TypeScript for C# developers

Making JavaScript manageable

Agenda

- What is TypeScript
- OO in TypeScript
- Closure
- Generics
- Iterators
- Asynchronous programming
- Modularisation
- Debugging TypeScript

What is TypeScript

- A language that compiles to JavaScript, designed by Microsoft
- Used by Google (Angular 2)
- Run in the browser and on the server (Node JS)
- Provides similar OO and type safety to C#
- But its not C#

JavaScript

- Issues when building large scale apps
  - No type safety
  - No language support for modularization
  - Encapsulation not promoted by the language
  - Developer needs to use a variety of techniques and patterns to make it happen.
- Advantages
  - Runs in browsers on many platforms
  - Can be used to build server apps NodeJS
  - Lots of existing frameworks

TypeScript and JavaScript

- TypeScript is a thin veneer on JavaScript providing type safety and modularisation
- Understand JavaScript and lots of TypeScript oddities make sense
  - Function scope
  - Array indexes are just properties
  - Objects only have public fields
  - No function overloading
- Later versions of JavaScript fix some of these “features”
  - Block scope
  - Property getters and setters
  - Class

Using JavaScript frameworks from TypeScript

- DefinitelyTyped files (.ts) for all major JavaScript frameworks
- Provides strong typing and intelli sense for JavaScript frameworks via TypeScript
- Definitions found on GitHub and Nuget

Classes

- Similar to C#
  - Class keyword
  - Public, Private encapsulation
  - Explicitly named Constructor method
  - Properties as of EcmaScript 5
- Conventions
  - Private fields start with an _
  - Public fields start with lower case, and then camel case
  - Methods start with lower case, and then camel case
- Method return type can be left to the compiler to determine

Primitive types

- For int,double,float,long use **number**
- For string use **string**
- For bool use **boolean**
- For object use **any**
- For void use **void**

Private means what ?

- TypeScripts creates the notion of private
- JavaScript does not have the notion of private
- Objects created in “TypeScript” consumed via JavaScript have no privacy

Polymorphism

- Uses extends keyword to derive from class
- Classes and methods can be marked as abstract
- Use super() not base to call base type methods
- No need to overload constructor if simply want base type behaviour.

Exceptions

- Anything can be thrown
- Best practice is to throw instance of Error/derived from Error
- Catch blocks can’t filter by type
- Use instanceof to get type

any the dynamic in TypeScript

- Variables marked as any behave like regular JavaScript objects
- No real intellisense

```
build(options: any): string {
      var url: string = "";

      if (options.protocol !== undefined) {
          url = options.protocol + "://";
      }

      url += options.path;

      return url;
  }
```

Interfaces

- Used to describe the shape of objects
- Useful for passing parameters and object literal

```
interface SuccessCallback {
    (data: string): void;
}

interface AjaxOptions {
    url: string;
    verb?: string;
    success: SuccessCallback;
}

class AjaxClient {
    call(options: AjaxOptions) {
        if (options.verb == undefined) {
            options.verb = "POST";
        }
    }
}
```

Interfaces for methods, not quiet Delegates

- Interfaces used to build **Function Types**
- Type safety similar to Delegates
- WARNING, invoking functions ‘this’ is not what you may expect.

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}
```

Overloading

- JavaScript does not support overloading
- TypeScript has a kind of overloading
- Assists with intellisense
- Many signatures
- One Implementation
- Signatures must be compatible with implementation
- Implementation has to examine parameters to vary behaviour

```
createUrl(options: IUrlParts): string;
createUrl(path:string):string;
createUrl(options: any): string {
        …
}
```

Closures

- The var keyword provides method based scoping
- The let keyword provides block based scoping
- Most noticeable when using closure
- Two ways to define closures
  - Anonymous functions
  - Lambda's known as Arrow functions
- Arrow functions capture 'this'
  - Gives you same behaviour as C#
- Anonymous functions use normal JavaScript rules for this.

Generics

- Generic functions, classes and interfaces possible
- Utilises constraints similar to C# to define functionality against type arguments

```
class Utils {
    public static min<T extends  IComparable>(items:T[]):T {
        var min:T = items[0];
        for (var i = 1; i < items.length; i++) {
            if (min.compareTo(items[i]) > 0) {
                min = items[i];
            }
        }
        return min;
    }
}
```

Better than .NET generics

- Sometimes hard to build generics around .NET shipped types
- JavaScript uses prototypical inheritance
  - **Add methods to objects** that have already been created

```
interface IComparable<T> {
    compareTo(other:T):number;
}

interface Number extends IComparable<number> {
}

Number.prototype.compareTo = function(rhs: number)  {
    var lhs:number = this;

    if (lhs === rhs) return 0;
    if (lhs < rhs) return -1;
    return 1;
}
```

Iterators

- **for in** syntax iterators over properties in JavaScript/TypeScript

```
var values = { name: 'andy', age: 21 };
for (var value in values) {
        console.log(value);
}

name
age
```

```
var collection = [10, 20, 30];

for (var item in collection) {
        console.log(item);
}

1
2
3
```

Modern JavaScript for..of, for iteration

- Provides equivalent of foreach in C#
  - Iterable<T> equivalent to IEnumerable<T>
  - Iterator<T> equivalent to IEnumerator<T>
- Built in types Array,Map,Set supported

```
var collection = [10, 20, 30];

for (var item of collection) {
        console.log(item);
}
```

```
10
20
30
```

Generators

- Can build custom iterators by hand
- Automated using the **yield** keyword

```
class FibGeneratation {
    *getFibs(nFibs: number) {
        var current = 1;
        var prev = 0;
        var prevPrev = 0;

        for (var nFib = 0; nFib < nFibs; nFib++) {
            yield current;
            prevPrev = prev;
            prev = current;
            current = prev + prevPrev;
        }
    }
}
```

Delegating generators

- Generators can delegate to other generators to combine sequences

```
class Enumerable {
    *combine<T>(first:Iterable<T>,second:Iterable<T>){
        yield* first;
        yield* second;
    }
}
var enumerable = new Enumerable();
var sequence = enumerable.combine([1,2,3],[4,5,6]);
```

Asynchronous operations with Promise<T>

- Built around the Promise type similar to Task<T>
- Promise provides fluent api to setup continuations
- Create the promise and provide the **executor** function
- Executor function is supplied **callbacks** for outcome

```
class Async {
    delay(time: number):Promise<any> {

        var executor =
        (resolve: (result: any) => void, reject: (error: any) => void) => {
            window.setTimeout(() => resolve(null), time);
        };

        var timerPromise = new Promise<any>(executor);

        return timerPromise;
    }
}
```

Working with Promises

- Use fluent api to register continuations in case of
  - Ran to completion
  - Failure

```
var asyncOperations = new Async();

asyncOperations.delay(2000).then(_ => {

        console.log('delay of 2 seconds done');
}).catch(err => {
        console.log('something went wrong');
});
```

Simplified with async and await

- Provides similar pattern of use to that of C#
- Keeps code simple,

```
async function() {
        var asyncOperations = new Async();

        await asyncOperations.delay(2000);
        console.log('delay completed');
    }
```

And finally just remember TypeScript is JavaScript

- 1.3 + 2.1 = ?
- typeof NaN = ?
- NaN == NaN = ?

Summary

- TypeScript offers type safety
- Encapsulation out of the box
- Compiles to JavaScript and so behaves like JavaScript