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

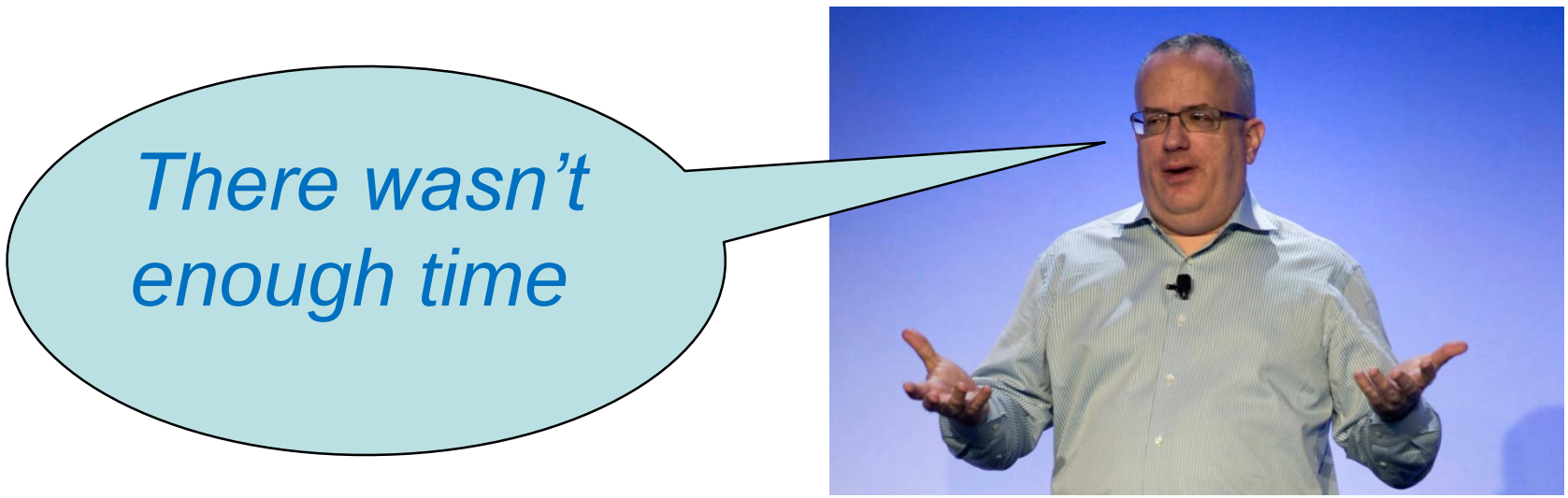
- Intro
- Block Scoped Variables
- Arrow Functions
- Rest Parameters
- Template Strings
- Default Parameters
- Computed Property Names
- Destructuring Assignment
- for...of

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

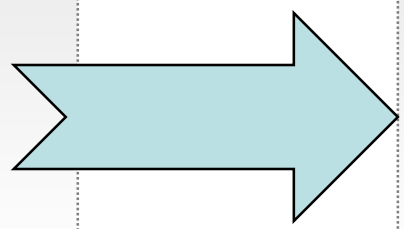
var vs. let

```
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
```

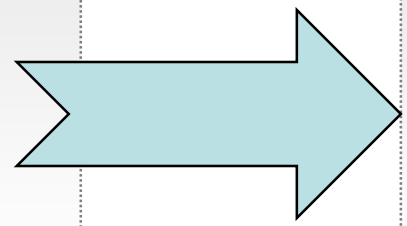


```
// fooya.js

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);
}
```

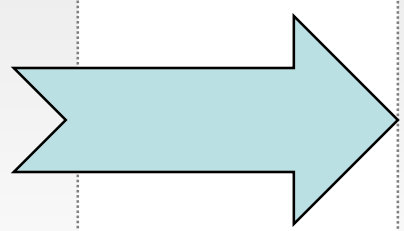


```
// loopy.js

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

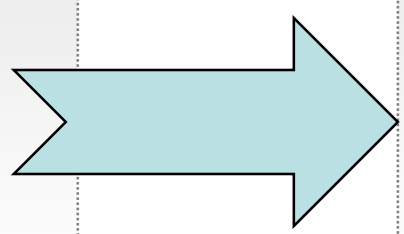
const – Examples

```
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

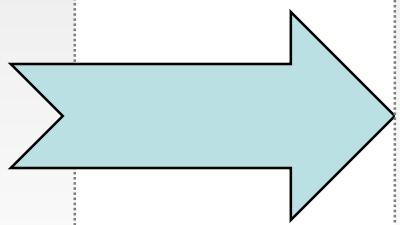
- The *this* reference is captured from outside the function body

```
// 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

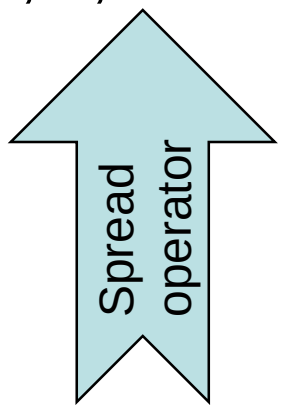
ECMAScript 6 & TypeScript

- 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

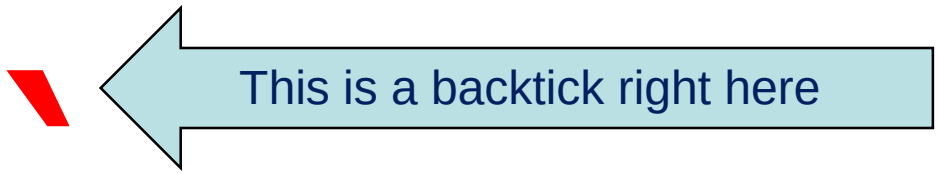
Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	47	15.0 (15.0)	No support	No support	No support	47

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

ECMAScript 6 & TypeScript

-- PART 2 --

ECMAScript 6 & TypeScript

- Enums
- Modules
- Types
- Classes
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Enums

- A way to organize a collection of related values
- Enum members have *numeric* values associated with them and can be either constant or computed
- TS Only. JS does not provide enums
- Enums are number based
- Syntax:

```
enum EnumName {
    elem1 [= initializer1],
    elem2 [= initializer2],
    …
    elemN [= initializerN]
}
```

Example

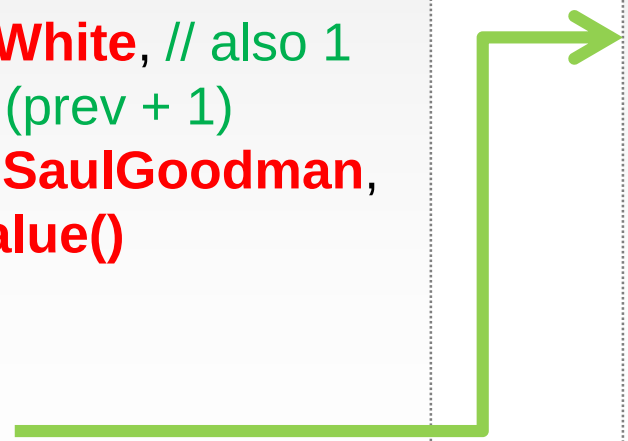
```
enum Characters {
    WalterWhite,
    SkylerWhite,
    SaulGoodman,
    JessePinkman,
    GusFring
}

var main = Characters.WalterWhite;

console.log(main);  // 0
console.log(main === Characters.WalterWhite);  // true
```

Enum Values

- Enum members have numeric values associated with them
- Generally:
  - First element receives a default value of 0
  - Other elements receive previous element's value + 1
  - Multiple elements can have same value
- However values can also be computed:
  - Expressions evaluating to a number
  - Expressions using previous members
  - Function calls
- Some computed values are known & defined at compile time, others only at runtime (e.g. function calls)

Example

```
function getGusValue() {
    return 99;
}

enum Characters {
    WalterWhite = 1,
    SkylerWhite = WalterWhite, // also 1
    SaulGoodman, // = 2  (prev + 1)
    JessePinkman = 10 * SaulGoodman,
    GusFring = getGusValue()
}

console.dir(Characters);
```



```
Object
    1:"SkylerWhite"
    2:"SaulGoodman"
    20:"JessePinkman"
    99:"GusFring"
    GusFring:99
    JessePinkman:20
    SaulGoodman:2
    SkylerWhite:1
    WalterWhite:1
```

Which Transpiles to...

```
function getGusValue() {
    return 99;
}

var Characters;

(function (Characters) {
    Characters[Characters["WalterWhite"] = 1] = "WalterWhite";
    Characters[Characters["SkylerWhite"] = 1] = "SkylerWhite";
    Characters[Characters["SaulGoodman"] = 2] = "SaulGoodman";
    Characters[Characters["JessePinkman"] = 20] = "JessePinkman";
    Characters[Characters["GusFring"] = getGusValue()] = "GusFring";
})(Characters || (Characters = {}));

console.dir(Characters);
```

Const Enums

- TS generated an object with both forward (name -> value) and reverse (value -> name) mappings, as we've seen earlier
- References to enum members are always emitted as property accesses, example:
  - console.log(*Characters*.JessePinkman);
- For a performance boost we can create *const* enums
- Const enum references use inline values
  - But then we can't use computed members ☹

Example

```
const enum Characters {
    WalterWhite = 1,
    SkylerWhite = WalterWhite,
    SaulGoodman,
    JessePinkman = 10 * SaulGoodman,
    // sorry, no computed values allowed ☹ */
    GusFring = 99 /* getGusValue() */
}

console.log(Characters.JessePinkman);
```

Which Transpiles to...

```
// no enum object created
// values are inlined

console.log(20 /* JessePinkman */);
```

Browser Compatibility

- This is a TypeScript specific feature not supported natively by JS

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Classes
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

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.

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
```

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
```

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*

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

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);
```

Modules & TS – Default Exports

```
/* calculator.js */

var COEFFICIENT = 42;

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

// module mode marker for interoperability
Object.defineProperty(exports, "__esModule", { value: true });
exports.default = calculate; // actual exported default value
```

```
/* application.js */

var calculator_1 = require("./calculator");
console.log(calculator_1.default(10, 20)); // imported as “default”
```

Browser Compatibility - import

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Edge	Opera	Safari
Basic support	No support	No support[1]	No support	Build 14342	No support	No support

Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	36.0	No support	No support	No support	No support	36.0

Browser Compatibility - export

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	No support	No support	No support	No support	No support

Desktop | Mobile

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

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Classes
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Types

- One of the main goals/reasons for using TypeScript
- Types enhance code quality and understandability, and allow us to catch errors at compile time
- JS natively has six (6) primitive data types:
  - string
  - number
  - boolean
  - null
  - undefined
  - symbol ← new in ES6
- TS uses these (and more e.g. interfaces, classes, arrays, …) for its typing system

Implicit Typing

- TS does not force using types; they are intentionally optional
- Any plain JS can be renamed .js -> .ts an will still compile
- TS attempts to infer types from the code and provide compile time type safety

```
var mynumber = 42; // TS implicitly infers type number

mynumber = "great number“; // assignment of a string value into a number typed var

// will compile but raise a compile time error
// Error: TS2322: Type 'string' is not assignable to type 'number'.
```

- However, using *explicit* types (next slide) greatly improve TS's ability to warn us of potential errors/bugs

Explicit Typing

- The basic *type annotations* are as follows:

```
var base: number = 123;

function multiply (num: number): number {
    return base * num;
}
```

- Anything that is available in the type *declaration space* can be used as a type annotation
- The declaration space includes JS primitive types, interfaces, enums, functions, classes, arrays

Primitives & Arrays

```
var num: number;
var str: string;
var bool: boolean;

num = 123;
num = 123.456;
num = '123'; // Error

str = '123';
str = 123; // Error

bool = true;
bool = false;
bool = 'false'; // Error
```

```
var boolArray: boolean[];

boolArray = [true, false];
console.log(boolArray[0]); // true
console.log(boolArray.length); // 2
boolArray[1] = true;
boolArray = [false, false];

boolArray[0] = 'false'; // Error!
boolArray = 'false'; // Error!
boolArray = [true, 'false']; // Error!
```

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
```

Inline Type Annotation

- Instead of creating an interface we can annotate inline

```
var name: {
    first: string;
    second: string;
};

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

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

- Great for quickly providing a one off type annotation
- However, if repeatedly used consider refactoring into an interface (or a *type alias* covered later)

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;
```

Special Types – null & undefined

- treated the same as something of type *any*
- These literals can be assigned to any other type

```
var num: number;
var str: string;

// these literals can be assigned to anything
num = null;
str = undefined;
```

Special Types – void

- Use :*void* to signify that a function has no return type (and value)

```
function log(message: string): void {
    console.log(message);
}
```

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
```

Function Overloading

- Allows us to define two or more functions with the same name but different signatures

```
class Person {
    constructor(public name:string, public age:number) {}
}

function getAge (x: Person[]): number;
function getAge (x: Person): number {
    if (x instanceof Person) {
        return x.age;
    }
    return group.map(p => p.age ).reduce((a1, a2) => (a1 + a2), 0);
}

var group = [];
group.push(new Person('Jack', 30));
group.push(new Person('Jill', 28));
group.push(new Person('Dave', 15));

console.log(getAge(group[0])); // 30
console.log(getAge(group)); // 73
```

Type Guards

- Allows narrowing down an object type within conditional block
- TS understands the variable type within that conditional block

```
// as seen in previous example

if (x instanceof Person) {    // TS understands that within this block x is a of type Person
      return x.age;   // and therefore allows us to access the 'age' property
}
```

- We can even create user defined type guards (out of scope)

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'
```

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[];
```

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 …
}
```

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)

Tuple Type

- Tuples are finite ordered list of elements
- Syntax ***:[type1, type2, … typeN]***

```
var nameNumber: [string, number];

nameNumber = ['Saul Goodman', 5055034455];  // Okay
nameNumber = ['Saul Goodman', '5055034455']; // Error!

var [name, num] = nameNumber; // destructure
```

Type Alias

- Used for providing names for reusable type annotations
- Syntax ***type someName = anyValidTypeAnnotation***

```
type StrOrNum = string|number;

var sample: StrOrNum; // used like any other notation

sample = 123;  // okay
sample = '123'; // okay
sample = true;  // error
```

- Type aliases can be created for any type really

```
type Text = string | { text: string };  //  union
type Coordinates = [number, number];  // tuple
type Callback = (data: string) => void;  // callback
```

lib.d.ts

- A special declaration file that ships with every TS installation
- Contains the *ambient declarations* (next slide) for common JS constructs (JS runtimes and the DOM)
  - Automatically included in compilation context of TS projects
  - Makes it easy for us to start writing type checked JS code

```
var foo = 123;
var bar = foo.toString(); // okay since lib.d.ts is included

// but if we set compiler flag to noLib: true in tsconfig.json then …

var bar = foo.toString();  // ERROR: Property 'toString' does not exist on type 'number'.
```

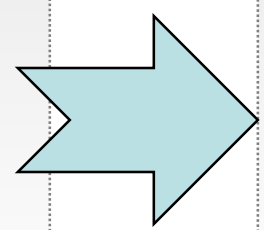
Ambient Declarations

- Used to provide type information (definitions) for existing JS code / libraries, either 3rd party or our own
- Contain the type information but not the implementation
- This provides us with type-checking and auto-completion without the need to re-write the code in TS
- Files usually end with a **.d.ts** extension
- There are many ambient declarations already written for us (jquery, angular, moment, …)
- We can use dev tools such as *Typings* for fetching existing .d.ts files
- .d.ts files are actually a great source for documentation and good declaration practices to learn from

JS & Types

- The typing system we covered is TS specific
- JS knows nothing about it
- All types are completely removed when transpiled

```
class Person {
    constructor(public name:string,
                public age:number) { }
}

function getAge(x: Person[]): number;
function getAge(x: Person): number {
    if (x instanceof Person) {
        return x.age;
    }
    return group.map(p => p.age )
       .reduce((a1, a2) => (a1 + a2), 0);
}
```



```
var Person = (function () {
    function Person(name, age) {
        this.name = name;
        this.age = age;
    }
    return Person;
}());

function getAge(x) {
    if (x instanceof Person) {
        return x.age;
    }
    return group.map(function (p) { return p.age; })
        .reduce(function (a1, a2) { return (a1 + a2); }, 0);
}
```

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Classes
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

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);
```

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;
}
```

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
```

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);
    }
}
```

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
```

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?
```

Browser Compatibility - Classes

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Edge	Internet Explorer	Opera	Safari
Basic support	42.0[1] 49.0	45	13	No support	No support	9.0

Desktop | Mobile

Feature	Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	45	?	?	9	42.0[1] 49.0

Classes & TS

- TypeScript's classes have some additional features which do not exist in ES6:
- **Types**: covered in previous section
- **Properties:** class value members (as opposed to methods)
- **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

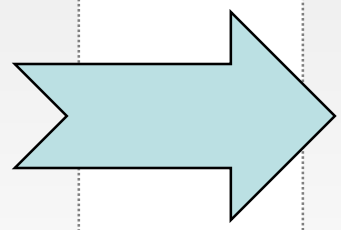
** At runtime these have no significance, but will raise errors in compile time if you incorrectly used.*

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
```

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;

}());
```

Classes – Define Using Constructor

- A very common class member initialization is:

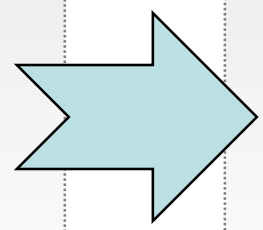
```
class Foo {
    x: number;
    constructor(x:number) {
        this.x = x;
    }
}
```

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

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

Define Using Constructor – TS - Transpiled

```
class Foo {

    constructor(public x:number) {
    }

}
```



```
var Foo = (function () {

    function Foo(x) {
        this.x = x;
    }

    return Foo;

}());
```

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Types
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Iterators

- Iterators are a Behavioral Design Pattern common for OOP languages
- Used for processing/going over collations, which is a very common task
- *Iterators* bring the iteration concept directly into core JS
- Provide a mechanism for customizing the behavior of *for…of* loops
- Iterators are objects that know how to access collection items one at a time, keeping track of the current item
- An iterator's *next()* method returns an object with two properties:
  - *done* – boolean indicating whether no more items left
  - *value* – the item value

Example

```
function makeOddIterator (array){
    var nextIndex = 0;

    return { // the iterator
        next: function() {
            var retval = nextIndex < array.length ?  {value: array[nextIndex], done: false} : {done: true};
            nextIndex += 2;
            return retval;
        }
    }
}

var iter = makeOddIterator(['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']);

for (var item = iter.next(); !item.done; item = iter.next()) {
    console.log(item.value);
}
// one, three, five, seven
```

Iterables

- An object is *iterable* if it defines its iteration behavior
  - Such as which values are looped over in a *for..of* construct
- To be *iterable*, an object must implement the @@iterator method
- Some built-in types, such as Array or Map, have a default iteration behavior (e.g. Array, Map, String), while others (e.g Object) do not
- Some statements and expressions actually expect iterables:

```
for(let value of ["a", "b", "c"]){ // for...of loop
        // ...
}

[..."abc"]; // ["a", "b", "c"] // spread operator

[a, b, c] = new Set(["a", "b", "c"]); // destructuring assignment
```

User Defined Iterable - ES6

```
let iterable = {
    0: 'a',
    1: 'b',
    2: 'c',
    length: 3,
    [Symbol.iterator]() {
        let index = 0;
        return {
            next: () => {
                let value = this[index];
                let done = index >= this.length;
                index++;
                return { value, done };
            }
        };
    }
};
for (let item of iterable) {
    console.log(item); // 'a', 'b', 'c'
}
```

User Defined Iterable - TS

```
class IterableStuffCollection implements IterableIterator<any> {

    private pointer = 0;
    constructor(private stuff:any[]) { }

    public next():IteratorResult<any> {
        let value = this.stuff[this.pointer];
        let done = this.pointer >= this.stuff.length;
        this.pointer++;
        return {value, done};
    }

     [Symbol.iterator]():IterableIterator<any> {
         return this;
     }
}

var myStuff = new IterableStuffCollection(['XBox One', 42, Math.PI, 'pokemon go', {oh: 'yeah'}]);
for (let item of myStuff) {
    console.log(item);
}

// XBox One, 42, 3.141592653589793, pokemon go, { oh: 'yeah' }
```

User Defined Iterable – TS - Notes

- Previous code example require ES6 target

```
// tsconfig.json

"compilerOptions": {
      "target": "ES6“
 }
```

- It could also work with ES5 target, but will require:
  - JS engine supporting Symbol.iterator (nodejs 4+, Google Chrome)
  - Using ES6 lib with ES5 target (add *es6.d.ts* to your project)

```
// lib.es6.d.ts

interface IterableIterator<T> extends Iterator<T> {
    [Symbol.iterator](): IterableIterator<T>;
}
```

TS Transpiled Code

```
class IterableStuffCollection {
    constructor(stuff) {
        this.stuff = stuff;
        this.pointer = 0;
    }
    next() {
        let value = this.stuff[this.pointer];
        let done = this.pointer >= this.stuff.length;
        this.pointer++;
        return { value: value, done: done };
    }
     [Symbol.iterator]() {  // note that ES6 Symbol.iterator / iteration protocol is required
        return this;
    }
}
var myStuff = new IterableStuffCollection(['XBox One', 42, Math.PI, 'pokemon go', { oh: 'yeah' }]);
for (let item of myStuff) {
    console.log(item);
}
```

Browser Compatibility

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	39.0	27.0 (27.0)	No support	26	No support
`IteratorResult` object instead of throwing	(Yes)	29.0 (29.0)	No support	(Yes)	No support

Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	(Yes)	27.0 (27.0)	No support	No support	No support	39.0
`IteratorResult` object instead of throwing	No support	?	29.0 (29.0)	No support	No support	No support	(Yes)

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Types
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Generators

- Generators are a new breed of functions in JS, with a new syntax:

  function *

- Calling a generator function does not execute its body immediately
  - Instead, an *iterator* object for the function is returned
- We then iterate the generator by repeatedly calling *next()*
- next() executes the body function until the next *yield* expression returns a value
- Since the generator is really a function, we can call next() with arguments
- Execution can be further delegated to another generator function using a *yield * generator* expression

Generators - Motivation

1. **Lazy Iterators** – examples:
   - Return a finite or infinite list of values
   - Lazy execution/loading

2. **Externally Controlled Execution**
   - Allows a function to pause execution and pass control to the caller
   - Re-entering the function again later, while keeping context (variable bindings) across re-entrances
   - We can control its behavior by passing arguments to the generator

Generators – Lazy Iteration

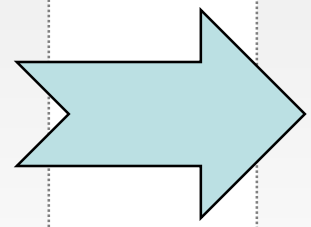
```
function* idMaker() {   // generator function
  var index = 0;
  while(index < 3)   // note this is a finite iterator
    yield index++;
}

var gen = idMaker(); // returns iterator

console.log(gen.next().value);      // 0
console.log(gen.next().value);      // 1
console.log(gen.next().value);      // 2
console.log(gen.next().value);      // undefined
```

Generators – Function Args

```
function* addCallNumber (base) {
    var callNumber = 0;
    while (true) {
        yield base + callNumber++;
    }
}

var gen = addCallNumber(10); // invoke generator with argument(s)
console.log(gen.next().value);  // 10
console.log(gen.next().value);  // 11
console.log(gen.next().value);  // 12
```

Passing Arguments Into Generators

```
"use strict";

function* showPrevCurrGenerator() {

    var prev, curr;
    while (true) {
        console.log('-------');
        prev = curr;
        curr = yield;
        console.log('prev = ' + prev);
        console.log('curr = ' + curr);
    }
}

var gen = showPrevCurrGenerator();

gen.next(); // executes until the first yield
gen.next('First');
gen.next('Second');
gen.next('Third');
```



```
-------
prev = undefined
curr = First
-------
prev = First
curr = Second
-------
prev = Second
curr = Third
-------
```

Generators – yield*

```
function* anotherGenerator(i) {
    yield i + 0.1;
    yield i + 0.2;
    yield i + 0.3;
}

function* generator(i){
    yield '0.01';
    yield* anotherGenerator(i);
    yield i * 10;
}

var gen = generator(10);

console.log(gen.next().value); // 0.01
console.log(gen.next().value); // 10.1
console.log(gen.next().value); // 10.2
console.log(gen.next().value); // 10.3
console.log(gen.next().value); // 100
```

TS Transpiled Code

```
// This is a pure ES6 feature, TS doesn't do any magic
// Generators require ES6 support in the browser/node

function* idMaker() {
    var index = 0;
    while (index < 3) {
        yield index++;
    }
}
```

Browser Compatibility - Desktop

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Edge	Opera	Safari (WebKit)
Basic support	39.0	26.0 (26.0)	No support	13	26	No support
`yield*`	(Yes)	27.0 (27.0)	No support	13	26	No support
`IteratorResult` object instead of throwing	(Yes)	29.0 (29.0)	No support	13	(Yes)	No support
Not constructable with `new` as per ES2016	(Yes)	43.0 (43.0)	?	?	?	?

Browser Compatibility - Mobile

Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	(Yes)	26.0 (26.0)	No support	No support	No support	39.0
`yield*`	No support	(Yes)	27.0 (27.0)	No support	No support	No support	(Yes)
`IteratorResult` object instead of throwing	No support	?	29.0 (29.0)	No support	No support	No support	(Yes)
Not constructable with `new` as per ES2016	?	?	43.0 (43.0)	?	?	?	?

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Types
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

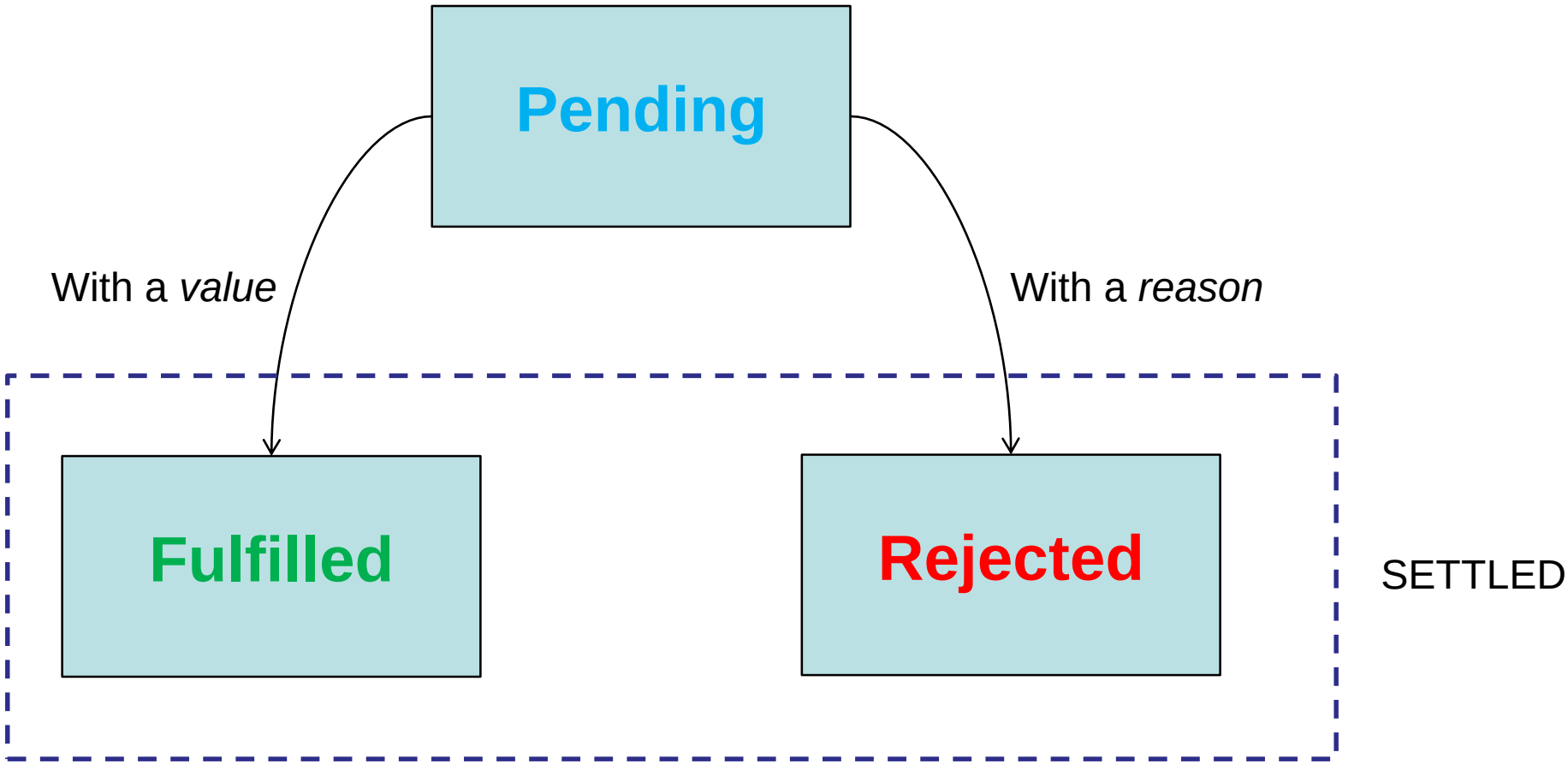
Promises

- A Promise represents an operation that hasn't completed yet, but is expected in the future
- Used for asynchronous computations
- Promises are chainable. This is a key benefit
- Syntax:

```
new Promise(function(resolve, reject) { ... } );
```

- The promise Ctor takes a single argument: an executor function
  - Executed immediately (even before returning the new Promise object)
  - *resolve* and *reject* functions are bound to the promise and calling them fulfills or rejects the promise, respectively
  - The executor function is expected to initiate some async work, and then invoke either *resolve* or *reject*

Promise States



Methods - *then*

- **Promise.then(onFulfilled, onRejected)**
  - Appends fulfillment and rejection handlers to the promise
  - Returns a new promise resolving to
    - The return value of the called handler (onFulfilled / onRejected)
    - Or to its original settled value if the promise was not handled (i.e. if the relevant handler onFulfilled or onRejected are not a function)
  - We say that promises are "thenable" objects
  - Allows us to create chains since *then()* returns a promise
    - We call this "composition"

Methods – *catch*

- **Promise.catch(onRejected)**
  - Appends a rejection handler callback to the promise
  - Returns a new promise resolving to
    - The return value of the callback if it is called
    - Or to original fulfillment value if the promise is fulfilled
  - Allows us to create chains since *catch()* returns a promise

Methods – Other

- **Promise.all(iterable)**
  - Takes a list of promises and returns a promise that
    - Resolves when all promises resolve
    - Or rejects as soon as any promise fails
- **Promise.race(iterable)**
  - Takes a list of promises and returns a promise that
    - Resolves as soon as any promise resolves
    - Or rejects as soon as any promise rejects
- **Promise.resolve(value) / Promise.reject(reason)**
  - Shortcuts returning an already resolved/rejected promise
  - Useful for example for initiating a chain

Example - Chaining

```
Promise.resolve(123)
    .then((res) => {
        console.log(res); // 123
        return 456;
    })
    .then((res) => {
        console.log(res); // 456
        return Promise.resolve(123);
    })
    .then((res) => {
        console.log(res);  // 123 : Notice `this` is called with the resolved value
        return Promise.resolve(123);
    })
```

Example – Aggregated Error Handling

```
Promise.reject(new Error('something bad happened'))
    .then((res) => {
        console.log(res); // not called
        return 456;
    })
    .then((res) => {
        console.log(res); // not called
        return Promise.resolve(123);
    })
    .then((res) => {
        console.log(res); // not called
        return Promise.resolve(123);
    })
    .catch((err) => {
        console.log(err.message); // something bad happened
    });
```

Example – *catch* Chaining

```
Promise.reject(new Error('something bad happened'))
    .then((res) => {
        console.log(res);  // not called
        return 456;
    })
    .catch((err) => {
        console.log(err.message);  // something bad happened
        return Promise.resolve(123);
    })
    .then((res) => {
        console.log(res);  // 123
    });
```

TS & Promises

- TS understands promises flow of values

```
Promise.resolve(123)
    .then((res)=>{
        // res is inferred to be of type `number`
        return true;
    })
    .then((res) => {
        // res is inferred to be of type `boolean`
    });
```

TS & Promises – cont.

- TS also understands unwrapping function calls that return a promise

```
function iReturnPromiseAfter1Second():Promise<string> {
    return new Promise((resolve)=>{
        setTimeout(()=>resolve("Hello world!"), 1000);
    });
}

Promise.resolve(123)
    .then((res)=>{
         // res is inferred to be of type `number`
         return iReturnPromiseAfter1Second();
    })
    .then((res) => {
        // res is inferred to be of type `string`
        console.log(res); // Hello world!
    });
```

TS – Converting CB to Promise

```
import fs = require('fs');

function readFileAsync (filename:string):Promise<any> {
    return new Promise((resolve,reject)=> {
        fs.readFile(filename,(err,result) => {
            if (err) reject(err);
            else resolve(result);
        });
    });
}
```

TS – Transpiled Code

- TS does not do any "magic" with promises
- It relies on ES6 promises or a polyfill

```
function iReturnPromiseAfter1Second() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Hello world!"), 1000);
    });
}
Promise.resolve(123)
    .then((res) => {
    return iReturnPromiseAfter1Second();
})
    .then((res) => {
    console.log(res); // Hello world!
});
```

Browser Compatibility - Desktop

Desktop | Mobile

Feature	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Servo
Promise	32.0	(Yes)	29.0	No support	19	7.1	No support
Contructor requires new	32.0	(Yes)	37.0	No support	19	10	No support
Promise.all	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype.catch	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype.then	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.race	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.reject	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.resolve	32.0	(Yes)	29.0	No support	19	7.1	No support

Browser Compatibility - Mobile

Desktop | Mobile

Feature	Android	Chrome for Android	Edge Mobile	Firefox for Android	IE Mobile	Opera Mobile	Safari Mobile
Promise	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Contructor requires new	4.4.4	32.0	(Yes)	37.0	No support	(Yes)	10
Promise.all	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.prototype	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.prototype.catch	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.prototype.then	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.race	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.reject	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.resolve	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Types
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Maps

- The Map object is a simple key/value dictionary
- Any value (both objects and primitive values) may be used as either a key or a value
- Syntax:

```
new Map([iterable])
```

- *Iterable* is an optional other iterable object whose elements are key-value pairs
- for…of looping on a map returns an [key, value] array (in insertion order) each iteration
- Key equality is based on "same value" algorithm
  - NaN is considered same as Nan (although in JS they're not)
  - All other values go by the === semantics

Maps vs. JS Objects

- Similar in that both let us set/retrieve/delete/check values by keys
- The main differences are:
  - An Object has a prototype, so we might have default keys
  - Object keys are Strings or Symbols, but can be any value for Map
  - Map's size can be retrieved easily, difficult with an Object
- Still, in many cases it is perfectly okay to continue using Objects

Map Properties & Methods

- ***size*** – Returns the number of k/v pairs in the Map object
- ***clear()*** – Removes all k/v pairs
- ***delete(key)*** – Removes value, returns true/false if deleted/not-found
- ***entries()*** – returns a new Iterator containing an array of [k,v] pairs per each iteration, in insertion order
- ***keys()*** – Returns a new Iterator containing keys in insertion order
- ***values()*** – Returns a new Iterator containing values in insertion order
- ***forEach(cbFn)*** – calls cbFn for each k/v pair in insertion order
- ***has(k)*** – Returns true if key exists in the Map
- ***get(k)*** – Returns the value if key k exists, undefined otherwise
- ***set(k, v)*** – Sets the value for the key, returns the Map (for chaining)
- ***[@@iterator]()*** – Returns a new Iterator containing [k,v] array for each element in insertion order

Maps – Example: Simple

```
var myMap = new Map();

var keyString = "a string",
    keyObj = {},
    keyFunc = function () {};

// setting the values
myMap.set(keyString, "value associated with 'a string'");
myMap.set(keyObj, "value associated with keyObj");
myMap.set(keyFunc, "value associated with keyFunc");

myMap.size;  // 3

// getting the values
myMap.get(keyString);       // "value associated with 'a string'"
myMap.get(keyObj);          // "value associated with keyObj"
myMap.get(keyFunc);         // "value associated with keyFunc"

myMap.get("a string");      // "value associated with 'a string'" because keyString === 'a string'
myMap.get({});              // undefined, because keyObj !== {}
myMap.get(function() {}) ;  // undefined, because keyFunc !== function () {}
```

Maps – Example: Iterating

```
var myMap = new Map();

myMap.set(0, "zero");
myMap.set(1, "one");

for (var [key, value] of myMap) {  // 0 = zero, 1 = one
    console.log(key + " = " + value);
}

for (var key of myMap.keys()) {  // 0, 1
    console.log(key);
}

for (var value of myMap.values()) {  // zero, one
    console.log(value);
}

for (var [key, value] of myMap.entries()) {  // 0 = zero, 1 = one
    console.log(key + " = " + value);
}

myMap.forEach(function(value, key) {  // 0 = zero, 1 = one
   console.log(key + " = " + value);
});
```

TS – Transpiled Code

- TS does not do any "magic" with Maps
- It relies on ES6 Maps or a polyfill

```
var myMap = new Map();

myMap.set(0, "zero");
myMap.set(1, "one");

for (var [key, value] of myMap) {
    console.log(key + " = " + value);
}
for (var key of myMap.keys()) {
    console.log(key);
}
for (var value of myMap.values()) {
    console.log(value);
}
for (var [key, value] of myMap.entries()) {
    console.log(key + " = " + value);
}
myMap.forEach(function (value, key) {
    console.log(key + " = " + value);
});
```

Browser Compatibility - Desktop

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	38 [1]	13 (13)	11	25	7.1
Constructor argument: `new Map(iterable)`	38	13 (13)	No support	25	No support
iterable	38	17 (17)	No support	25	7.1
`Map.clear()`	31 38	19 (19)	11	25	7.1
`Map.keys()`, `Map.values()`, `Map.entries()`	37 38	20 (20)	No support	25	7.1
`Map.forEach()`	36 38	25 (25)	11	25	7.1
Key equality for -0 and 0	34 38	29 (29)	No support	25	No support
Constructor argument: `new Map(null)`	(Yes)	37 (37)	?	?	?
Monkey-patched `set()` in Constructor	(Yes)	37 (37)	?	?	?
`Map[@@species]`	?	41 (41)	?	?	?
`Map()` without `new` throws	?	42 (42)	?	?	?

Browser Compatibility - Mobile

Desktop | Mobile

Feature	Android	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	No support	38 [1]	13.0 (13)	No support	No support	8
Constructor argument: `new Map(iterable)`	No support	38	13.0 (13)	No support	No support	No support
iterable	No support	No support	17.0 (17)	No support	No support	8
`Map.clear()`	No support	31 38	19.0 (19)	No support	No support	8
`Map.keys()`, `Map.values()`, `Map.entries()`	No support	37 38	20.0 (20)	No support	No support	8
`Map.forEach()`	No support	36 38	25.0 (25)	No support	No support	8
Key equality for -0 and 0	No support	34 38	29.0 (29)	No support	No support	No support
Constructor argument: `new Map(null)`	?	(Yes)	37.0 (37)	?	?	?
Monkey-patched `set()` in Constructor	?	(Yes)	37.0 (37)	?	?	?
`Map[@@species]`	?	?	41.0 (41)	?	?	?
`Map()` without `new` throws	?	?	42.0 (42)	?	?	?

Sets

- Set objects are collections of values, which we can iterate according to insertion order
- Sets let us store unique values of any type, whether primitive values or object references
- Syntax:

```
new Set([iterable])
```

- If an iterable object is passed, all of its elements will be added to the new Set
- Value equality is similar to ===

```
var set = new Set();
set.add({a:1});
set.add({a:1});
console.log(set.size) // 2
console.log([...set.values()]); // Array [ Object, Object ]
```

TS – Transpiled Code

- TS does not do any "magic" with Sets
- It relies on ES6 Sets or a polyfill

TODO

Browser Compatibility - Desktop

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	38 [1]	13 (13)	11	25	7.1
Constructor argument: `new Set(iterable)`	38	13 (13)	No support	25	9.0
iterable	38	17 (17)	No support	25	7.1
`Set.clear()`	38	19 (19)	11	25	7.1
`Set.keys()`, `Set.values()`, `Set.entries()`	38	24 (24)	No support	25	7.1
`Set.forEach()`	38	25 (25)	11	25	7.1
Value equality for -0 and 0	38	29 (29)	No support	25	No support
Constructor argument: `new Set(null)`	(Yes)	37 (37)	?	?	?
Monkey-patched `add()` in Constructor	(Yes)	37 (37)	?	?	?
`Set[@@species]`	?	41 (41)	?	?	?
`Set()` without `new` throws	?	42 (42)	?	?	?

Browser Compatibility - Mobile

Desktop | Mobile

Feature	Android	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	No support	38 [1]	13.0 (13)	No support	No support	8
Constructor argument: `new Set(iterable)`	No support	38	13.0 (13)	No support	No support	No support
iterable	No support	No support	17.0 (17)	No support	No support	8
`Set.clear()`	No support	38	19.0 (19)	No support	No support	8
`Set.keys()`, `Set.values()`, `Set.entries()`	No support	38	24.0 (24)	No support	No support	8
`Set.forEach()`	No support	38	25.0 (25)	No support	No support	8
Value equality for -0 and 0	No support	38	29.0 (29)	No support	No support	No support
Constructor argument: `new Set(null)`	?	(Yes)	37.0 (37)	?	?	?
Monkey-patched `add()` in Constructor	?	(Yes)	37.0 (37)	?	?	?
`Set[@@species]`	?	?	41.0 (41)	?	?	?
`Set()` without `new` throws	?	?	42.0 (42)	?	?	?