ECMAScript & TypeScript

trainologic

copyright 2016 Trainologic LTD

ECMAScript & TypeScript

- Intro
- Block Scoped Variables
- Arrow Functions
- Rest Parameters
- Template Strings
- Default Parameters
- Computed Property Names
- Destructuring Assignment
- for...of

copyright 2016 Trainologic LTD

ECMA Who?

- **ECMAScript** (or ES)
  - A trademarked scripting language specification
  - Owned by ECMA International
- **ECMA International**
  - **E**uropean **C**omputer **M**anufacturers **A**ssociation
  - A private, non-profit international standards organization
  - Develop standards & reports to facilitate and standardize the use of information communication technology and consumer electronics
  - Members: Adobe, HP, Google, IBM, PayPal, MS, Intel, Hitachi, …
- Spec implementations include:
  - JavaScript
  - ActionScript (Macromedia)
  - JScript (Microsoft)

copyright 2016 Trainologic LTD

ECMAScript – Bit of History

- **1995**: Mocha (JavaScript's original name) developed at Netscape
  - Developed in only 10 days. Interestingly, they soon after also released a server-side scripting version
- **1996**: JS taken to ECMA for standartization
- **1997**: ECMAScript standard edition 1 released
- **1998**: edition 2, ISO alignments (no new features)
- **1999**: edition 3, introducing regex, better string handling, new control statements, try/catch ex. handling and more.
- **In-between**: Edition 4 dropped due to political differences
- **2009**: edition 5, introducing "strict mode", JSON support, object properties reflection and more.
- **2011**: edition 5.1, ISO-3 alignments (no new features)
- **2015**: edition 6, a.k.a. ES6 / ECMAScript 2015 / ES6 Harmony
- **June 2016**: edition 7, with only two features: exponentiation operator (**) and Array.prototype.includes

copyright 2016 Trainologic LTD

ES2017

- Async functions
- Shared memory & atomics

copyright 2016 Trainologic LTD

var's Function Scope

- One of the common complaints has been JavaScript's lack of block scope
- Unlike other popular languages (C/Java/…), blocks ({...}) in JavaScript (pre-ES6) do not have a scope
- Variables in JavaScript are scoped to their nearest parent function, or globally if there is no function present

copyright 2016 Trainologic LTD

let Semantics

- The new ES6 keyword **let** allows scoping variables at the block level (the nearest curly brackets)
- limited in scope to the block, statement, or expression on which it is used

```
var fruit = “guava”;

if (true) {
    let fruit = “mango”;
    console.log(fruit); // mango
}
console.log(fruit); // guava
```

```
var listItems = document.querySelectorAll(‘li’);

for (let i = 0; i < listItems.length; i++) {
    let element =  listItems[i];

    element.addEventListenet(‘click’, function() {
        alert(‘Clicked item number ‘ + i);
    });
}
```


copyright 2016 Trainologic LTD

const

- Syntax:

const *name1 = value1 [, name2 = value2 [, ... [, nameN = valueN]]]*;

- Creates a read-only reference to a value
- Doesn’t mean the value is immutable; only the variable identifier can’t be reassigned
- Constant declarations must be initialized
- Constants are block-scoped, similar to let variables
- Constants values cannot be re-assigned nor re-declared
- All “temporal dead zone” considerations applying to “let” apply here too

copyright 2016 Trainologic LTD

const – Examples

```
const PI = 3.141592;
const API_KEY = 'super*secret*123';
const HEROES = [];

HEROES.push('Jon Snow');  // okay
HEROES.push('Tyrian Lannister');  // okay
HEROES = ['Ramsay Bolton', 'Walder Frey'];   // error
```


copyright 2016 Trainologic LTD

When Do We Use Which?

- One recommendation:
  - Use **const** by default
  - Use **let** if you have to rebind a variable
  - Use **var** to signal untouched legacy code

- But other opinions exist:
  - Use **var** to signal variables used throughout the function (i.e. function scope)

copyright 2016 Trainologic LTD

Arrow Functions

- A.k.a. "Fat Arrow" (because **->** is a thin arrow and **=>** is a fat arrow)
- A.k.a. "Lambda Function" (because of other languages)
- Promotes the functional programming paradigm in JS
- Addresses a JS pain-point of losing the meaning of *this*

- Motivation:
  - No need to keep typing *function*
  - Lexically captures *this* from the surrounding context
  - Lexically captures *arguments* of a function

copyright 2016 Trainologic LTD

Examples

```
var f_1 = (x) => x + 1;  // increment by 1

let f_2 = x => 2 * x;  // muiply by 2

// zero arguments requires using parentheses
const f_3 = () => console.log('look ma, no arguments');

// as anonymous timer callback
setTimeout(() => { console.log('well, it is about time'); }, 1000);
```


copyright 2016 Trainologic LTD

The Lexical *this*

- Until arrow functions, every new function defined its own *this* value:
  - Constructor: new object
  - Strict Mode: undefined
  - "Object Method": the context object
- We had to use a capture variable to keep hold of *this*

copyright 2016 Trainologic LTD

Using an Arrow Function

- The *this* reference is captured from outside the function body

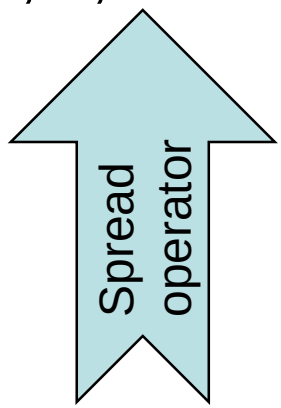
```
// relaxing.js

function QuoteMaster() {

    this.quote = 'luckily we have arrow functions';
    this.sayIt = () => console.log(this.quote);

    setTimeout(this.sayIt, 1000);
}
```

copyright 2016 Trainologic LTD

Rest Parameters

- Convenient way to accept multiple parameters as array
- Denoted by *...restArgsName* as the last argument
- The ellipsis notation (…) is a new *spread operator*
- Reduce boilerplate code induced by the arguments
- Can be used in any function (plain function / fat arrow)
- Syntax:

```
function(a, b, ...allTheRest) {  // … }
```



copyright 2016 Trainologic LTD

Rest Parameters

- Differences between rest parameters and arguments object:

	Rest Parameters	arguments Object
Parameters received	Only those not given separate name	All arguments passed to the function
Is Array?	A real array (supports sort, map, forEach, pop)	Not a real array
Special Properties	None	Has specific functionality, e.g. *callee*
Can be used by arrow function	Yes	No

copyright 2016 Trainologic LTD

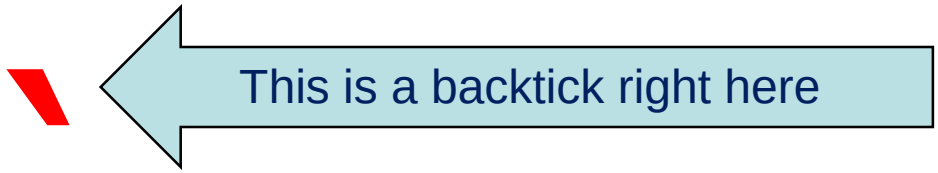
Example – Rest Parameters

```
function getTheOthers(first, second, ...allOthers) {
    console.log(allOthers);
}

// [] empty array since first two args are named (“first”, “second”)
getTheOthers('Cersei Lannister', 'Daenerys Targaryen');

// ['Khal Drogo', 'Roose Bolton', 'Robert Baratheon']
getTheOthers('Cersei Lannister', 'Daenerys Targaryen',
                    'Khal Drogo', 'Roose Bolton', 'Robert Baratheon');
```


copyright 2016 Trainologic LTD

Template Strings (also: String Literals)

- Syntactically these are strings that use backticks



- Motivation:
  - Multiline strings
  - String interpolation (i.e. parameterized)
  - Tagged templates

copyright 2016 Trainologic LTD

Template Strings – Syntax

```
`string text` // simple string literal

`string text line 1
 string text line 2`  // multiline string literal

`string text ${expression} string text`  // interpolation literal

tag `string text ${expression} string text` // tagged template
```


copyright 2016 Trainologic LTD

Default Parameters – cont.

- Replaces the common strategy of testing values in function body:

```
function multiply(a, b) {
    var b = b !== undefined ? b : 1; // yuck!
    ...
```

- Instead we can more elegantly write:

```
function multiply(a, b = 1) {
```


copyright 2016 Trainologic LTD

Destructuring Assignment

- De-structuring literally means breaking up a structure
- Expressions that extract array/object data → distinct variables
- Two destructuring types are supported: Array and Object

- Syntax:

```
// array destructuring assignment
[a, b] = [1, 2]; // a=1, b=2
[a, b, ...rest] = [1, 2, 3, 4, 5] // a=1, b=2, rest= [3,4,5]

// object destructuring assignment
({a, b} = {a:1, b:2}) // a=1, b=2
({a, b, ...rest} = {a:1, b:2, c:3, d:4}); // a=1, b=2, rest={c: 3,d: 4}
```


copyright 2016 Trainologic LTD

Examples – Object Destructuring

```
var lastEpisode = { season: 6, episode: 10, title: "The Winds of Winter", aired: "2016-06-26" };

// dstructuring assignment of all properties
//
var {season, episode, title, aired} = lastEpisode;
console.log(season, episode, title, aired); // 6, 10, "The Winds of Winter", "2016-06-26“

// dstructuring assignment of only few properties
//
var {title, aired} = lastEpisode;
console.log(title, aired); // "The Winds of Winter", "2016-06-26“

// assign extracted variable to new variable name
//
var {title, aired: releaseDate} = lastEpisode;
console.log(releaseDate); // "2016-06-26“
```


copyright 2016 Trainologic LTD

Examples – Array Destructuring

```
var x = 1, y = 2, z = "Zed";
var a, b, others;

// array destructuring + variable renaming
[a, b] = [x, y];
console.log(a, b);  // 1,2

// swap variables
[y, x] = [x, y];
console.log(x, y);  // 2,1

// destructuring with rest parameters
[x, ...others] = [x, y, z];
console.log(others);  // [1, "Zed"]
```


copyright 2016 Trainologic LTD

for...of

- Creates a loop iterating over all values of an iterable object
  - Iterable: Array, Map, Set, String, TypedArray, arguments
- Each iteration invokes a custom iteration hook (callback)
- Syntax:

```
for (variable of iterable) {
    {statement}
}

for ([k, v] of iterable) {  // key-value destructuring for Maps
    {statement}
}
```

☝ Note that for…of iterates over the iterable's values, as opposed to for…in which iterates the iterable's enumerable properties (keys)

copyright 2016 Trainologic LTD

Examples - for...of

- Arrays and for...in vs. for…of

```
var houses = ["Lannister", "Bolton", "Greyjoy", "Arryn", "Baratheon", "Frey"];

// 0, 1, 2, 3, 4, 5
for (var house in houses) {
    console.log(house);
}

// "Lannister", "Bolton", "Greyjoy", "Arryn", "Baratheon", "Frey"
for (var house of houses) {
    console.log(house);
}
```


copyright 2016 Trainologic LTD

Examples - for...of

- for...of with Maps

```
var books = new Map();

books.set(1, "A Game of Thrones");
books.set(2, "A Clash of Kings");
books.set(3, "A Storm of Swords");

// [1, "A Game of Thrones"], [2, "A Clash of Kings"], [3, "A Storm of Swords"]
for (var book of books) {
    console.log(book);
}

// "A Game of Thrones", "A Clash of Kings", "A Storm of Swords"
for (var [sequence, name] of books) {
    console.log(name);
}
```


copyright 2016 Trainologic LTD

Modules

- Before ES6, JS did not have modules, and so libraries were used instead. Now, ES6 finally introduced modules.
- Modules are executed within their own scope: declarations do not pollute the global namespace
- Modules are stored in files: one module per file
- Module name is the file name (w/o extension)
- The *export* and *import* statements are used to import/export module declarations respectively
- Two export types exist: named and default
  - Named exports are useful to export several values
  - Default exports are considered the “main” exported module value. Limited to single default per module.

copyright 2016 Trainologic LTD

Example – Named Exports

```
/* calculator.js */

const COEFFICIENT = 42;

export function calculate(x, y) {
    return x + COEFFICIENT * y;
}

export { COEFFICIENT };
```

```
/* application.js */

import { calculate, COEFFICIENT } from "./calculator";

console.log(calculate(10, 20)); // 42
console.log(COEFFICIENT); // 850
```


copyright 2016 Trainologic LTD

Example – Default Exports

```
/* calculator.js */

const COEFFICIENT = 42;

export default function calculate(x, y) {
    return x + COEFFICIENT * y;
}
```

```
/* application.js */

import calculate from "./calculator";   // no curly braces around calculate

console.log(calculate(10, 20)); // 850
```


copyright 2016 Trainologic LTD

A Word about Module Loaders

- As we've seen, modules can import/use one another
- The actual module files loading is performed by a *module loader*, responsible for:
  - Locating the module files
  - Fetching/loading them into memory
  - Handling module dependencies
  - Executing their code
- This is usually done in runtime (although can be done in compile time e.g. for dist bundling)
- Common module loaders include *requirejs* and *systemjs*

copyright 2016 Trainologic LTD

TS & Modules

- TS needs to know which module loader we will be using, as the compilation output differs for each one
- We define it using the compiler *module* option:

```
// tsconfig.json

{
    "compilerOptions": {
        "target": “es6",
        "module": "commonjs“ // other options: amd, system, es6, umd
    }
}
```

- We will now see how TS transpiles modules to be used for commonjs

copyright 2016 Trainologic LTD

Modules & TS – Named Exports

```
/* calculator.js */

var COEFFICIENT = 42;
exports.COEFFICIENT = COEFFICIENT;

function calculate(x, y) {
    return x + COEFFICIENT * y;
}
exports.calculate = calculate;
```

```
/* application.js */

var calculator_1 = require("./calculator");

console.log(calculator_1.calculate(10, 20));
console.log(calculator_1.COEFFICIENT);
```


copyright 2016 Trainologic LTD

Interfaces

- TS's primary way for composing multiple type annotations into a single named annotation

```
interface Name {
    first: string;
    second: string;
}

var name: Name;
name = { first: 'John', second: 'Doe'  }; // Okay

name = { first: 'John'  }; // Error : `second` is missing
name = { first: 'John', second: 1337 }; // Error : `second` is the wrong type
```


copyright 2016 Trainologic LTD

Special Types - any

- Beyond the primitive types there are few types with special meaning in TS: *any, null, undefined, void*
- ***any***:
  - Compatible with all types
  - Tells the compiler not to do any meaningful static analysis

```
var power: any;

// takes any and all types
power = '123';  // number
power = 123;   // string

// compatible with all types
var num: number;
power = num;
num = power;
```


copyright 2016 Trainologic LTD

Function Types

- Parameter & Return Type annotations

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

function getAge (person: Person): number {
    return person.age;
}
```

- Optional Parameters

```
function addCharacter (name: string, age?: number): void {
    // ..
}

addCharacter('Jon Snow', 24);
addCharacter('Sansa Stark');  // okay, age is optional
```


copyright 2016 Trainologic LTD

Generics

- Many algorithms and data structures in computer science do not depend on the *actual type* of the object
- Allows us to define functions, classes and interfaces that are based on *type parameters*

```
// function based on the type parameter T
function reverse<T>(items: T[]): T[] {
    var reversed = [];
    for (let i = items.length - 1; i >= 0; i--) {
        reversed.push(items[i]);
    }
    return reversed ;
}

var numArr = [1, 2, 3];  // implicitly typed as :number[]
var numArrRev = reverse(numArr);  // returns an array of type :number[] , with values = 3,2,1

var strArr = ['one', 'two'];  // implicitly typed as :string[]
var strArrRev = reverse(strArr);  // returns an array of type :string[] , with values = 'two', 'one'
```


copyright 2016 Trainologic LTD

Generics

- As a matter of fact, JS string's prototype already has a .reverse() function
- TS itself uses generics to define its structure (in lib.d.ts)
- Meaning we get type safety when calling .reverse() on any array

```
/////////////////////////////
/// ECMAScript Array API (specially handled by compiler)
/////////////////////////////

interface Array<T> {

    /**
     * Reverses the elements in an Array.
     */
    reverse(): T[];
```


copyright 2016 Trainologic LTD

Union Type

- Allows a property to be one of multiple types (e.g string or a number)
- Denoted by the pipe sign | in a type annotation (e.g. string|number)

```
// can take a string or array of strings
function formatCommandline(command: string[ ]|string) {
    var line = '';
    if (typeof command === 'string') {
        line = command.trim();
    } else {
        line = command.join(' ').trim();
    }

    // do stuff with line:string …
}
```


copyright 2016 Trainologic LTD

Intersection Type

- Allows us to define a type having members of several types

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U> {};
    for (let id in first) {
        result[id] = first[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            result[id] = second[id];
        }
    }
    return result;
}

var x = extend({ a: "hello" }, { b: 42 }); // x now has both `a` and `b`
console.log(x.a, x.b); // hello 42
```

- Commonly used for mixins (which are convenient replacement for multiple inheritance we don't have in JS)
- Note we're not limited to two types only (e.g. T & U & V & W)

copyright 2016 Trainologic LTD

Classes

- ES5 classes are syntactic sugar over prototypical inheritance
- Classes provide simpler & clearer syntax for dealing with inheritance
- Classes can be defined in similar manner to function expressions and function declarations:

```
// class declaration
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}

var p = new Point(10, 20);
```

```
// class expression
var Point  = class {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}

var p = new Point(10, 20);
```


copyright 2016 Trainologic LTD

Classes – Hoisting

- As opposed to function declarations, class declarations are not hoisted
- Thus class declarations cannot be used before the declaration

```
// ReferenceError !
var p = new Point(10, 20);

// class declaration
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
```

```
// Okay
var f = calc(10, 20);

// function declaration
function calc (x, y) {
    return x * y;
}
```


copyright 2016 Trainologic LTD

Classes – Body & CTor

- The body class is the part within the curly braces {}
- This is where we define properties and methods
- Body code is executed in strict mode
- One special method is the *constructor*, for creating and initializing a class object instance

```
class Point { // body starts here
    constructor(x, y) {
        this.x = x;
        this.y = y;
        console.log('new point created');
    }
} // body ends here
```


copyright 2016 Trainologic LTD

Classes – Prototype Methods

- Methods are defined within the body as follows

```
class Westeros {

    this.kingdoms = [];
    this.maxKingdoms = 7;

    constructor() {
        console.log("Westeros initialized");
    }

    addKingdom(name) {
        if (this.kingdoms.length >= 7) {
            console.log("Sorry, max kingdoms reached");
            return;
        }
        this.kingdoms.push(name);
    }
}
```


copyright 2016 Trainologic LTD

Classes - Sub Classing

- The *extends* keyword is used to create a child class (sub-class)
- A class can only have a single superclass (i.e. single inheritance)
- The *super* keyword is used to access the parent class
  - *super()* invokes the object's parent constructor
  - *super.someMethod()* invokes *someMethod* on the object's parent

```
class Dothraki {
    constructor(name) {
        this.name = name;
        console.log(
                name + " created");
    }
}
```

```
class DothrakiWarrior extends Dothraki{
    constructor(name, weapon) {
        super(name);
        this.weapon= weapon;
        console.log("Weapon = " + weapon);
    }
}
```

```
var khalDrogo = new DothrakiWarrior("Khal Drogo", "Sword");

// Khal Drogo created \n Weapon = Sword
```


copyright 2016 Trainologic LTD

Classes – Static Methods

- The *static* keyword defines static methods (shared across all class instances)
- They are called using the class name (not an instance)

```
class Dothraki {

    constructor(name) {
        this.name = name;
        console.log(name + " created");
    }

    static greet() {
        console.log("Hello, kirekosi are yeri?");
    }
}

console.log(Dothraki.greet()); // Hello, kirekosi are yeri?
```


copyright 2016 Trainologic LTD

Classes & TS

- TypeScript's classes have some additional features which do not exist in ES6:
- **Types**: covered in previous section
- **Access Modifiers*** determine accessibility to class members:

Accessible On	public	private	protected
Class instances	yes	no	no
Class	yes	yes	yes
Class children	yes	no	yes

** At runtime these have no significance, but will raise errors in compile time if you incorrectly used.*

copyright 2016 Trainologic LTD

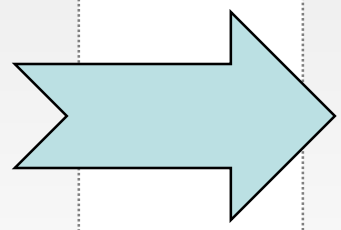
Classes – TS - Example

```
class Point {
    x: number;
    y: number;
    static instances: number = 0;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
        Point.instances++;
    }

    add(point: Point) {
        return new Point(this.x + point.x, this.y + point.y);
    }

    static printNumInstances() {
        console.log("There are " + Point.instances + " points");
    }
}

var p1 = new Point(0, 10);
var p2 = new Point(10, 20);
var p3 = p1.add(p2); // {x:10,y:30}
Point.printNumInstances(); // There are 3 points
```


copyright 2016 Trainologic LTD

Classes – TS - Transpiled

```
class Point {
    x: number;
    y: number;
    static instances: number = 0;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
        Point.instances++;
    }

    add(point: Point) {
        return new Point(
                this.x + point.x, this.y + point.y);
    }

    static printNumInstances() {
        console.log("There are " +
                Point.instances + " points");
    }
}
```



```
var Point = (function () {

    function Point (x, y) {
        Point.instances++;
    }

    Point.prototype.add = function (point) {
        return new Point(
                this.x + point.x, this.y + point.y);
    };

    Point.printNumInstances = function () {
        console.log("There are " +
                Point.instances + " points");
    };

    Point.instances = 0;

    return Point;

}());
```


copyright 2016 Trainologic LTD

Classes – Define Using Constructor

- A very common class member initialization is:

```
class Foo {
    x: number;
    constructor(x:number) {
        this.x = x;
    }
}
```

- TS thus provides a convenient shorthand annotation that does the same:

```
class Foo {
    constructor(public x:number) {
    }
}
```


copyright 2016 Trainologic LTD

Summary

- JavaScript is getting serious
- Typescript takes JavaScript to enterprise level
- ECMAScript 6/7/8 has many interesting features