Typescript Intro

Introduction to Angular 6

Basic overview

- TypeScript is a superset of JavaScript.
  - allows programmers to use new features in their code, which is then translated (transpiled) to JavaScript.
  - developed by Microsoft and comes standard with Visual Studio
- TypeScript is portable across browsers, devices, and operating systems
- It can run on any environment that JavaScript runs on.
- TypeScript is aligned with ECMA2015/ES6.
  - language features like modules and class-based orientation are in line with the ES6 specification. Additionally, TypeScript also embraces features like generics and type decorators that aren’t a part of ES6 yet

Working with TypeScript

- TypeScript files are identified by the *.ts extension
  - are compiled to vanilla javascript *.js using tsc or an automated task runner like Gulp.
  - Uses a configuration file tsconfig.json to specify the root files and compiler options for a Typescript project
- Compiled TypeScript can be consumed from any JavaScript code
  - TypeScript can use and be used by other JS libraries, tools and frameworks
  - Any valid .js file can be renamed to .ts and compiled with other TypeScript files.
- When a TypeScript script gets compiled, there is an option to generate a declaration file
  - This acts as an interface to the components in the compiled JavaScript to provide support for libraries like jQuery, MooTools.

Benefits of TypeScript

- Static typing
  - makes TypeScript code more predictable and easier to debug than JavaScript.
  - Many type based errors can be caught at compile time.
  - Automatic type inference is supported as well to make code more compact
- Features like modules and namespaces
  - make organizing large code bases more manageable compared to Javascript
- Full support for Object-orientation
  - This is supported to a nearly the same degree as C# / Java, providing the associated advantages such as reusability and maintainability

Basic types

- Static typing
  - you can declare the types of variables, and the compiler will make sure that they aren't assigned the wrong types of values.
  - If type declarations are omitted, they will be inferred automatically from your code
- Most commonly used data types:
- Number
  - All numeric values are represented by the number type, there aren't separate definitions for integers, floats or others
- String
  - The text type, similar to vanilla JS strings

Basic types

- Boolean
  - true or false
- Any
  - A variable with this type can have it's value set to a string, number, or anything else.
  - Allows opt-out of type-checking and let the values pass through compile-time checks
- Arrays
  - Normal or generic declaration
- Void
  - Used for functions that don't return anything.
- undefined and null
  - These values actually have their own types with the same name

Template strings

- This can span multiple lines and have embedded expressions of the form ${ expr }
- They are surrounded by the backquote (`) character

Functions

- Types for function parameters and return types can be explicitly declared
- Functions can be created both as a named function or as an anonymous function.
- Every parameter is assumed to be required by the function.
  - The number of arguments given to a function has to match the number of parameters the function expects.
- Adding a ? to the end of parameters makes them optional, their value will then be undefined.
  - Any optional parameters must follow required parameters.

Functions

- Default-initialized parameters
  - set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined instead.
  - Default-initialized parameters that come after all required parameters are treated as optional and can be omitted when calling their respective function.
- Rest parameters allow you to work with multiple parameters as a group
  - When you do not know how many parameters a function will ultimately take.

Arrow functions

- Arrow functions (fat arrow functions) are a more concise syntax for writing function expressions
  - They are anonymous and change the way the keyword this binds in functions
  - Make code more concise, and simplify function scoping
- In classic function expressions, the **this** keyword is bound to different values based on the function's execution context
  - In the global scope, **this** refers to the global object
- When **this** is used inside of a declared object
  - the value of **this** is set to the closest parent object the method is called on.
- With arrow functions however, **this** is lexically bound
  - Means that it uses **this** from its original context.

Classes

- Similar syntax to C# / Java
- Classes have members, which are usually properties, constructors and methods
  - The constructor is a special method that runs when the new keyword is used and returns an instance of the class
- Classes are available in ES6 and are not specific to TypeScript
  - TypeScript is a little more strict in type-checking

Class inheritance

- Inheritance extends existing classes to use new ones.
- The child (derived) classes inherit all the public and protected members of the parent (base) class
  - The child classes can override the methods of the parent class with more specialized versions
- Each derived class that contains a constructor function must call super() which will execute the constructor of the base class
  - Before we ever access a property on this in a constructor body, we have to call super()

Working with class members

- There are 3 access modifiers for members of a class
- Public
  - Members are accessible everywhere (default modifier if none is specified)
- Protected
  - Members are accessible only in the class itself and its descendant classes
- Private
  - Members are accessibly only in the class itself and nowhere else
- You can make properties readonly by using the readonly keyword.
  - Readonly properties must be initialized at their declaration or in the constructor.

Working with class members

- Static members of a class are those that are visible on the class itself rather than on the instances.
- Parameter properties provide a shortcut that allows you to create and initialize a member at the same time
  - Parameter properties are declared by prefixing a constructor parameter with an accessibility modifier or readonly, or both.
- Getters/setters provide a way to control accesses to a member of an object to prevent incorrect modification

Interfaces

- Interfaces are used to type-check whether an object fits a certain structure.
- Defining an interface provides a way to name a combination of certain properties, making sure that they will always go together.
  - Useful way of defining contracts between code
- The order of the properties does not matter.
  - We just need the required properties to be present and to be the right type.

Interfaces

- Not all properties of an interface may be required.
  - Interfaces with optional properties have each optional property denoted by a ? at the end of the property name
  - The advantage of optional properties is that you can describe these possibly available properties while still also preventing use of properties that are not part of the interface.
- Interfaces are also capable of describing function types.
  - To do this, give the interface a call signature.
- Classes can also implement an interface
  - They must incorporate the properties and methods defined in that interface

Generics

- Generics is useful for creating reusable components
  - Allows the execution of certain generic operations on a different range of data types
- A function could be made more flexible by allowing its arguments to be of any particular type
  - Involves the use of type variables which captures of the type of an argument so that it can be reused in the function body
- Generic constraints allow the arguments to be only from a range of specific types
  - This makes generic functions much more useful by allowing a wider range of operations to be performed involving these types

Decorators

- Provide a way to add both annotations and a meta-programming syntax for class declarations and members.
- Decorators are proposed for future versions of JavaScript (ES7), and are available as an experimental feature of TypeScript.
- A Decorator is a special kind of declaration that can be attached to a class declaration, method, property, or parameter.
- Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

Decorators

- Decorators use the form @expression
  - expression must evaluate to a function that will be called at runtime with information about the decorated declaration.
- The decorator function is supplied information about the thing that it is attached to
  - It returns something in its place, or manipulates its target in some way.
  - Typically the "something" a decorator returns is the same thing that was passed in, but augmented in some way.

Class decorators

- A Class Decorator is declared just before a class declaration.
  - It can be used to observe, modify, or replace a class definition.
- The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument.
  - If the class decorator returns a value, it will replace the class declaration with the provided constructor function.
- Should you chose to return a new constructor function, you must explicitly maintain the original prototype.

Property decorator

- A Property Decorator is declared just before a property declaration.
- The expression for the property decorator will be called as a function at runtime, with the following two arguments:
  - Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
  - The name of the member.

Parameter Decorators

- A Parameter Decorator is declared just before a parameter declaration.
- The parameter decorator is applied to the class constructor or method declaration.
- The expression for the parameter decorator will be called as a function at runtime, with the following three arguments:
  - Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
  - The name of the member.
  - The ordinal index of the parameter in the function's parameter list.