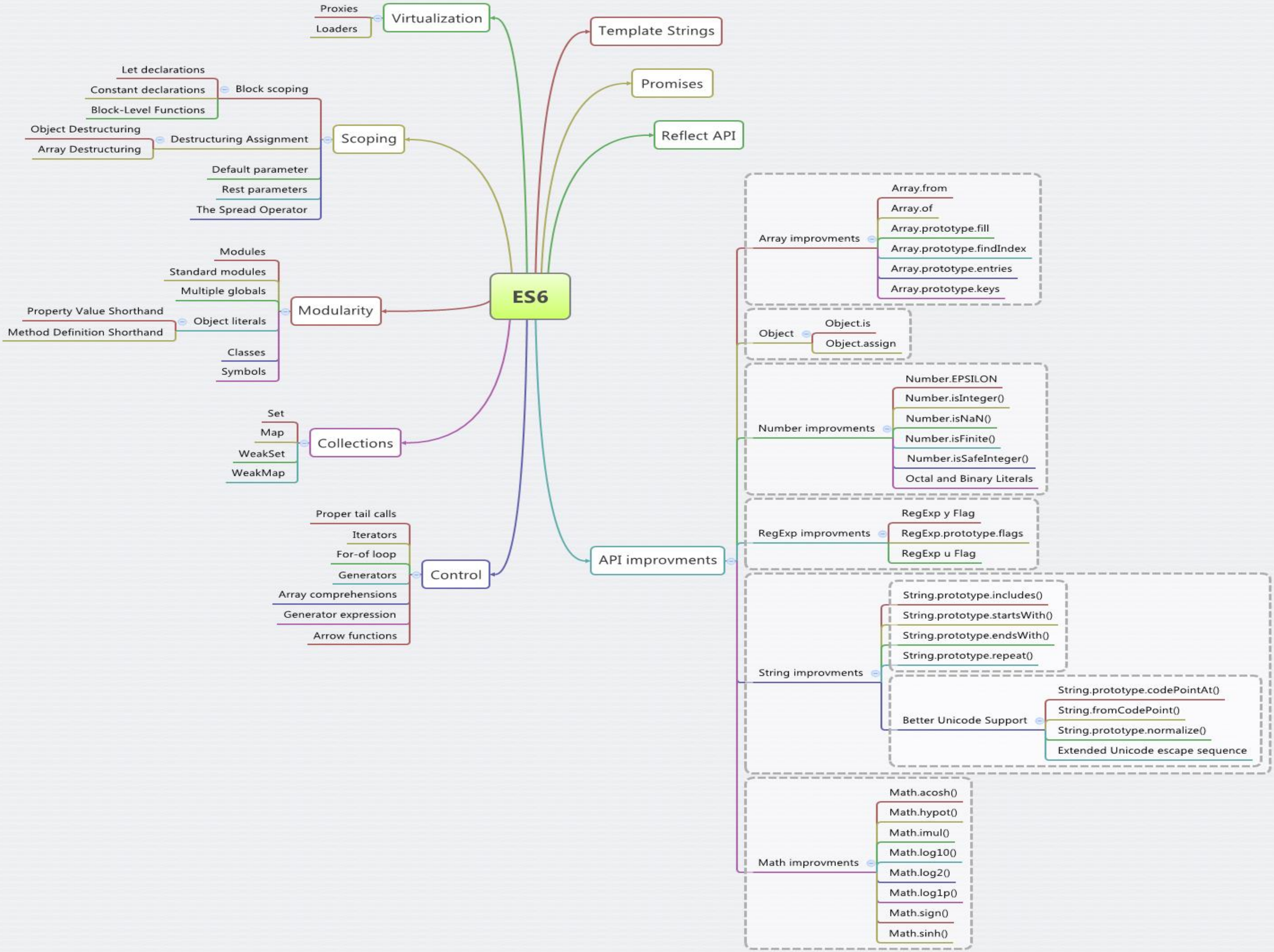
ES.next

Amazing New Features In JavaScript

Eng. Niveen Nasr El-Den
SD & Gaming CoE
iTi

Day 3

A Better JavaScript for the Ambient Computing Era



ES6

- Virtualization
  - Proxies
  - Loaders
- Template Strings
- Promises
- Reflect API
- Scoping
  - Block scoping
    - Let declarations
    - Constant declarations
    - Block-Level Functions
  - Destructuring Assignment
    - Object Destructuring
    - Array Destructuring
  - Default parameter
  - Rest parameters
  - The Spread Operator
- Modularity
  - Modules
  - Standard modules
  - Multiple globals
  - Object literals
    - Property Value Shorthand
    - Method Definition Shorthand
  - Classes
  - Symbols
- Collections
  - Set
  - Map
  - WeakSet
  - WeakMap
- Control
  - Proper tail calls
  - Iterators
  - For-of loop
  - Generators
  - Array comprehensions
  - Generator expression
  - Arrow functions
- API improvments
  - Array improvments
    - Array.from
    - Array.of
    - Array.prototype.fill
    - Array.prototype.findIndex
    - Array.prototype.entries
    - Array.prototype.keys
  - Object
    - Object.is
    - Object.assign
  - Number improvments
    - Number.EPSILON
    - Number.isInteger()
    - Number.isNaN()
    - Number.isFinite()
    - Number.isSafeInteger()
    - Octal and Binary Literals
  - RegExp improvments
    - RegExp y Flag
    - RegExp.prototype.flags
    - RegExp u Flag
  - String improvments
    - String.prototype.includes()
    - String.prototype.startsWith()
    - String.prototype.endsWith()
    - String.prototype.repeat()
    - Better Unicode Support
      - String.prototype.codePointAt()
      - String.fromCodePoint()
      - String.prototype.normalize()
      - Extended Unicode escape sequence
  - Math improvments
    - Math.acosh()
    - Math.hypot()
    - Math.imul()
    - Math.log10()
    - Math.log2()
    - Math.log1p()
    - Math.sign()
    - Math.sinh()

ES6 Features

- let + const
- default Parameters
- rest parameters
- spread operator
- Destructuring (array/object)
- Arrow Functions
- Enhanced object literals
- Template strings
- for..of
- Data Structure/Collection
  - map
  - set
  - weakmap
  - weakset

- Binary and Octal literals
- Classes
- iterators
- Generators
- Symbols
- Proxies
- Modules
- Module loaders
- Promises
- math API
- number API
- string API
- array API
- object API
- etc…

http://es6-features.org/#Constants

https://github.com/lukehoban/es6features

https://kangax.github.io/compat-table/es6/

Reminder

- Property descriptors hold descriptive information about object properties. It allows developer to control some of the internal attributes of the object properties.

- It is defined via
  - Object.defineProperty(obj,”prop”,{})
  - Object.defineProperties(obj,{})

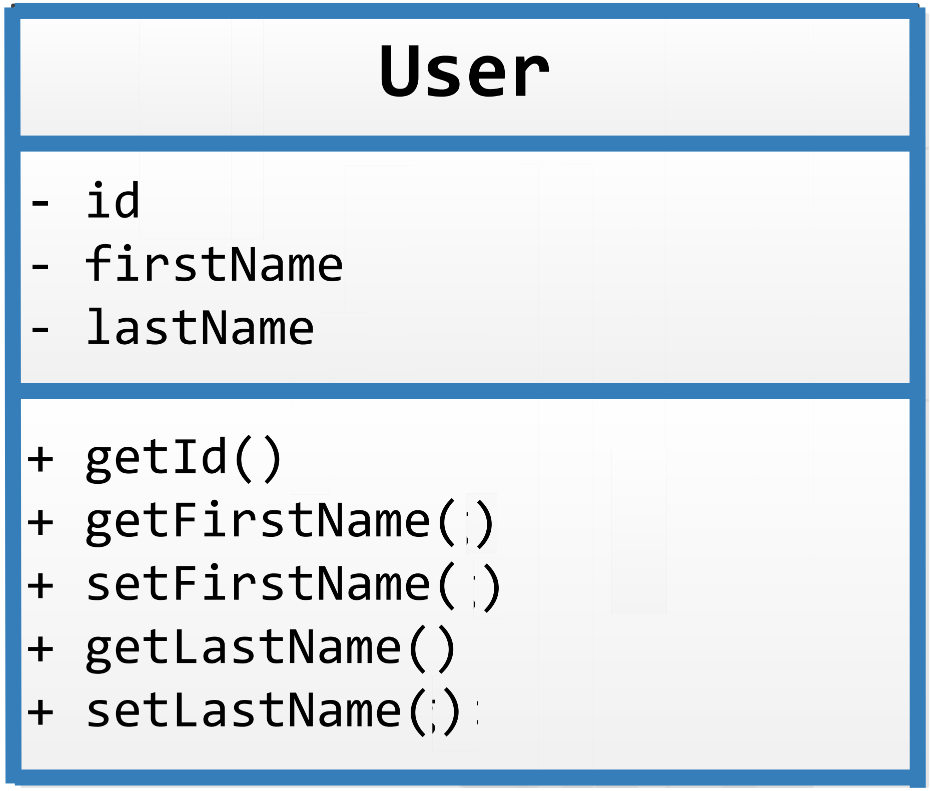
- It can be either
  - Data Descriptor or,
  - Accessor Descriptor

Reminder Example

```
var Employee = function(name, age){
    var person = {};

    Object.defineProperty (person, “name”, {
        value : name,
        writable : true,
        configurable: true,
        enumerable: true
    } );

    Object.defineProperty (person, “age”, {
        get : function() { return age; },
        set : function(val) { age = val; }
    } );

    return person;
    }
```

```
var Employee = function(name, age){
    var person = {};
    Object.defineProperties
      (person,{
        name:{
            value : name,
            writable : false},
            age:{…..},
            show:{…..}
      ……..

      } );
    return person;
    }
```

User Class Diagram

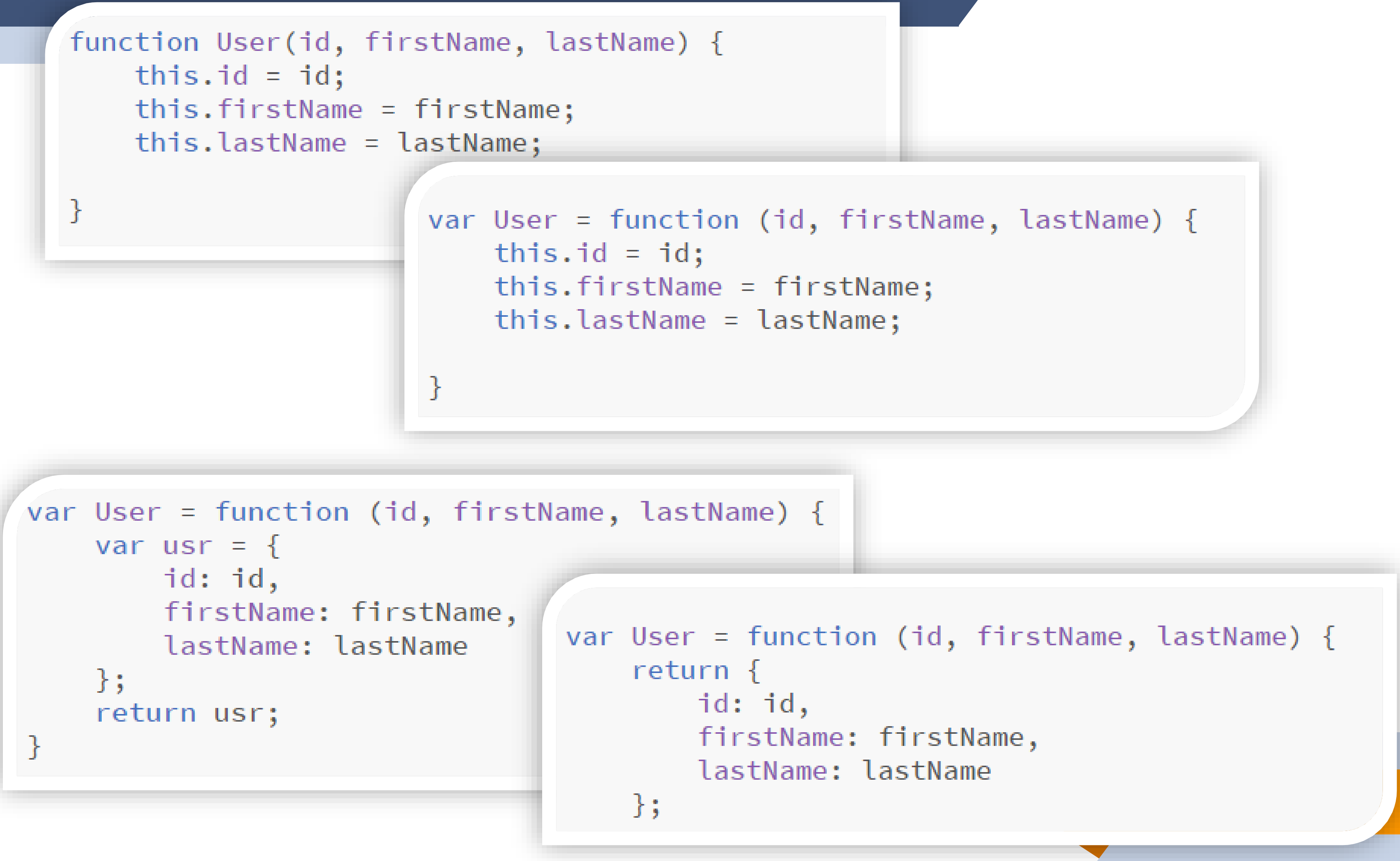


User

- id
- firstName
- lastName

```
+ getId()
+ getFirstName()
+ setFirstName()
+ getLastName()
+ setLastName()
```

Reminder: User Creation

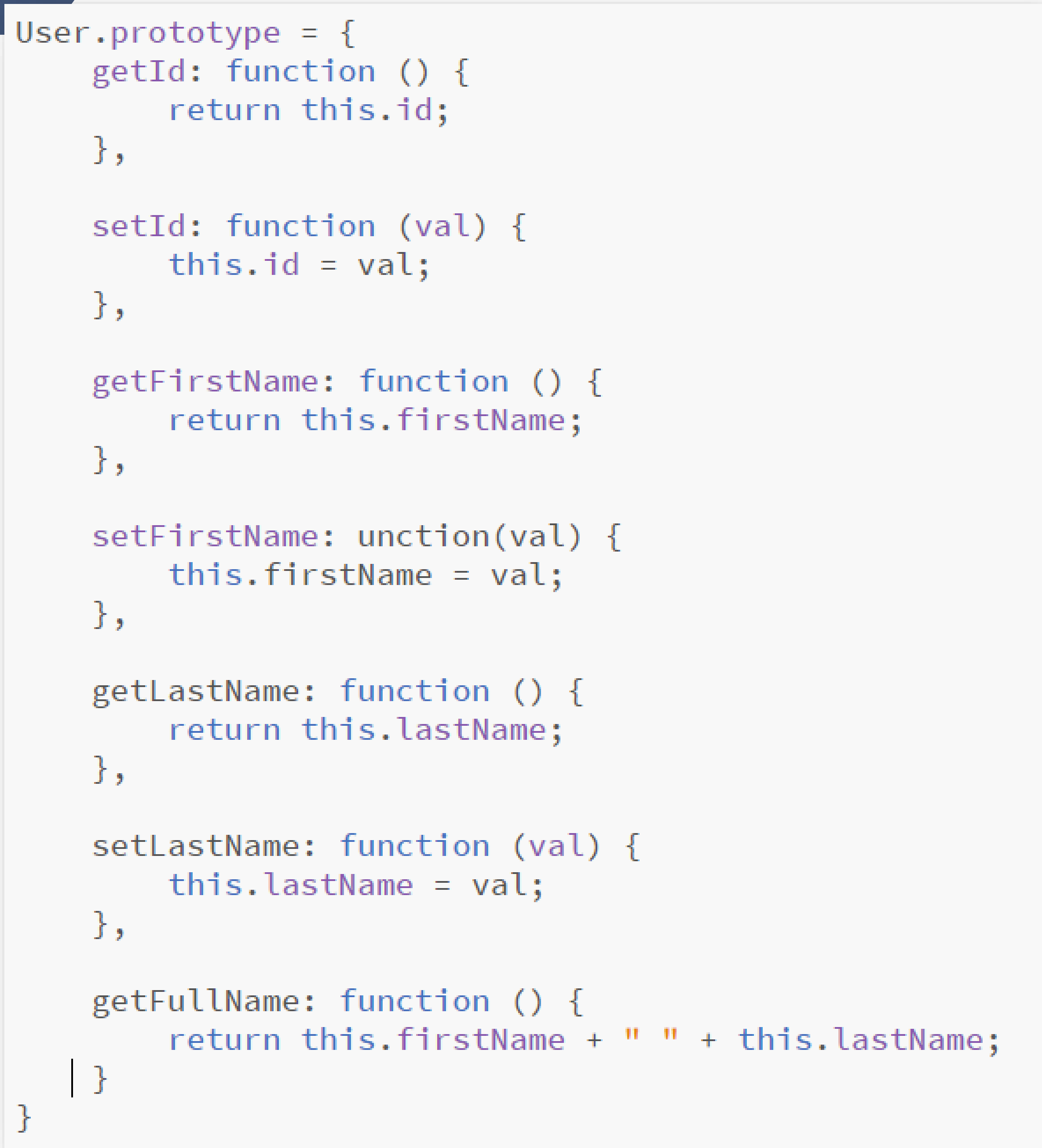


```
function User(id, firstName, lastName) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;

}
```

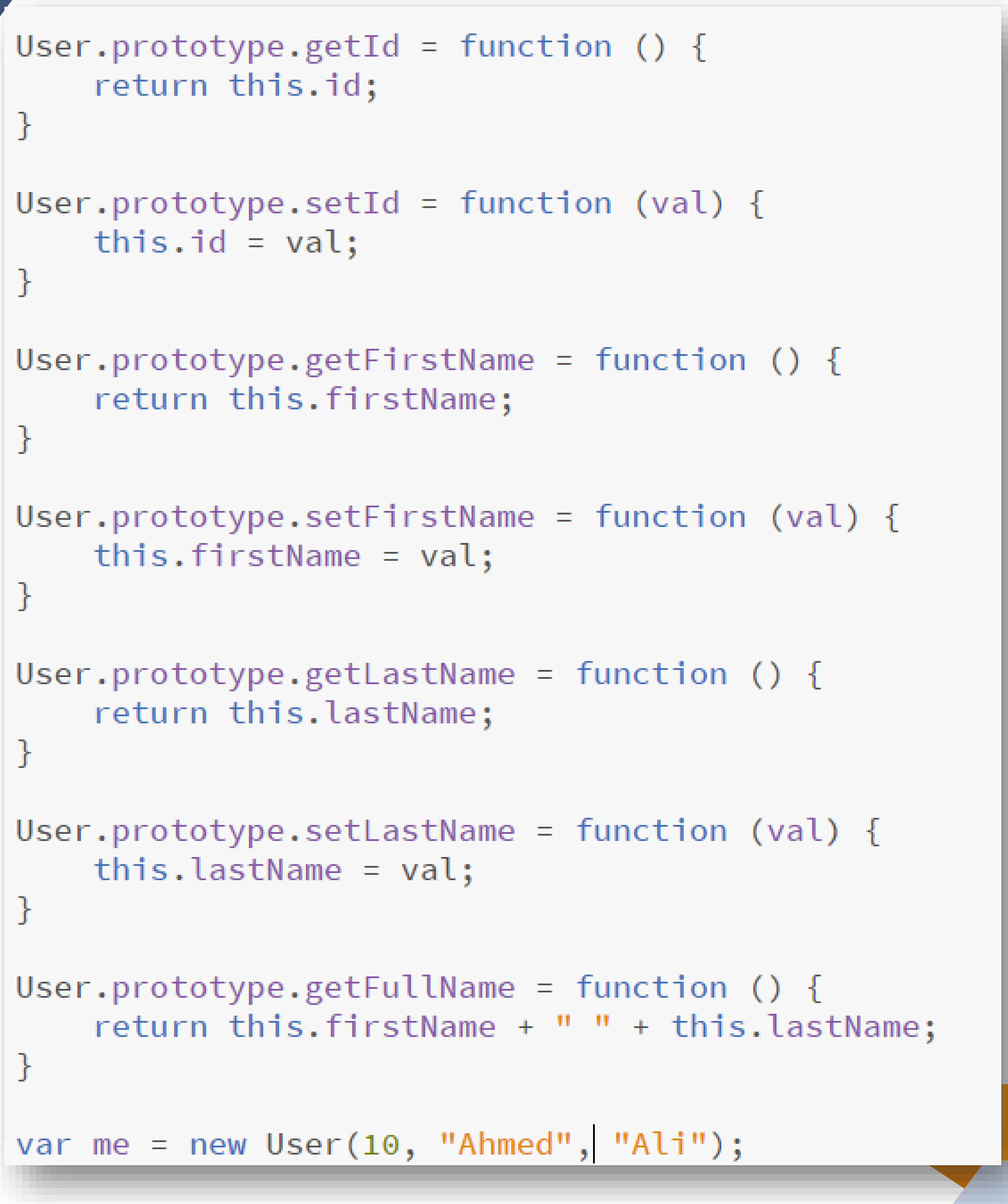
```
var User = function (id, firstName, lastName) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;

}
```

```
var User = function (id, firstName, lastName) {
    var usr = {
        id: id,
        firstName: firstName,
        lastName: lastName
    };
    return usr;
}
```

```
var User = function (id, firstName, lastName) {
    return {
        id: id,
        firstName: firstName,
        lastName: lastName
    };
```



```
User.prototype = {
    getId: function () {
        return this.id;
    },

    setId: function (val) {
        this.id = val;
    },

    getFirstName: function () {
        return this.firstName;
    },

    setFirstName: unction(val) {
        this.firstName = val;
    },

    getLastName: function () {
        return this.lastName;
    },

    setLastName: function (val) {
        this.lastName = val;
    },

    getFullName: function () {
        return this.firstName + " " + this.lastName;
    }
}
```



```
User.prototype.getId = function () {
    return this.id;
}

User.prototype.setId = function (val) {
    this.id = val;
}

User.prototype.getFirstName = function () {
    return this.firstName;
}

User.prototype.setFirstName = function (val) {
    this.firstName = val;
}

User.prototype.getLastName = function () {
    return this.lastName;
}

User.prototype.setLastName = function (val) {
    this.lastName = val;
}

User.prototype.getFullName = function () {
    return this.firstName + " " + this.lastName;
}

var me = new User(10, "Ahmed", "Ali");
```

ES6 Class Implementation

http://es6console.com/

```
class classnm{
_p3=10;
~~#p3=20;~~ //es.next for private prop
  constructor (p1, p2) {
    ~~this.p1=p1;~~
    this._p1=p1;
    this._p2 = p2;
  }
//properties getters & setters
  get p1(){return this.p1;}
  set p1(val){this.p1=val; ;}

//Function Declaration
   prototypeFn(){return  ;}
   static staticFn(){return  ;}
   static get staticprop(){return  ;}
}
```

```
var classnm = class {
  constructor (p1, p2) {
    this.p1 = p1;
    this.p2 = p2;
  }
```

```
var classnm = class [classnm]{
  constructor (p1=1, p2=2) {
    this.p1 = p1;
    this.p2 = p2;
  }
classnm.prototype.fun=function(){}
```

JavaScript remains prototype-based
This is **syntactic sugar**
private members are prefixed by _
the name of the class is local to the class body only

ES6 Class User Implementation

```
class User {
 constructor (id, firstName, lastName) {
   this.id = id
   this.firstName = firstName
   this.lastName = lastName
 }

 getId() { return this.id   }

 getFirstName() { return this.firstName  }
 setFirstName(firstName) { this.firstName = firstName  }

 getLastName() { return this.lastName  }
 setLastName(lastName) { this.lastName = lastName  }

 getLastName() { return this.firstName +””+ this.lastName }
}
```

```
let firstNameSymbol = Symbol();
let lastNameSymbol = Symbol();

class User {
  constructor (id, firstName, lastName) {
    this.id = id
    this[firstNameSymbol] = firstName
    this[lastNameSymbol] = lastName
  }

  getId() { return this.id   }

  get firstName() { return this[firstNameSymbol]  }
  set firstName(firstName) {this[firstNameSymbol] = firstName  }

  get lastName() { return this[lastNameSymbol]; }
  get fullName() { return this.firstName + " " + this.lastName;  }
}
```

ES6 Inheritance Implementation

```
class Square extends Shape {
  constructor (sideLength) {
    super (sideLength, sideLength);
  }
  get area() {//property
    return this.calcArea() ;
  }
  set sideLength(newLength) {//property
    this.height = newLength;
    this.width = newLength;
  }
  calcArea(){ //method
     return this.height * this.width;
  }
  toString(){
      return `${super.hToString()} w:${this.width}`;}
}
```

```
class Shape {
  constructor (height, width) {
    this.height = height;
    this.width = width;
  }
  hToString(){return `h:${this.height}`;}
}
```

```
var square = new Square(2);
console.log(square.area);//4
console.log(square.calcArea());//4
```

ES9 new Features

http://exploringjs.com/es2018-es2019/ch_overview.html#ecmascript-2018

- Array.prototype.flat()
- Array.prototype.flatMap()
- Symbol.prototype.description
- try{} catch{}
- Promise.then().catch().finally()

Module

http://www.2ality.com/2014/09/es6-modules-final.html

- Modular programming is the process of subdividing a computer program into separate sub-programs.
- Modularity is needed to package and encapsulate the code
- Module is a JavaScript file that exports object that can be used in a page
  - One module is only one JavaScript file
- We can only access those that were exported
- Modules allow loading code on demand

Module

http://exploringjs.com/es6/ch_modules.html

- export entities in the module where declared
- import entities from a module in a module
- A JavaScript file is a module
- There are two kinds of exports:
  - named exports (several per module) and
  - default exports (one per module).
    - This could be anonymous class or constructor function.
- Imports are hoisted
- <script> must have type=“module”

Module

```
//------index.html ------
<script src="main.js" type="module">
</script>
```

```
//------ lib.js ------
export function square (x) {
   return x * x;
}
export function diag (x, y) {
   return sqrt(square(x) +
    square(y));
}
export const sqrt = Math.sqrt;
```

```
//------ main.js ------
import {square,diag} from ‘./lib.js';
import {square as s,diag} from ‘./lib.js';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

Must be either relative or absolute path

```
//------ main.js ------
import * as mod from ‘./lib.js';
console.log(mod.square(11)); // 121
console.log(mod.diag(4, 3)); // 5
```

Module

```
//------ lib.js ------
export class c {
    constructor(){}
    square(x) {
        return x * x;
    }
    diag(x, y) {
        return sqrt(this.square(x) + this.square(y));
    }
}
```

Must be either **relative** or absolute path

```
//------ main.js ------
import * as mod from './lib.js';
var obj= new mod.c();
console.log(obj.square(11)); // 121
console.log(obj.diag(4, 3)); // 5
```

```
//------index.html ------
<script src="main.js" type="module">
</script>
```

Module

```
//------ lib.js ------
export class c {
    constructor(){}
    square(x) {
        return x * x;
    }
    diag(x, y) {
        return sqrt(this.square(x) + this.square(y));
    }
}
```

```
//------index.html ------
<script type="module">
  import * as mod from './lib.js';
  var obj= new mod.c();
  console.log(obj.square(11)); // 121
  console.log(obj.diag(4, 3)); // 5
</script>
```

Must be either relative or absolute path

Module

```
//------ lib.js ------
export default class c {
    constructor(){}
    square(x) {
        return x * x;
    }
    diag(x, y) {
        return sqrt(this.square(x) + this.square(y));
    }
}
```

```
//------ main.js ------
import c from './lib.js';
var obj= new c();
console.log(obj.square(11)); // 121
console.log(obj.diag(4, 3)); // 5
```

Must be either **relative** or absolute path

Proxies

- Proxy is dynamic/virtual object that doesn't have properties
- Proxy is used to define custom behavior for fundamental operations
- A proxy object sits between a real object and the calling code. The calling code interacts with the proxy instead of the real object
- To create a **proxy** object we need to pass target object and a handler object that act as its placeholder and has some traps
  - Traps are the same as the methods used in the **Reflect API**.

Proxies

- It provides custom implementations for properties
- Proxy returns a new object which wraps the passed in object, but anything you do with either effects the other

```
var handler={
    get:function(proxy,prop){return proxy[prop]},
    set:function(proxy,prop,val){proxy[prop]=val},
    has:function(prop){}

}

var p= new Proxy({},handler);

p.x=10
console.log(p.x)
```

Proxies

- There are many real-world applications for **Proxies**
  - validation
  - value correction
  - property lookup extensions
  - tracing property accesses
  - revocable references
  - etc..

Proxies

```
var target = { name: "targetName"}

var handler = {
        //prop lookup behavior
        get: function (obj, prop) {
        return prop in obj ? obj[prop] : "new value";
        }

}
//enforce value type validation
handler.set = function (obj, prop, val) {
        if (prop === "name") {
        if (typeof val !== "string")
                throw new TypeError("not req type")
         }
         obj[prop] = val;
}
```

```
var p = new Proxy(target, handler)

console.log(p.name) // targetName
console.log(p.newProp) // new value

p.prop=10
p.name=100 //

console.log(p.prop)
console.log(p.name)
```

Proxies

```
var handler={
    get:function(proxy,prop){return proxy[prop]},
    set:function(proxy,prop,val){proxy[prop]=val},
    has:function(prop){}

}

var p= new Proxy({},handler);

p.x=10
console.log(p.x)
```

Promise

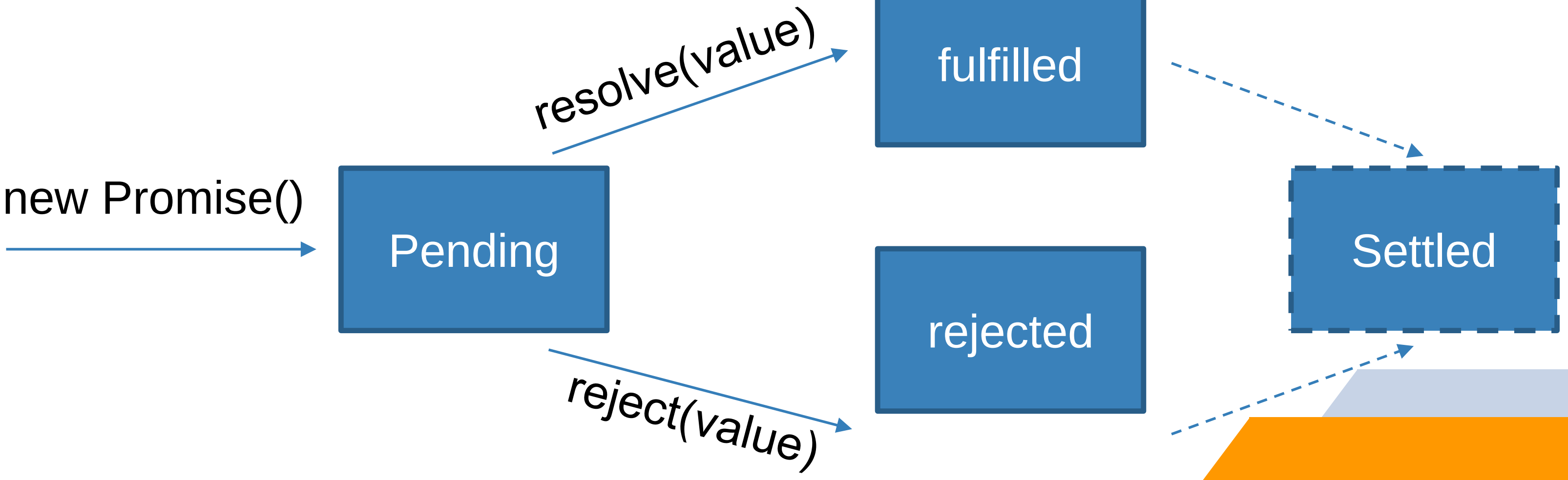
- The word 'asynchronous', aka 'async' just means 'takes some time' or 'happens in the future, not right now.

- Usually done by callbacks that are only used when doing I/O, e.g. downloading things, reading files, talking to databases

http://sporto.github.io/blog/2012/12/09/callbacks-listeners-promises/

Promise

- Creating a new Promise automatically sets it to the pending state. Then, it can do 1 of 2 things: become fulfilled or rejected. After which it is said to be in settled state.
- It starts as pending then it can either be resolved or rejected



https://developers.google.com/web/fundamentals/primers/promises

Promise

```
var myPromise = new Promise((resolve, reject) => {
    if (any_condition)
        resolve("fine"); // fire then
    else reject("error"); //fire catch

});

myPromise.then((data) => console.log(data))
myPromise.catch((err) => console.log(err))
```

```
new Promise((resolve, reject) => {
    if (any_condition)
        resolve("fine"); // fire then
    else reject("error"); //fire catch

}).then((data) => console.log(data))
   .catch((err) => console.log(err))
```

Promise

- To avoid duplicating code in both the promise's .then() & .catch() handlers use .finally() that will execute when promise is settled

- .finally() is new in ES9

Promise Static Properties & Methods

- Promise.length
  - number of constructor arguments which is always 1
- Promise.all([])
  - Returns either resolved promise if all passed promises are resolved or rejected promise if as soon as one of these promises is rejeced
- Promise.reject(reason)
  - Returns rejected promise object with the given reason
- Promise.resolve(reason)
  - Returns resolved promise object with the given reason
- Promise.race([])
  - Returns rejected or resolved promise as soon as one of the passed promises is settled

async & await

- async function writes promise-based code as behaves if it were synchronous code, but without blocking the main thread.
- When using await, the function is paused in a non-blocking way until the promise settles.
- await may only be used in functions marked with the async keyword

async & await

https://javascript.info/async-await

- async function writes promise-based code as behaves if it were synchronous code, but without blocking the main thread.
- If no promise return statement it automatically wraps it into a resolved promise with that value.
  - It always returns a promise
- When using await, the function is paused in a non-blocking way until the promise settles.
- await may only be used in functions marked with the async keyword

async & await

```
async function f() {
       let promise = new Promise((resolve, reject) => {
              setTimeout(() => resolve("done!"), 1000)
             });

       let promise2 = new Promise((resolve, reject) => {
              setTimeout(() => resolve("done2!"), 1000)
             });

      let result = await promise; // wait until promise resolves
      //let result = await Promise.all([promise,promise2]);//.then(alert);

      promise.then(console.log)//done
      //alert(result); // "done!"
}

f();
```

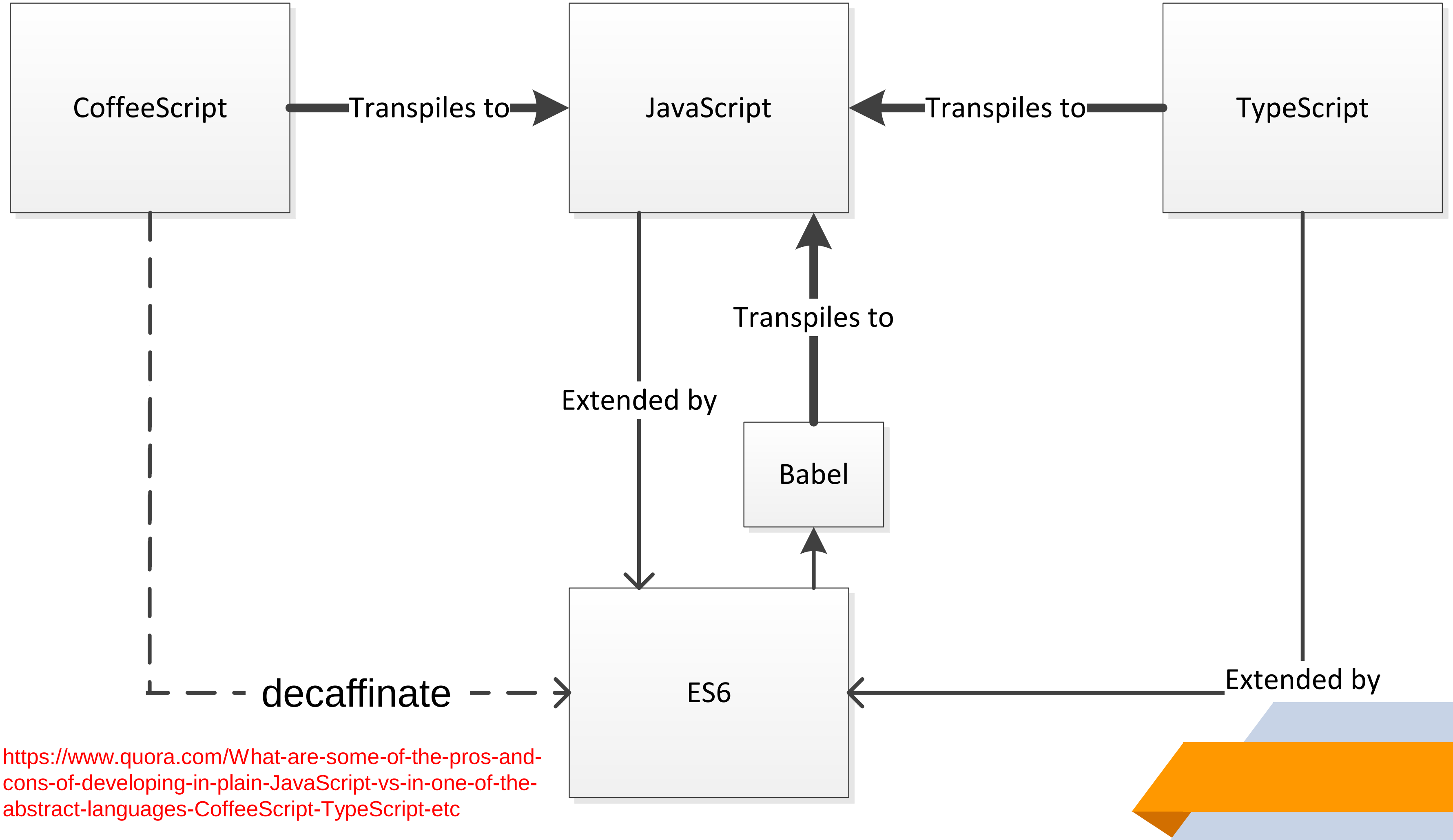
async & await

- await may only be used in functions marked with the async keyword

- await: suspends execution until the promise settles.
  - Note: If the awaited expression isn’t a promise, its casted into a promise.

- async function can be
  - Assigned to variable
  - Written as IIFE

Languages & Transpilers



https://www.quora.com/What-are-some-of-the-pros-and-cons-of-developing-in-plain-JavaScript-vs-in-one-of-the-abstract-languages-CoffeeScript-TypeScript-etc

Transpile

- A transpile is a type of compilers that converts the syntax from language to another.

- Previously no engine runs es6 so we needed a transpiler that generates vanilla JavaScript

https://kangax.github.io/compat-table/es6/

Transpilers

- Traceur
  - http://github.com/google/traceur-compiler
- Babeljs
  - http://babeljs.io
- Etc.

- Online transpiler
  - google.github.io/traceur-compiler/demo/repl.html
  - http://babeljs.io/repl/
  - http://es6console.com

Assignments