ECMAScript 6 & TypeScript

-- PART 1 --

trainologic

ECMAScript 6 & TypeScript

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

About This Part

- Focuses on ES6 specific features
- That is: features that exist in ECMAScript 2015
- These features provide extended capabilities
- When the TS compiler *target* option is set to "es5" (*), TS code is transpiled into standard ES5 JS

  * *(this is what we want until ES6 is fully supported)*

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)

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

ES7 – Why So Small?

- ES7 / ECMAScript 2016 is so small due to the new release process, which is actually good
- New features are only included after they are completely ready and after there were at least two implementations that were sufficiently field-tested.
- Releases will now happen much more frequently (once a year) and will be more incremental

Atwood's Law

"Any application that can be written in JavaScript will eventually be written in JavaScript"

Note about Sloppy Mode

- In this presentation you might see mentions of the term “Sloppy Mode”
- This is a common (but unofficial) term referring to the normal, non-strict mode of JavaScript

ECMAScript 6 & TypeScript

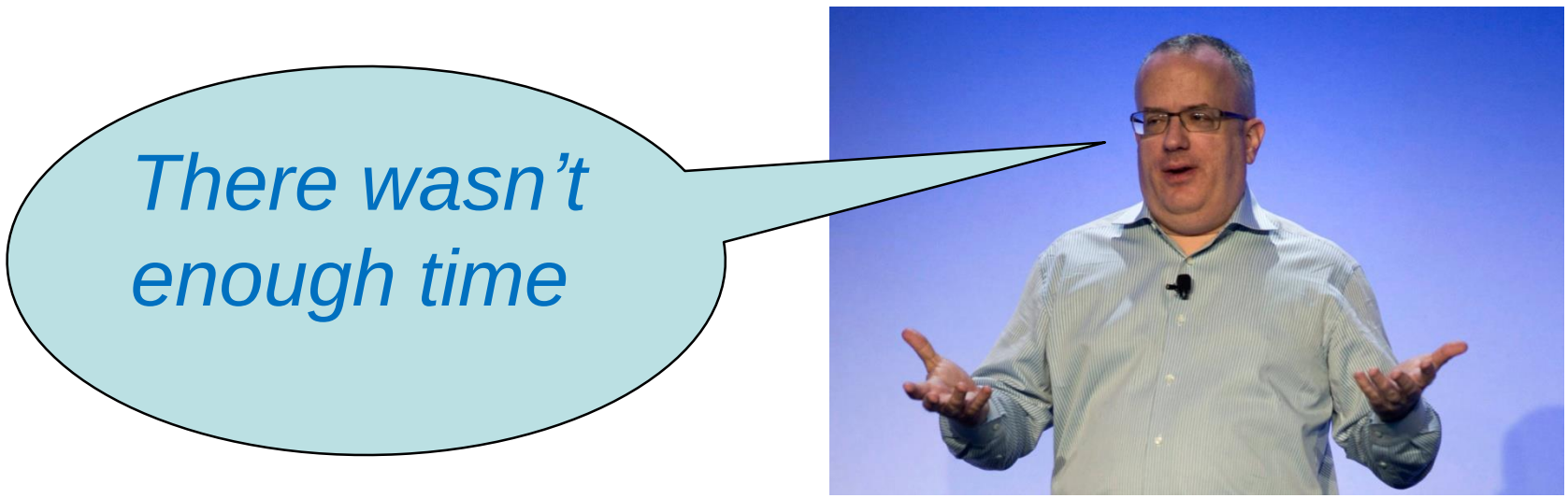
- Intro
- Block Scoped Variables
- Arrow Functions
- Rest Parameters
- Template Strings
- Default Parameters
- Computed Property Names
- Destructuring Assignment
- for...of

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

Why No Block Scope?

- JavaScript was created in 10 days in May 1995 by Brendan Eich, then working at Netscape
- When asked why JavaScript does not have block scopes, Brendan replied:



var Challenges

- Scoping is confusing for developers coming from other languages
- Local vs. Global confusion, accidental shadowing
- Confusing workaround patterns: IIFE
- Misconceptions about hoisting

```
function blocky() {
    if (!hoisty) {
        var hoisty = “gotcha";
    }
    alert(hoisty);  // alerts “gotcha" instead of reference error
}
blocky();
```

The let Statement

- Using "let" instead (ECMAScript 6) is more intuitive

```
function blocky() {
    if (!hoisty) {
        let hoisty = "gotcha";
    }
    alert(hoisty);  // reference error: hoisty is not defined
}
blocky();
```

- let Syntax (similar to "var"):

```
let var1 [= value1] [, var2 [= value2]] [, ..., varN [= valueN]];
```

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);
    });
}
```

let Limitations

- Cannot be re-declared in same block scope
  - SyntaxError: Identifier … has already been declared
  - Also applies in switch-case blocks
  - Also applies to using **var x** after **let x** statement
  - Can't shadow function argument names

- let variables cannot be referenced before their declaration
  - The variable is hoisted to top of block
  - however it is in "temporal dead zone" and cannot be accessed
  - Will result in ReferenceError

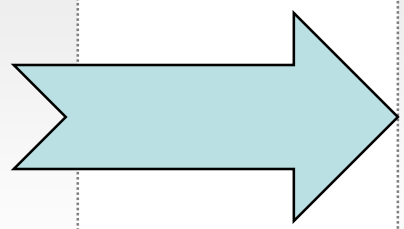
var vs. let

```
var x = ‘global’;
let y = ‘not global’;

console.log(this.x);  // “global”
console.log(this.y);  //  undefined
```

Block Scopes & TS - let

- TS transpiles *let* to *var* declarations
- Renames variable name if it already exists in surrounding scope

```
// fooya.ts

var foo = 'fooya!';

If (true) {
    let foo = 'nununu!';
}

console.log(foo); // fooya
```

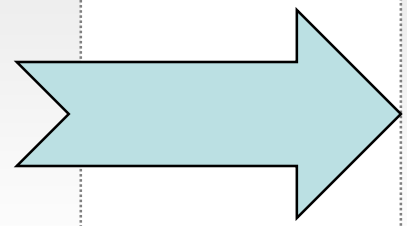


```
// fooya.js

var foo = 'fooya!';

If (true) {
    var foo_1 = 'nununu!';
}

console.log(foo); // fooya
```

Block Scopes & TS – let cont.

- TS transpiles *let* to *var* in for loops too
- For simple loops, TS ends it there

```
// loopy.ts

for (let i = 0; I < 5; i++) {
    console.log(i);
}
```

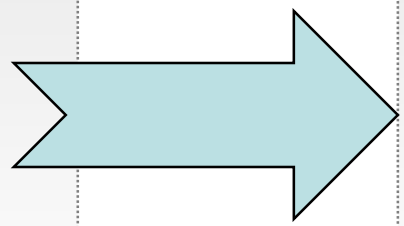


```
// loopy.js

for (var i = 0; I < 5; i++) {
    console.log(i);
}
```

Block Scopes & TS – let cont.

- When closures in for loops are detected however, TS will extract the call out to a separate function

```
// closure.ts

var funcs = [];

for (let i = 0; I < 5; i++) {
    funcs.push(function() {
        console.log(i);
    });
}
```



```
// closure.js

var funcs = [];

var _loop_1 = function(i) {
    funcs.push(function() {
        console.log(i);
    });
};

for (var i = 0; I < 5; i++) {
    _loop_1(i);
}
```

Browser Compatibility - let

Desktop | Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	41.0	12	44 (44)	11	17	?
Temporal dead zone	?	12	35 (35)	11	?	?
`let` expression ⚠	No support	No support	No support	No support	No support	No support
`let` block ⚠	No support	No support	No support	No support	No support	No support
Allowed in sloppy mode	49.0	?	44 (44)	?	?	?

Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	?	41.0	44.0 (44)	?	?	?	41.0
Temporal dead zone	?	?	35.0 (35)	?	?	?	?
`let` expression ⚠	No support	?	No support	No support	No support	No support	No support
`let` block ⚠	No support	?	No support	No support	No support	No support	No support
Allowed in sloppy mode	No support	49.0	44 (44)	?	?	?	49.0

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

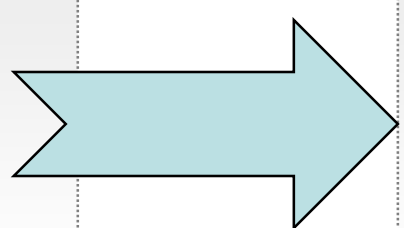
const – Examples

```
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
```

Block Scopes & TS - const

- TS simply transpiles *const* to *var* declarations

```
// constantin.ts

const c = 84;

function funk() {
    const c = 42;
}
```



```
// constantin.js

var c = 84;

function funk{
    var c = 42;
}
```

Browser Compatibility - const

Desktop | Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	21	(Yes)	36 (36)	11	12	5.1
Reassignment fails	20	(Yes)	13 (13)	11	?	?
Allowed in sloppy mode	49.0					

Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	(Yes)	?	?	(Yes)	?	(Yes)
Reassignment fails	No support	(Yes)	?	?	(Yes)	?	(Yes)
Allowed in sloppy mode	No support	49.0					49.0

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)

ECMAScript 6 & TypeScript

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

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

Basic Syntax

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
        // equivalent to:  => { return expression; }

// Parentheses are optional with a single parameter:
(singleParam) => { statements }
singleParam => { statements }

// A function with no parameters requires parentheses:
() => { statements }
```

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);
```

Advanced Syntax

```
// Parenthesize the body to return an object literal expression
params => ({foo: bar})

// Rest parameters and default parameter values
(param1, param2, ...rest) => { statements }
(param1 = defaultValue1, param2, …, paramN = defaultValueN) =>
   { statements }

// Destructuring within the parameter list
var f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;
f();  // 6
```

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*

Using a Capture Variable

- That can become very annoying, especially with OOP

```
// annoying.js

function QuoteMaster() {

    var self = this;
    this.quote = 'if only we had arrow functions';

    this.sayIt = function() {
        console.log(self.quote);
     };

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

Using an Arrow Function

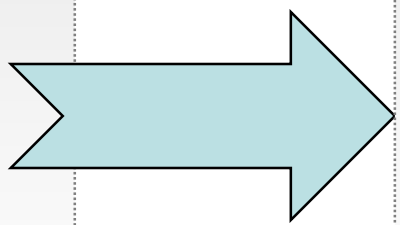
- The *this* reference is captured from outside the function body

```
// relaxing.js

function QuoteMaster() {

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

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

Arrow Functions & TS

- Behind the scenes TS generates a capture variable

```
// relaxing.ts

function QuoteMaster() {

    this.quote = 'fat arrow rulez';
    this.sayIt = () =>
            console.log(this.quote);

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



```
// relaxing.js

function QuoteMaster() {

    var _this = this;  // capture variable
     this.quote = 'fat arrow rulez';

     this.sayIt = function () {
          return console.log(_this.quote);
     };

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

Browser Compatibility – Arrow Function

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Edge	IE	Opera	Safari
Basic support	45.0	22.0 (22.0)	(Yes)	No support	32	No support

Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	45.0	22.0 (22.0)	No support	No support	No support	45.0

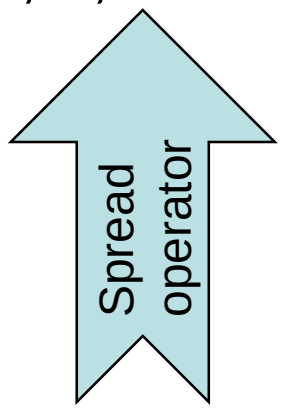
ECMAScript 6 & TypeScript

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

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) {  // … }
```



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*

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');
```

Rest Parameters & TS

```
// reverts to using the arguments object

function getTheOthers(first, second) {
    var allOthers = [];
    for (var _i = 2; _i < arguments.length; _i++) {
        allOthers[_i - 2] = arguments[_i];
    }
    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');
```

Browser Compatibility - Rest Parameters

Desktop | Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	47	(Yes)	15.0 (15.0)	No support	34	No support

Desktop | Mobile

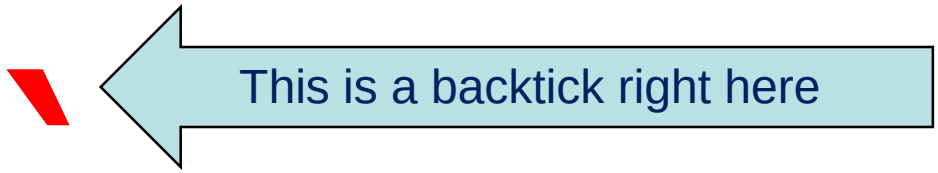
Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	47	15.0 (15.0)	No support	No support	No support	47

ECMAScript 6 & TypeScript

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

Template Strings (also: String Literals)

- Syntactically these are strings that use backticks



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

Template Strings – cont.

- **Mutiline Strings**
  - Allows us to easily create a string spanning multiple lines
- **String Interpolation**
  - Allow us to create string templates with placeholders
  - Placeholder expressions are evaluated into the resulting string
- **Tagged Templates**
  - Allow us to place a function (called a *tag*) before the template string
  - The tag function gets the opportunity to pre-process the template string literals and placeholder expressions
  - Can be used for example for escaping the string

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
```

Examples – Multiline & Interpolation

```
// multiline
var debugLyrics = `Catch, catch, catch a bug.
Put it in a jar.
Sometimes they fly, sometimes they die,
but most get squashed on your car.`;

// interpolation
let htmlString = `<div class="song">${debugLyrics}</div>`;

// hack, we can practically interpolate any expression
const theAnswer = `2 times 21 make ${2 * 21}`;
```

Example – Tagged Template

```
var animal = "dog";
var result = myTagFunc `${animal}s are the best!`;

function myTagFunc(literals, ...values) {   // a sample tag function
    let result = "";

    for (let i = 0; i < values.length; i++) {   // interleave the literals with the values
        result += literals[i];
        result += values[i] === animal ? 'literal string' : values[i];   // replace dawg
}

    result += literals[literals.length - 1]; // add the last literal
    return result;
}

console.log(result);  // literal strings are the best!
```

Template Strings & TS

- TS transpiles multiline strings → escaped strings

```
var debugLyrics = "Catch, catch, catch a bug.\u00A0\v\nPut it in a jar.\u00A0\v\nSometimes they fly, \nsometimes they die,\u00A0\n\vbut most get squashed on your car.";
```

- TS transpiles string interpolations → string concatenations

```
var theAnswer = "2 times 21 make " + 2 * 21;
```

Template Strings & TS

- TS transpiles tagged templates → function calls

```
var animal = "dog";
var result = (_a = ["", "s are the best!"], _a.raw = ["", "s are the best!"], myTagFunc(_a, animal));

function myTagFunc(literals) {   // a sample tag function
    var values = [];
    for (var _i = 1; _i < arguments.length; _i++) {
        values[_i - 1] = arguments[_i];
    }
    var result = "";
    for (var i = 0; i < values.length; i++) {   // interleave the literals with the values
        result += literals[i];
        result += values[i] === animal ? 'literal string' : values[i]; // replace dawg
    }

    result += literals[literals.length - 1];   // add the last literal
    return result;
}
console.log(result);   // literal strings are the best!
var _a;
```

Browser Compatibility – Template Strings

Desktop | Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	41	(Yes)	34 (34)	No support	28	9

Desktop | Mobile

Feature	Android	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	No support	41	34.0 (34)	No support	28	9

ECMAScript 6 & TypeScript

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

Default Parameters

- In JavaScript, parameters of functions default to undefined
- It is useful in some situations to set different defaults
- Default function parameters allow formal parameters to be initialized with default values if no value or undefined is passed

- Syntax:

```
function [name]([param1[ = defaultValue1 ]
                [, ..., paramN[ = defaultValueN ]]])
                { statements }
```

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) {
```

Default Parameters – Example

```
function sendRaven(to, body, subject = 'New Raven Mail') {
    console.log(`Sending mail with subject "${subject}"`);
}

var recipients = ['Lord Commander<lord.commander@castleblack.org',
                  'Maester<maester@castleblack.org'];

// Sending mail with subject "New Raven Mail"
sendRaven(recipients, "The winter is coming");

// Sending mail with subject "New Raven Mail"
sendRaven(recipients, "The winter is coming", undefined);

// Sending mail with subject “Winter Sale!"
sendRaven(recipients, "The winter is coming", “Winter Sale!");
```

Default Parameters – cont.

- Default parameters are available to consequent default parameters

```
function runWeirdCalc(a, b, c = 42, d = c / 2) {
    console.log(`Calculation yields: ${a * b + c + d} (${a} * ${b} + ${c} + ${d})`;

}

runWeirdCalc();  // Calculation yields: NaN (undefined * undefined + 42 + 21)
runWeirdCalc(1);  // Calculation yields: NaN (1 * undefined + 42 + 21)
runWeirdCalc(1, 2);  // Calculation yields: 65 (1 * 2 + 42 + 21)
runWeirdCalc(1, 2, 3);  // Calculation yields: 6.5 (1 * 2 + 3 + 1.5)
runWeirdCalc(1, 2, 3, 4);  // Calculation yields: 9 (1 * 2 + 3 + 4)
```

Default Parameters – cont.

- Default parameters can even accept other default values, such as function calls, *this* and the *arguments* object

```
function getD() {
    return "You got Dee!"
}

function checkThisOut(a, b = 5, c = b, d = getD(), e = this,
                      f = arguments, g = this.whatsThis) {
    return [a,b,c,d,e,f,g];
}

// ["Whoa", 5, 5, "You got Dee!", Window, Arguments[1], undefined]
// (Note: Arguments only contains “Whoa”)
console.log(checkThisOut("Whoa"));
```

Default Parameters – cont.

- As opposed to other languages (C# et al.), defaults can be provided to any parameter(s), not necessarily consecutive or in any particular order

```
function func (a = 42, b, c = a, d, e = “Cool”) {
    return [a,b,c,d,e];
}

console.log(func(undefined, 15, “Yeah”)); // [42, 15, "Yeah", undefined, "Cool"]
```

Default Parameters – cont.

- Destructured parameter with default value assignment

```
function func([x, y] = [1, 2], {z: z} = {z: 3}) {
  return x + y + z;
}

func(); // 6
```

Default Parameters & TS

- TypeScript transpiles to code which sets defaults in case the argument value is undefined
- (it uses the void operator to obtain the *undefined* primitive)

```
function sendRaven(to, body, subject) {
    if (subject === void 0) { subject = 'New Raven Mail'; }
    console.log("Sending mail with subject \"" + subject + "\"");
}
```

```
function runWeirdCalc(a, b, c, d) {
    if (c === void 0) { c = 42; }
    if (d === void 0) { d = c / 2; }
    console.log("Weird calculation yields: " + (a * b + c + d) +
                    " (" + a + " * " + b + " + " + c + " + " + d + ")");
```

Default Parameters & TS – cont.

```
function checkThisOut(a, b, c, d, e, f, g) {
    if (b === void 0) { b = 5; }
    if (c === void 0) { c = b; }
    if (d === void 0) { d = getD(); }
    if (e === void 0) { e = this; }
    if (f === void 0) { f = arguments; }
    if (g === void 0) { g = this.whatsThis; }
    return [a, b, c, d, e, f, g];
}
```

Browser Compatibility – Default Parameters

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	49	15.0 (15.0)	No support	No support	No support
Parameters without defaults after default parameters	49	26.0 (26.0)	?	?	?
Destructured parameter with default value assignment	No support	41.0 (41.0)	?	?	?

Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	49	15.0 (15.0)	No support	No support	No support	49
Parameters without defaults after default parameters	No support	49	26.0 (26.0)	?	?	?	49
Destructured parameter with default value assignment	No support	?	41.0 (41.0)	?	?	?	?

ECMAScript 6 & TypeScript

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

Computed Property Names

- ES6 introduces the ability to define object property names based on computed keys

- Syntax:

```
obj[{computed_expression}] = {value}

// usage in object literals
obj = {
    [{computed_expression}]: {value}
};
```

Examples

```
var x = 100, y = "abc";

function getPropName() {
    return ++x;
}
//
// object literal
//
var literal = {
   ["prop_" + getPropName()]: "Example 1",
   ["prop_" + y]: "Example 2"
};
console.log(literal);   // {prop_101: "Example 1", prop_abc: "Example 2"}

//
// create a new computed property name (member) on the function object
//
getPropName["static_" + getPropName()] = y;

console.log(getPropName.static_102); // abc
```

Computed Property Names & TS

- Computed literal property names are transpiled into separate expressions, one per property

```
var x = 100;
var y = "abc";
function getPropName() {
    return ++x;
}
var literal = (_a = {}, // new empty object is constructed
    _a["prop_" + getPropName()] = "Example 1", // computer property 1
    _a["prop_" + y] = "Example 2", // computer property 2
    _a // final statement returns the object
);
console.log(literal);
getPropName["static_" + getPropName()] = y; // no special handling for this case
console.log(getPropName.static_102);
var _a; // object reference variable is declared as _a
```

Browser Compatibility – Dynamic Property Names

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Computed property names	(Yes)	34 (34)	No support	No support	7.1

Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Computed property names	No support	(Yes)	34.0 (34)	No support	No support	No support	No support

ECMAScript 6 & TypeScript

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

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=[3,4]
```

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“
```

Examples – Deep Object Destructuring

```
// create an object with nested properties
var lastEpisodeWithInfo = {
    season: 6, episode: 10, title: "The Winds of Winter", aired: "2016-06-26",  extraInfo: {
        chapter: 60, director: "Miguel Sapochnik", author: "David Benioff & D.B. Weiss“
    }
};

//  note the deep object destructuring
var {extraInfo: {chapter, "director": directedBy}} = lastEpisodeWithInfo;

console.log("directed by " + directedBy);  //  directed by Miguel Sapochnik
```

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"]
```

Examples – Array Destructuring – cont.

- We can ignore any index by using a sparse assignments array
- Ignore particular values by leaving a location empty (i.e. , ,) in the left hand side of the assignment

```
var v1 = "take me", v2 = "ignore me", v3 = "take me too",
      v4 = "I'm in", v5 = "last but not least";

var one, three, others;

[one,    , three, ...others] = [v1, v2, v3, v4, v5];
//       ^-- note the empty location here. v2 will be ignored

console.log(one, three, others);
// "take me", "take me too", ["I'm in", "last but not least"]
```

Object Destructuring & TS

- TS transpiles into simple value extraction and variable assignment

```
var lastEpisodeWithInfo = {
    season: 6, episode: 10, title: "The Winds of Winter", aired: "2016-06-26", extraInfo: {
        chapter: 60, director: "Miguel Sapochnik", author: "David Benioff & D. B. Weiss"
    }
};

var _a = lastEpisodeWithInfo.extraInfo, // temporary handle
    chapter = _a.chapter, // simple property destructure
    directedBy = _a["director"]; // destructure + rename

console.log("directed by " + directedBy);
```

Array Destructuring & TS

```
var v1 = "take me", v2 = "ignore me", v3 = "take me too",
    v4 = "I'm in", v5 = "last but not least";

var one, three, others;

// temporary array is used from which destructured variables are cherry picked
_a = [v1, v2, v3, v4, v5], one = _a[0], three = _a[2], others = _a.slice(3);
// note the slice from index 3 to get the …rest parameters ----------------------^

console.log(one, three, others);

var _a;
```

Destructuring Assignment & TS

- Computed literal property names are transpiled into separate expressions, one per property

```
var x = 100;
var y = "abc";
function getPropName() {
    return ++x;
}
var literal = (_a = {}, // new empty object is constructed
    _a["prop_" + getPropName()] = "Example 1", // computer property 1
    _a["prop_" + y] = "Example 2", // computer property 2
    _a // final statement returns the object
);
console.log(literal);
getPropName["static_" + getPropName()] = y; // no special handling for this case
console.log(getPropName.static_102);
var _a; // object reference variable is declared as _a
```

Browser Compatibility - Destructuring Assignment

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Edge	Internet Explorer	Opera	Safari
Basic support	49.0	2.0 (1.8.1)	14[1]	No support	No support	7.1

Desktop | Mobile

Feature	Android	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	49.0	1.0 (1.0)	No support	No support	8	49.0

ECMAScript 6 & TypeScript

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

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)

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);
}
```

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);
}
```

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");

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

for...of & TS

- TS transpiles for…of loops into a standard for (…) loop

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

// no special transpiling with for…in loops
for (var house in houses) {
    console.log(house);
}

// transpiled into simple for (…) loop
for (var _i = 0, houses_1 = houses; _i < houses_1.length; _i++) {
    var house = houses_1[_i];
    console.log(house);
}
```

for...of & TS

- Same for 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");

for (var _i = 0, books_1 = books; _i < books_1.length; _i++) {
    var book = books_1[_i];
    console.log(book);
}
```

Browser Compatibility – for...of

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Edge	Opera	Safari
Basic support	38 [1] 51 [3]	13 (13) [2]	12	25	7.1

Desktop | Mobile

Feature	Android	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	?	38 [1]	13.0 (13) [2]	?	?	8