ES6 - Session

Agenda

- What is ES6?
- New features
- Browser compatibility
- Block Scope
- Variable scoping with let
- Running the ES6 applications in ES5 browsers
- Arrow functions
- Template Strings and Literals
- Working with arrays
- Define classes and inheritance

Agenda

- Class properties
- Computed property names
- Constants
- Decorators
- Default parameters
- De-structuring
- Generators and Iterators
- Symbol Types
- The Object Properties and state

What is ES6?

- **ECMAScript = ES:**
  - ECMAScript is a Standard for a scripting languages.
  - Languages like Javascript are based on the ECMAScript standard.
  - ECMA Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft).
  - ECMA means European Computer Manufacturer's Association

- **JavaScript = JS:**
  - JavaScript is the most popular implementations of the ECMAScript Standard.
  - The core features of Javascript are based on the ECMAScript standard, but Javascript also has other additional features that are not in the ECMA specifications/standard.
  - ActionScript and JScript are another languages that implements the ECMAScript.
  - JavaScript was submitted to ECMA for standardization but due to trademark issues with the name Javascript the standard became called ECMAScript.
  - Every browser has a JavaScript interpreter.

ES6 - New Features

ES6 - New features

- All the new features of ES6 can be categorized in 7 categories. Here is the list of categories:
- Variables and Parameters
- Classes
- Proxies
- Build-in Objects and Prototype based features
- Promises
- Reflect API
- Modules
- Iterators and Generators

Browser Compatibility

- Kangax has created a ES6 compatibility table which lists the ES6 features and browsers which support or don't support them.
- You can also use caniuse.com to find the support of various ES6 features on various different browsers.

- **Running ECMAScript 6 in an Incompatible Browser**

- If you are writing ES6 for your website on development phase then you can embed the Traceur compiler in your webpages which will compile the ES6 to simple browser supportable JavaScript code on the fly in the browser.

- Here is how to embed Traceur in your website

Browser Compatibility

```
<!doctype html>
<html>
    <head>...</head>
    <body>
        ...

        <!-- Load Traceur Compiler -->
        <script src="https://google.github.io/traceur-compiler
/bin/traceur.js"></script>
        <!-- Bootstrap.js finds script tags with type module and com-
piler them using the interfaces provided by traceur.js -->
        <script src="https://google.github.io/traceur-compiler
/src/bootstrap.js"></script>
        <script type="module">
            //Place ES6 code here
        </script>
    </body>
</html>
```

- On production site compiling ES6 to browser supportable JS on every page load can be a resource & time consuming task and can effect the site performance. Its recommend that you use Traceur's node compiler to compile once for all time and embed the compiled JS in your pages.

- If you don't like this compile idea then you can use ES6 polyfills. Polyfill is not available for every ES6 feature.

Terminology

- JavaScript: the language name
- ECMAScript: the language standard
- TC 39: the technical committee
- ECMAScript Harmony: post ES 5 improvements
- ECMAScript.Next: the code name for next version
- ECMAScript 6: the final name for next version

History

- ECMAScript 1: 1997
- ECMAScript 2: 1998
- ECMAScript 3: 1999
- ECMAScript 4: Abandoned
- ECMAScript 5: 2009
- ECMAScript 6: ?

Transpilers

- Babel
  http://babeljs.io
- Traceur
  https://github.com/google/traceur-compiler
- TypeScript
  (Superset of JavaScript that aims to align with ECMAScript 6)
  http://www.typescriptlang.org

var

Function-Scoped

```
function divide(x, y) {
    if (y !== 0) {
        var result;
        result = x / y;
    }
    return result;
}
console.log(divide(10, 2)); // 5
```

let

Block-Scoped

```
function divide(x, y) {
    if (y !== 0) {
        let result;
        result = x / y;
    }
    return result;  // throws Error
}
console.log(divide(10, 2));
```

const

Block-Scoped

```
const PI = 3.14159265359;
const COLOR = {
    name: "Red",
    hexValue: "#FF0000"
};
```

Cannot be reassigned or redeclared

Creating Objects from Variables

ECMAScript 5

```
var firstName = "Christophe";
var lastName = "Coenraets";
var twitterId = "@ccoenraets";

var speaker = {
    firstName: firstName,
    lastName: lastName,
    twitterId: twitterId
};
```

Creating Objects from Variables

ES6 Object Literals

```
let firstName = "Christophe";
let lastName = "Coenraets";
let twitterId = "@ccoenraets";

let speaker = {firstName, lastName, twitterId};
```

Works with var too

Creating Variables from Array Elements

ECMAScript 5

```
var colors = ["red", "green", "blue"];

var primary   = colors[0];
var secondary = colors[1];
var tertiary  = colors[2];

console.log(primary);   // red
console.log(secondary); // green
console.log(tertiary);  // blue
```

Creating Variables from Array Elements

ES 6 Array Destructuring

```
var colors = ["red", "green", "blue"];

var [primary, secondary, tertiary] = colors;

console.log(primary);   // red
console.log(secondary); // green
console.log(tertiary);  // blue
```

Spread Operator

```
var colors = ["red", "green", "blue", "yellow", "orange"];

var [primary, secondary, ...otherColors] = colors;

console.log(primary);      // red
console.log(secondary);    // green
console.log(otherColors); // [ 'blue', 'yellow', 'orange']
```

Functions with Multiple Return Values

```
function getDate() {
    var d = new Date();
    return [d.getDate(), d.getMonth() + 1, d.getFullYear()];
}

var [day, month, year] = getDate();
console.log(day);   // 4
console.log(month); // 5
console.log(year);  // 2015
```

Creating Variables from Object Properties

ECMAScript 5

```
var speaker = { firstName: "Christophe",
                lastName: "Coenraets",
                twitterId: "@ccoenraets" };

var firstName = speaker.firstName;
var lastName = speaker.lastName;
var twitterId = speaker.twitterId;

console.log(firstName); // Christophe
console.log(lastName);  // Coenraets
console.log(twitterId); // @ccoenraets
```

Creating Variables from Object Properties

ES6 Object Destructuring

```
var speaker = { firstName: "Christophe",
                lastName: "Coenraets",
                twitterId: "@ccoenraets" };

var {firstName, lastName, twitterId} = speaker;

console.log(firstName); // Christophe
console.log(lastName);  // Coenraets
console.log(twitterId); // @ccoenraets
```

```
var {fName: firstName, lName: lastName, twId:twitterId} = speaker;
console.log(fName); // Christophe
```

ES5 Function

```
var greeting = function(message, name) {
    return message + ' ' + name;
}

console.log(greeting('Hello', 'Christophe'));
```

ES6 Arrow Function

```
var greeting = (message, name) => {
    return message + ' ' + name;
}
```

```
var greeting = (message, name) => message + ' ' + name;
```

```
var greeting = name  => 'Hello ' + name;
```

Another Example

```
var array = [1, 2, 3];
var total = 0;
array.forEach(function(item) {
    total = total + item;
});
console.log(total);
```

```
var array = [1, 2, 3];
var total = 0;
array.forEach(item => total = total + item);
console.log(total);
```

ECMAScript 5

"var self = this" *aka* "var that = this"

```
var obj = {
    init: function () {
        var self = this;
        setTimeout(function() {
            self.doSomething();
        }, 1000);
    },
    doSomething: function() {
        console.log("doing something in ES5");
    }
};
obj.init();
```

ES 6 Arrow Functions

Lexical “this”

```
var obj = {
    init: function() {
        setTimeout(() => this.doSomething(), 1000);
    },
    doSomething: function() {
        console.log("doing something in ES6");
    }
};
obj.init();
```

Default Params

```
var greeting = (name, msg="Hello") => msg + ' ' + name;

console.log(greeting('Christophe', 'Hi')); // Hi Christophe
console.log(greeting('Christophe'));       // Hello Christophe
```

Works with "function" too:

```
function greeting(name, message="Hello") {
    return message + ' ' + name;
}
```

Callback Pyramid of Doom

```
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Do something with value4
            });
        });
    });
});
```

Callback Issues

1. Hard to compose (sequence and parallel)
2. Hard to handle error
3. Violates input params/return value semantic

Promises

Promise

Proxy for value that's not yet available

1. Async function returns promise
2. Caller attaches event handler to returned promise
3. Event handler handles result when it becomes available

→ Clean input params/return value function semantic

q Promises
Using

```
Q.fcall(step1)
    .then(step2)
    .then(step3)
    .then(step4)
    .then(function(value4) {
        // Do something with value4
    })
    .catch(function(error) {
        // Handle any error from all above steps
    })
    .done();
```

think of then(handler) as addEventListener("done", handler)

q Promises
Defining

```
function step1() {
    var deferred = Q.defer();
    FS.readFile("foo.txt", "utf-8", function (error, text) {
        if (error) {
            deferred.reject(new Error(error));
        } else {
            deferred.resolve(text);
        }
    });
    return deferred.promise;
}
```

jQuery Promises

Using

```
service.getList()
    .done(function(list) {
        console.log(list);
    })
    .fail(function(error) {
        console.log(error);
    });
```

jQuery Promises

Defining

```
function getList() {
    var deferred = $.Deferred();
    if (list) {
        deferred.resolve(list);
    } else {
        deferred.reject("no list");
    }
    return deferred.promise();
}
```

ECMAScript 6 Promises

Defining

```
function timeout(millis) {
    var promise = new Promise(function (resolve, reject) {
        setTimeout(function() {
            resolve();
        }, millis);
    });
    return promise;
}
```

Promisified setTimeout()

ECMAScript 6 Promises

Using

```
timeout(1000).then(function() {
    console.log('done waiting');
});
```

Promisified setTimeout()

ECMAScript 6 Promises
Defining Mock Data Service

```
var employees;

function findAll() {
    return new Promise(function (resolve, reject) {
        if (employees) {
            resolve(employees);
        } else {
            reject("employees is not defined");
        }
    });
}
```

ECMAScript 6 Promises

Using Mock Data Service

```
findAll()
    .then(function(employees) {
        console.log(employees);
    })
    .catch(function(error) {
        console.log(error);
    });
```

AMD Module

```
define(function (require) {

    var $ = require('jquery');

    var findAll = function() {
        // implementation
    };

    var findById = function(id) {
        // implementation
    };

    return {
        findAll: findAll,
        findById: findById
    };

});
```

Common.js Module

```
var findAll = function () {
    // implementation
};

var findById = function(id) {
    // implementation
};

module.exports = {
    findAll: findAll,
    findById: findById
};
```

ECMAScript 6 Modules

Defining

```
// datelib.js
export const today = new Date();

export var shortDate = function() {
    return today.getMonth() + 1 + '/' +
           today.getDate() + '/' +
           today.getFullYear();
}
```

ECMAScript 6 Modules

Using

```
import {today, shortDate} from './datelib';
console.log(today);         // Mon May 4 2015 11:04:06...
console.log(shortDate()); // 5/4/2015
```

```
import * as date from "./datelib";
console.log(date.today);
console.log(date.shortDate());
```

ECMAScript 6 Modules

Defining Mock Data Service

```
export function findAll() {
    // implementation
}

export function findById(id) {
    // implementation
}

export var endpoint = "http://localhost:5000";
```

ECMAScript 6 Modules

Using Mock Data Service

```
import * as employeeService from "employee";

employeeService.findAll().then(function(employees) {
    console.log(employees);
});
```

ECMAScript 6 Classes

```
class Mortgage {

    constructor(amount, years, rate) {
        this.amount = amount;
        this.years = years;
        this.rate = rate;
    }

    calculate() {
        // No interest mortgage for ES6 fans
        return this.amount / (this.years * 12);
    }

}

let m = new Mortgage(200000, 30, 3);
console.log(m.calculate());
```

ECMAScript 6 Classes

```
class Shape {

    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    render() {
        console.log('Rendering shape');
    }

}
```

ECMAScript 6 Classes

```
class Circle extends Shape {

    constructor(x, y, radius) {
        super(x, y);
        this.radius = radius;
    }

    static get pi() {
        return 3.14159265359;
    }

    get circumference() {
        return 2 * Circle.pi * this.radius;
    }

    render() {
        console.log('Rendering circle');
    }

}
```

ECMAScript 6 Classes

```
var c = new Circle(0, 0, 10);
console.log(c.x);                // 0
c.x = 5;
console.log(c.x);                // 5
console.log(Circle.pi);          // 3.14159265359
console.log(c.circumference); // 62.8318530718
c.render();                      // Rendering circle
```

Template String

Template Strings

```
var str = `Hello World!`
console.log(str);
```

```
var multiLine = `This is an example
    of a multiline string`;
console.log(multiLine);
```

Template String

String Substitution

```
var name = "Christophe";
console.log(`Hello, ${name}!`);
```

```
var user = {firstName: "Lisa", lastName: "Wong"};
console.log(`Hello, ${user.firstName} ${user.lastName}!`);
```

Template String

Expressions

```
console.log(`Today is ${new Date()}!`);
```

```
var price = 100;
var exchangeRate = 0.89;
console.log(`Price in Euro: ${price * exchangeRate}`);
```

```
console.log(`Hello, ${user.firstName} ${user.lastName}!
Today is ${new Date()}`);
```

What is a Generator?

- Factory for special type of function declared using function*
- Can be exited at a particular point using yield
- Resumes at same point and in same state when invoked again

Generator Example

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

var gen = idMaker();

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

ES5 "Map"

```
var settings = {};
settings.server = "http://localhost";
settings.userName = "Christophe";
settings["language"] = "EN";
```

Keys have to be strings

ES6 Map

```
var map = new Map();
map.set("server", "http://localhost");
map.set("userName", "Christophe");
map.get("userName");
map.has("userName");
map.delete("userName");
```

Keys can be anything

WeakMap

- Similar to Map
- Keys must be objects
- Doesn't prevent keys from being garbage-collected
- Useful to prevent memory leaks
- Can't be iterated

Set

Ordered list of unique arbitrary values

```
var colors = new Set();
cards.add("red");
cards.add("blue");
console.log(colors.size); // 2;
console.log(colors.has("red")); // true;
items.delete("red");
```

WeakSet

- Similar to Set
- Values must be objects
- Doesn't prevent values from being garbage-collected
- Can't be iterated over

With Babel & npm

- Create project folder
- Run the following command
  - npm init
  - npm install babel-cli babel-core --save-dev
  - npm install babel-preset-es2015 --save-dev
  - npm install http-server --save-dev
- Modify package.json
  - "scripts": { "babel": "babel --presets es2015 js/main.js -o build/main.bundle.js", "start": "http-server -p 9000" },
- Create ‘build’ folder
- Run build
  - npm run babel
- Add into index.html
  - <script src="build/main.bundle.js"></script>
- Run application
  - npm start
- Use URL in browser – http://localhost:9000

With babel with .babelrc

- npm install babel-cli
- npm install babel-preset-es2015
- Create .babelrc
- Add

```
{ "presets": ["es2015"] }
```

- Transpile

- babel src -d dest

With babel & webpack

- Create project folder
- Run the following command
  - npm init
  - npm install babel-core babel-loader babel-preset-es2015 webpack --save-dev
  - npm install http-server --save-dev
- Create webpack.config.js

- var path = require('path'); var webpack = require('webpack'); module.exports = { entry: './js/app.js', output: { path: path.resolve(__dirname, 'build'), filename: 'app.bundle.js' }, module: { loaders: [{ test: /\.js$/, loader: 'babel-loader', query: { presets: ['es2015'] } }] }, stats: { colors: true }, devtool: 'source-map' };

- Modify package.json
  - "scripts": { "webpack": "webpack", "start": "http-server" },

- Create ‘build’ folder

With babel & webpack

- Run build
  - npm run webpack
- Add into index.html
  - <script src="build/app.bundle.js"></script>
- Run application
  - npm start
- Use URL in browser – http://localhost:8080

For -> for-in-> forEach -> for-of

- for-of is a new loop in ES6 that replaces both for-in and forEach() and supports the new iteration protocol
- Use it to loop over *iterable* objects (Arrays, strings, Maps, Sets, etc.
  - ```
    const iterable = ['a', 'b'];
    for (const x of iterable) {
        console.log(x);
    } // Output: // a // b
    ```
- break and continue work inside for-of loops:
  - ```
    for (const x of ['a', '', 'b']) {
     if (x.length === 0) break;
         console.log(x);
      } // Output: // a
    ```
- Access both elements and their indices while looping over an Array.
  - ```
    const arr = ['a', 'b'];
    for (const [index, element] of arr.entries()) {
         console.log(`${index}. ${element}`);
     } // Output: // 0. a // 1. b
    ```
    ```

For -> for-in-> forEach -> for-of

- const map = new Map([[false, 'no'], [true, 'yes'],]);

```
for (const [key, value] of map) {
    console.log(`${key} => ${value}`);
}
```

Private Data

- Keeping private data in the environment of a class constructor
- Marking private properties via a naming convention (e.g. a prefixed underscore)
- Keeping private data in WeakMaps
- Using symbols as keys for private properties

Private Data- class constructor

```
class Countdown {
    constructor(counter, action) {
        Object.assign(this, {
            dec() {
                if (counter < 1) return;
                counter--;
                if (counter === 0) {
                    action();
                }     }
        });
    } }
```

Using Countdown looks like this:

```
> const c = new Countdown(2, () => console.log('DONE'));
> c.dec();
> c.dec();
```

Private Data – class constructor

- Pros:
  - The private data is completely safe
  - The names of private properties won't clash with the names of other private properties (of superclasses or subclasses).
- Cons:
  - The code becomes less elegant, because you need to add all methods to the instance, inside the constructor (at least those methods that need access to the private data).

Private Data - Marking - naming convention

```
class Countdown {
    constructor(counter, action) {
        this._counter = counter;
        this._action = action;
    }
    dec() {
        if (this._counter < 1) return;
        this._counter--;
        if (this._counter === 0) {
            this._action();
        }
    }
}
```

Private Data - Marking - naming convention

Pros:

- Code looks nice.
- We can use prototype methods.

Cons:

- Not safe, only a guideline for client code.
- The names of private properties can clash.

Private Data - Weakmaps

```
const _counter = new WeakMap();
const _action = new WeakMap();
class Countdown {
    constructor(counter, action) {
        _counter.set(this, counter);
        _action.set(this, action);
    }
    dec() {
        let counter = _counter.get(this);
        if (counter < 1) return;
                            counter--;
        _counter.set(this, counter);
        if (counter === 0) {
            _action.get(this)();
        }
    } }
```

Private Data - symbols

- Storage location for private data are properties whose keys are symbols:

```
const _counter = Symbol('counter');
const _action = Symbol('action');
class Countdown {
    constructor(counter, action) {
        this[_counter] = counter;
        this[_action] = action;
    }
    dec() {
        if (this[_counter] < 1) return;
        this[_counter]--;
        if (this[_counter] === 0) {
            this[_action]();
        }
    }}
```

Private Data

- Each symbol is unique, which is why a symbol-valued property key will never clash with any other property key.

```
const c = new Countdown(2, () => console.log('DONE'));
console.log(Object.keys(c));     // []
console.log(Reflect.ownKeys(c));   //[ Symbol(counter), Symbol(action)]
```

- Pros:
- We can use prototype methods.
- The names of private properties can’t clash.

- Cons:
- Code is not as elegant as a naming convention.
- you can list all property keys (including symbols!) of an object via Reflect.ownKeys().

Private Data

```
const privateMethods = {
  privateMethod () {
    console.log(this.say);
  }
}

export default class Service {
  constructor () {
    this.say = "Hello";
  }

  publicMethod () {
    privateMethods.privateMethod.call(this);
  }
}
```

Symbols

- *A **symbol** is a unique and immutable data type and may be used as an identifier for object properties. The* symbol object *is an implicit object wrapper for the symbol primitive data type.*

```
const privateMethod = Symbol('privateMethod');
export default class Service {
  constructor () {
    this.say = "Hello";
  }
  [privateMethod] () {
    console.log(this.say);
  }
  publicMethod () {
    this[privateMethod]()
  }
}
```

Symbols

- *// Uncaught TypeError: (intermediate value).privateMethod is not a function*
- *new Service().privateMethod()*

- *// Uncaught TypeError: (intermediate value)[Symbol(...)] is not a function*
- *new Service()[Symbol('privateMethod')]();*

Void keyword

- When using an immediately-invoked function expression, void can be used to force the function keyword to be treated as an expression instead of a declaration.

- Consider the following example –

```
void function iife_void() {
        var msg = function () {
            console.log("hello world")
        };
        msg();
}();
```

Reflection API

- Reflect.defineProperty

```
var yay = Reflect.defineProperty(target, 'foo', { value: 'bar' })
if (yay) {
  // yay!
} else {
  // oops.
}
```

- *OLD :*

```
try {
  Object.defineProperty(target, 'foo', { value: 'bar' })
  // yay!
} catch (e) {
  // oops.
}
```

Reflection API

- Reflect.deleteProperty

```
var target = { foo: 'bar', baz: 'wat' }
Reflect.deleteProperty(target, 'foo')
console.log(target)
// <- { baz: 'wat' }
```

Reflection API

- Reflect.construct
- Passing arbitrary argument list while object creation
- OLD:

```
var proto = Dominus.prototype
Applied.prototype = proto
function Applied (args) {
  return Dominus.apply(this, args)
}
function apply (a) {
  return new Applied(a)
}
```

=➔

```
apply(['.foo', '.bar'])
```

Reflection API

- Reflect.construct
- Passing arbitrary argument list while object creation
- NEW:

new Dominus(...args)

Another alternative is to go with Reflect .

Reflect.construct(Dominus, args)

Reflection API

- Reflect.construct -> Factory Method

```
class Greeting {
    constructor(name) {
        this.name = name;
    }
    greet() {
      return `Hello ${this.name}`;
    }
}
const greetingFactory = (name) => Reflect.construct(Greeting, [name]);
Console.log(greetingFactory('abc').greet()));

// ES5 style factory:
function greetingFactory(name) {
    var instance = Object.create(Greeting.prototype);
    Greeting.call(instance, name);
```

Reflection API

- Reflect.construct -> Factory Method

```
// ES6 style factory
function greetingFactory(name) {
    return Reflect.construct(Greeting, [name], Greeting);
}

// Or, omit the third argument, and it will default to the first argument.
function greetingFactory(name) {
  return Reflect.construct(Greeting, [name]);
}

// Super slick ES6 one liner factory function!
const greetingFactory = (name) => Reflect.construct(Greeting, [name]);
```

Reflection API

- Reflect.apply
- Passing arbitrary argument list to function
- OLD:

```
fn.apply(ctx, [1, 2, 3])
```

fn might be shadow apply.
more verbose alternative.

```
Function.prototype.apply.call(fn, ctx, [1, 2, 3])
```

Reflection API

- Reflect.apply
- Passing arbitrary argument list to function
- NEW:

fn(...[1, 2, 3])

PROBLEM in defining a this context.

Reflect.apply(fn, ctx, args)

Proxies

- Proxy to determine behavior whenever the properties of a target object are accessed.
- A handler object can be used to configure *traps* for your Proxy.

```
var target = {}
var handler = {}
var proxy = new Proxy(target, handler)
proxy.a = 'b'
console.log(target.a)
// <- 'b'
console.log(proxy.c === undefined)
// <- true
```

Proxies

- With trap - get

The proxy below is able to track any and every property access event because it has a handler.get trap.

```
var handler = {
  get (target, key) {
    console.info(`Get on property "${key}"`)
    return target[key]
  }
}
var target = {}
var proxy = new Proxy(target, handler)
proxy.a = 'b'
proxy.a    // <- 'Get on property "a"'
proxy.b   // <- 'Get on property "b"'
```

Proxies

- With trap – set -> Restricting private members from accessing.

```
var handler = {
  get (target, key) {
    invariant(key, 'get')
    return target[key]
  },
  set (target, key, value) {
    invariant(key, 'set')
    return true
  } }
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}"
property`)
  }
}
```

Proxies

- With trap – set -> Restricting private members from accessing.

```
var target = {}
var proxy = new Proxy(target, handler)
proxy.a = 'b'
console.log(proxy.a)
// <- 'b'
proxy._prop
// <- Error: Invalid attempt to get private "_prop" property
proxy._prop = 'c'
// <- Error: Invalid attempt to set private "_prop" property
```

Proxies

- Restricting direct access of target.

```
function proxied () {
  var target = {}
  var handler = {
    get (target, key) {
      invariant(key, 'get')
      return target[key]
    },
    set (target, key, value) {
      invariant(key, 'set')
      return true
    }
  }
  return new Proxy(target, handler)
}
```

Proxies

- Restricting direct access of target, only via proxy.

```
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}"
property`)
  }
}
```

Proxies

- Schema Validation using proxy.

```
var validator = {
  set (target, key, value) {
    if (key === 'age') {
      if (typeof value !== 'number' || Number.isNaN(value)) {
        throw new TypeError('Age must be a number')
      }
      if (value <= 0) {
        throw new TypeError('Age must be a positive number')
      }
    }
    return true
  }
}
```

Proxies

- Schema Validation using proxy.

```
var person = { age: 27 }
var proxy = new Proxy(person, validator)
proxy.age = 'foo'
// <- TypeError: Age must be a number
proxy.age = NaN
// <- TypeError: Age must be a number
proxy.age = 0
// <- TypeError: Age must be a positive number
proxy.age = 28
console.log(person.age)
// <- 28
```

Proxies Revocable

- Schema Validation using proxy.

```
var target = {}
var handler = {}
var {proxy, revoke} = Proxy.revocable(target, handler)
proxy.a = 'b'
console.log(proxy.a)
// <- 'b'
revoke()
revoke()
revoke()
console.log(proxy.a)
// <- TypeError: illegal operation attempted on a revoked proxy
```

Decorators

- decorator is simply a way of wrapping one piece of code with another – literally “decorating” it.

```
function doSomething(name) {
  console.log('Hello, ' + name);
}
function loggingDecorator(wrapped) {
  return function() {
    console.log('Starting');
    const result = wrapped.apply(this, arguments);
    console.log('Finished');
    return result;
  }
}
```

Decorators

- decorator is simply a way of wrapping one piece of code with another – literally “decorating” it.

```
const wrapped = loggingDecorator(doSomething);
```

This example produces a new function – in the variable wrapped – that can be called exactly the same way as the doSomething function, and will do exactly the same thing. The difference is that it will do some logging before and after the wrapped function is called.

```
doSomething('Graham');
// Hello, Graham
wrapped('Graham');
// Starting
// Hello, Graham
// Finished
```

Decorators - class members

Property decorators are applied to a single member in a class – whether they are properties, methods, getters, or setters.

This decorator function is called with three parameters:

target – The class that the member is on.

name – The name of the member in the class.

descriptor – The member descriptor. This is essentially the object that would have been passed to Object.defineProperty.

The classic example used here is @readonly. This is implemented as simply as:

```
function readonly(target, name, descriptor) {
  descriptor.writable = false;
  return descriptor;
}
```

Decorators - class members

Literally updating the property descriptor to set the “writable” flag to false.

This is then used on a class property as follows:

```
class Example {
  a() {}
  @readonly
  b() {}
}
const e = new Example();
e.a = 1;
e.b = 2;
// TypeError: Cannot assign to read only property 'b' of object '#<Example>'
```

But we can do better than this. We can actually replace the decorated function with different behavior. For example, let’s log all of the inputs and outputs:

Decorators - class members

```
function log(target, name, descriptor) {
  const original = descriptor.value;
  if (typeof original === 'function') {
    descriptor.value = function(...args) {
      console.log(`Arguments: ${args}`);
      try {
        const result = original.apply(this, args);
        console.log(`Result: ${result}`);
        return result;
      } catch (e) {
        console.log(`Error: ${e}`);
        throw e;
      }    } }
  return descriptor;
}
```

Decorators - class members

This replaces the entire method with a new one that logs the arguments, calls the original method and then logs the output.

We can see this in use as follows:

```
class Example {
    @log
    sum(a, b) {
        return a + b;
    }
}
const e = new Example();
e.sum(1, 2);
// Arguments: 1,2
// Result: 3
```

the apply function allows you to call the function, specifying the this value and the arguments to call it with.

Decorators - class members

we can arrange for our decorator to take some arguments. For example, let's re-write our log decorator as follows:

```
function log(name) {
  return function decorator(t, n, descriptor) {
    const original = descriptor.value;
    if (typeof original === 'function') {
      descriptor.value = function(...args) {
        console.log(`Arguments for ${name}: ${args}`);
        try {          const result = original.apply(this, args);
          console.log(`Result from ${name}: ${result}`);
          return result;
        } catch (e) {
          console.log(`Error from ${name}: ${e}`);
          throw e;
        }     }   }
    return descriptor;   }; }
```

Decorators - class members

This is getting more complex now, but when we break it down what we have is:

```
class Example {
  @log('some tag')
  sum(a, b) {
    return a + b;
  }
}

const e = new Example();
e.sum(1, 2);  // Arguments for some tag: 1,2
// Result from some tag: 3
```