ECMAScript 6

trainologic

ECMAScript 6

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

ECMAScript 6

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

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

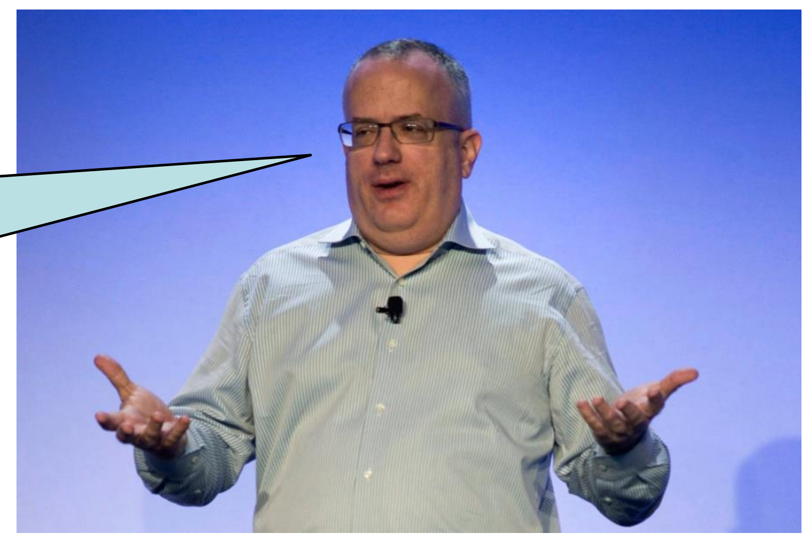
var’s Function Scope

- 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:

There wasn't enough time



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 <u>is</u> 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
```

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
```

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)

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);
```

trainologic

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
```

trainologic

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);
}
```

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**) { // … }

Spread operator

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');
```

Template Strings (also: String Literals)

- Syntactically these are strings that use backticks

`

This is a backtick right here

- 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!
```

ECMAScript 6

- 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
```

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
```

Destructuring Assignment

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

- Syntax:

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

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

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"]
```

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 <u>values</u>, as opposed to for...in which iterates the iterable's <u>enumerable properties</u> (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);
}
```

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*

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 <u>not hoisted</u>
- 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?
```

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'
}
```

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
```

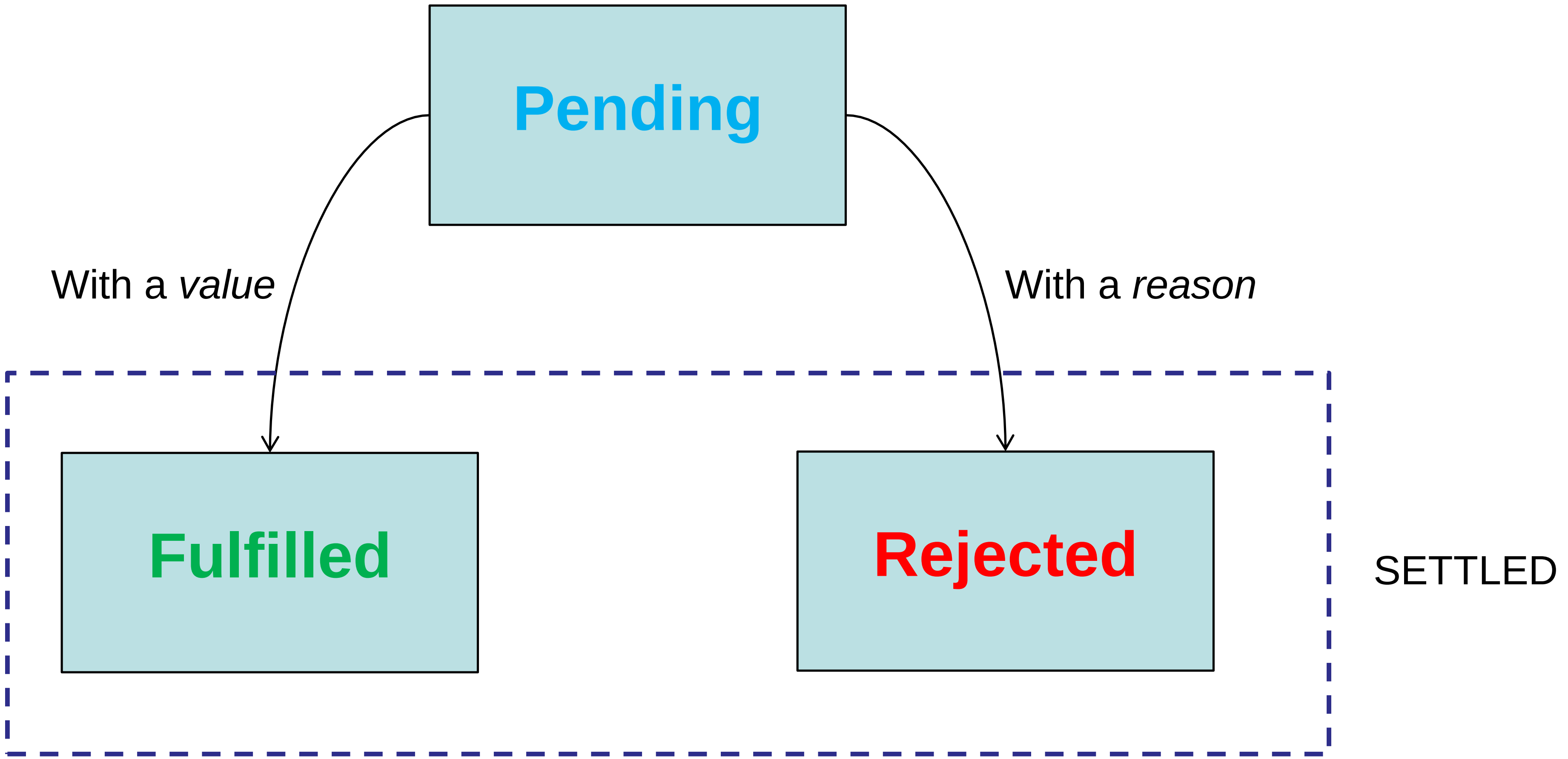
Promises

- 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
    });
```

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 [, this])*** – calls cbFn for each k/v pair in insertion order. If this is provided, will be applied to cbFn
- ***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);
});
```

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 ===
- two objects are equal only if they refer to the exact same object

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

Set Properties & Methods

- ***size*** – Returns the number of elements pairs in the Set object
- ***add()*** – Appends a new element
- ***clear()*** – Removes all elements from the Set object
- ***delete(value)*** – Removes element and returns true/false if value existed(deleted) or not
- ***entries()*** – Returns a new Iterator object containing an array of [value, value] for each element, in insertion order
- ***forEach(cbFn [, this])*** – Calls cbFn for each value in the Set object in insertion order. If this is provided – will be applied to cbFn
- ***has(value)*** – Returns a boolean indicating whether value exists
- ***values()*** – Returns a new Iterator containing all element values
- ***keys()*** – Same as *values()*
- ***[@@iterator]()*** – Returns a new Iterator containing all values in insertion order

Sets – Example: Simple

```
var mySet = new Set();

mySet.add(1);
mySet.add(1); // does nothing, 1 is already in the set

mySet.add(5);
mySet.add("some text");
var o = {a: 1, b: 2};
mySet.add(o);

mySet.has(1); // true
mySet.has(3); // false
mySet.has(Math.sqrt(25));  // true (5 exists)
mySet.has("Some Text".toLowerCase()); // true
mySet.has(o); // true


mySet.size; // 4

mySet.delete(5); // removes 5 and returns true (5 existed before deletion)
mySet.has(5);    // false, 5 has been removed

mySet.size; // 3, we just removed one value
```

Sets – Example: Iterating

```
// ... continuing our previous example

for (let item of mySet) console.log(item); //  1, some text, Object {a: 1, b: 2}

for (let item of mySet.keys()) console.log(item); //  1, some text, Object {a: 1, b: 2}

for (let item of mySet.values()) console.log(item); //  1, some text, Object {a: 1, b: 2}

for (let [key, value] of mySet.entries()) console.log(key); //  1, some text, Object {a: 1, b: 2}

mySet.forEach(e => console.log(e)); //  1, some text, Object {a: 1, b: 2}

console.log([...mySet]); // [1, “some text”, Object]
```

WeakMap & WeakSet

- The “Weak” counterparts of Map and Set
- Weakly hold references to keys/values stored
- Adding an element to the collection does’nt increase reference count
- When the element is freed up, the collection will no longer contain that element
- Syntax:

```
new WeakMap([iterable])
new WeakSet([iterable])
```

WeakMap & WeakSet – Cont.

- When there are no more references (in our code) to an object stored in the collection, it is garbage collected
- That means there is no list of objects stored in the collection
- Therefore weak collections are not enumarable
- Available methods – WeakMap:
  - ***delete(), get(key), has(key), set(key, value)***
- Available methods – WeakSet:
  - ***add(value), get(value), has(value)***

WeakMap - Example

```
var wm = new WeakMap();

var keys = {
    key1: {}
};

wm.set(keys.key1, "some value associated with key");

console.log(wm.get(keys.key1)); // "some value associated with key"

delete keys.key1; // we'll now delete the key object

console.log(wm.get(keys.key1)); // undefined
```

WeakSet - Example

```
var ws = new WeakSet();

var keys = {
    key1: {}
};

ws.add(keys.key1);

console.log(ws.has(keys.key1)); // true

delete keys.key1; // we'll now delete the key object

console.log(ws.has(keys.key1)); // false
```