ECMAScript 2015

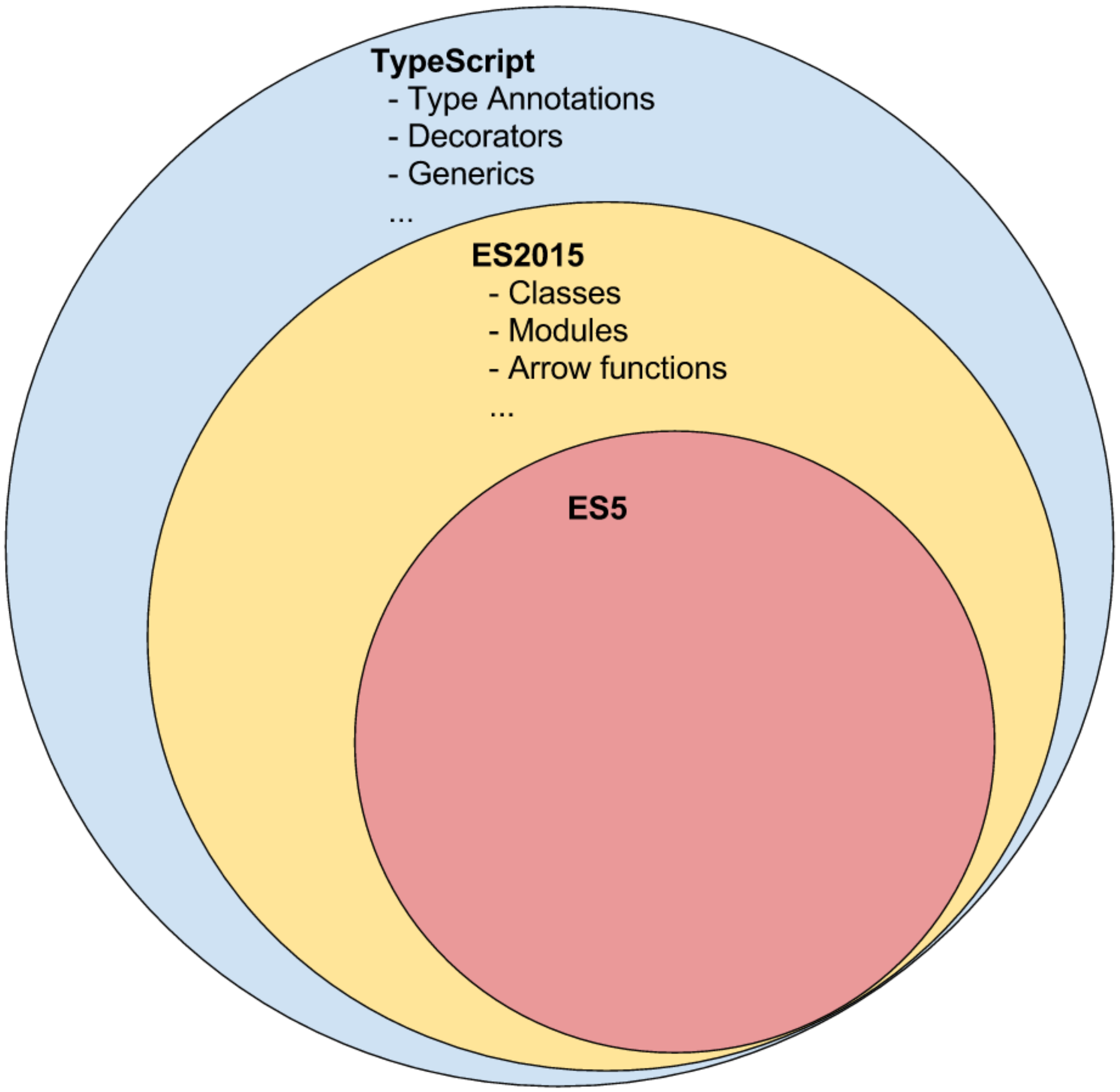
&

TypeScript

ECMAScript 2015 (ES2015)

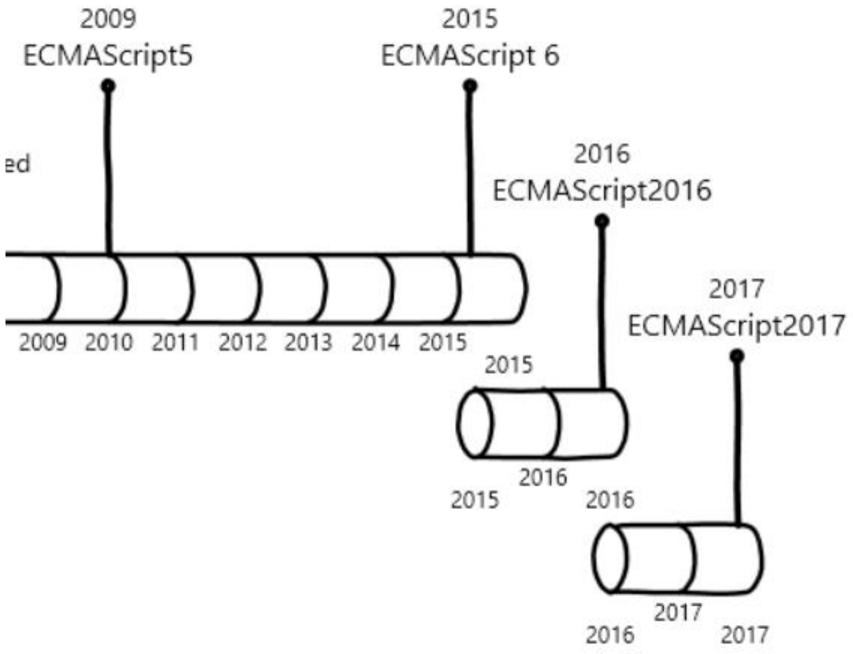
< Back

ES2015 & TypeScript Big Picture



ES2015 Status

- Approved June 17, 2015
- Largest Update in JavaScript's History
  - Future updates will be much smaller and more frequent



ECMAScript 2015 features

- Variables: var, let, const
- OOP: classes, inheritance, super, get/set
- Functions: generators, iterators, arrow functions
- Loops: for-of
- Data Structures: set/weakset, map/weakmap
- Async operations: built-in promises
- Modules: imports, exports
- Objects: computed properties, shorthand properties, Object.is(), Object.assign(), proxies
- Others: templates, Math and Number extensions

ES2015 basic new features

< Back

ES2015 Variables

- ES2015 introduces new ways to declare variables:
  - **let** – creates a block scope variable
    - Accessible only in its scope

```
for(let number of [1, 2, 3, 4]){
  console.log(number);
}
//accessing number here throws exception
```

- **const – creates a constant variable**
  - **Its value is read-only and cannot be changed**

```
const MAX_VALUE = 16;
MAX_VALUE = 15; // throws exception
```

For-of loop

- The for-of loop iterates over the values
  - Of an array

```
let sum = 0;
for(let number of [1, 2, 3])
  sum+= number;
```

  - Of An iteratable object

```
function* generator(maxValue){
  for(let i = 0; i < maxValue; i+=1){
    yield i;
  }
}
let iter = generator(10);
for(let val of iter()){
  console.log(val);
}
```

Templated Strings in ES2015

- ES2015 supports templated strings
  - i.e. strings with placeholders:

```
let people = [new Person('Samir', 'Saghir'), … ];
for(let person of people){
   log(`Fullname: ${person.fname} ${person.lname}`);
}
```

Arrow Functions

- Arrow functions easify the creation of functions:

```
numbers.sort(function(a, b){
  return b – a;
});
```

Becomes

```
numbers.sort((a, b) => b – a);
```

```
var fullnames =
   people.filter(function (person) {
     return person.age >= 18;
   }).map(function (person) {
     return person.fullname;
   });
```

Becomes

```
var fullnames2 =
  people.filter(p => p.age >= 18)
    .map(p => p.fullname);
```

Also called LAMBDA expressions

Arrow Functions – Example

```
let arr = [1, 2, 3];
let sum = arr
  .map(x => x * 2)
  .reduce((sum, x) => sum + x);

log(sum); // ==> 12
```

Object Literals

- ES2015 adds a new feature (rule) to the way of defining properties:
  - Instead of

```
let name = 'Samir Saghir',
    age = 25;
let person = {
  name: name,
  age: age
};
```

- We can do just:

```
let name = 'Samir Saghir';
age = 25;
let person = {
  name,
  age
};
```

Destructuring Array

- Destructuring assignments allow to set values to objects in an easier way:
  - Destructuring assignments with arrays:

```
var [a,b] = [1,2]; //a = 1, b = 2
var [x, , y] = [1, 2, 3] // x = 1, y = 3
var [x, y, ...rest] = = [1, 2, 3, 4]
             // x = 1, y = 2, rest = [3, 4]
```

- Swap values: `[x, y] = [y, x]`
- Result of method:

```
function get(){ return [1, 2]; }
var [x, y] = get();
```

Destructuring Object

- Destructuring assignments allow to set values to objects in an easier way:
  - Destructuring assignments with objects:

```
var person = {
  name: 'Samir Saghir',
  address: {
    city: 'Doha',
    street: 'University'
  }
};

var {name, address: {city}} = person;
console.log(name, city);
```

Classes and Inheritance

The way of OOP in ES2015

< Back

Classes and Inheritance in ES2015

- ES2015 introduces classes and a way to create classical OOP

```
class Person extends Mammal{
  constructor(fname, lname, age){
    super(age);
    this._fname = fname;
    this._lname = lname;
  }
  get fullname() {
    //getter property of fullname
  }
  set fullname(newfullname) {
    //setter property of fullname
  }
  // more class members…
}
```

Constructor of the class

Getters and setters

Maps and Sets

< Back

Maps and Sets

- ES2015 supports maps and sets natively

```
var names = new Set();
names.add('Samir');
names.add('Fatima');
names.add('Mariam');
names.add('Ahmed');
names.add('Samir'); // won't be added

for (let name of names) {
    console.log(name);
}
```

Using Iterators – Example

```
let map = new Map();
map.set(1, 'a');
map.set(2, 'b');
for(let pair of map) {
     log(pair)
}
for(let key of map.keys()) {
     log(key)
}
for(let value of map.values()) {
     log(value)
}
```

ES2015 Modules

Back

Modules

- ES2015 introduces modules that enables us to write modular code.
  - Each JS file has its own scope (not the global)
  - Each file decides what to export from its module
- Export the objects you want from a module:

```
// Car.js
export class Car { ... }
export class Convertible extends Car { ... }
```

- Use the module in another file:

```
// App.js
import {Car, Convertible} from 'Car';
let bmw = new Car();
let cabrio = new Convertible();
```

Asynchronous Patterns in JavaScript

Callbacks

Promises

ES 2015 Generators

See ‘14-promises-advanced’ example

< Back

Synchronous vs. Asynchronous

Buying A Book

- **Synchronous**: You go to your local book store, wait impatiently in a queue, then pay for the book and take it home.
- **Asynchronous**: You order the book on Amazon, and then get on with other things in your life. At some indeterminate point in the future, when the book is ready for you, the postman raises a knocking event on your door so that the book can be delivered to you.

Sync Programming is Easy

```
function getStockPrice(name) {
    var symbol = getStockSymbol(name);
    var price = getStockPrice(symbol);
    return price;
}
```

For async need to use either:

- Callbacks
- Promises
- ES 2015 Generators

Synchronous Programming Problems

- CPU demanding tasks delay execution of all other tasks => UI may become unresponsive
- Accessing resources blocks entire program
  - Especially problematic with web resources
    - Resource may be large
    - Slow connections mean slow loading
    - Server may hang
  - While loading, UI blocks

Why use Async Programming?

- **To prevent blocking** on long-running operations
  - Client-side UI remains responsive vs. locking up

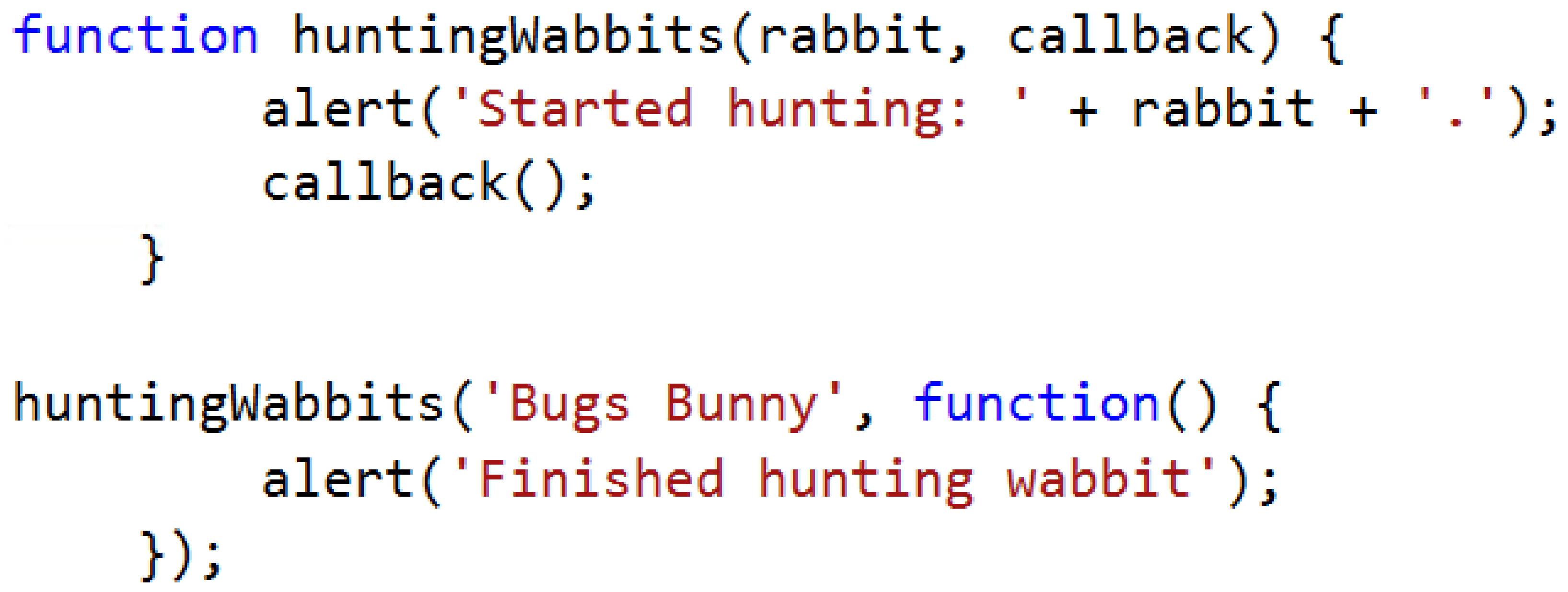
    => Doesn’t lock UI on long-running computations
  - Server-side scales vs. denying service

- **Use cases:**
  - ***Responsiveness**: prevent blocking of the UI*
  - ***Scalability**: prevent blocking of request-handling threads*

Callback-oriented Programming

- A callback is a function that is passed to another function as a parameter:
  - To continue the work
  - To process values
- Examples of callbacks:
  - Event handlers are sort-of callbacks
  - setTimeout and setInterval take a callback argument
- Problems:
  - Heavily nested functions are hard to understand => **Callback hell** i.e., Non-trivial to follow path of execution
  - Errors and exceptions are a nightmare to process

Example



```
function huntingWabbits(rabbit, callback) {
        alert('Started hunting: ' + rabbit + '.');
        callback();
    }

huntingWabbits('Bugs Bunny', function() {
        alert('Finished hunting wabbit');
    });
```

Callback Hell...

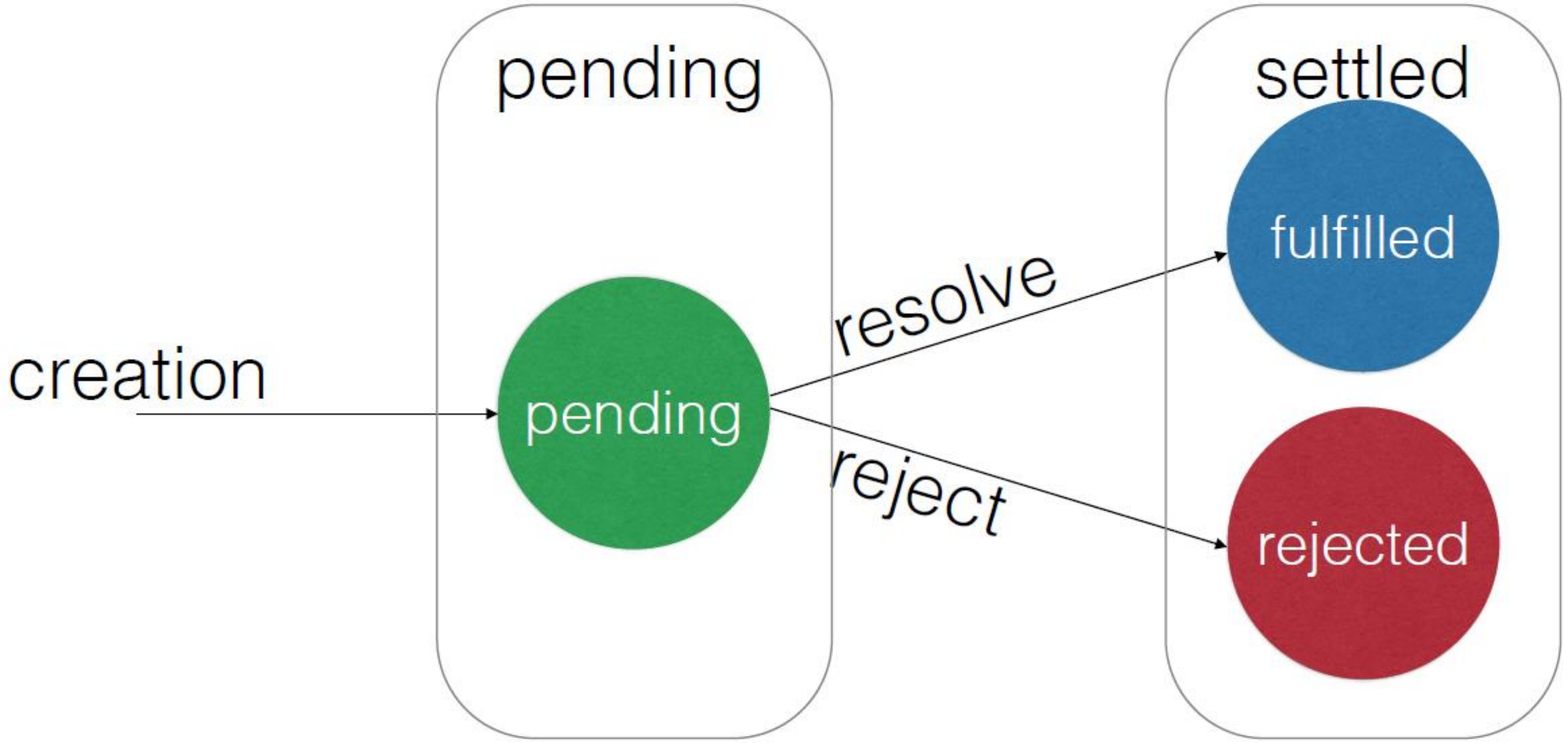
```
function getStockPrice(name, cb) {
    getStockSymbol(name, (error, symbol) => {
        if (error) {
            cb(error);
        }
        else {
            getStockPrice(symbol, (error, price) => {
                if (error) {
                    cb(error);
                }
                else {
                    cb(price);
                }
            })
        }
    })
}
```

Promises

- Promise = object that represents an eventual (future) value
- A producer returns a promise which it can later fulfill or reject
- Promise has one of three states: pending, fulfilled, or rejected
- Consumers listen for state changes with .then method:

```
promise.then(onSuccess, onError);
```

State of a Promise



sync vs. async

- **sync**

```
function getStockPrice(name) {
    var symbol = getStockSymbol(name);
    var price = getStockPrice(symbol);
    return price;
}
```

- **async**

```
function getStockPrice(name) {
    return getStockSymbol(name).
     then(symbol => getStockPrice(symbol));
}
```

Example

In jQuery the Promise pattern can be used with $.when(object) where object is a deferred object or a promise

```
$.when($.ajax("test.aspx")).then(function (data, textStatus, jqXHR) {

    alert(jqXHR.status); // Alerts 200

});
```

Promises – Example

```
// producer creates a promise, resolves when ready
function timeout(ms) {
  return new Promise(resolve => {
    setTimeout(resolve, ms);
  });
}

log("start");

// consumer gets a promise, is notified when resolved
let p = timeout(1000);
p.then(() => log("end"));
```

Promise

```
let getUser = function() {
  return new Promise(function(resolve, reject) {
 // async stuff, like fetching users from server,
returning a response
    if (response.status === 200) {
      resolve(response.data);
    } else {
      reject('No user');
    }
  });
};
```

Promise Usage

```
getUser()
  .then(function(user) {
    return getRights(user);
  })
  .then(function(rights) {
    updateMenu(rights);
  })
```

Better Syntax

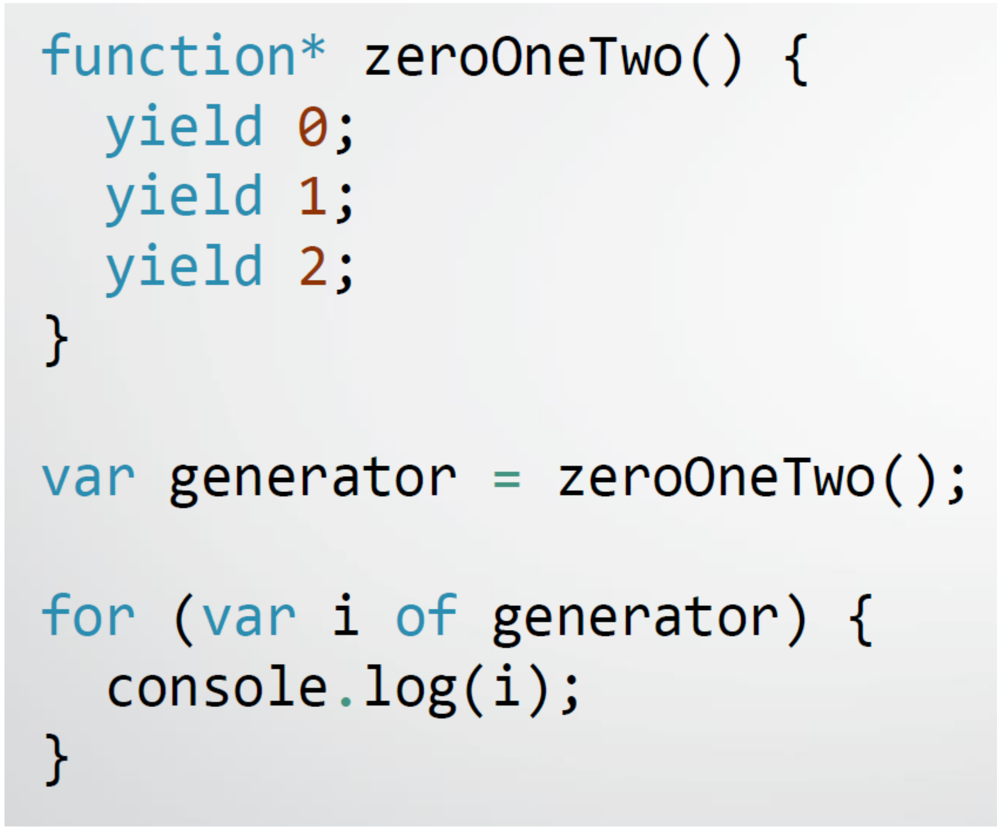
```
getUser()
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

Why Generators?

- Programming with asynchronous callbacks is hard
- ECMAScript 2015 provides generators, which allows us to alleviate some of this pain…
- Can use generators to move from **continuation-passing style** to direct style programming
- Generators = **Suspendable functions** with multiple entry points
- Benefits:
  - Easier to follow path of execution
  - Easier exception handling

Generators

- Suspending execution until someone calls next()
- **Run..Stop..Run**



```
function* zeroOneTwo() {
  yield 0;
  yield 1;
  yield 2;
}

var generator = zeroOneTwo();

for (var i of generator) {
  console.log(i);
}
```

Learn more @ https://www.youtube.com/watch?v=QO07THdLWQo
See '***14-promises-advanced***' example

sync vs. generator

- **sync**

```
function getStockPrice(name) {
    var symbol = getStockSymbol(name);
    var price = getStockPrice(symbol);
    return price;
}
```

- **async**

```
function* getStockPrice(name) {
    var symbol = yield getStockSymbol(name);
    var price = yield getStockPrice(symbol);
    return price;
}
```

sync vs. ES 2016 generator

- **sync**

```
function getStockPrice(name) {
    var symbol = getStockSymbol(name);
    var price = getStockPrice(symbol);
    return price;
}
```

- **async**

```
async function getStockPrice(name) {
    var symbol = await getStockSymbol(name);
    var price = await getStockPrice(symbol);
    return price;
}
```

Resources

- Best ES 2015 eBook

http://exploringjs.com/es6/

- Best ES 2015 Learning Resources

https://github.com/ericdouglas/ES2015-Learning

- More Examples

http://www.es6fiddle.net/

TypeScript

< Back

What is TypeScript?

Strongly Typed

Classes

Interfaces

Generics

Modules

Type Definitions

Compiles to JavaScript

EcmaScript 6 & 7 Features

Type Annotations

- Type annotations provide optional static typing. Applied using **: T** syntax

```
var height:number = 6;
var isDone:boolean = true;
var name:string = 'thoughtram';

var list:number[] = [1, 2, 3];
var list:Array<number> = [1, 2, 3];

function add(x: number, y: number): number {
  return x+y;
}
```

Decorators

- A decorator is an **expression** that is evaluated after a class has been defined, that can be used to **annotate or modify** the class in some fashion.

```
import {Component, View} from 'angular2/core';



@Component({
  selector: 'contacts-app'
})
@View({
  template: 'Hello World!'
})
class ContactsApp {


}
```

Will be discussed when learning AngularJS

Resources

- TypeScript Playground

http://www.typescriptlang.org/Playground

- TypeScript Handbook

http://www.typescriptlang.org/Handbook