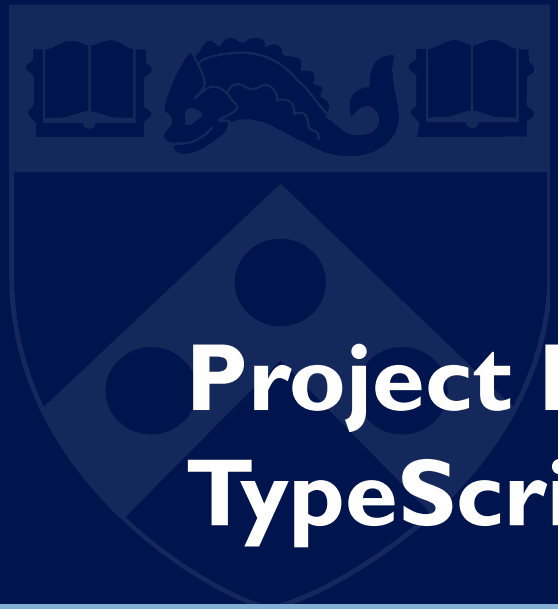


Project Management & Intro to TypeScript

CIS 1962 (Winter 2026)
January 29th, 2026

Penn Engineering

Lesson Plan

Project Management

6	CommonJS and ES6 Modules
15	Package Management
25	Linting and Formatting
30	Unit Testing

Intro to TypeScript

36	TypeScript
42	Type Syntax
52	Configuration

Weekly Logistics

Homework 1: Data Analysis DUE TONIGHT*

- *Late Policy Waived: You can submit up to February 1st without penalty

Homework 2: Project Scaffolding RELEASED

- Due on February 5th @ 11:59 PM
- No Starter code (you're setting it all up yourself!)

Review Activity

https://edstem.org/us/courses/91614/lessons/158392/slides/930663

Let's review some content from the previous lecture before we start!

Project Management

CommonJS and ES6 Modules

How do we make use of modules and CommonJS to allow better code organization?

Introduction to Modules

As JavaScript projects grow larger, developers need ways to encapsulate and organize code for better reusability and maintenance.

Rather than having all code in one single script file, we can break code up into smaller reusable pieces and only export and import what is necessary.

Two ways to do this are **CommonJS** and **ES6 Modules**.

CommonJS

CommonJS was a standard module system introduced with Node.js, that works synchronously for server-side JavaScript.

It does NOT work with browsers natively, it is only suitable for Node.

Syntax:

- Import: `require()`
- Export: `module.exports` or `exports` (**cannot** be mixed)

```
// math.js
function add(a, b) { return a + b;}

function subtract(a, b) { return a - b;}

module.exports = {
  add: add,
  subtract: subtract
};
```

```
// app.js
const math = require('./math');
console.log(math.add(2, 3)); // 5
console.log(math.subtract(5, 3)); // 2
```

ES6 Modules

ES6 Modules is an asynchronous, browser-compatible module system. It's becoming a more standard way to manage dependencies.

Syntax:

- Import: `import`
- Export: `export` for named exports, `export default` for singular exports (**can** be mixed)

```
// math.js
export function add(a, b) {
  return a + b;
}

export default function subtract(a, b) {
  return a - b;
}
```

```
// app.js

import sub, { add } from './math.js';

console.log(add(2, 3)); // 5
console.log(sub(5, 2)); // 3
```

ES6 Module Exports

```
// math.js
export function add(a, b) {
  return a + b;
}

export default function subtract(a, b) {
  return a - b;
}

// ALTERNATIVE SYNTAX (named exports)
// export {add, subtract}
```

```
// app.js

import sub, { add } from './math.js';

console.log(add(2, 3)); // 5
console.log(sub(5, 2)); // 3
```

You can only have 1 `export.default` per file, while you can have as many named `export` items as you want.

When importing:

- A default export will be associated with the file, thus you can rename the item
- A named export must be exported with its exact name in curly braces.

Using Modules

By default, a script file will use CommonJS. You can specify what module system you are using in 3 ways:

File Extensions

.cjs specifies CommonJS.
.mjs specifies ES6 Modules.

```
// math.mjs
export default function
subtract(a, b) {
  return a - b;
}
```

package.json

Within the package.json of a project, you can change the module system of an **entire project** to use ES6 modules.

```
// package.json
{
  "type": "module"
}
```

Browsers

If you are using a browser and an HTML file is available, you can specify ES6 modules with the script tag.

```
<script type="module"
src="main.js"></script>
```

Will it Blend?

While not recommended, .cjs modules can be imported into .mjs modules without problem.

However, .mjs modules cannot be imported into .cjs modules without some extra help.

```cjs
// math.cjs
export function add(a, b) {
  return a + b;
}

export default function subtract(a, b) {
  return a - b;
}
```

```mjs
// app.mjs
import math from "./math.cjs"
console.log(math.add(2, 3)); // 5
console.log(math.subtract(5, 3)); // 2
```

Static Analysis of ES6 Modules

Another benefit of ES6 modules is that they are statically analyzable due to imports and exports being fixed and at top level of scripts, not allowed inside functions/code blocks like CommonJS.

This provides benefits like:

- Code IntelliSense in IDEs
- Dependency Graphs
- Tree Shaking in web bundlers

Activity: ES6 Export/Import

https://edstem.org/us/courses/91614/lessons/158392/slides/930660

You've been given an app that organizes a small library system. Your task is to properly modularize the code using ES6 modules.

You are given 3 starter files: books.js, users.js, and libraryUtils.js. You must export the objects and functions inside these files, import them into app.js, and then print each user's fullName along with the title of their favorite book.

Package Management

What is package.json and how do we use it to manage projects and dependencies?

Packages and Dependencies

Modern JavaScript projects rely on many third-party libraries. These libraries provide reusable modules that solve problems and streamline programs.

Instead of manually managing modules, JavaScript has many **package managers** in order to handle code dependencies.

Packages & Package Managers

Packages are bundles of reusable code that may include modules, libraries, metadata, documentation, and configurations.

Packages in JavaScript are published to **package managers** like npm or yarn to be installed by other developers.

https://www.npmjs.com/

https://yarnpkg.com/

Activity: Explore lodash

https://www.npmjs.com/package/lodash

https://lodash.com/

Lodash is a popular package for working with arrays, objects, and strings in JavaScript.

Explore the npm page for lodash above for installation and documentation instructions!

package.json

The heart of a modern JS project is the `package.json` file.

It defines various items about a project:

- Metadata
- Scripts
- Dependencies

For working in teams on a project, sharing a `package.json` file is key to keeping up-to-date dependency installations.

package.json Structure

You can create a `package.json` using `npm init`.

This will prompt you to specify details about a project to populate the `package.json`. You can also write these details yourself, resulting in the file contents on the right:

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test
specified\" && exit 1"
  },
  "author": "voravich",
  "license": "ISC"
}
```

package.json Contents

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test
specified\" && exit 1"
  },
  "author": "voravich",
  "license": "ISC"
}
```

main: entry point of the package, the file that will be loaded when someone imports and runs the package

scripts: custom commands for certain tasks, run through `npm run <script_name>`, like `npm run test`.

dependencies: list of packages required at runtime

devDependencies: list of packages only used for development, like testing

Installing Dependencies with npm

`npm install <package>`: Installs a package as a dependency

`npm install <package> --save-dev`: Installs a package as a devDependency

`npm uninstall <package>`: Removes a package from the project

`npm update`: Updates all dependencies specified in the package.json to newer versions is available

All packages are installed into a local `node_modules` folder in the project, that is usually not pushed to git repos!

Custom Scripts

Sometimes you want to automate parts of your project that usually require complex terminal commands. This is where scripts come in handy.

Aside from `npm start` and `npm test`, custom scripts can be defined and run with `npm run <script>`.

```
"scripts": {
     "format": "prettier --write ./src",
     "lint": "eslint ./src",
     "dev": "serve ./src -p 3000"
},
```

Versioning and package-lock.json

When you install dependencies, a `package-lock.json` is automatically generated. This file records the exact versions of installed packages.

This file is useful for collaboration- to make sure multiple developers will have the exact same dependencies and dependency trees.

It also speeds up installation and improves debugging dependencies!

Linting and Formatting

What tools are available to improve code readability and keep consistent code style?

Linting and Formatting

Linting and Formatting code is a central part of many JavaScript projects, and is often automated during development.

Keeping consistent formatting and linting allows code to be more readable and consistent across members of a project.

Formatting with Prettier

Formatting involves making the code style of files consistent. This is often handled with the prettier package, which also uses a json file of rules. Formatting is often done automatically upon detecting errors consistent with the rules set.

https://prettier.io/

```
// .prettierrc.json
{
    "singleQuote": true,
    "semi": true,
    "useTabs": false,
    "tabWidth": 4,
    "endOfLine": "auto"
}
```

```
function foo(){console.log("bar")}
```

```
function foo() {
  console.log('bar');
}
```

Linting with ESLint

Linting is the act of analyzing code for errors and style issues. JavaScript popularly uses ESLint to detect errors with many built-in rules and extra libraries of rules.

https://eslint.org/docs/latest/rules/

```
rules: {
     'no-lonely-if': 'error',
     eqeqeq: 'error',
     'prefer-const': 'error',
     'no-var': 'error',
     'prefer-template': 'error',
     'prefer-arrow-callback': 'error',
     'no-alert': 'warn',
     'no-unused-vars': 'warn',
     'consistent-return': 'off'
},
```

Activity: Fix Bad Style

https://edstem.org/us/courses/91614/lessons/158392/slides/930667

Let's fix some bad style! The provided code has a badly formatted and styled file that we can fix using ESLint and prettier.

Unit Testing

How do we perform testing on JavaScript code to make sure it works as intended with specific inputs and outputs?

Testing in JavaScript

Like many other languages, JavaScript hosts a variety of ways to test your code.

As JavaScript was built for the web, in addition to unit testing functions that manipulate data within an application, we can also perform DOM testing to make sure certain items render correctly.

Unit Testing Tools

There are many libraries built to test code in JS, for instance:

- **Mocha**: Simple test runner without assertions
- **Jest**: All-in-one testing framework with a runner, assertions, mocks, and DOM manipulation.

```
// test.js
const assert = require('assert');
const { add } = require('../index.js');
describe('add', function() {
  it('adds two numbers', function() {
    assert.strictEqual(add(2, 3), 5);
  });
});
```

Mocha

```
// test.js
const { add } = require('../index.js');

test('adds two numbers', () => {
  expect(add(2, 3)).toBe(5);
});
```

Jest

Activity: Write some Tests!

We'll now install Jest to write some tests.

`npm i jest` (ignore the warnings)

Write your function and imports as CommonJS, as Jest requires some extra steps to use ES6 modules.

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

module.exports = {
    add
}
```

5-Minute Break!

Introduction to TypeScript

TypeScript

What is TypeScript and why should we learn to use it alongside JavaScript?

Types in JavaScript

We know JavaScript can be very loose with types. This poses a few problems:

- Functions can be unclear what types they should have as inputs and outputs
- The properties of an object aren't typed, thus can be overwritten with different types
- Type coercion causes unintended behaviors with type mixing
- Undefined and null method calls

TypeScript!

TypeScript is a **superset** of JavaScript to provide static typing to the language. As a superset, it contains all the features of JS with more syntax for types.

Unlike JavaScript, a .ts file using TypeScript must be transpiled into JavaScript before use.

TypeScript comes with its own compiler for this purpose.

TypeScript's Toolchain

Installation: `npm install typescript --save-dev`

File Paths: `.ts` marks a file to use TypeScript.

Compilation: `npx tsc` runs the compiler for TypeScript

- turns selected .ts files into .js files
- Can specify input and output directories in configurations

Configuration: `tsconfig.json` includes options to configure TS, including compiler options.

TypeScript in Action

```
function makeImportant(arr) {
  return arr.map((item) => `${item.toUpperCase()}!!!`);
}
// Problems:
// Parameter 'arr' implicitly has an 'any' type.
// Parameter 'item' implicitly has an 'any' type.
```

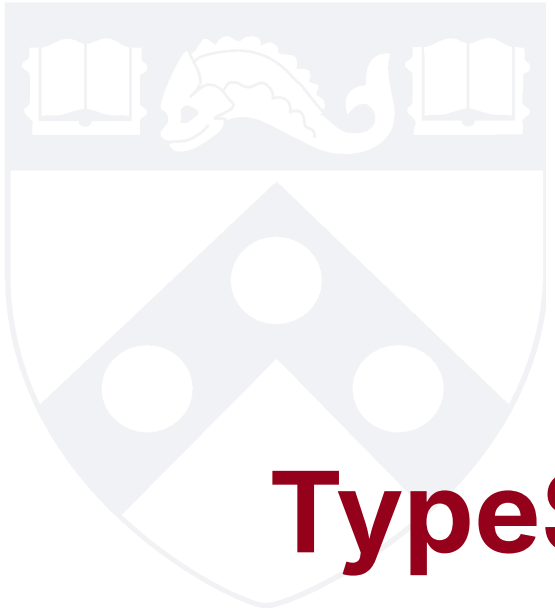
```
function makeImportant(arr: string[]) {
  return arr.map((item) => `${item.toUpperCase()}!!!`);
}
```

TypeScript in Action

```
function makeImportant(arr: string[]) {
  return arr.map((item) => `${item.toUpperCase()}!!!`);
}


const veryImportantPeople = makeImportant(["brennan",
"grant", "zac"])
// ^ Allowed, types match

const notSoImportantPeople = makeImportant(["voravich", 42])
// Problems:
// Type 'number' is not assignable to type 'string'.
```



TypeScript Type Syntax

How do we define types for various values within our code?

Implicit Typing and Inference

TypeScript is good at **inferring types**, even when not defined.

```
const x = 42 // number
const x: number = 42 // also a number
```

There are limits to this inference, if some value doesn't match any available types:

```
let x; // TypeScript doesn't know what type
to use, so it defaults to "any"
```

TypeScript Types

TypeScript has all of JavaScript's built-in types, arrays, and objects. This includes some extra syntax for Tuples, essentially fixed order arrays.

```
let age: number = 30;
let firstName: string = "Matthew";
let isActive: boolean = true;

let primes: number[] = [2, 3, 5, 7, 11];
let tuple: [string, number] = ["Alice", 42];

let user: { name: string; age: number } = {
  name: "Giuseppe",
  age: 57
};
```

Unique TypeScript Types

There are a variety of types unique to TypeScript:

```ts
// any: opts out of type checking
let value: any = 42;
value = "Hello";
value = { a: 1 };


// unknown: must be narrowed before use
let result: unknown =
couldBeStringOrNumber();
if (typeof result === "string") {
  console.log(result.toUpperCase());
}
```

```ts
// never: this value will never be reached
function endlessLoop(): never {
  while (true) { /* ... */ }
}


// void: for functions that don't return
anything
function logMessage(msg: string): void {
  console.log(msg);
}
```

Custom Types: type

You can define your own types with the `type` keyword.

Declaring with `type` is more flexible for unions and intersections.

```
type HasEmail = { email: string };
type HasPhone = { phone: string };

type Contact = HasEmail & HasPhone;

const support: Contact = { email:
"help@example.com", phone: "555-5555" };
```

Custom Types: interface

You can also define types with the `interface` keyword.

They allow unspecified fields and can be extended.

They cannot represent union/intersection types themselves.

```ts
interface Person {
  name: string;
  age: number;
  isStudent?: boolean; // optional
}

const julie: Person = {
  name: "Julie",
  age: 22,
};
```

```ts
interface Employee extends Person {
  employeeID: string;
}

const watson: Employee = {
  name: "Watson",
  age: 35,
  employeeID: "A1234",
};
```

Unions

A union type using the pipe | operator allows a value to be one of several types.

You can use either normal types or object literals, allowing items to behave like enums with only some specific allowed values.

```
type ID = string | number;

let userId: ID;

userId = "user42"; // OK
userId = 42;       // Also OK
// userId = true;  // Error!
```

```
type Compass = "north" | "east" |
"south" | "west";

let compass: Compass ;
compass = "north";    // OK
compass = "south"; // OK
compass= "northeast"; // Error!
```

Discriminated Unions

Unions allow us to make a unique type within TS called a **discriminated union**.

This allows a single type to represent multiple distinct types with a common property to discriminate between them.

```ts
interface Square {
  kind: 'square';
  size: number;
}

interface Rectangle {
  kind: 'rectangle';
  width: number;
  height: number;
}

interface Circle {
  kind: 'circle';
  radius: number;
}

type Shape = Square | Rectangle | Circle;
```

Intersections

An intersection type using the and & operator has all properties of the intersecting types. This includes methods as well.

Any items from an intersection must include all properties from both types.

```
type CanFly = { fly(): void };
type CanSwim = { swim(): void };

type Duck = CanFly & CanSwim;

const daffy: Duck = {
  fly() { console.log("Flying!"); },
};
// Problems:
// Type '{ fly(): void; }' is not
assignable to type 'Duck'.
// Property 'swim' is missing in type
'{ fly(): void; }' but required in type
'CanSwim'.
```

Functions and Narrowing

Functions arguments and return types can be defined as well.

Below you will see an example of **narrowing**, where an uncertain type, like one defined by a union, needs to be checked before use.

```
function padLeft(value: string, padding: number | string): string {
  if (typeof padding === "number") {
    return " ".repeat(padding) + value;
  } else {
    return padding + value;
  }
}

console.log(padLeft("hello", 5));
console.log(padLeft("hello", ">>> "));
```

Configuration

How do we configure TypeScript to perform properly?

tsconfig.json

The tsconfig.json file will define various options for how TypeScript interprets and tranpiles your code.

We'll go through a few `compilerOptions` here, but you can review them on your own at the link below:

https://www.typescriptlang.org/tsconfig/

Compiler Option: target

The **target** option allows you to define what JavaScript feature set to target for code outputted through compiling.

es6, es2016, es2020, or es2022 are good options.

```
"compilerOptions": {
    "target": "es2016",
    ...
}
```

Compiler Option: lib

The **lib** option allows you to define JavaScript APIs that are available in your TypeScript codebase for types, in including items like "DOM", "ESNext", and "WebWorker"

```
"compilerOptions": {
    "lib": ["ESNext", "DOM"]
    ...
}
```

Compiler Option: module

`module` determines what system the emitted code uses, either "commonjs" for Node projects and "es6" for browser projects.

```
"compilerOptions": {
    "module": "es6"
    ...
}
```

Compiler Option: strict

`strict` is by default false in TypeScript.

Set to true, this enables a lot more type-checking rules to find exceptions and errors. This also fixes some issues due to type strictness and narrowing.

```
"compilerOptions": {
    "strict": true
    ...
}
```

Compiler Option: outDir

`outDir` defines what directory the transpiled code will be output to. If none is specified, code will be output to a directory called "dist".

```
"compilerOptions": {
    "outDir": "my_folder"
    ...
}
```

Top-Level: includes

Alongside `compilerOptions`, `includes` is another top level configuration that defines what folder to target for compilation. By default the compiler will target any .ts files in the working directory.

```
{
  "compilerOptions": {
    ...
  },
  "include": ["my_project_dir"]
}
```

Activity: Play with TypeScript

Let's show the process of using TypeScript, both in Ed Lessons and VSCode (to show compilation).

Imagine you're building a message system for a basic messaging app using TypeScript. In the following activities, you will define the types for this system.

https://edstem.org/us/courses/91614/lessons/158392/slides/930677

More TypeScript!

TypeScript has many more features that aim to fix many type issues with JS and provide lots of type utilities:

- Class Syntax (that works with JS classes)
- Utility Types
- Generics
- Enums, conditional types, mapped types, template union types

Some examples and exercises for these will be posted on Ed Lessons.

TypeScript Cheat Sheet!

Overwhelmed with syntax? Use the cheat sheet!

https://www.typescriptlang.org/cheatsheets/

The TypeScript documentation provides a cheatsheet for various syntax, including a lot of what we covered today.