Intro to TypeScript

Not the hero we want, but the one we need right now, but still pretty cool

- Multi-Paradigm but mostly OO Transpired to JS Language
- Backwards compatible with JS
- Optionally statically typed

Transpiled to JavaScript

- Compiles to configurable ECMA Script version (default 5)
- Allows use of JS libraries with help of .d.ts files
- Run on both server and node

Backwards compatible with JavaScript

- Used ECMA Script 6 as a starting point for syntax
- All Valid JavaScript is valid TypeScript
- Familiar syntax for JS devs
- Also means it has JS baggage
- And some occasional weird syntax because of backward compatibility/transpilation

Defining a variable

```
var x: number; // Declare a variable with a type

var x = 1; // Contextual TypeInference

var x: number = 1; // Explicitly typed with a value
```

Object initializer syntax

```
var staticallyTyped = <MyClass>{
    someProperty: 1
};

var staticallyTyped: MyClass = {
    someProperty: 1
};

var dynamicallyTyped = {
    someProperty: 1
};
```

Basic Built in Types

- number
- boolean
- string
- void

Arrays

```
var array: Array<number> = [1, 2, 3];

var array: number[] = [1, 2, 3];

var array: any[] = [1, '2', 3];
```

Enums

```
enum Color {Red = 1, Green, Blue};
var c: Color = Color.Green;

enum Color {Red = 1, Green, Blue};
var colorName: string = Color[2];
```

any

- Dynamically Typed
  - Can contain anything
  - Can attempt to get any member regardless of whether it exists
  - Can change type that's in it

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

var array: any[] = [1, '2', 3];
```

Basic Function Syntax

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

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

// Add types
function doSomething(x: number, y: any, z): any {
    return x + y + z;
}
```

Optional Params

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

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

Default Params

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

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

Additional Params

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

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

Overrides

- Multiple interfaces, BUT can only have one implementation

```
function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    if (typeof x == "object") {
        // Do Something
    }
    else if (typeof x == "number") {
        // Do Something
    }
}
```

Lamdas and Function Types

```
var lamdaAdd = (x: number, y: number) => x + y;

var combine = (...params: string[]) => { return params.join(' ‘);};

var functionVariable: (x:number, y:number)=>number;
```

Lamda syntax uses a ‘this’ differently then ‘function’ syntax: more like C# than JavaScript

http://www.typescriptlang.org/Handbook#functions-lambdas-and-using-39this39

Inline Interfaces

```
function printLabel(labelledObj: {label: string}) {
    console.log(labelledObj.label);
}
```

Class Interfaces

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}
```

Optional Items

```
interface Shape {
    color: string;
    resize(multiplier: number): void;
}

interface Square extends Shape {
    sideLength: number;
}
```

Function Interfaces

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

var mySearch: SearchFunc;

mySearch = function(source: string, subString: string) {
    // Do Something
}
```

Array Interfaces

```
interface StringArray {
    [index: number]: string;
}

var myArray: StringArray;
myArray = ["Bob", "Fred"];
```

Classes

```
class Animal {
    name:string;
    constructor(theName: string) { this.name = theName; }
    move(meters: number = 0) {
        alert(this.name + " moved " + meters + "m.");
    }
}
var greeter = new Animal("fido");
```

Inheritance

```
class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 5) {
        alert("Slithering...");
        super.move(meters);
    }
}
```

Generics

```
var output = identity("myString");

var output = identity<string>("myString");
```

Generics

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

var myGenericNumber = new GenericNumber<number>();
```

Modules

- Uses either require js or commonjs
- Same syntax
- Configured using a compile option

Modules

- Module can mean two different things
  - The ‘module’ keyword
  - File modules included with ‘require’

Module Keyword

- Creates it as ‘variable’
- Export contents with export keyword (important!)

```
module SomeModule {
    export class SomeClass {

    }
}
```

File Modules

- Use export and require
- Export individual things
- Import and require into a variable
- All the exported things are properties of that variable

File Modules

```
export class MyClass {

}

export function myFunction() {

}
```

```
import myModule = require('modules');
```

Common Pitfalls

- make sure you export a class
- you can only have one implementation of an override

Upcoming featurs

- Async/Await - write code as synchronous, compiles as async
- Annotations - Include meta-data with object