Organizing Your Code with Modules

Brice Wilson

Overview

Why use modules?

Supporting technologies

Import and export syntax

Module resolution

Why Use Modules?

Encapsulation

Reusability

Create higher-level abstractions

Supporting Technologies

TypeScript Compiler

JavaScript Modules
- AMD
- CommonJS
- UMD
- System
- ES2015

Loader/Bundler
- Node
- RequireJS
- SystemJS
- Webpack

Exporting a Declaration

```
// person.ts

export interface Person { }

export function hireDeveloper(): void { }

export default class Employee { }

class Manager { } // not accessible outside the module
```

Export Statements

```ts
// person.ts

interface Person { }

function hireDeveloper(): void { }

class Employee { }

class Manager { } // not accessible outside the module

export { Person, hireDeveloper, Employee as StaffMember };
```

Importing from a Module

```
// player.ts
import { Person, hireDeveloper } from './person';
let human: Person;

import Worker from './person';
let engineer: Worker = new Worker();

import { StaffMember as CoWorker } from './person';
let emp: CoWorker = new CoWorker();

import * as HR from './person';
HR.hireDeveloper();
```

Relative vs. Non-relative Imports

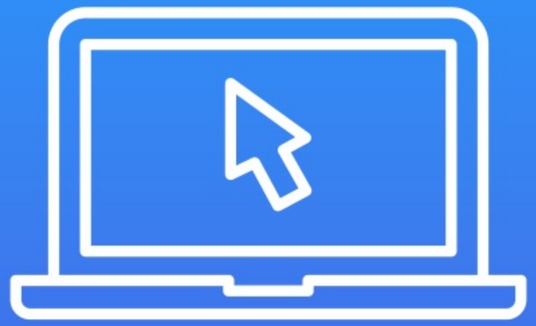
```
// relative imports
import { Laptop } from '/hardware';
import { Developer } fro  './person';
import { NewHire } from  ../HR/recruiting';

// non-relative imports
import * as $ from 'jquery';
import * as lodash from 'lodash';
```

Module Resolution Strategies

```
tsc --moduleResolution Classic | Node
```

Classic	vs	Node
Default when emitting AMD, UMD, System, or ES2015 modules		Default when emitting CommonJS modules
Simple		Closely mirrors Node module resolution
Less configurable		More configurable

Demo



Converting the demo app to use modules

Summary

Modules provide higher-level abstractions

Simple syntax

Flexible usage

Configurable resolution strategies

Up Next:

Writing Asynchronous Code