Best Practices

- Use **interfaces** to define contracts for classes and objects.
- Prefer `readonly` and `private` for encapsulation and immutability.
- Use inheritance sparingly; prefer composition when possible.
- Leverage generics for reusable, type-safe code.
- Keep classes small and focused; follow the Single Responsibility Principle (SRP).