TypeScript

Naveen Pete

Why TypeScript?

- Superset of JavaScript
  - Any valid JavaScript code is also valid TypeScript code
- Offers more features over vanilla JavaScript
  - Types, Classes, Interfaces, Modules, etc.
- TypeScript does not run in the browser, it is compiled to JavaScript (by CLI)
- Chosen as main language by Angular
- By far most documentation & example-base uses TypeScript

- Why TypeScript?
  - Strong Typing
    - reduces compile-time errors, provides IDE support
  - Next Gen JS Features
    - Modules, Classes, Import, Export, …
  - Missing JS Features
    - Interfaces, Generics, …

Why TypeScript?

- Benefits
  - Strong typing
    - More predictable and easier to debug
  - Object-oriented features
    - Classes, Interfaces, Modules, Properties, Generics, etc.
  - Compile-time errors
  - Great tooling
    - Intelli-sense
- Installing TypeScript
  - npm install –g typescript
- Transpile TS to JS
  - tsc some-program.ts

Data types

- Used to provide classification to the data
- Informs the compiler how the programmer intends to use the data

- TypeScript supports following data types:
  - Boolean
  - Number
  - String
  - Array
  - Any
  - Tuple
  - Enum
  - Void

Data types

- Syntax for variable declaration

```
let <variable-name>: <data-type> = <value>;

const <variable-name>: <data-type> = <value>;
```

- Boolean

```
let isDone: boolean = false;
```

- Number
  - TypeScript numbers are floating point values

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

Data types

- String

```
let color: string = "blue";
color = 'red';
```

  - Template string example

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;
```

- Array

```
let list: number[] = [1, 2, 3];

(or)

let list: Array<number> = [1, 2, 3];
```

Data types

- Tuple
  - Allows you to express an array where the type of a fixed number of elements is known

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK

// Initialize it incorrectly
x = [10, "hello"]; // Error

x[3] = true; // Error, 'boolean' isn't 'string | number'
```

Data types

- Enum
  - A way of giving more friendly names to sets of numeric values
  - By default, enums begin numbering their members starting at 0

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

```
enum Color {Red = 1, Green, Blue}

enum Color {Red = 1, Green = 2, Blue = 4}
```

Data types

- Any
  - We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

- Void
  - The opposite of any: the absence of having any type

```
function warnUser(): void {
    alert("This is my warning message");
}
```

Type Assertions

- Is like a type cast in other languages
- A way to tell the compiler “trust me, I know what I’m doing.”

- Two forms
  - Angle-bracket Syntax

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

  - “as” Syntax

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

Variable Declaration

- let
  - declares a block scope local variable
  - usually called lexical scoping or block scoping
  - block-scoped variables
    - are not visible outside of their nearest containing block
    - can’t be read or written to before they’re actually declared
    - unlike var, same variable names cannot be used in multiple declarations
- const
  - block-scoped, much like variables defined using the let statement
  - value of a constant cannot change through re-assignment, and it can't be redeclared

```
const name = 'Hari';
let age = 25;
const dateOfBirth = '10/10/2017';
```

Functions

- Basic building blocks of an app
- Follow DRY (Don’t Repeat Yourself) principle
- Promote code reuse

- Named functions
  - Similar to JavaScript with a small difference

```
function add(x, y) {
    return x + y;
}
```

```
function add(x: number, y: number): number {
    return x + y;
}
```

Functions

- Anonymous functions
  - We assign a function to a variable
  - We then can call the function using the variable name

```
// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

```
let myAdd = function(x: number, y: number): number {
  return x + y;
};
```

Functions

- Arrow functions
  - Very similar to regular functions in behavior, but are quite different syntactically
  - Are anonymous functions
  - Shorter syntax
  - Does not have its own ‘this’; the this value of the enclosing execution context is used

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter name:
(singleParam) => { statements }
singleParam => { statements }

// The parameter list for a function with no parameters should be
// written with a pair of parentheses.
() => { statements }
```

Functions

- Optional parameters
  - By default, every parameter is assumed to be required by the function
  - The compiler also assumes that these parameters are the only parameters that will be passed to the function
  - The number of arguments given to a function has to match the number of parameters the function expects

```
function buildName(firstName: string, lastName: string) {
  return firstName + " " + lastName;
}

let result1 = buildName("Bob");  // error, too few params
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many params
let result3 = buildName("Bob", "Adams");  // just right
```

Functions

- Optional parameters (Continued)
  - By adding a ? to the end of a parameter, we can make it optional
  - Any optional parameters must follow required parameters

```
function buildName(firstName: string, lastName?: string) {
  if (lastName)
    return firstName + " " + lastName;
  else
    return firstName;
}

let result1 = buildName("Bob");  // works correctly now
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many params
let result3 = buildName("Bob", "Adams");  // just right
```

Functions

- Default parameters
  - We can set a value that a parameter will be assigned if the user does not provide one. These are called default-initialized parameters

```
function buildName(firstName: string, lastName = “Smith”) {
  return firstName + " " + lastName;
}

let result1 = buildName("Bob");  // works correctly, returns “Bob Smith”
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many params
let result3 = buildName("Bob", "Adams");  // just right
```

Functions

- Rest parameters
  - The rest parameter syntax allows us to
    - represent an indefinite number of arguments as an array
    - gather variables together
  - Rest collects multiple elements and 'condenses' them into a single element

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Interfaces

- Can be used to define complex types
- Are a way to define contracts within your code
- Implementation detail of members are not present within the interface

```
interface RectangleOptions {
  length: number;
  width: number;
  height?: number;
}

function drawRectangle(obj: RectangleOptions) {
  ...
  ...
  ...
}
```

Interfaces

```
interface CommonFeatures {
  name: string;
  create(); // save an object
  getAll(); // gets all the objects
}
```

```
class Product implements CommonFeatures
{
  name: string;
  price: number;

  create() {
    console.log('Product created');
  }

  getAll() {
    console.log(
      'Returning all products'
    );
  }
}
```

```
class Student implements CommonFeatures
{
  name: string;
  std: number;

  create() {
    console.log('Student created');
  }

  getAll() {
    console.log(
      'Returning all students'
    );
  }
}
```

Classes

- Enables you to create your own custom types
- Is a blueprint
- Defines the data and the behavior of a type
- TypeScript allows us to create a class and add properties, methods, constructors, getter and setter functions

- Constructor
  - A special function of a class
  - Called when we ‘new’ up instances of the class
  - Used to set the default or user supplied values to properties of an object

Classes

```
class Animal {
  constructor(public name: string) {
    this.name = name;
  }

  speak() {
    console.log(this.name +
        ' makes a noise.');
  }
}
```

- Inheritance
  - Used to extend existing classes to create new ones
  - One of the most fundamental object-oriented patterns
  - Use ‘extends’ keyword to extend one class from another

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(this.name +
        ' makes a noise.');
  }
}
```

```
class Dog extends Animal {
  constructor(name, color = 'white') {
    super(name);
    this.color = color;
  }

  speak() {
    console.log(this.name + ' barks.');
  }
}
```

```
let a = new Animal('Snowy');
a.speak();

let d = new Dog('Tommy');
d.speak();

let l = new Lion('Leo');
l.speak();
```

```
class Lion extends Animal {
  constructor(name, color = 'ochre') {
    super(name);
    this.color = color;
  }

  speak() {
    console.log(this.name + ' roars.');
  }
}
```

Classes

- Super
  - Is used to refer to constructor, properties or methods of base class from the derived class
  - To execute the constructor function of the base class, we can call super() from the constructor of derived class. Optionally, arguments can be passed

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

```
class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}
```

Classes

- Abstract Classes
  - Base classes from which other classes may be derived
  - May not be instantiated directly
  - May contain implementation details for its members
  - ‘abstract’ keyword is used to define abstract classes as well as abstract methods within an abstract class

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log("roaming the earth...");
    }
}
```

Classes

- Access Modifiers
  - Public
    - Default access modifier
    - Provides public visibility to a member
  - Private
    - When ‘private’ used, the member cannot be accessed from outside of the containing class
  - Protected
    - Similar to ‘private’ with the exception that members declared ‘protected’ can also be accessed by instances of derived classes

Classes

- Static members
  - Are visible on the class itself rather than on the instances
  - Can be assigned to a property or a method of the class

```
// Static methods
class Plane {
  constructor(numEngines) {
    this.numEngines = numEngines;
    this.enginesActive = false;
  }

  startEngines() {
    console.log('starting engines…');
    this.enginesActive = true;
  }

 static badWeather(planes) {
    for (plane of planes) {
      plane.enginesActive = false;
    }
  }
}
```

```
// Calling a static method
Plane.badWeather([
   plane1,
   plane2,
   plane3
]);
```

Generics

- Components that we build should be
  - Reusable
  - Capable of working on the data of today as well the data of tomorrow
    - Provides flexible capabilities for building up large software systems
- Generics
  - Enables you to create components that can work over a variety of types rather than a single one
  - Allows users to consume these components and use their own types

Generics

```
function reverse<T>(items: T[]): T[] {
    var toreturn = [];
    for (let i = items.length - 1; i >= 0; i--) {
        toreturn.push(items[i]);
    }
    return toreturn;
}
```

```
class Queue<T> {
  private data = [];
  push = (item: T) => this.data.push(item);
  pop = (): T => this.data.shift();
}
```

Q & A

- Thank you!