Angular 2 and TypeScript
Web Application Development
Trayan Iliev, http://www.iproduct.org/

Object-Oriented JavaScript. ECMAScript 6.
JS Design Patterns. JS Module Systems

Trayan Iliev

IPT – Intellectual Products & Technologies
e-mail: tiliev@iproduct.org
web: http://www.iproduct.org

Oracle®, Java™ and JavaScript™ are trademarks or registered trademarks of Oracle and/or its affiliates.
Microsoft .NET, Visual Studio and Visual Studio Code are trademarks of Microsoft Corporation.
Other names may be trademarks of their respective owners.
Agenda

1. JavaScript™ history and main features
2. JavaScript™ datatypes
3. Functions in JavaScript™
4. Closures
5. Objects in JavaScript™
6. Using this
7. Object-oriented inheritance in JavaScript™
8. JavaScript™ design patterns
9. EcmaScript 6 (Harmony, ES 2015) novelties
10. Bootstrapping an ES6 project using Webpack and Babel
Where is The Code?

Angular 2 and TypeScript Web App Development code is available @GitHub:
https://github.com/iproduct/course-angular2
Brief History of JavaScript™

- **JavaScript™** created by Brendan Eich from Netscape for less than 10 days!
- Initially was called **Mocha**, later **LiveScript** – Netscape Navigator 2.0 - 1995
- December 1995 Netscape® и Sun® agree to call the new language **JavaScript™**
- “JS had to 'look like Java' only less so, be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JS would have happened.”

B. E. (http://www.jwz.org/blog/2010/10/every-day-i-learn-something-new-and-stupid/#comment-1021)

The Language of Web

- JavaScript™ success comes fast. Microsoft® create own implementation called JScript to overcome trademark problems. JScript was included in Internet Explorer 3.0, in August 1996.

- In November 1996 Netscape announced their proposal to Ecma International to standardize JavaScript → ECMA Script

- JavaScript – most popular client-side (in the browser) web programming language („de facto“ standard) and one of most popular programming languages in general.

- Highly efficient server-side platform called Node.js based on Google V8 JS engine, compiles JS to executable code Just In Time (JIT) during execution (used at the client-side also).
The Language of Big Contrasts

JavaScript – a language of big contrasts: from beginner web designers (copy-paste) to professional developers of sophisticated JS libraries and frameworks.

Douglas Crockford: “JavaScript is may be the only language the people start to code in before learning the language :)

This was a reason for many to consider JavaScript as „trimmed version of object-oriented programming language“

Popularity of AJAX (Asynchronous JavaScript and XML) and shift towards dynamic (asynchronous) client side applications returned JavaScript in the spotlight.
JavaScript / ECMAScript Now

- **JS Reusable Design Patterns**, modular component-oriented software engineering, **Test Driven Development (TDD) and Continuous Integration (CI)**.
- **Model View Controller** (*Model-View-Presenter - MVP, Model-View-ViewModel - MVVM – or generally MV* ) libraries and application frameworks available → single page web and mobile applications using standard components and widgets.
- January 2009: **CommonJS** => to use of JS outside of browser
- June 2015: **ES6 (Harmony)** → classes, lambdas, promises, ...
- October 2012: **Typescript** → Type checking + @Decorators
Object-Oriented JavaScript

Three standard ways to create objects in JavaScript:

- Using **object literal**:
  ```javascript
  var newObject = {};
  ```
- Using **Object.create(prototype[, propertiesObject])** (prototypal)
  ```javascript
  var newObject = Object.create(Object.prototype);
  ```
- Using **constructor function** (pseudo-classical)
  ```javascript
  var newObject = new Object();
  ```
Object Properties

- **Object-Oriented (OO)** – object literals and constructor functions
- Objects can have **named properties**

Ex.: `MyObject.name = 'Scene 1';
     MyObject ['num-elements'] = 5;
     MyObject.prototype.toString = function() {
         return "Name: " + this.name + ": " + this['num-elements']
     }

- **Configurable object properties** – e.g. read only get/set etc.

Ex.: `Object.defineProperty( newObject, "someKey", {
         value: "fine grained control on property's behavior",
         writable: true, enumerable: true, configurable: true
     });`
Property Getters and Setters

Ex.: function PositionLogger() {
    var position = null, positionsLog = [];
    Object.defineProperty(this, 'position', {
        get: function() {
            console.log('get position called');
            return position;
        },
        set: function(val) {
            position = val;
            positionsLog.push({ val: position });
        }
    });
    this.getLog = function() { return positionsLog; };  
}
JavaScript Features

- The state of objects could be changed using JS functions stored in object's **prototype**, called **methods**.
- Actually in JavaScript **there were no real classes**, - only objects and constructor functions before ES6 (ES 2015, Harmony).
- JS is **dynamically typed language** – new properties and methods can be added runtime.
- JS supports object inheritance using **prototypes** and **mixins** (adding dynamically new properties and methods).
- **Prototypes** are objects (which also can have their prototypes) → inheritance = traversing prototype chain
- Main resource: Introduction to OO JS YouTube video https://www.youtube.com/watch?v=PMfcsYzj-9M
JavaScript Features

- Supports `for ... in` operator for iterating object's properties, including inherited ones from the prototype chain.
- Provides a number of predefined datatypes such as: `Object, Number, String, Array, Function, Date` etc.
- **Dynamically typed** – variables are universal containers, no variable type declaration.
- Allows dynamic script evaluation, parsing and execution using `eval()` – discouraged as a bad practice.
Datatypes in JavaScript

- **Primitive datatypes:**
  - `boolean` – values `true` и `false`
  - `number` – floating point numbers (no real integers in JS)
  - `string` – strings (no `char` type -> string of 1 character)

- **Abstract datatypes:**
  - `Object` – predefined, used as default prototype for other objects (defines some common properties and methods for all objects: `constructor`, `prototype`; methods: `toString()`, `valueOf()`, `hasOwnProperty()`, `propertyIsEnumerable()`, `isPrototypeOf()`)
  - `Array` – array of data (really dictionary type, `resizable`)
  - `Function` – function or object method (defines some common properties: `length`, `arguments`, `caller`, `callee`, `prototype`)
Datatypes in JavaScript

- Special datatypes:
  - `null` – special values of **object type** that does not point anywhere
  - `undefined` – a value of variable or argument that have not been initialized
  - `NaN` – Not-a-Number – when the arithmetic operation should return numeric value, but result is not valid number
  - `Infinity` – special numeric value designating infinity \( \infty \)

- Operator **typeof**

Example: `typeof myObject.toString` //-->'function'
Functional JavaScript

- Functional language – functions are “first class citizens”

- Functions can have own properties and methods, can be assigned to variables, pass as arguments and returned as a result of other function's execution.

- Can be called by reference using operator ()

- Functions can have embedded inner functions at arbitrary depth

- All arguments and variables of outer function are accessible to inner functions – even after call of outer function completes

- Outer function = enclosing context (Scope) for inner functions  → Closure
Closures

Example:

```javascript
function countWithClosure() {
    var count = 0;
    return function() {
        return count ++;
    }
}

var count = countWithClosure(); <-- Function call – returns inner function which keeps reference to count variable from the outer scope

console.log( count() );  <-- Prints 0;
console.log( count() );  <-- Prints 1;
console.log( count() );  <-- Prints 2;
```
Default Values & RegEx

- Functions can be called with different number of arguments. It is possible to define default values – Example:
  
  ```javascript
  function Polygon(strokeColor, fillColor) {
    this.strokeColor = strokeColor || "#000000";
    this.fillColor = fillColor || "#ff0000";
    this.points = [];
    for (i=2;i < arguments.length; i++) {
      this.points[i] = arguments[i];
    }
  }
  ```

- Regular expressions – Example: `/a*/.match(str)`
Functions in JavaScript

- Embedded functions – define their own scope `scope` - Ex.:

```javascript
function getBoundingRectangle(pts) {
  var points = pts || [];
  function minX() {
    var x, min = Number.POSITIVE_INFINITY;
    for(var i = 0; i < points.length; i++){
      x = points[i].x;
      if( x < min){
        min = x;
      }
    }
    return min;
  }
  return {
    x: minX(),
    y: minY(),
    width: maxX() - minX(),
    height: maxY() - minY()
  }
}
```

Local variables

Object literal
Inner functions define their own scopes:

```javascript
function getBoundingRectangle(pts) {
    var points = pts || [];
    var minX = function () {
        var x, min = Number.POSITIVE_INFINITY;
        for(i = 0; i < points.length; i++){
            x = points[i].x;
            if( x < min){
                min = x;
            }
        }
        return min;
    }
    return {
        x: minX,
        y: minY,
        width: function() {return maxX() - minX();},
        height: function() {return maxY() - minY();}
    }
}
```
Object Literals. Using `this`

Object literals – example:
```javascript
var point1 = { x: 50, y: 100 }
var rectangle1 = { x: 200, y: 100, width: 300, height: 200 }
```

Using `this` calling a function /D. Crockford/:

Pattern „Method Call“:
```javascript
var scene1 = {
    name: 'Scene 1',
    numElements: 5,
    toString: function() {
        return "Name: " + this.name + ", Elements: " + this['numElements']
    }
}
console.log(scene1.toString())   // --> 'Name: Scene 1, Elements: 5'
```
Accessing **this** in Inner Functions

- Using **this** calling a function /D. Crockford/:
  - Pattern „Function Call“:

```javascript
var scene1 = {
    ...?
    log: function(str) {
        var that = this;  
        var createMessage = function(message) {
            return "Log for ", that.name, ": ", message;
        };
        console.log( createMessage(str) );
    }
}
```

It's necessary to use an additional variable, because **this** points to the global object (window) **undefined** in strict mode.
Creating Objects Using **Constructors**

Pattern „**Constructor Call**“:

```javascript
function Shape(sx, sy, width, height, strokeColor, fillColor) {
    this.x = sx || 0;  this.y = sy || 0;
    this.strokeColor = strokeColor || "#000000";
    this.fillColor = fillColor || "#ff0000"
    this.width = width || 0; this.height = height || 0;
}
Shape.prototype.toString = function() {
    return "x: " + this.x + ", y: " + this.y + ", strokeColor: " +
    this.strokeColor + ", fillColor: " + this.fillColor;
}
When constructing object with **new** a hidden link is created: __proto__
or [[Prototype]] pointing to constructor's **prototype** property:

```javascript
shape1 = new Shape(50, 100, 30, 30, "red", "green");
console.log(shape1.toString());
```
"Classical" Inheritance, call() apply() & bind()

Pattern "Calling a function using special method"

Function.prototype.apply(thisArg, [argsArray])
Function.prototype.call(thisArg[, arg1, arg2, ...])
Function.prototype.bind(thisArg[, arg1, arg2, ...])

function Point(x, y, color){
    Shape.apply(this, [x, y, 1, 1, color, color]);
}
extend(Point, Shape);

function extend(Child, Parent) {
    Child.prototype = new Parent;
    Child.prototype.constructor = Child;
    Child.prototype.supper = Parent.prototype;
}
„Classical“ Inheritance. Using call() & apply()

```javascript
Point.prototype.toString = function() {
  return "Point(\" + this.super.toString.apply(this,[]) + \")\";
}
Point.prototype.draw = function(ctx) {
  ctx.Style = this.strokeColor;
  ctx.fillRect(this.x, this.y, 1, 1);
}
point1 = new Point(200,150, "blue");
console.log(point1.toString());
```
JavaScript Design Patterns

- **Software design patterns** gained popularity after the book *Design Patterns: Elements of Reusable Object-Oriented Software* [1994], GoF: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

- **Def: Software design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design

- **Proven solutions** – proven techniques that reflect the experience and insights the developers

- **Easily reused** – out of the box solutions to common problems

- **Expressiveness** – define common vocabulary and structure
JavaScript Design Patterns

- Prototype ( Object.create() / Object.clone() )
- Constructor (using prototypes)
- Singleton (literals, lazy instantiation)
- Module
- Observer (publish/subscribe events)
- Dynamic loading of JS modules
- DRY (Don't Repeat Yourself)

- Command
- Facade
- Factory
- Mixin
- Decorator
- Function Chaining
Examples Using JavaScript Design Patterns

Learning JavaScript Design Patterns
A book by Addy Osmani
Volume 1.6.2:
JS Design Patterns: Prototype

**Intent:** creates objects based on a template of an existing object through cloning: `Object.create(prototype[, propertiesObject])`
JS Design Patterns: Constructor

**Intent:** *constructor* is a special function used to initialize properties of a new object once memory allocated.

```javascript
function Vehicle( model, year, kilometers ) {
    this.model = model;
    this.year = year;
    this.kilometers = kilometers;
    this.toString = function () {
        return this.model + " (" + this.year + ") has travelled " + this.kilometers + " kilometers";
    };
}
```

Var focus = new Vehicle( "Ford Focus", 2010, 90000 );
Var jazz = new Vehicle( "Honda Jazz", 2005, 170000 );
JS Design Patterns: Module I

- **Intent**: Group several related elements, such as singletons, properties and methods, into a single conceptual entity.

- A portion of the code must have **global or public access** and be designed for use as global/public code. Additional **private or protected code** can be executed by the main public code.

- A module must have an **initializer/finalizer** functions that are equivalents to, or complementary to object constructor/destructor methods.

- In JavaScript, there are several options for implementing modules: **Module pattern**, as Object literal, **AMD modules**, **CommonJS modules**, **ECMAScript Harmony modules**.
JS Design Patterns: Singleton

**Intent:** Ensure a class has only one instance, and provide a global point of access to it.

Object literals `{ }` in JavaScript are a natural way to implement Singletons

Often Singletons are lazily initialized, like:

```javascript
getInstance: function( myOptions ) {
    if( instance === undefined ) {
        instance = new MySingleton( myOptions );
    }
    return instance;
}
```
JS Design Patterns: Observer (Publish/Subscribe)

**Intent:** Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.

```
Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
```

```
Observer
+notify()
```

```
ConcreteObserverA
+notify()
```

```
ConcreteObserverB
+notify()
```

notifyObservers() for observer in observerCollection call observer.notify()
JS Design Patterns: Factory Method

**Intent:** Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses (dependency injection).
JS Design Patterns: Abstract Factory

**Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

![Image of Abstract Factory Diagram](https://commons.wikimedia.org/w/index.php?curid=5962476)

*By DoktorMandrake - Recreation of Abstract_factory.png by Bdean42 in SVG format., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5962476*
JS Design Patterns: Mixin

**Intent:** Mixins as a means of collecting functionality through extension – simple alternative to multiple inheritance

**Example:**

```javascript
var o1 = { a: 1, b: 1, c: 1 };  
var o2 = { b: 2, c: 2 };  
var o3 = { c: 3 };  

var obj = Object.assign({}, o1, o2, o3);  
console.log(obj);  // { a: 1, b: 2, c: 3 }
```

- In ECMAScript 6 there is `Object.assign(target, ...sources)`
Intention: Attach additional responsibilities to an object dynamically keeping the same interface.

Decorators provide a flexible alternative to subclassing for extending functionality.

Conclusions – OO JavaScript Development

JavaScript™ provides everything needed for contemporary object-oriented software development. JavaScript supports:

- **Data encapsulation** (separation of public and private parts) – How?: Using design patterns **Module** or **Revealing Module**

- **Inheritance** – before ES 6 there were no classes but several choices for constructing new objects using object templates (“pseudo-classical” using `new`, OR using functions, OR `Object.create(baseObject)`, OR **Mixin**)

- **Polimorphism supported** – there are methods with the same name and different implementations – **duck typing**
EcmaScript 6 – ES 2015, Harmony
[https://github.com/lukehoban/es6features]

A lot of new features:
- arrows
- classes
- enhanced object literals
- template strings
- destructuring
- default + rest + spread
- let + const
- iterators + for..of
- Generators
- unicode
- Modules + module loaders
- map + set + weakmap + weakset
- proxies
- symbols
- subclassable built-ins
- Promises
- math + number + string + array + object APIs
- binary and octal literals
- reflect api
- tail calls
ES6 Classes [http://es6-features.org/]

class Shape {
    constructor (id, x, y) {
        this.id = id
        this.move(x, y)
    }
    move (x, y) {
        this.x = x
        this.y = y
    }
}

class Rectangle extends Shape {
    constructor (id, x, y, width, height) {
        super(id, x, y)
        this.width = width
        this.height = height
    }
}

class Circle extends Shape {
    constructor (id, x, y, radius) {
        super(id, x, y)
        this.radius = radius
    }
}
Block Scope Vars: let [http://es6-features.org/]

```javascript
for (let i = 0; i < a.length; i++) {
    let x = a[i]
    ...
}
for (let i = 0; i < b.length; i++) {
    let y = b[i]
    ...
}

let callbacks = []
for (let i = 0; i <= 2; i++) {
    callbacks[i] = function () { return i * 2 }
}
callbacks[0]() === 0
callbacks[1]() === 2
callbacks[2]() === 4
```
ES6 Arrow Functions and this

- **ECMAScript 6:**
  ```javascript
  this.nums.forEach((v) => {
    if (v % 5 === 0)
      this.fives.push(v);
  });
  ```

- **ECMAScript 5:**
  ```javascript
  var self = this;
  this.nums.forEach(function (v) {
    if (v % 5 === 0)
      self.fives.push(v);
  });
  ```

Source: [http://wiki.commonjs.org/wiki/Modules/1.1](http://wiki.commonjs.org/wiki/Modules/1.1)
ES6 Promises [http://es6-features.org/]

```javascript
function msgAfterTimeout (msg, who, timeout) {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(`$${msg} Hello ${who}!`), timeout)
    })
}
msgAfterTimeout("", "Foo", 1000).then((msg) => {
    console.log(`done after 1000ms:${msg}`);
    return msgAfterTimeout(msg, "Bar", 2000);
}).then((msg) => {
    console.log(`done after 3000ms:${msg}`)
})
```
ES6 Promises

Combining ES6 Promises

```javascript
function fetchAsync (url, timeout, onData, onError) { … }
fetchPromised = (url, timeout) => {
    return new Promise(((resolve, reject) => {
        fetchAsync(url, timeout, resolve, reject)
    }));
}
Promise.all([fetchPromised("http://backend/foo.txt", 500),
              fetchPromised("http://backend/bar.txt", 500),
              fetchPromised("http://backend/baz.txt", 500)]).then((data) => {
    let [ foo, bar, baz ] = data
    console.log(`success: foo=${foo} bar=${bar} baz=${baz}`)
}, (err) => {
    console.log(`error: ${err}`)
})
```
JavaScript Module Systems - CommonJS

- math.js:
  ```javascript
  exports.add = function() {
    var sum = 0, i = 0, args = arguments, len = args.length;
    while (i < len) {
      sum += args[i++];
    }
    return sum;
  };
  ```

- increment.js:
  ```javascript
  var add = require('./math').add;
  exports.increment = function(val) {
    return add(val, 1);
  };
  ```

Source: http://wiki.commonjs.org/wiki/Modules/1.1
JavaScript Module Systems – AMD I

// Calling define with module ID, dependency array, and factory function
define('myModule', ['dep1', 'dep2'], function (dep1, dep2) {
    // Define the module value by returning a value.
    return function () {};
});

define(['alpha'], function (alpha) {
    return {
        verb: function () {
            return alpha.verb() + 2;
        }
    };
});
Asynchronous module definition (AMD) – API for defining code modules and their dependencies, loading them asynchronously, on demand (lazy), dependencies managed, client-side.

```javascript
define("alpha", ["require", "exports", "beta"],
    function(require, exports, beta) {
        exports.verb = function() {
            return beta.verb();
        }
    });

define(function (require) {
    require(["a", "b"], function (a, b) {
    });
});
```
JavaScript Module Systems – ES6

// lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

// someApp.js
import * as math from "lib/math"
console.log("2\pi = " + math.sum(math.pi, math.pi))

// otherApp.js
import { sum, pi } from "lib/math"
console.log("2\pi = " + sum(pi, pi))
### EcmaScript 6 Compatibility


<table>
<thead>
<tr>
<th>Feature name</th>
<th>Current browser</th>
<th>IE 11</th>
<th>Edge 13(4)</th>
<th>Edge 14(4)</th>
<th>FF 45 ESR</th>
<th>FF 49</th>
<th>FF 50 Beta</th>
<th>FF 51 Aurora</th>
<th>FF 52 Nightly</th>
<th>CH 53, OP 40(1)</th>
<th>CH 54, OP 41(1)</th>
<th>CH 55, OP 42(1)</th>
<th>SF 9</th>
<th>SF 10</th>
<th>SF TP</th>
<th>WK</th>
<th>KQ 4.14</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Optimisation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proper tail calls (tail call optimisation)</td>
<td>2/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td></td>
<td>0/2</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td></td>
<td>7/7</td>
<td>4/7</td>
<td>4/7</td>
<td>5/7</td>
<td>0/7</td>
<td>0/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td></td>
<td>4/7</td>
</tr>
<tr>
<td>default function parameters</td>
<td></td>
<td>7/7</td>
<td>4/7</td>
<td>4/7</td>
<td>5/7</td>
<td>0/7</td>
<td>0/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td></td>
<td>4/7</td>
</tr>
<tr>
<td>rest parameters</td>
<td></td>
<td>7/7</td>
<td>4/7</td>
<td>4/7</td>
<td>5/7</td>
<td>0/7</td>
<td>0/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td>4/7</td>
<td></td>
<td>4/7</td>
</tr>
<tr>
<td>spread (... ) operator</td>
<td></td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>0/15</td>
<td>0/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
<td>15/15</td>
</tr>
<tr>
<td>object literal extensions</td>
<td></td>
<td>0/16</td>
<td>0/6</td>
<td>0/6</td>
<td>0/6</td>
<td>0/16</td>
<td>0/6</td>
<td>0/6</td>
<td>0/6</td>
<td>0/6</td>
<td>0/6</td>
<td>0/6</td>
<td>0/6</td>
<td>0/6</td>
<td>0/6</td>
<td></td>
<td>0/6</td>
</tr>
<tr>
<td>for...of loops</td>
<td></td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td>0/9</td>
<td></td>
<td>0/9</td>
</tr>
<tr>
<td>template literals</td>
<td></td>
<td>5/5</td>
<td>4/5</td>
<td>4/5</td>
<td>3/5</td>
<td>2/5</td>
<td>0/5</td>
<td>2/5</td>
<td>2/5</td>
<td>2/5</td>
<td>2/5</td>
<td>2/5</td>
<td>2/5</td>
<td>2/5</td>
<td>2/5</td>
<td></td>
<td>2/5</td>
</tr>
<tr>
<td>RegExp &quot;y&quot; and &quot;u&quot; flags</td>
<td></td>
<td>5/5</td>
<td>3/5</td>
<td>3/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td>0/5</td>
<td></td>
<td>0/5</td>
</tr>
<tr>
<td>destructuring declarations</td>
<td></td>
<td>22/22</td>
<td>20/22</td>
<td>22/22</td>
<td>18/22</td>
<td>19/22</td>
<td>0/22</td>
<td>15/22</td>
<td>19/22</td>
<td>19/22</td>
<td>21/22</td>
<td>22/22</td>
<td>22/22</td>
<td>22/22</td>
<td>22/22</td>
<td></td>
<td>22/22</td>
</tr>
<tr>
<td>destructuring.parameters</td>
<td></td>
<td>23/23</td>
<td>19/23</td>
<td>20/23</td>
<td>17/23</td>
<td>15/23</td>
<td>0/23</td>
<td>13/23</td>
<td>15/23</td>
<td>15/23</td>
<td>15/23</td>
<td>15/23</td>
<td>15/23</td>
<td>15/23</td>
<td>15/23</td>
<td></td>
<td>15/23</td>
</tr>
<tr>
<td>Unicode code point escapes</td>
<td></td>
<td>2/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td></td>
<td>1/2</td>
</tr>
<tr>
<td>new.target</td>
<td></td>
<td>2/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td>0/2</td>
<td></td>
<td>0/2</td>
</tr>
<tr>
<td><strong>Bindings</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>const</td>
<td></td>
<td>16/16</td>
<td>14/16</td>
<td>14/16</td>
<td>15/16</td>
<td>14/16</td>
<td>0/16</td>
<td>16/16</td>
<td>16/16</td>
<td>16/16</td>
<td>16/16</td>
<td>16/16</td>
<td>16/16</td>
<td>16/16</td>
<td>16/16</td>
<td></td>
<td>16/16</td>
</tr>
<tr>
<td>let</td>
<td></td>
<td>12/12</td>
<td>10/12</td>
<td>10/12</td>
<td>10/12</td>
<td>10/12</td>
<td>0/12</td>
<td>12/12</td>
<td>12/12</td>
<td>12/12</td>
<td>12/12</td>
<td>12/12</td>
<td>12/12</td>
<td>12/12</td>
<td>12/12</td>
<td></td>
<td>12/12</td>
</tr>
<tr>
<td>block-level function declaration</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arrow functions</td>
<td></td>
<td>17/13</td>
<td>17/13</td>
<td>17/13</td>
<td>17/13</td>
<td>9/13</td>
<td>0/13</td>
<td>17/13</td>
<td>17/13</td>
<td>17/13</td>
<td>17/13</td>
<td>17/13</td>
<td>17/13</td>
<td>17/13</td>
<td>17/13</td>
<td></td>
<td>17/13</td>
</tr>
<tr>
<td>class</td>
<td></td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td>0/24</td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td>24/24</td>
<td></td>
<td>24/24</td>
</tr>
</tbody>
</table>

**Note:** The data is subject to change and may not be accurate at the time of this text extraction.
TypeScript
[http://www.typescriptlang.org/]

- Typescript → since October 2012, Anders Hejlsberg (lead architect of C# and creator of Delphi and Turbo Pascal)
- Targets large scale client-side and mobile applications by compile time type checking + @Decorators -> Microsoft, Google
- TypeScript is strictly superset of JavaScript, so any JS is valid TS

Source: Google Trends comparison
Developing Single Page Apps (SPA) in 3 steps

1) Setting up a build system – *npm, webpack, gulp* are common choices, *babel, typescript, JSX, CSS preprocessors (SASS, SCSS, LESS)*, *jasmine, karma, protractor, Yeoman/Slush, live servers*

2) Designing front-end architecture components – *views & layouts* + *view models* (presentation data models) + *presentation logic* (event handling, messaging) + *routing paths* (essential for SPA)

   Better to use component model to boost productivity and maintainability.

3) End-to-end application design – front-end: wireframes → views, data entities & data streams → service API and models design, sitemap → router config
Webpack Project Bootstraping

N. Dabit – Beginner’s guide to Webpack:
https://medium.com/@dabit3/beginner-s-guide-to-webpack-b1f1a3638460

Webpack: An Introduction (Angular 2 website):
https://angular.io/docs/ts/latest/guide/webpack.html

SurviveJS – Webpack tutorial (more advanced):
http://survivejs.com/webpack/introduction/
Webpack Demo Structure

- `index.js`
- `AbstractComponent`
  - `FlickrSearchComponent`
  - `AbstractService`
    - `FlickrService`
  - `WikiSearchComponent`
  - `WikiService`
Resources

Thanks for Your Attention!

Questions?