

Lecture 06

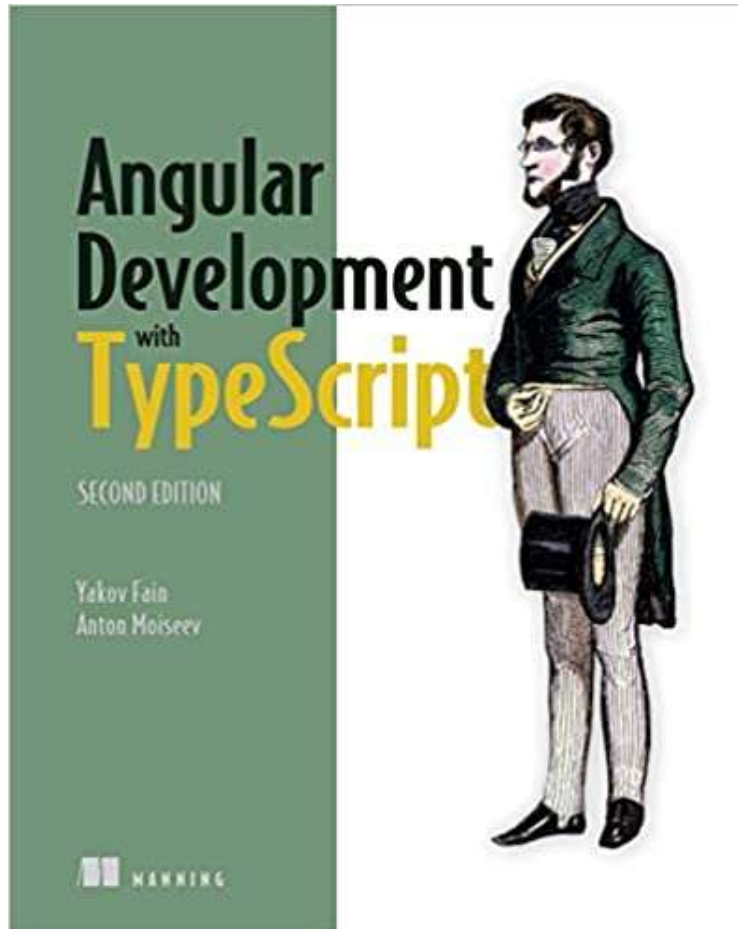
# TypeScript & Angular

Cs701 A1, Rich Internet Application Development  
Spring 2022

Zoran B. Djordjević

# Reference

- This lecture follows Appendix C and Appendix B of book:  
Angular Development with TypeScript, 2<sup>nd</sup> Edition,  
by Yakov Fain & Anton Moiseev, O'Reilly 2019



# What is Angular

- Angular is a development platform, built on TypeScript.
- As a platform, Angular includes:
  - A component-based framework for building scalable web applications
  - A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more
  - A suite of developer tools to help you develop, build, test, and update your code
  - When you build applications with Angular, you're taking advantage of a platform that can scale from single-developer projects to enterprise-level applications. Angular is designed to make updating as easy as possible, so you can take advantage of the latest developments with a minimum of effort. Best of all, the Angular ecosystem consists of a diverse group of over 1.7 million developers, library authors, and content creators.
- Angular originated in the previous framework called AngularJS
- For code samples, documentation and many useful links and references, please go to <https://angular.io/>
-

# What is TypeScript

## JavaScript and More

- TypeScript is an open-source language which builds on JavaScript, one of the world's most used tools, by adding static type definitions.
- Types provide a way to describe the shape of an object, providing better documentation, and allowing TypeScript to validate that your code is working correctly.
- Writing types can be optional in TypeScript, because type inference allows you to get a lot of power without writing additional code.
- All valid JavaScript code is also TypeScript code. You might get type-checking errors, but that won't stop you from running the resulting JavaScript. While you can go for stricter behavior, that means you're still in control.
- TypeScript code is transformed into JavaScript code via the TypeScript compiler or Babel. This JavaScript is clean, simple code which runs anywhere JavaScript runs: In a browser, on Node.JS or in your apps.
- For code samples, documentation and many useful links and tool references, please go to: <https://www.typescriptlang.org/>
- Typescript was originally developed in 2012 by Microsoft. Its core developer was Anders Hejlsberg.

# Why Angular and TypeScript

The main reasons that Angular and TypeScript are the right tools for developing web applications are:

- **Clean separation of UI and app logic.** There's a clean separation between the code that renders the UI and the code that implements application logic. The UI doesn't have to be rendered in HTML, and there are already products that support native UI rendering for iOS and Android.
- **Modularization.** There's a simple mechanism for modularizing applications with support for the lazy loading of modules.
- **Navigation.** The router supports complex navigation scenarios in single-page applications.
- **Loose coupling.** Dependency injection gives you a clean way to implement loose coupling between components and services. Binding and events allow you to create reusable and loosely coupled components.
- **Component lifecycle.** Each component goes through a well-defined lifecycle, and hooks for intercepting important component events are available for application developers.
- **Change detection.** An automatic (and fast) change-detection mechanism spares you from the need to manually force UI updates while providing you a way to fine-tune this process.
- **No callback hell.** Angular comes with the RxJS library, which allows you to arrange subscription-based processing of asynchronous data, eliminating callback hell.
- **Forms and validation.** Support for forms and custom validation is well designed. You can create forms either by adding directives to form elements in the template or programmatically.

# Why Angular and TypeScript

- **Testing.** Unit and end-to-end testing are well supported, and you can integrate tests into your automated build process.
- **Tooling.** Angular is well supported by IDEs. The TypeScript code analyzer warns you about errors as you type. Angular CLI quickly generates a new project and various artifacts (such as components and services), comes with a web server, and performs dev and prod builds, sparing developers from dealing with configuration scripts.
- **Concise code.** Using TypeScript classes and interfaces makes your code concise and easy to read and write.
- **Compilers.** The TypeScript code can be compiled into ES3, ES5, or the latest versions of JavaScript. Ahead-of-time compilation of app templates makes the initial app rendering faster and eliminates the need to package the Angular compiler with your app.
- **Server-side rendering.** Angular Universal turns your app into HTML in an offline build step that can be used for server-side rendering, which in turn greatly improves indexing by search engines and SEO.
- **Modern-looking UI components.** Angular Material offers more than 35 well designed UI components.

# Node.js and npm

# Node.js, npm

- Node.js (or simply Node) is a framework and a library.
- Node.js is also a JavaScript runtime environment.
- Node.js can be used to develop JavaScript programs that run outside the browser. You can develop the server-side layer of a web application in JavaScript or Typescript.
- We can create a web server using Node.js and Express frameworks.
- Google developed a high-performance V8 JavaScript engine for the Chrome browser, and it can also be used to run code written using the Node.js API.
- The Node.js framework includes an API to work with the file system, access databases, listen to HTTP requests, and more.
- We use the Node runtime for running various utilities like npm or launching JavaScript code without a browser. npm is Node.js package manager. We use npm scripts to automate building, testing, and deploying Angular apps.
- Go to <https://nodejs.org/en/download/> and download Node.js installer for your operating system. On Windows 10, I am using `node-v16.14.0-x64.msi`
- The installation also installs npm and adds Node.js and npm to the PATH.
- I allowed the installation of the newest version of Python, 3.9.2, and tools for Visual Studio Code. That is up to you.



# Verify Node and npm are there

- On the command prompt, type:

```
C:\Users\Zoran>node
Welcome to Node.js v16.14.0.
Type ".help" for more information.
>> .help
.break      Sometimes you get stuck, this gets you out
.clear      Alias for .break
.editor      Enter editor mode
.exit        Exit the REPL
.help        Print this help message
.load        Load JS from a file into the REPL session
.save        Save all evaluated commands in this REPL session to a file
Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
>
```

# Verify npm is there

```
C:\Users\Zoran>npm
```

```
Usage: npm <command>
```

```
where <command> is one of:
```

```
access, adduser, audit, bin, bugs, c, cache, ci, cit, clean-install,  
clean-install-test, completion, config, create, ddp, dedupe, deprecate, dist-tag,  
docs, doctor, edit, explore, fund, get, help, help-search, hook, i, init,  
install, install-ci-test, install-test, it, link, list, ln, login, logout,  
ls, org, outdated, owner, pack, ping, prefix,  
profile, prune, publish, rb, rebuild, repo, restart, root,  
run, run-script, s, se, search, set, shrinkwrap, star,  
stars, start, stop, t, team, test, token, tst, un,  
uninstall, unpublish, unstar, up, update, v, version, view,  
whoami
```

```
npm <command> -h  quick help on <command>
```

```
npm -l            display full usage info
```

```
npm help <term>   search for help on <term>
```

```
npm help npm      involved overview
```

```
Specify configs in the ini-formatted file:
```

```
C:\Users\Zoran\.npmrc
```

```
or on the command line via: npm <command> --key value
```

```
Config info can be viewed via: npm help config
```

```
npm@6.14.11 D:\ProgramData\nodejs\node_modules\npm
```

```
C:\Users\Zoran>node --version
```

```
v16.14.0
```

# Installing TypeScript

- To install a JavaScript library, we run the command `npm install`, or `npm i` for short.
- To install the TypeScript compiler locally, open the terminal in any directory and type `.\> npm i typescript`
- After this command completes, you'll see a new subdirectory named `node_modules`, where the TypeScript compiler has been installed. `npm` always installs packages in the `node_modules` directory. If such a directory doesn't exist, `npm` will create it.
- If you want to install a package globally, add the `-g` option. You could first uninstall that local installation by typing: `npm uninstall typescript`. Now proceed with global installation.

```
npm i typescript -g
```

- This time the TypeScript compiler will be installed not in the current directory, but globally in the `lib/node_modules` subdirectory of your Node.js installation.
- If you want to install a specific version of the package, add the version number to the package name after the `@` sign. For example, to install Typescript 2.9.0 globally, use the following command:
- `npm i typescript@2.9.0 -g`
- All available options of the `npm install` command are described at <https://docs.npmjs.com/cli/install>

```
tsc -version      # will tell you the version of the install compiler.
```

# Multiple versions of `tsc` compiler

- In some cases, you may want to have the same package installed both locally and globally. As an example, you may have the TypeScript compiler 2.7 installed locally, and TypeScript 2.9 installed globally.
- To run the global version of this compiler, you enter the `tsc` command in your terminal or command window, and to run the locally installed compiler, you could use the following command from your project directory:

```
node_modules/.bin/tsc
```

- A typical Node-based project may have multiple dependencies, and we don't want to keep running separate `npm i` commands to install each package.
- Creating a `package.json` file is a better way to specify all project dependencies.

# Transpile TypeScript to JavaScript

- Code written in TypeScript has to be *transpiled* into JavaScript so web browsers can execute it.
- TypeScript code is saved in files with the `.ts` extension. Say you write a script and save it in the `main.ts` file. The following command will transpile `main.ts` into `main.js`:

```
tsc main.ts
```

# Specifying project dependencies in `package.json`

- To start a new Node-based project, create a new directory (e.g., `my-project`), open your terminal or command window, and change the current working directory to the newly created one.
- Then run the `npm init -y` command, which will create the initial version of the `package.json` configuration file.
- Normally, `npm init` asks several questions while creating the file, but the `-y` flag makes it accept the default values for all options. The following example shows this command running in the empty `my-node-project` directory:

```
$ npm init -y
```

```
Wrote to D:\code3\cs701A1\lec07\package.json:
```

```
{
  "name": "lec07",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

# Custom package.json

- Most of the generated configuration is needed for publishing the project into npm registry or while installing the package as a dependency for another project. We use npm only for managing project dependencies and for automating development and build processes.
- Because we will not publish new package into the npm registry, remove all the properties except name, description, and scripts. Also, add a "private": true property, because it's not created by default. That will prevent the package from being accidentally published to the npm registry. The package.json file should look like this:

```
{  
  "name": "my-project",  
  "description": "",  
  "private": true,  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  }  
}
```

# Command Aliases

- The scripts configuration allows you to specify command aliases that you can run in your command window. By default, `npm init` creates the `test` alias, which can be run like:  
`npm test`
- The generated script command include double ampersands, `&&`, which are used as a separator between commands. When you run `npm test`, it runs two commands: `echo \"Error: no test specified\"` and `exit -1`.
- `npm` scripts support about a dozen command names like `test`, `start`, and others. A list of these commands can be found at <https://docs.npmjs.com/misc/scripts>.
- You can create your own command aliases with any names:

```
"scripts": {  
  "deploy": "copyfiles -f dist/** ../server/build/public",  
}
```

- Because `deploy` is a custom alias name, you need to run this command by adding the keyword `run`:

```
npm run deploy
```



# Specifying Dependencies

- When we generate `package.json` using the `npm init` command, it'll be missing two important sections: `dependencies` and `devDependencies`. Figure bellow shows a fragment from a `package.json` file of a typical Angular project with specified dependencies.
- You don't need to remember all these packages and their versions, because you'll be generating projects with Angular CLI, which will create the file `package.json` with the proper content.
- The `dependencies` section lists all the packages required for your application to run. As you can see, the Angular framework comes in several packages with names starting with `@angular`. You may not need to install all of them. If your app doesn't use `forms`, for example, there's no need to include `@angular/forms` in your `package.json`.
- The `devDependencies` section lists all the packages required to be installed on a developer's computer. It includes several packages, and none of them is needed on the production server. There is no need to have testing frameworks or a TypeScript compiler on the production machine, for example.

```
"dependencies": {
  "@angular/animations": "^6.0.0",
  "@angular/common": "^6.0.0",
  "@angular/compiler": "^6.0.0",
  "@angular/core": "^6.0.0",
  "@angular/forms": "^6.0.0",
  "@angular/http": "^6.0.0",
  "@angular/platform-browser": "^6.0.0",
  "@angular/platform-browser-dynamic": "^6.0.0",
  "@angular/router": "^6.0.0",
  "core-js": "^2.5.4",
  "rxjs": "^6.0.0",
  "ws": "^5.2.0",
  "zone.js": "^0.8.26"
},
"devDependencies": {
  "@angular/compiler-cli": "^6.0.0",
  "@angular-devkit/build-angular": "~0.6.1",
  "typescript": "~2.7.2",
  "@angular/cli": "~6.0.1",
  "@angular/language-service": "^6.0.0",
  "@types/jasmine": "~2.8.6",
  "@types/jasminewd2": "~2.0.3",
  "@types/node": "~8.9.4",
  "codemlizer": "~4.2.1",
  "jasmine-core": "~2.99.1",
  "jasmine-spec-reporter": "~4.2.1",
  "karma": "~1.7.1",
  "karma-chrome-launcher": "~2.2.0",
  "karma-coverage-istanbul-reporter": "~1.4.2",
  "karma-jasmine": "~1.1.1",
  "karma-jasmine-html-reporter": "~0.2.2",
  "protractor": "~5.3.0",
  "ts-node": "~5.0.1",
  "tslint": "~5.9.1"
}
```

# Installing packages

- To install a single package with `npm`, list the name of this package. For example, to add the `lodash` library to your `node_modules` directory, run the following command:

```
npm i lodash
```

- After this command, `project.json`, reads like:

```
{  "name": "lec07",    "description": "",
  "scripts": {      "test": "echo \"Error: no test specified\" && exit 1"
    },  "private": true,
  "dependencies": {
    "lodash": "^4.17.21"
  }
}
```

- To add the package to `node_modules` and add the corresponding dependency to the `dependencies` section of your `package.json` file, explicitly specify the `--save-prod` option :

```
npm i lodash --save-prod
```

- You can also use the abbreviation `-P` for `--save-prod`. If no options are specified, the
- `npm i lodash` command updates the `dependencies` section of the `package.json` file.
- To add the package to `node_modules` and add the corresponding dependency to the `devDependencies` section of your `package.json` file, use the `--save-dev` option:

```
npm i protractor --save-dev
```

- You can also use the abbreviation `-D` for `--save-dev`.

# Fetching the latest packages from GitHub

- Sometimes the GitHub version of a package has an important bug fix that hasn't been released yet on `npmjs.org`.
- If you want install such a package from GitHub, you need to replace the version number in `package.json` dependencies with its location on GitHub.
- Changing the dependency to include the name of the GitHub organization and repository should allow you to install the latest builds from GitHub versions of this library. For example:

```
"@angular/flex-layout": "angular/flex-layout-builds"
```

- The preceding configuration will work, assuming that the master branch of the `flex-layout` library has no code issues preventing `npm` from installing it.

# Semantic versioning

- Numbering of Angular releases uses a set of rules known as semantic versioning.
- The version of a package consists of three digits—for example, 6.1.2.
- **The first digit** denotes a *major release* that includes new features and potentially breaking changes in the API. **The second digit** represents a *minor release*, which introduces new backward-compatible APIs but has no breaking changes.
- **The third digit** represents *backward-compatible patches with bug fixes*.
- In slide 17 every package has a three-digit version, and many of them have additional symbols: ^ or ~. If the specified version has just the three digits, that means you instruct npm to install exactly that version. For example, the following line in `package.json` tells npm to install Angular CLI version 6.0.5, ignoring the newer versions even if they're available:

```
"@angular/cli": "6.0.5"
```

- Many packages in that `package.json` file have the hat sign ^ in front of the version. For example:

```
"@angular/core": "^6.0.0"
```

- This means you allow npm to install the latest minor release of version 6 if available. If the latest version of the Angular Core package is 6.2.1, it will be installed.
- The tilde ~ means you want to install the latest patch for the given major and minor version:

```
"jasmine-core": "~2.99.1"
```

- There are many other symbols you can use with version numbers with npm—see <http://mng.bz/YnyW> for details.

# Yarn as an alternative to npm

- Yarn (see <https://yarnpkg.com>) is yet another package manager that can be used as an alternative to npm. Prior to version 5, npm was slow, which was one reason developers were using the faster Yarn.
- Now npm is fast too, but Yarn has an additional benefit: it creates the file `yarn.lock` that keeps track of exact versions of packages that were installed in your project. Let's say your `package.json` file has a `"@angular/core": "^6.0.0"` dependency, and your project has no `yarn.lock` file. If version 6.1.0 is available, it'll be installed, and `yarn.lock` is created with a record about the version 6.1.0. If you run `yarn install` in a month, and if `yarn.lock` exists in your project, Yarn will use it and install version 6.1.0, even if version 6.2.0 is available.
- The following fragment from `yarn.lock` shows that although the dependency in `package.json` for the `@angular/core` package was set as `^6.0.0`, version 6.0.2 was installed:  

```
"@angular/core@^6.0.0":  
  version "6.0.2"  
  resolved "https://registry.yarnpkg.com/@angular/core/-/core-  
    6.0.2.tgz#d183..."  
  dependencies:  
    tslib "^1.9.0"
```
- In a team setup, you should check the `yarn.lock` file into the version control repository so every member of your team has the same versions of packages.

# npm package-lock.json

- npm also creates the package-lock.json file, but npm isn't designed to install exact package version(s) listed in this file if you run npm install (see <https://github.com/npm/npm/issues/17979>). The good news is that starting from version 5.7, npm supports the npm ci command, which ignores the versions listed in package.json, but installs the versions listed in the package-lock.json file.
- If at some point you decide to upgrade packages, overriding the versions stored in yarn.lock, run the yarn upgrade-interactive command, as shown in figure below:

```
$ yarn upgrade-interactive
yarn upgrade-interactive v1.3.2
info Color legend :
"<red>"      : Major Update backward-incompatible updates
"<yellow>"   : Minor Update backward-compatible features
"<green>"    : Patch Update backward-compatible bug fixes
? Choose which packages to update. (Press <space> to select, <a> to toggle all, <i> to inverse selection)
dependencies
  name                range      from    to      url
>o @angular/animations ^5.0.0    5.1.2  > 5.2.0  https://github.com/angular/angular#readme
o @angular/common     ^5.0.0    5.1.2  > 5.2.0  https://github.com/angular/angular#readme
o @angular/compiler   ^5.0.0    5.1.2  > 5.2.0  https://github.com/angular/angular#readme
o @angular/core       ^5.0.0    5.1.2  > 5.2.0  https://github.com/angular/angular#readme
o @angular/forms      ^5.0.0    5.1.2  > 5.2.0  https://github.com/angular/angular#readme
```

- Yarn works with your project's package.json file, so there's no need for any additional configuration. You can read more about using Yarn at <https://yarnpkg.com/en/docs>

# TypeScript

# TypeScript

- TypeScript was released in 2012 by Microsoft, and its core developer was Anders Hejlsberg. He's also one of the authors of Turbo Pascal and Delphi, and is a lead architect of C#. We will cover main elements of the TypeScript syntax.
- We will also learn how to turn TypeScript code into JavaScript (ES5) so it can be executed by any web browser or a standalone JavaScript engine.
- For a complete TypeScript documentation visit [www.typescriptlang.org/docs/home.html](http://www.typescriptlang.org/docs/home.html)



# TypeScript as a superset of JavaScript

- TypeScript supports ES5, ES6, and newer ECMAScript syntax.
- Just change the name extension of a file with JavaScript code from `.js` to `.ts`, and it'll become valid TypeScript code. Being a superset of JavaScript, TypeScript adds a number of useful features to JavaScript.

# How to run the code samples

- To run the code samples from appendix B of the reference book, locally on your computer, perform the following steps:
  - 1 Install Node.js from <https://nodejs.org/en/download/> (use the current version).
  - 2 Clone or download the <https://github.com/Farata/angulartypescript> repository into any directory.
  - 3 In the command window, change into this directory, and then go to the `code-samples/appendixB` subdirectory.
  - 4 Install the project dependencies (the TypeScript compiler) locally by running  
`npm install`.
  5. Use the locally installed TypeScript compiler to compile all code samples into the `dist` directory by running  
`npm run tsc`,  
which will transpile all code samples from the `src` directory into the `dist` directory.
  6. To run a particular code sample (such as `fatArrow.js`) use the following command:  
`node dist/fatArrow`

# Transpiling

- Web browsers do not understand any language but JavaScript. If the source code is written in TypeScript, it has to be *transpiled* into JavaScript before you can run it in a JavaScript engine, whether browser or standalone.
- *Transpiling* means converting the source code of a program in one language into source code in another language.
- Many developers prefer to use the word compiling, so phrases like “TypeScript compiler” and “compile TypeScript into JavaScript” are also valid.
- Figure below shows TypeScript code on the left and its equivalent in an ES5 version of JavaScript generated by the TypeScript transpiler on the right. In TypeScript, we declared a variable `foo` of type `string`, but the transpiled version doesn't have the type information. In TypeScript, we declared a class `Bar`, which was transpiled in a class-like pattern in the ES5 syntax.
- You can try the code in TypeScript playground at [www.typescriptlang.org/play](http://www.typescriptlang.org/play). If we had specified ES6 as a target for transpiling, the generated JavaScript code would look different; you'd see the `let` and `class` keywords on the right side as well.



The screenshot shows the TypeScript Playground interface. At the top, there are tabs for 'TypeScript', 'Share', 'Options', 'Run', and 'JavaScript'. The 'TypeScript' tab is active, showing the following code:

```
1 let foo: string;
2
3 class Bar {
4
5 }
```

The 'JavaScript' tab is also active, showing the transpiled ES5 code:

```
1 var foo;
2 var Bar = (function () {
3     function Bar() {
4     }
5     return Bar;
6 }());
```

# Benefit of TypeScript

- A combination of Angular with statically typed TypeScript simplifies the development of web applications.
- Good tooling and a static type analyzer substantially decrease the number of runtime errors and shorten the time to market.
- When complete, your Angular application will have lots of JavaScript code; and although developing in TypeScript may require you to write a little more code, you will reap benefits by saving time on testing and refactoring and minimizing the number of runtime errors.

# Starting with TypeScript

- Microsoft has open sourced TypeScript and hosts the TypeScript repository on GitHub at <https://github.com/Microsoft/TypeScript/wiki/Roadmap>.
- You can install the TypeScript compiler using npm.
- The TypeScript site [www.typescriptlang.org](http://www.typescriptlang.org) has the language documentation and has a web-hosted TypeScript compiler (under the Playground menu), where you can enter TypeScript code and compile it to JavaScript interactively.
- Enter TypeScript code on the left, and its JavaScript version (ES5) is displayed on the right.
- Click the Run button to execute the transpiled code (open the browser console to see the output produced by your code, if any).
- Such interactive tools will suffice for learning the language syntax, but for real-world development, you need better tooling to be productive.
- You may decide to use an IDE or a text editor but having the TypeScript compiler installed locally is a must for development.
- You have installed the TypeScript compiler and can run code samples using the Node JavaScript engine.
- You have Node.js and npm installed on your computer.

# Transpiling Code

- Code written in TypeScript has to be transpiled into JavaScript so web browsers can execute it. TypeScript code is saved in files with the .ts extension. Say you write a script and save it in the main.ts file. The following command will transpile main.ts into main.js:

```
tsc main.ts
```

- During compilation, TypeScript's compiler removes from the generated code all TypeScript types, interfaces, and keywords not supported by JavaScript. By providing compiler options, you can generate JavaScript compliant with ES3, ES5, ES6, or newer syntax.
- Here's how to transpile the code to ES5-compatible syntax (the `--t` option specifies the target syntax):

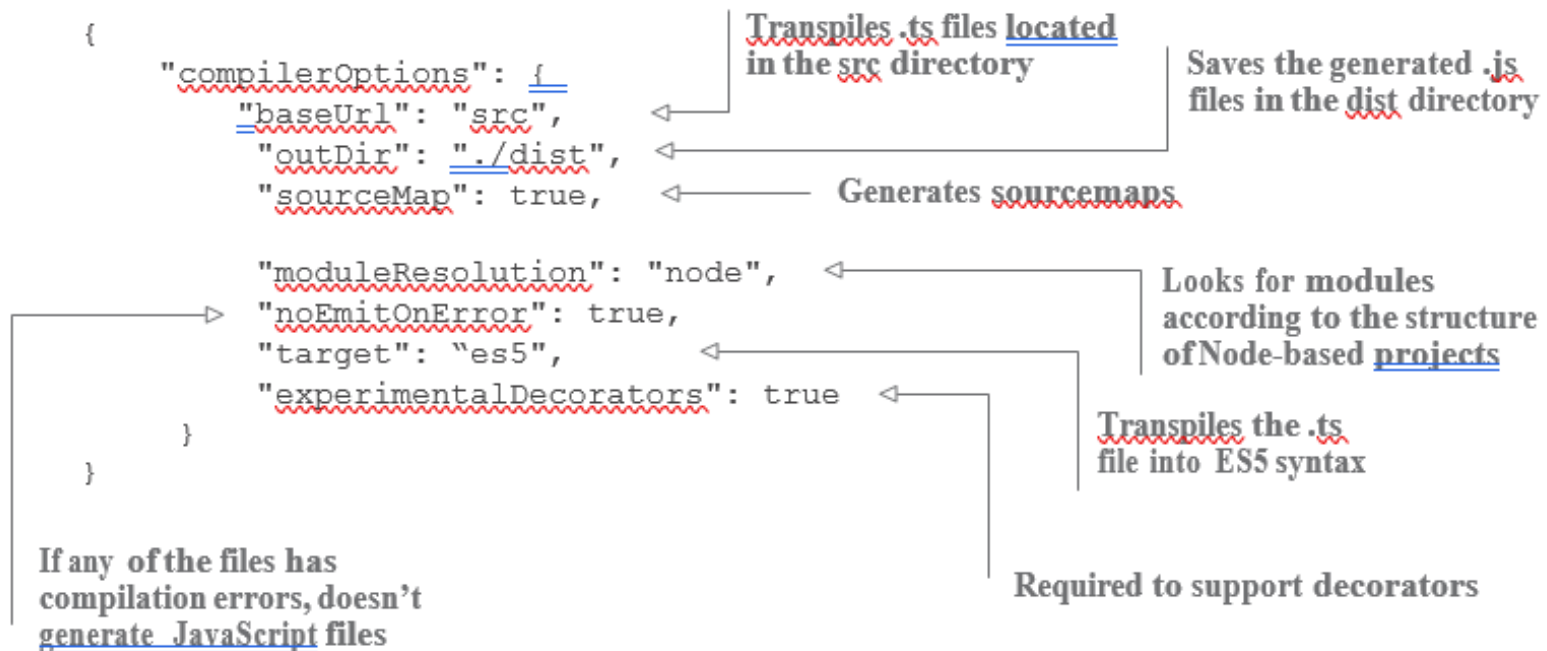
```
tsc --t ES5 main.ts
```

- You can start your TypeScript compiler in watch mode by providing the `-w` option. In this mode, whenever you modify and save your code, it's automatically transpiled into corresponding JavaScript files. To compile and watch all .ts files from the current directory, run the following command:

```
tsc -w *.ts
```

# File tsconfig.json

- `tsc` offers dozens of compilation options described at <http://mng.bz/rf14>. You can preconfigure the process of compilation (specifying the source and destination directories, source map generation, and so on).
- The presence of the `tsconfig.json` file in the project directory means you can enter `tsc` on the command line, and the compiler will read all the options from `tsconfig.json`. A sample `tsconfig.json` file from an Angular projects is shown below.



- To exclude some of project files from compilation, add the `exclude` property to `tsconfig.json`. This excludes the entire content of the `node_modules` directory:  
`"exclude": [ "node_modules" ]`

# Decorators

- Every Angular/TypeScript app uses decorators with classes or class members (such as `@Component()` and `@Input()`). If you want to exclude some of your project files from compilation, add the `exclude` property to `tsconfig.json`. This is how you can exclude the entire content of the `node_modules` directory:

```
"exclude": [ "node_modules"]
```



# Optional Types

- You can declare variables and provide types for all or some of them. The following two lines are valid TypeScript syntax:

```
let name1 = 'John Smith';  
let name2: string = 'John Smith';
```

- In the second line, specifying the type `string` is unnecessary. Since the variable is initialized with the string, TypeScript will guess (infer) that the type of `name2` is `string`.
- If you use types, TypeScript's transpiler can detect mismatched types during development, and IDEs will offer code completion and refactoring support. This will increase your productivity on any decent-sized project. Even if you don't use types in declarations, TypeScript will guess the type based on the assigned value and will still do type checking afterward. This is called type inference.
- The following fragment of TypeScript code shows that you can't assign a numeric value to a `name1` variable that was meant to be a string, even though it was initially declared without a type (JavaScript syntax). After initializing this variable with a string value, the inferred typing won't let you assign the numeric value to `name1`:

```
let name1 = 'John Smith';  
name1 = 123;
```

- Assigning a value of a different type to a variable is valid in JavaScript but invalid in TypeScript because of the **inferred type**.

# Optional Types

- In TypeScript, you can declare typed variables, function parameters, and return values. There are four keywords for declaring basic types: `number`, `boolean`, `string`, and `void`. The last one indicates the absence of a return value in a function declaration. A variable can have a value of type `null` or `undefined`, similar to JavaScript.

- Here are some examples of variables declared with explicit types:

```
let salary: number;  
let isValid: boolean;  
let customerName: string = null;
```

Starting from TypeScript 2.7, you need to either initialize variables during declaration or initialize (member variables) in the constructor.

- All of these types are subtypes of the `any` (ANY) type. You may explicitly declare a variable, specifying `any` as its type. In this case, inferred typing isn't applied. Both of these declarations are valid:

```
let name2: any = 'John Smith';  
name2 = 123;
```

- If variables are declared with explicit types, the compiler will check their values to ensure that they match the declarations. TypeScript includes other types that are used in interactions with the web browser, such as `HTMLElement` and `Document`. If you define a class or an interface, it can be used as a custom type in variable declarations.

# Functions

- TypeScript functions and function expressions are similar to JavaScript functions, but you can explicitly declare parameter types and return values.
- Bellow is a JavaScript function that calculates tax. It has three parameters and calculates tax based on the state, income, and number of dependents. For each dependent, the person is entitled to a \$500 or \$300 tax deduction, depending on the state the person lives in. The function is shown bellow.

```
function calcTax(state, income, dependents) {  
    if (state === 'NY') {  
        return income * 0.06 - dependents * 500;  
    } else if (state === 'NJ') {  
        return income * 0.05 - dependents * 300;  
    }  
}
```

- A person with an income of \$50,000 lives in the state of New Jersey and has two dependents. Let's invoke `calcTax()`
- `let tax = calcTax('NJ', 50000, 2);`
- The `tax` variable gets 1,900, which is correct. Even though `calcTax()` does not declare any types for the function parameters, you can guess them based on the parameter names. Now let's invoke it the wrong way, passing a string value for a number of dependents:
- `var tax = calcTax('NJ', 50000, 'two');`
- You will not know there is a problem until you invoke this function. The `tax` variable will have a NaN value (not a number). A bug sneaked in just because you didn't have a chance to explicitly specify the types of the parameters.

# A better Function

- The code bellow rewrites previous function in TypeScript, declaring types for parameters and the return value.

```
function calcTax(state: string, income: number, dependents: number): number
{
    if (state === 'NY'){
        return income * 0.06 - dependents * 500;
    } else if (state === 'NJ'){
        return income * 0.05 - dependents * 300;
    }
}
```

- Now there's no way to make the same mistake and pass a string value for the number of dependents:

```
let tax: number = calcTax('NJ', 50000, 'two');
```

- The TypeScript compiler will display an error saying, "Argument of type string is not assignable to parameter of type number." Moreover, the return value of the function is declared as number, which stops you from making another mistake and assigning the result of the tax calculations to a non-numeric variable:

```
let tax: string = calcTax('NJ', 50000, 'two');
```

- The compiler will catch this, producing the error "The type 'number' is not assignable to type 'string': var tax: string." This kind of type checking during compilation can save you a lot of time on any project.

# Default parameters

- While declaring a function, you can specify default parameter values. For example:

```
function calcTax(income: number,  
                dependents: number,  
                state: string = 'NY'): number{  
    // the code goes here  
}
```

- There's no need to change even one line of code in the body of `calcTax()`. You now have the freedom to invoke it with either two or three parameters:

```
let tax: number = calcTax(50000, 2);  
or  
let tax: number = calcTax(50000, 2, 'NY');
```

- The results of both invocations will be the same.

# Optional parameters

- In TypeScript, you can easily mark function parameters as optional by appending a question mark to the parameter name. The only restriction is that optional parameters must come last in the function declaration. When you write code for functions with optional parameters, you need to provide application logic that handles the cases when the optional parameters aren't provided.
- Let's modify the tax-calculation function in the following listing: if no dependents are specified, it won't apply any deduction to the calculated tax.
- Note the question mark in `dependents?: number`. Now the function checks whether the value for dependents was provided. If it wasn't, you assign 0 to the `deduction` variable; otherwise, you deduct 500 for each dependent.

# Optional parameters

```
function calcTax(income: number, state: string = 'NY', dependents?: number):number {
    let deduction: number;
    if (dependents) {
        deduction = dependents * 500;
    }else {
        deduction = 0;
    }
    if (state === 'NY') {
        return income * 0.06 - deduction;
    } else if (state === 'NJ') {
        return income * 0.05 - deduction;
    }
}

let tax: number = calcTax(50000, 'NJ', 3);
console.log(`Your tax is ${tax}`)
tax = calcTax(50000);
console.log(`Your tax is ${tax}`);
```

- Running the preceding script will produce the following output:

Your tax is 1000

Your tax is 3000

# Function overloading

- JavaScript doesn't support function overloading, so having several functions with the same name but different lists of arguments is not possible. TypeScript supports function overloading, but because the code has to be transpiled into a single JavaScript function, the syntax for overloading is not elegant.
- You can declare ***several signatures of a function with one and only one body***; you need to check the number and types of the arguments and execute the appropriate portion of the code:

```
function attr(name: string): string;
function attr(name: string, value: string): void;
function attr(map: any): void;
function attr(nameOrMap: any, value?: string): any {
    if (nameOrMap && typeof nameOrMap === "string") {
        // handle string case
    } else {
        // handle map case
    }

    // handle value here
}
```

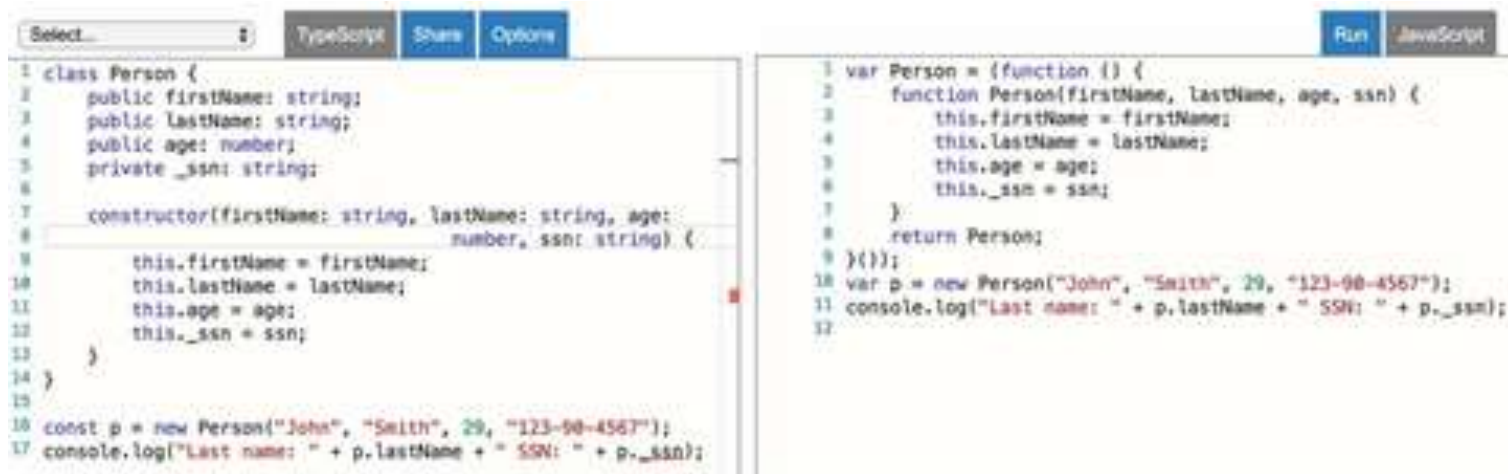


# Classes

- If you have Java or C# experience, you are familiar with the concepts of classes and inheritance in their classical form. In those languages, the definition of a class is loaded in memory as a separate entity (like a blueprint) and is shared by all instances of this class. If a class is inherited from another one, the object is instantiated using the combined blueprint of both classes.
- TypeScript is a superset of JavaScript, which only supports *prototypal inheritance*, where you can create an inheritance hierarchy by attaching one object to the prototype property of another. In this case, an inheritance (or rather, a linkage) of objects is created dynamically.
- In TypeScript, the `class` keyword is syntactic sugar to simplify coding. In the end, your classes will be transpiled into JavaScript objects with prototypal inheritance. In JavaScript, you can declare a *constructor* function and instantiate it with the `new` keyword.
- In TypeScript, you can also declare a class and instantiate it with the `new` operator.
- A class can include a constructor, fields (properties), and methods. Declared properties and methods are often referred to as class members.
- We will illustrate the syntax of TypeScript classes by showing a series of code samples and comparing them with the equivalent ES5 syntax.

# Classes, continued

- We will create a simple `Person` class that contains four properties to store the first and last name, age, and Social Security number (a unique identifier assigned to citizens and residents of the United States).
- At left in figure below, you can see the TypeScript code that declares and instantiates the `Person` class; on the right is a JavaScript closure generated by the tsc compiler.
- By creating a closure for the `Person` function, the TypeScript compiler enables the mechanism for exposing and hiding the elements of the `Person` object.
- TypeScript also supports class constructors that allow you to initialize object variables while instantiating the object.
- A class constructor is invoked only once during object creation. The left side shows the `Person` class, which uses the `constructor` keyword that initializes the fields of the class with the values given to the constructor. The right side shows transpiled JavaScript closure.



```
1 class Person {
2   public firstName: string;
3   public lastName: string;
4   public age: number;
5   private _ssn: string;
6
7   constructor(firstName: string, lastName: string, age:
8     number, ssn: string) {
9     this.firstName = firstName;
10    this.lastName = lastName;
11    this.age = age;
12    this._ssn = ssn;
13  }
14 }
15
16 const p = new Person("John", "Smith", 29, "123-98-4567");
17 console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
```

```
1 var Person = (function () {
2   function Person(firstName, lastName, age, ssn) {
3     this.firstName = firstName;
4     this.lastName = lastName;
5     this.age = age;
6     this._ssn = ssn;
7   }
8   return Person;
9 }());
10 var p = new Person("John", "Smith", 29, "123-98-4567");
11 console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
12
```

# person.ts

```
class Person {
  public firstName: string;
  public lastName: string;
  public age: number;
  private _ssn: string;

  constructor(firstName: string, lastName: string, age: number,
               ssn: string) {
    this.firstName == firstName;
    this.lastName = lastName;
    this.age = age;
    this._ssn = ssn;
  }
}

const p = new Person("John", "Smith", 29, "123-90-4567");
console.log(`Last name: ${p.lastName} `);
```

# JavaScript closure

- A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.
- Consider the following example code:

```
function init() {  
    var name = 'Mozilla'; // name is a local variable created by init  
    function displayName() { // displayName() is the inner function, a closure  
        alert(name); // use variable declared in the parent function  
    }  
    displayName();  
}  
init();
```

- `init()` creates a local variable called `name` and a function called `displayName()`. The `displayName()` function is an inner function that is defined inside `init()` and is available only within the body of the `init()` function. Note that the `displayName()` function has no local variables of its own. However, since inner functions have access to the variables of outer functions, `displayName()` can access the variable `name` declared in the parent function, `init()`.

# Access Modifiers

- JavaScript doesn't have a way to declare a variable or a method as private (hidden from external code). To hide a property (or a method) in an object, you need to create a closure that neither attaches this property to the `this` variable nor returns it in the closure's return statement.
- TypeScript provides `public`, `protected`, and `private` keywords to help you control access to object members during the development phase. By default, all class members have public access, and they're visible from outside the class. If a member is declared with the `protected` modifier, it's visible in the class and its subclasses. Class members declared as `private` are visible only in the class.
- Let's use the `private` keyword to hide the value of the `_ssn` property so it can't be directly accessed from outside of the `Person` object. We show two versions of declaring a class with properties that use access modifiers. The longer version of the class is presented on slide 43.
- Note that the name of the private variable starts with an underscore: `_ssn`. This is a naming convention for private properties.
- The last line on slide 43 attempts to access the `_ssn` private property from outside, so the TypeScript code analyzer will give you a compilation error: "Property `'_ssn'` is private and is only accessible in class `'Person'`."
- Unless you use the `--noEmitOnError` compiler option, the erroneous code will still be transpiled into JavaScript:

# Transpiled JavaScript code

- JavaScript transpiled from code on slide 43 reads:

```
const Person = (function () {  
    function Person(firstName, lastName, age, _ssn) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
        this._ssn = _ssn;  
    }  
    return Person;  
})();  
  
const p = new Person("John", "Smith", 29, "123-90-4567");  
console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
```

- The private keyword only makes it private in the TypeScript code, but the generated JavaScript code will treat all properties and methods of the class as public anyway.

# Access methods for constructor arguments

- TypeScript also allows you to provide access modifiers with constructor arguments, as shown in the short version of the Person class below

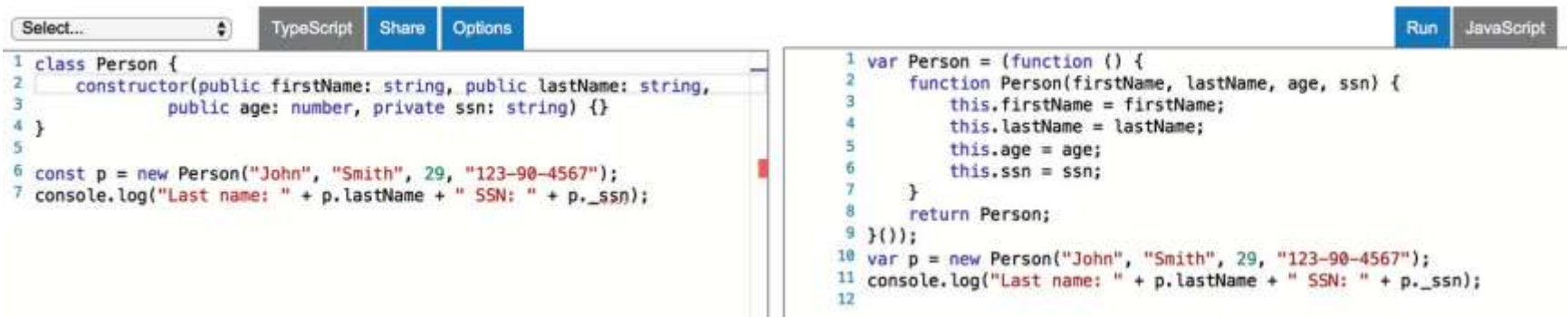
```
class Person {  
  
    constructor(public firstName: string,  
                public lastName: string, public age: number, private _ssn: string) {  
    }  
}
```

```
const p = new Person("John", "Smith", 29, "123-90-4567");
```

- When you use a constructor with access modifiers, the TypeScript compiler takes it as an instruction to create and retain class properties matching the constructor arguments. You don't need to explicitly declare and initialize them.
- Both the short and long versions of the Person class generate the same JavaScript, but we recommend using the shorter syntax

# Methods

- When a function is declared in a class, it's called a method. In JavaScript, you need to declare methods on the prototype of an object, but with a class, you declare a method by specifying a name followed by parentheses and curly braces, as you would in other object-oriented languages.



The image shows a side-by-side comparison of TypeScript and JavaScript code for a Person class. The left pane is labeled 'TypeScript' and the right pane is labeled 'JavaScript'. Both panes show the same logic: defining a class with a constructor that takes firstName, lastName, age, and ssn, and then creating an instance of the class and logging its properties.

```
Select... TypeScript Share Options Run JavaScript
```

```
1 class Person {
2   constructor(public firstName: string, public lastName: string,
3     public age: number, private ssn: string) {}
4 }
5
6 const p = new Person("John", "Smith", 29, "123-90-4567");
7 console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
```

```
1 var Person = (function () {
2   function Person(firstName, lastName, age, ssn) {
3     this.firstName = firstName;
4     this.lastName = lastName;
5     this.age = age;
6     this.ssn = ssn;
7   }
8   return Person;
9 }());
10 var p = new Person("John", "Smith", 29, "123-90-4567");
11 console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
12
```

- The next code listing shows how you can declare and use a `MyClass` class with a `doSomething()` method that has one argument and no return value.

```
class MyClass {
  doSomething(howManyTimes: number): void {
    // do something here
  }
}

const mc = new MyClass();
mc.doSomething(5);
```



# Static and instance members

- The code on previous slide creates an instance of the class first and then accesses its members using a reference variable that points at this instance:
- `mc.doSomething(5);`
- If a class property or method were declared with the `static` keyword, its values would be shared between all instances of the class, and you wouldn't need to create an instance to access static members. Instead of using a reference variable (such as `mc`), you'd use the name of the class:

```
class MyClass{  
    static doSomething(howManyTimes: number): void {  
        // do something here  
    }  
}
```

```
MyClass.doSomething(5);
```

- If you instantiate a class and need to invoke a class method from within another method declared in the same class, don't use the `this` keyword (as in, `this.doSomething(5)`), but still use the class name, as in `MyClass.doSomething(10);`.

# Inheritance

- JavaScript supports prototypical object-based inheritance, where one object can assign another object as its prototype, and this happens during runtime. TypeScript has the `extends` keyword for inheritance of classes, like ES6 and other object-oriented languages. But during transpiling to JavaScript, the generated code uses the syntax of prototypical inheritance.
- Figure below shows how to create an `Employee` class (line 9) that extends the `Person` class. On the right, you can see the transpiled JavaScript version, which uses prototypical inheritance. The TypeScript version of the code is more concise and easier to read.

TypeScript

Select... Share

```
1 class Person {
2
3     constructor(public firstName: string,
4         public lastName: string, public age: number,
5         private _ssn: string) {
6     }
7 }
8
9 class Employee extends Person{
10
11 }
```

Run JavaScript

```
1 var __extends = this.__extends || function (d, b) {
2     for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
3     function __() { this.constructor = d; }
4     __.prototype = b.prototype;
5     d.prototype = new __();
6 };
7 var Person = (function () {
8     function Person(firstName, lastName, age, _ssn) {
9         this.firstName = firstName;
10        this.lastName = lastName;
11        this.age = age;
12        this._ssn = _ssn;
13    }
14    return Person;
15 })();
16 var Employee = (function (_super) {
17     __extends(Employee, _super);
18     function Employee() {
19         _super.apply(this, arguments);
20     }
21     return Employee;
22 })(Person);
```

# Adding properties to Employee class, super

- Let's add a constructor and a department property to the Employee class in the next listing.

```
class Employee extends Person {  
    department: string;  
  
    constructor(firstName: string, lastName: string,  
        age: number, _ssn: string, department: string) {  
        super(firstName, lastName, age, _ssn);  
        this.department = department;  
    }  
}
```

- We added property `department` and a constructor with additional `department` argument.
- A subclass that declares a constructor must invoke the constructor of the superclass using `super()`.
- If you invoke a method declared in a superclass on the object of the subclass type, you can use the name of this method as if it were declared in the subclass. But sometimes you want to specifically call the method of the superclass, and that's when you should use the `super` keyword.

# super keyword

- The `super` keyword can be used two ways. In the constructor of a derived class, you invoke it as a method. You can also use the `super` keyword to specifically call a method of the superclass.
- Property `super` is typically used with method overriding. For example, if both a superclass and its descendant have a `doSomething()` method, the descendant can reuse the functionality programmed in the superclass and add other functionality as well:

```
doSomething() {  
    super.doSomething();  
  
    // Add more functionality here  
}
```

# Interfaces

- JavaScript doesn't support interfaces, which, in other object-oriented languages, are used to introduce a code contract that an API has to abide by.
- An example of a contract can be class X declaring that it implements interface Y. If class X does not include an implementation of the methods declared in interface Y, it is considered a violation of the contract and the class will not compile.
- TypeScript includes the keywords `interface` and `implements` to support interfaces, but interfaces are not transpiled into JavaScript code. They just help you avoid using the wrong types during development.
- In TypeScript, we use interfaces for two reasons:
  - Declare an interface that defines a custom type containing a number of properties. Then declare a method that has an argument of such a type. The compiler will check that the object given as an argument includes all the properties declared in the interface.
  - Declare an interface that includes abstract (non-implemented) methods. When a class declares that it implements this interface, the class must provide an implementation for all the abstract methods.
- On the following slides, we will illustrate those concepts

# Declaring custom types with interfaces

- When you use JavaScript frameworks, you may run into an API that requires some sort of configuration object as a function parameter. To figure out which properties must be provided in this object, either open the documentation for the API or read the source code of the framework. In TypeScript, you can declare an interface that includes all the properties, and their types, that must be present in a configuration object.
- Let's see how to do this in the `Person` class, which contains a constructor with four arguments: `firstName`, `lastName`, `age`, and `ssn`.
- This time, in the following code we declare an `IPerson` interface that contains the four members, and we modify the constructor of the `Person` class to use an object of this custom type as an argument.

```
interface IPerson { firstName: string; lastName: string;
                    age: number, ssn?: string;
}
```

- Note that `IPerson` interface has `ssn` as an optional member (note the question mark).
- The `Person` class has a constructor with one argument of type `IPerson`.

```
class Person {
    constructor(public config: IPerson) {} }

let aPerson: IPerson = {firstName: "John", lastName: "Smith", age: 29 }
let p = new Person(aPerson); # Instantiates the Person object, providing
                             # an object of type IPerson as an argument
console.log("Last name: " + p.config.lastName );
```

# TypeScript has a structural type system

- TypeScript has a structural type system, which means that if two different types include the same members, the types are considered compatible.
- In code on slide 55, even if you do not specify the type of the `aPerson` variable, it still would be considered compatible with `IPerson` and could be used as a constructor argument while instantiating the `Person` object.
- If you change the name or type of one of the members of `IPerson`, the TypeScript compiler will report an error.
- The `IPerson` interface didn't define any methods, but TypeScript interfaces can include method signatures without implementations.

# Using the `implements` keyword

- The `implements` keyword can be used with a class declaration to announce that the class will implement a particular interface. Say you have an `IPayable` interface declared as follows:

```
interface IPayable {  
    increase_cap: number;  
    increasePay(percent: number): boolean  
}
```

- Now the `Employee` class can declare that it implements `IPayable`:

```
class Employee implements IPayable {  
    // The implementation goes here  
}
```

- The questions one naturally asks is why not just write all required code in the class rather than separating a portion of the code into an interface?



# Benefits of Implementing Interfaces

- Let's say you need to write an application that allows increasing salaries for the employees of your organization.
- You can create an `Employee` class (that extends `Person`) and include the `increaseSalary()` method there. Then the business analysts may ask you to add the ability to increase pay to contractors who work for your firm. But contractors are represented by their company names and IDs; they have no notion of salary and are paid on an hourly basis.
- You can create another class, `Contractor` (not inherited from `Person`), that includes some properties and an `increaseHourlyRate()` method. Now you have two different APIs: one for increasing the salary of employees, and another for increasing the pay for contractors.
- A better solution is to create a common `IPayable` interface and have `Employee` and `Contractor` classes provide different implementations of `IPayable` for these classes, as illustrated in the following listing.

# Interface IPayable, Classes Person and Employee

- The Person class serves as a base class for Employee.
- The IPayable interface includes the signature of the increasePay() method that will be implemented by the Employee and Contractor classes

```
interface IPayable {  
    increasePay(percent: number): boolean  
}
```

- The Employee class inherits from Person and implements the IPayable interface. A class can implement multiple interfaces. The Employee class implements the increasePay() method. The salary of an employee can be increased by any amount, so the method prints the message on the console and returns true (allowing the increase).

```
class Person { // properties are omitted for brevity  
}  
  
class Employee extends Person implements IPayable {  
    increasePay(percent: number): boolean {  
        console.log(`Increasing salary by ${percent}`);  
        return true;  
    }  
}
```

# Contractor class

- The Contractor class includes a property that places a cap of 20% on pay increases.
- The implementation of `increasePay()` in the Contractor class is different, invoking `increasePay()` with an argument that's more than 20 results in the "Sorry" message and a return value of false.

```
class Contractor implements IPayable {
  increaseCap:number = 20;
  increasePay(percent: number): boolean {
    if (percent < this.increaseCap)
      console.log(`Increasing hourly rate by ${percent}`);
      return true;
    } else {
      console.log(`Sorry, the increase cap for contractors is
        ${this.increaseCap}`);
      return false;
    }
  }
}

workers: IPayable[] = [];
workers[0] = new Employee();
workers[1] = new Contractor();
workers.forEach(worker => worker.increasePay(30));
```

- You can invoke the `increasePay()` method on any object in the workers array. Note that you don't use parentheses with the fat-arrow expression that has a single worker argument.

- Declaring an array of type `IPayable` lets you place any objects that implement the `IPayable` type there. Preceding script produces the following output on the browser console

Increasing salary by 30, Sorry, the increase cap for contractors is 20

# Why declare classes with `implements` keyword

- If you remove `implements IPayable` from the declaration of either `Employee` or `Contractor`, the code will still work, and the compiler won't complain about lines that add these objects to the `workers` array. The compiler is smart enough to see that even if the class doesn't explicitly declare `implements IPayable`, it `implements increasePay()` properly.
- But if you remove `implements IPayable` and try to change the signature of the `increasePay()` method from any of the classes, you won't be able to place such an object into the `workers` array, because that object would no longer be of the `IPayable` type. Also, without the `implements` keyword, IDE support (such as for refactoring) will be broken.

# Generics

- TypeScript supports parameterized types, also known as `generics`, which can be used in a variety of scenarios. For example, you can create a function that can take values of any type; but during its invocation, in a particular context, you can explicitly specify a concrete type.
- Take another example: an array can hold objects of any type, but you can specify which particular object types (for example, instances of `Person`) are allowed in an array. If you were to try to add an object of a different type, the TypeScript compiler would generate an error.
- The following code listing declares a `Person` class and its descendant, `Employee`, and an `Animal` class. Then it instantiates each class and tries to store them in the `workers` array declared with the `generic` type. Generic types are denoted by placing them in angle brackets (as in `<Person>`).

# Using Generic Type

```
class Person {  
    name: string;  
}  
class Employee extends Person {  
    department: number;  
}  
class Animal {  
    breed: string;  
}  
let workers: Array<Person> = [];  
  
workers[0] = new Person();  
workers[1] = new Employee();  
workers[2] = new Animal(); // compile-time error
```

- By declaring the `workers` array with the generic type `<Person>`, you announce your plan to store only instances of the `Person` class or its descendants. An attempt to store an instance of `Animal` in the same array will result in a compile-time error.