ES6 & TypeScript

What we will cover

- this lesson will cover advanced features of JavaScript ES8 and Typescript
- The lesson will involve code examples
- the examples will be run with **node**

```
> node example.js
```

What is TypeScript?

- Programming language that compiles to JavaScript
- Open source and maintained by Microsoft
- Superset of JavaScript
- Optional static type and type checking
- ES6 Support
- Browsers can’t execute TypeScript files
- TypeScript comes with a compiler
- Better IDE completion than JS

Installing TypeScript

- We use npm to install TypeScript
- Need to init npm in a local folder: **npm init --yes**
- To install TypeScript
  - **npm install typescript --save-dev**
- You can verify TypeScript is installed by typing:
  - **tsc -v**

Facts About Javascript

- JavaScript (JS) - Dynamic language
- Scripting language for browsers and servers (JS for server side is run by node)
- JS is interpreted by the browser, no compilation
- Functional programming language
- Prototype version of JavaScript was written in **10 days** in1995
- Since 1996 ECMA is in charge of releasing the specification of JS, and browsers need to implement it.
- ES6 also known as ECMAScript2015 is the first major update since ES5 in 2009
- Since then every year there is a new version ES7-ECMAScript2016, ES8-ECMAScript2017
- Each browser implements the new version at his own pace
- Proposing new features to ECMAScript is open source and features are staged 0-4

Facts About Javascript

JOHN BRYCE
Leading in IT Education
a matrix company

- Browsers Support ES6
  - New Desktop browsers > 96%
  - New mobile browsers > 99%
  - Unfortunately you don’t know which browser will run your program so you can’t assume that the user browser will support ES6
- We can transpile our code from ES6,ES7,ES8 to ES5
- Babel is a popular transpiler

Hello World with TypeScript

- TypeScript files have a **.ts** extension
- Create **hello.ts** and using the **console.log** we will print **hello world**
- Browsers don’t understand TypeScript files and the file need to be turned to JS
- We use the TypeScript compiler to turn the TypeScript files to javascript files
- Basic usage of the compiler
  - **tsc hello.ts**
  - this will create **hello.js** from the TypeScript files we created

TSC options

- You can view additional options of the compiler by typing: **tsc --help**
- Some interesting options:
    - **target, outDir, sourcemap, watch, module**

TypeScript Configuration - tsconfig.json

- Configuration file for TypeScript
- Usually located at the root dir of the TypeScript project
- Some of the compiler options we seen earlier can be specified in this file
- Compiler will look in the tsconfig options and compile according to that file
- **tsc --init** will output that config file
- Compiling with no input file will search for tsconfig.json file in the current dir and it’s parents
- You can add a **--project (-p)** flag to specify the tsconfig.json directory
- The compilation will be according to the files listed in the tsconfig or if not listed in the tsconfig directory and all sub directory will look for **ts** files
- When input files are specified in the compiler cli then tsconfig is ignored
- Lets create a default tsconfig to examine popular options.

Debugging TypeScript Files

- Browsers can’t run TypeScript files so they will run our compiled JS files
- We want to debug with our written code and not generated code
- Source map is a file that maps from the transformed source to the original source
- Using the source map the browser can reconstruct the original source and place that source in the debugger
- The browser will know about the source map by a special comment at the end of the js
- There is an option that you can place in **tsconfig** that will create the source map: **compilerOptions.sourceMap = true**

Variable Scope

- Variables in TypeScript are defined like JS with: **var**
- Syntax: **var <variableName> = <assignment>;**
- Assignment is optional
- **variableName** should be camelcased
- Variable type can change dynamic language/loosly typed
- Example:

```
var myString = ‘hello world’

myString = 10;
```

- What is the scope of var?
- Can I declare a variable without using var?

TypeScript Type Inference

- TypeScript will try to guess the type of variable when doing the following
  - assignment
  - default arguments to functions
  - function return types
- The following code won’t compile

```
var stam = ‘hello world’;

stam = 10;
```

- The first assignment TypeScript will assume that the variable is of type string

Scope of var

- What will this print?

```
for (var i = 0; i < 10; i++) {
  for (var i = 0; i < 10; i++) {
    console.log(i);
  }
}
```

- What will this code print?

```
function printMe(isPrint) {
  if (isPrint) {
     var message = 'hello world';
  }
  console.log(message);
}
printMe(true);
printMe(false);
```

Block Scoped Variables

- Another way to define variables in ES6 & TypeScript is:

```
const <variableName> = <assignment is a must>;
let <variableName > = <assignment is optional>
```

- Const has a single assignment
- Let can have multiple assignment
- Is single assignment mean immutable?
- The scope of let and const is inside the block
- What’s the result of the previous example when changing var to let?

Static Types, Type Checking

- With TypeScript you can optionally specify the type of variable
- The syntax is:
  - **const/let/var variableName: <type> = assignmentIsOptionalUnlessConst**
- When compiling the file TypeScript will check that the type is matching

Basic Types - String

- **let myString: string = ‘hello world’;**
- Define a string: “”, ‘’, ``
- `` backticks are used for multiple lines (Template string)
- `` with backticks you can inject javascript variables by using ${}
- You can concat strings with the **+**
- String is an array of characters so you can access a character like array syntax: **myStr[i]**
- can you change a character? what will happen if you try?
- You can iterate on a string like array

- Some common functions: **indexOf, substr, split**

Basic Types - Numbers

- Single number type that represents: float, positive, negative numbers
- Operators: +, -, *, /, %, **, ++, --
- **toString** will convert number to string
- **parseInt, parseFloat** will convert from string to number if fails will return **NaN**
- Numbers are immutable
- Number constants: **NaN, Infinity, -Infinity**

Basic Types - Booleans

- **const myBoolean: boolean = true;**
- True, False
- Common tricks with boolean:
  - if (<var>) { … } // ‘’, 0, null, undefined, NaN are false
  - const myVar = expressionIfTrue || -1
- Booleans are immutable
- Logical operators: !, ==, ===, ||, &&, !=, !==

Basic Types - Miscellaneous

- Undefined
- Null
  - null and undefined are subtype of everything unless --**strictNullChecks true**
- NaN
- Infinity
- -Infinity
- Void
- Any
- Object

Type Assertion - Casting

- **Example:**

```
var person : any =  ‘yariv kayz’;

var nameLength : number = (<string>person).length;

var nameLength2 : number = (person as string).length;
```

Advanced Types - Arrays

Syntax:

- **const myNumArray: number[] = [];**
- **const myStringArray: Array<string> = []**

- TypeScript will check when you push to the array that the type match
- If you want to support different types you can: **const myAnyArray: any[] = [];**
- Common methods: **forEach, push, pop, splice,**
- Common properties: **length**

Advanced Types - Object/Dictionary

- Syntax:
  - **const dict: {[key: string]: any} = {<string key>: <value>, <string key2>: <value2>}**
- Access values: **dict.key1** or **dict[‘key1’]**
- key must be a string or a number
- Add value: **dict[‘newkey’] = <new value>**
- Get an array of all the keys: **Object.keys(dict)**
- Delete a key: **delete dict[‘newkey’]**
- Is key in object? **dict.hasOwnProperty(‘newkey’)**
- The key can be a string or a number
- The key can be computed you need to place the key in square brackets:
  - **const** ***computedKeys2***: {[**key**: **number**]: **any**} = {[*createRandom*()]: **'stam'**};

Advanced Types - Object/Dictionary

- You can define getters and setters for computed property values

```
const computedProperty: {[key: string]: any} = {

  sayHello: function sayHello() {return 'hi call me'},

  get sayHello2() {return 'hi';},

  set wat(val: string) {this.sayHello = val;}

}
```

```
console.log(computedProperty.sayHello());

console.log(computedProperty.sayHello2);

computedProperty.wat = 'i changed the function'

console.log(computedProperty.sayHello);
```

Arrays/Objects - Destructuring Assignment

- Goal is to easily unpack values from arrays and objects into variables
- Arrays:

  var [a, b, ...rest] = [1,2,3,4,5,6] // a=1, b=2, rest=[3,4,5,6]

  - Can u think of a way to swap variables with a single line?

- Objects

  var {a, b} = {a: ‘foo’, b: ‘bar’}

for .. of / for .. in

- What does each loop is used for?
- Is one of them dangerous and if so how?

Advanced Types - Symbol

- Problem 1: you want to set a property for objects that your library will support
  - you want to allow users to change that property
  - you don’t want programmers to accidently overwrite that property
  - you don’t want the property to be printed in **Object.keys** or iterated in **for..of**
- Problem 2: reflection… you want to sometime implement your own logic for iterator and ES6 needs to expose property for you to overwrite
- Symbols are always unique
- They are primitives data type
- They are immutable
- To create a symbol: **Symbol(‘description’)**
- Symbols won’t be printed in **Object.keys** and **for..of**
- You can change a symbol property only if you have access to that symbol

Advanced Types - Symbol

- There is a global registry of symbols
- You can access that global registry with: **Symbol.for('key')**
- ES6 expose certain global symbol you can override in the **Symbol** object
    - **Symbol.iterator -** we will see example later
- In TypeScript you can set a variable of type **Symbol**
- TypeScript will error out if you try to put that key in a dictionary without casting (known issue in TypeScript)

Advanced Types - Map

- Object can have key of type: **string, number, Symbol**
- The key of a map can be **any** including: **Objects, Functions. Arrays**
- Map's are iterable
- Key equality will be with **===**
- to use maps in typescript you need to edit the **tsconfig.json** and add **compilerOptions.lib = ["es6"]**

Advanced types - Set

- Set contains unique values
- Comparing values is with ===
- Set is iterable
- With TypeScript you can contain the type of members in the set

Function

- Function can return type
  - **function(x: number, y: number): number { return x + y; }**
- You can define a variable to accept a function
  - **let pokeFunc : (message : string) => void = function(msg){console.log(msg);}**
- The compiler will check if you call the function with the correct number of arguments

Function Arguments

- Arguments can get type
- Compiler will check the types that are passed to functions
- You can access the function arguments from: **arguments** array
- You can pass default value to arguments
- Default arguments don’t have to be the last ones
- You can supply an optional param by adding **?**
  - **function(x: number, y?: number): number { return x + y; }**
  - The optional params must be last

Function Quiz

- Parameters to function are passed by reference or by value?
- Can you name 3 ways to call a function? can you tell the difference between them?

this

- This behaves differently in JS then in other languages
- By default **this === window**
- When a function is called this is equal to window
- When a function has ‘use strict’ this is equal to undefined
- Typescript will use “use strict” based on the option in **tsconfig:**
  - **compilerOptions.alwaysStrict = true**
- This will be determined at call time
- When a function is part of an object this will be the object
- When a function is called with the **new** then **this** will be the new object of the function (good for dealing with classes)
- You can use **bind** to set what **this** will be

Lambda Functions

- Syntax
  - (arg1, arg2) => { … }
  - arg1 => { … }
  - (arg1, arg2) => 3 // return 3
  - arg1 => 3
- Doesn’t have a **this**

Generator Object / Iterator

- the generator object contains the following methods

```
interface Generator extends Iterator<any> { }
```

```
interface Iterator<T> {

   next(value?: any): IteratorResult<T>;

   return?(value?: any): IteratorResult<T>;

   throw?(e?: any): IteratorResult<T>;

}
```

```
interface IteratorResult<T> {

   done: boolean;

   value: T;

}
```

Generator Function

- A function which returns a **Generator Object**
- Generator function can be exited and later re-entered
- The body of the function won’t run until you hit next
- The function will run until the it reaches the yield and will return the yield value to a generator object
- The next function can get an argument which can be used as a returned argument for the previous yield

Generator Function

- the syntax of a generator function

```
const myGenFun = function* (startIndex: number = 0): Generator {
    const item = yield startIndex + 1;
    console.log('this will run on second yield');
    yield `${startIndex + 2}${item}`;
    yield startIndex + 3;
    return 100;
};
```

```
const gen: Generator = myGenFun();
console.log(gen.next().value); // 1
console.log(gen.next('tofu').value); // 2tofu
console.log(gen.next().value); // 3
console.log(gen.next().value); // 100
```

Iterables

- An object is iterable if it defines its iteration behavior
- For example an iterable can be used in the **for..of** loop
- To be an iterable an object has to implement the **Symbol.iterator** property
- The **Symbol.iterator** property need to be function that returns **Iterator**
- You can create a regular function that returns an object with **next**
- What did we learn that creates a **Generator object?**

Custom Iterable - ES6

```
class SortedArray extends Array {
  *[Symbol.iterator]() {
    const clone = this.splice(0);
    clone.sort();
    for(let i=0; i<clone.length; i++) {
      yield clone[i];
    }
  }
}
```

```
const temp = new SortedArray(1,0,5,-1);
for(let item of temp) {
  console.log(item);
}
```

Prototype

- JavaScript doesn't have a subclass and inheritance like traditional languages
- JavaScript uses prototype to achieve this
- The base prototype is: **Object.prototype** nearly all object are instances of **Object**
- Some of the inherited methods: **toString, hasOwnProperty, create, getPrototypeOf, constructor**
- **Array** and **Function** has prototype as well which inherits from **Object.prototype**
- When searching for a property it will start from the nearest prototype and then search in the next one and next one (prototype chaining)
- The next prototype is saved in the **__proto__**
- We can use prototype to create classes and inheritance
- We can take advantage of prototype chaining and override methods in the chain

Class

- Class is a syntax sugar for creating a class and inheritance like common languages and not by using prototype
- The feature was added in ES6
- You can define **constructor** in the class
- In the constructor arguments you can specify if an argument will be saved as **private, public, protected**
- Inheritance is done with the **extend** keyword
- You can call base function by using **super** (in constructor it has to be the first statement)
- You can define static methods with the keyword **static**
- You can define getters and setters
- Abstract class

Async actions

- Some actions in JS do not return an answer right away and will return the answer after a period of time
    - request data from rest server
    - setTimeout
    - setInterval
    - web workers
- We need a way to subscribe an event that will run when the response is back

What are promises?

- promise is an object which implements an action that returns data in an async way
- subscribers can choose to listen for the promise when the data arrives
- promise is built in in JS since ES6
- promise can also send fail event for the subscribers
- To use promises with TypeScript you have to include the **es6** lib
- promise in typescript:
  - **const myStringPromise: Promise<string> = new Promise((resolve, reject) => {...})**
- Let’s create a simple promise which returns a string data

Promise behaviour

- The constructor for the promise get a function
- the constructor function is called right away with 2 arguments: resolve and reject which are both functions
- we call the resolve when we want to return the data to the listeners
- we call the reject to inform listeners of a fail event
- that function will run right away, no matter if there are any subscribers
- the function will run once no matter if there are multiple subscribers
- the promise is not cancelable

then - promise listener

- we subscribe a listener to a promise by calling the instance method **then**
- the method **then** can optionally get a success method that will run when we call resolve (1st argument) **onsuccess**
- the second argument is a method that will run when we call reject **onreject**
- the **onsuccess** and **onreject** can return a value or a promise with a value
- **then** will return a promise with the data from what the **onsuccess** and **onreject** returned
- if a promise is already resolved when we attach **then** the **then** function will be called after script tasks are finished (similar to setTimeout with 0)

Promise chaining

- **then** returns a promise with altered data
- common use case:
  - **promise A** requests a rest server for a json data
  - a listener is attached to **promise A** and gets the json data
  - the listener is turning the json data to a class model and return it to make **promise B** with classes instead of json
  - listeners can subscribe to **promise B** and deal with classes and not raw json

fetch

- with fetch we can send an ajax request to a server
- fetch returns a promise
- fetch gets url as first argument
- second argument is optional options for the request
- let’s try and use fetch to grab all the tasks from the todo rest server
- url: https://nztodo.herokuapp.com/api/task/?format=json
- after we grab the json data from the server we will use promise chaining to create another promise with array of task class

Problem with promises

- async tasks repeats often in JS apps
- async tasks are often prone to errors
- difficult to manage exceptions
- data in async tasks need to be customizable and easy to alter and manage

async await

- async functions are functions that return a promise
- async functions can use the await on promises and will only continue when that promise is resolved
- async functions

What is Reflection?

- The ability to examine object properties and methods and change them during runtime.
- Simple reflection example:

```
> var dict = {};

> dict.toString(); //result: '[object Object]'

> Object.getPrototypeOf(dict).toString = () => 'Well hello reflection, how do u do?'

> dict.toString(); // result: 'Well hello reflection, how do u do?'
```

Proxy

- With proxies we can set traps that will run when performing certain actions on a **target**
- A trap allows us to run our own action before we run the original action
- Only if the action is performed on the proxy then the traps will run
- To create a proxy: **var proxy = new Proxy(target, handler)**
  - target is the item we want to set traps for
  - handler contains the traps we want to define

Proxy - Handler

- We have the following traps we can set:
    - has? (target: T, p: PropertyKey): **boolean**;
    - get? (target: T, p: PropertyKey, receiver: **any**): **any**;
    - set? (target: T, p: PropertyKey, value: **any**, receiver: **any**): **boolean**;
    - deleteProperty? (target: T, p: PropertyKey): **boolean**;
    - defineProperty? (target: T, p: PropertyKey, attributes: PropertyDescriptor): **boolean**;
    - enumerate? (target: T): PropertyKey[];
    - ownKeys? (target: T): PropertyKey[];
    - apply? (target: T, thisArg: **any**, argArray?: **any**): **any**;

Reflect

- Reflect holds a collection of internal methods
- Until es6 most of the reflection was done using the static methods in Object
- Few caveats:
  - next version of ecma script will add additional reflection methods to the Reflect object and not Object
  - Object reflection methods will exist in future versions but they are deprecated
  - Reflect has much more useful return types
  - Safer way to call Function.prototype.apply
- Expose methods for object reflection
- Contains the default handlers which can be returned from proxy

Interfaces

- Syntax:
  - **interface IInterfaceName {...}**
- can include optional properties
- can define a function
- can define methods that a class needs to implement

Enum

- Giving a friendly name to numeric values
- Syntax:
  - **enum Color {Red, Green, Blue};**
  - **var c : Color = Color.Red;**
- By default the numbering starts from zero
- You can change the by specifying the first item
- After you specify an item the rest will be start incrementing from that value
- You can specify the values of all the items
  - **enum Color {Red = 1, Green = 4, Blue = -1}**
- You can also get the string from the key
  - **var colorName: string = Color[2];**

Generic Type

- Generic type can be added to **function, interface, class**
- With generic type your object behaviour changes according to the type sent
- You can restrict the generic type by using **extends**

Decorators

- Used to annotate or modify class or class members
- Currently at **stage 2** (still experimental and syntax might change)
- To enable in **tsconfig** specify **compilerOptions.experimentalDecorators=true**
- Decorator is a function that gets called with the decorated object
- Decorators are more common to use with a decorator factory which allows to add configuration to the decorator
- Decorators can be attached to: **class, method, accessor, property**

Todo Rest Server

- Our rest server is located at this url: https://nztodo.herokuapp.com
- The server is connected to a database with a single table called task
- The task table api is in this path: **/api/task/**
- The server returns a **json** response

Task JSON

- A single task json looks this:

```
{"id":8529,"title":"mytitle","description":"mydescription","group":"mygroup"
,"when":"2016-12-12T21:20:00Z"}
```

- **id** is the primary key and automatically created by the server
- **when** is an **ISOString** representing date time

CORS

- Stands for **Cross-Origin Resource Sharing**
- As a security measure browsers restrict cross-origin HTTP requests initiated from within scripts
- Using CORS spec we can do cross domain communication between browser and server
- CORS are used with HTTP headers
- CORS headers has **Access-Control-*** prefix
- **Access-Control-Allow-Origin** - is required in the response from the server
- Certain Requests for the server are considered simple and are sent directly to the server
- Some requests like **PUT, DELETE** the browser will automatically send a preflight request

Get Tasks from Server

host: https://nztodo.herokuapp.com

path: /api/task/?format=json

method: GET

- Fetch will work with promise
- Fetch will return promise even on bad response

Get a Single Task

- **host: https://nztodo.herokuapp.com**
- **path: /api/task/:id/?format=json**
- **method: GET**

Insert New Task

- host: https://nztodo.herokuapp.com
- path: /api/task/
- method: POST
- request body: {title: …, description: …, when: …, group: ...}

Delete Task

- host: https://nztodo.herokuapp.com
- path: /api/task/:id/
- method: DELETE

Update Task

- host: https://nztodo.herokuapp.com
- path: /api/task/:id/
- method: PUT
- request body: {title: …, description: …, when: …, group: ...}

Modules

- With modules we can split our project to multiple files
- You can tell the compiler to concat all the files to a single file with the option: **compilerOptions.outFile**
  - this will work with module system **amd** or **system**
- Concat to a single file is not recommended
- For now to use modules we need to include the entry point file in the index with type **module**

Modules - Export

- Using export you can expose **function, class, constant** that can be imported from other module
- You can use export to chain export from other files (good for barrel files)
- You can use export default then import name can change
- If using regular export then name is important

Modules - Import

- You can import exported **functions, const, class**
- Exported default items can be imported with any name
- Export without default name should persist in import as well
- You can use **import * as name from …** to import everything in a module
- You can change the name of the import with alias
- Import can be relative or non relative
  - relative will start with / ./ ../ - use it to point to your own modules, relative to the importing file
  - non relative: **import * as $ from ‘jquery’;**
  - non relative used for external dependency
  - **tsconfig compilerOptions.moduleResolution** will determine how non relative will be searched
- Import without specifying what will just run the file

Teaching the Compiler

- Some API’s are not recognized by the compiler
  - example **fetch**
- We still want to use them and we still want the compiler to check that we are using them correctly
- In the **tsconfig** you can add **lib** array with string of additional packages that the compiler should know about
- You can use the **declare** to make the compiler aware of something global
  - **declare var fetch : any;**
- You can use definetly typed **@types** to download to the compiler interfaces for popular packages.

Teaching the Compiler

- Some API’s are not recognized by the compiler
  - example **fetch**
- We still want to use them and we still want the compiler to check that we are using them correctly
- In the **tsconfig** you can add **lib** array with string of additional packages that the compiler should know about
- You can use the **declare** to make the compiler aware of something global
  - **declare var fetch : any;**
- You can use definetly typed **@types** to download to the compiler interfaces for popular packages.

Student Ex. Iterable

- Create a class called SortedString
- the class gets a string
- make the class iterable
- the iterable function should print the string by character order

Student Ex.

- Create a class called Task
- the constructor of the class will get an object similar to what the todo rest server is returning
- the class will have properties like our todo task

Student Ex.

- install **node-fetch**

  npm install node-fetch
- Create a function which returns a promise, the promise returned will be from the **fetch** api querying the todo server
- using promise chaining transform the list you get from the server to a list of task class you created the previous ex.

Student Ex.

- Create an async await function that waits for the promise you created before and returns the number of items in the list

Student Ex.

- Create a fibonacci generator function
- every number after the first two is the sum of the two preceding ones

  1, 1, 2, 3, 5, 8, 13 …

```
function* fib() { ... }
const generator: Iterator<number> = fib();
console.log(generator.next().value); // 1
console.log(generator.next().value); // 1
console.log(generator.next().value); // 2
console.log(generator.next().value); // 3
```

Summary

- Every year EcmaScript is releasing a new version of JS
- the language is evolving at a great pace and JS and TypeScript are becoming powerful languages
- even working every day with JS there are many advanced features that can improve our day to day work.