ECMAScript 6 Features

JavaScript Classes

Definitions

- A class is a blueprint / general description which individual objects can be created (objects are instances of classes)
- Example: *Bobby* is an *object* in the *Person* class

- In JavaScript, a class definition is done using a function and an object is created using the new keyword (this was discussed in an earlier example).
- To define a class, we need a class Declaration along with a constructor function

JavaScript Classes

Declaration

- Use the class keyword and a constructor

```
class <class_name> {
  // attributes
  constructor(){
    // constructor code
  }
}
```

```
class Person {
    constructor(n,b){
        this.name = n;
        this.birthday = b;
    }
}

var person_object =
    new Person('Bobby', '1980-04-19');
```

- Much more on this later …

Default Parameters

Example function declarations

- **Default function parameters** allow named parameters to be initialized with default values if no value or undefined is passed
- Function parameters default to `undefined`. However, it's often useful to set a different default value. This is where default parameters can help.
- The alternative is to test parameter values in the function body and assign a value if they are undefined

```
function <name>(p1,p2=<def_val>) {
    // statements
}
```

Template Strings

Format preserved String

- Templates are string literals allowing embedded expressions.
  You can use multi-line strings and string interpolation features with them.
- They are formatted string including NEWLINEs, TABs, and other special characters, much like triple quotes in Python.
- The content is placed inside a pair of “back-ticks”:

  ```
  `<formatted text>`
  ```

- Template strings can contain placeholders. indicated by

  ```
  ${expression}
  ```


`var`, `const`, `let`, and default variables

Difference of Scopes

- `var` declarations are function scoped
- `let` and `const` declarations are block-scoped and cannot be redeclared
- Default declarations are globally scoped
- `var` and `let` variables are initialized with `undefined`, `const` variables must be initialize at declaration.

`for` loops

for … in, for … of

- The `for … in` loops before ES6 iterates through a sequence's index
- The `for … of` loops in ES6 iterates over the sequence objects (or values) rather than indexes
  - Both applicable to String, Array, Collections, etc.
- Syntax:

```
for (variable of iterable) { // statements }
for (variable in iterable) { // statements }
```

`variable` can be declared with `let` or `var`

`iterable` is an Object that can be iterated

`for` loops

forEach

- The `forEach()` method executes a provided callback function once for each array element.
  - Applicable only to Arrays
- Syntax

```
arr.forEach(callback(currentValue) {
  // statements
});
```

`callback` is the function to be executed during each iteration, it accepts an argument of the `current_item`

Arrow Functions

- In most cases, the arrow functions are used as a shorthand for creating anonymous callback functions

  ```
  var myFunc = function(a,b,c){...}
  ```
  now looks like this:
  ```
  var myFunc = (a,b,c) => {...}
  ```

- However, arrow functions do not retain the `this` keyword, and thus will lexically go up the scope until it finds a `this`.

TypeScript Basics

Types and Functions

TypeScript Types

… extended from JavaScript

- Primitives
  - Number
  - Boolean
  - String
  - Void
  - Symbol
  - Null
  - Undefined
- Any
- Array, Tuple
- Enum
- Object
- Never
- Unknown

TypeScript Variables

Declaration

- To declare a type, we can use the following syntax

```
<variable_declaration>: <datatype> [= initial_value]
```

- For example:

```
let total: number = 0
```

- TypeScript will then enforce the type throughout the application
- TypeScript variables cannot be redeclared

TypeScript Functions

- TypeScript functions are the same as JavaScript functions with the ability to specify a return type.
- In the JavaScript function `makeAnimalSpeak()`, the function doesn't check to see if the `animal` object is a certain type or if the `speak()` function/method exists before calling it.

```
var pig = {
    name: "Porker",
    speak: function(){
        return "oink!";
    }
}

function makeAnimalSpeak(animal){
    console.log(animal.speak());
}
```


TypeScript Functions

Inferred Type

- Even through we did not explicitly state the return type, TypeScript looks through the code, doing its best to guess or infer what type any given object could be.
- This is known as type inference

```
4   var pig = {
5       (property) speak: () => string
6       speak: function(){
7           return "oink!";
8       }
9   }
```

TypeScript Functions

Type Declarations

- We can specify the datatypes of the parameters and return values

```
function add(x : any[], y : any[]): number {
    return x.length + y.length;
}
```

OR

```
let add = function (x:any[], y:any[]): number {
    return x.length + y.length;
};
```

- This indicates to TypeScript that the return value of a function must be a number

Union Types

- We can also specify that a parameter can be one of several types by using a union of types (denoted |)

```
function add(x : any[] | string, y : any[] | string): number {
    return x.length + y.length;
}
```

- We can create a new type by using the `type` keyword

```
type thingsWithLengthProp = any[] | string;

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

Interface and ENUM

Basic TypeScript Structures

Interface

TypeScript core principle

- An interface defines a structure for your data
  - Think of it as defining a new type of data
- It cannot be made into a JavaScript object nor does it transpile to JavaScript code.
  - Purpose is to provide metadata to TypeScript
- Consider interfaces a way to tell TypeScript information about objects to help you catch more errors at build time (not at run-time)
- General Syntax:

```
interface <Interface name> {
    <prop1> : <type1>;
    <prop2> : <type2>;
    …
}
```

```
interface Person {
    name: string;
    age: number;
}
```

Enumerations and Anonymous Types

Limiting value domains

- An enumeration defines a type with only specified values available
  - Usually encoded using integers (i.e. 0 = value1, 1 = value2, etc.)
- A useful way to define a set of constant values that can be used to replace the magic strings and numbers in your code.

```
enum <enum_name> {
    Value1,
    Value2,
    etc.
}
```

- Anonymous Types allow for general forms to be declared within a function definition

TypeScript Classes

TypeScript Class

- A TypeScript class has the same syntax as the ES6 standard with the addition of optional type information for the parameters and return value
- A class can have members (or attributes or properties) and methods (or functions)
- A class can have a `constructor()` function which is executed during the creation of the object instance

TypeScript Class

Static Members

- A static member is an attribute that is associated with the class
- It is usually used to keep track of class metadata
- We can declare an attribute as static by invoking the `static` keyword before the variable name
- We access a static member by using the form `<class_name>.<static_variable>`

JavaScript Prototypical Inheritance

… An Aside

- **Prototypal Inheritance: *A prototype is a working object instance.*** Objects inherit directly from other objects.

```
function Animal(){
    fur: true
};
Animal.prototype.speak = function() {
    console.log("hello");
}

let dog = new Animal();
dog.speak() // "hello"

let scruffy = {
    name: "scruffy",
    __proto__: dog
}

scruffy.speak() // "hello"
```


TypeScript Class

… and Class Inheritance

- Classes (objects) can often be described with respect to other more general classes (objects)
- A Car describes a vehicle driven on the road with certain characteristics (i.e. can be driven, has mileage, has color etc.)
- Tesla and BMW are Cars
  - They inherit all the characteristics of a Car *and* have their own unique characteristics:
    - Teslas have battery capacity
    - BMWs have a fuel tank size
- Car is known as the Superclass (instatiated as the general object)
- Tesla and BMW are Subclasses (instantiated as the specified object)

TypeScript Class

The "IS-A" Relationship

- A subclass is a class that extends another super (or "base") class
- The subclass inherits all the attributes and functionality of the super
- Subclasses are typically described using the "is-a" relationship as subclass "is-a" superclass
  - i.e. A Tesla is-a Car
- When specifying a super class and sub-class relationship, both classes can be instantiated. Meaning that we can make objects out of either class using the new keyword.

```
class Car {
    name: string
}
class Tesla extends Car {
    battery: number
}
let mycar = new Tesla();
```


TypeScript Class

Abstract Classes

- Consider the case that our intent was that a class was only ever intended to serve as a base class, and never instantiated
- For example, we do not want to instantiate a Person only a Student or a FacultyMember
- In this case, define Person as an abstract class
- An abstract class gives the class definition but does not include the implementation. It acts as a contract that any class that inherits from it must implement the defined function headers

```
abstract class Person {
    name: string;
    age: number;
}
class Student extends Person {
    studentID: number;
}
let s = new Student();
let p = new Person(); // error
```

TypeScript Class

Access Modifiers

- Access modifiers determine who can access the attributes and functions within a class. It is used for the purposes of encapsulation and data hiding.
- There are three access modifiers in TypeScript
  - Public
  - Private
  - Protected
- Unfortunately, the trans-piled JavaScript will not have access modifier definitions since they do not exist in JavaScript

Writing a TypeScript Application

Putting together a Service

- A service controls the flow of data and manipulates the state of an application.
- For example, we can write a service to allow for adding, removing, showing, and viewing a collection people.
- We can define the structure of the service (members and functions) in an interface, then "implement" it in a class.

```
class <Service_name> implements <Interface> { … }
```

```
class PeopleController implements PersonServices {

    add(per:Person){
        // implement...
    }
    showAll(){
        // implement...
        return
    }
}
```

```
interface PersonServices {
    add(p: Person): void;
    showAll(): Person;
}
```