

รีวิฟรีเจอร์ใหม่ใน

# จาวาสคริปต์มาตรฐาน ES7, ES8

(ECMAScript 2016 กับ ECMAScript 2017)

แก้ไขครั้งที่ 3.1

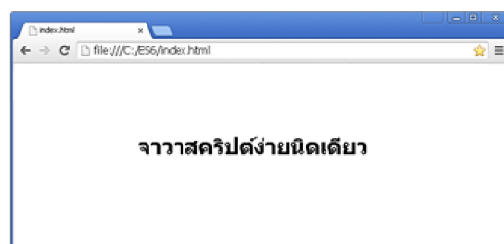
เขียนโดย แอดมินโฮ โอน้อยออก

JAVASCRIPT  
(ECMAScript)

*EBook เล่มนี้สงวนลิขสิทธิ์ตามกฎหมาย ห้ามมิให้ผู้ใด นำไปเผยแพร่ต่อสาธารณะ เพื่อประโยชน์ในการค้า หรืออื่นๆ โดย  
ไม่ได้รับความยินยอมเป็นลายลักษณ์อักษรจากผู้เขียน*

## ให้ความรู้เพิ่มเติมนิดหนึ่ง เผื่อคนไม่รู้จักภาษา JavaScript

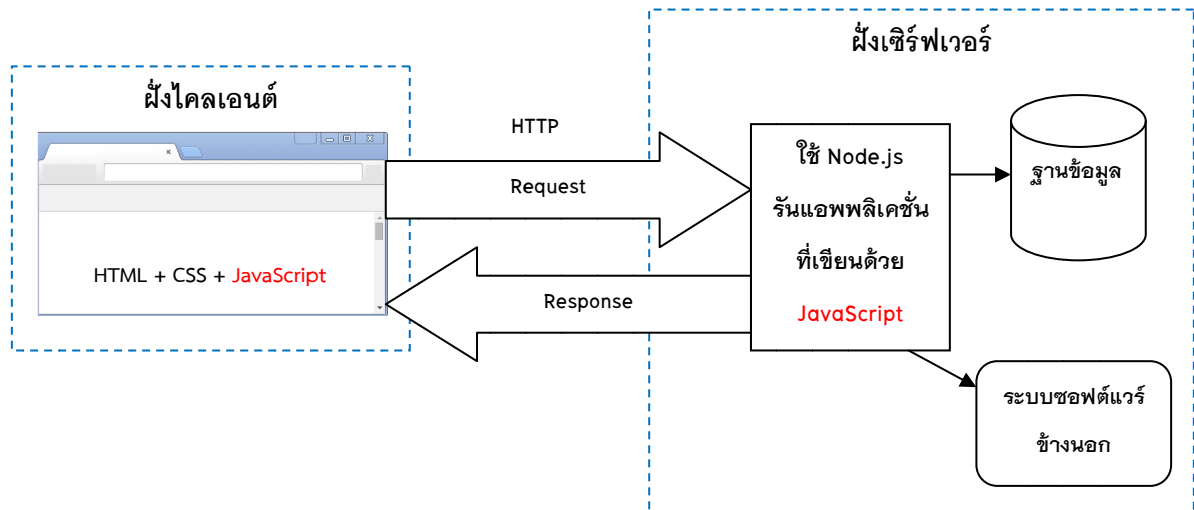
- ภาษา JavaScript เป็นภาษาโปรแกรมเชิงวัตถุแบบไดนามิกไทป์ (Dynamic types) ซึ่งไวยากรณ์ของมันได้นำโครงสร้างมาจากภาษายอดนิยมอย่าง Java กับภาษาซี C
- โปรแกรมที่เขียนขึ้นด้วย JavaScript จะต้องทำงานอยู่บน JavaScript engine ที่เป็นทั้งตัวแปลภาษา (Interpreter) และใช้รันโปรแกรม สำหรับการทำงานของ JavaScript ที่เราค้นเคยกันดี จะทำงานอยู่บนเว็บเบราว์เซอร์ เช่น Google Chrome, Firefox และ Internet Explorer เป็นต้น ซึ่งจะมี JavaScript engine ติดตั้งมาให้อยู่แล้ว



- นักพัฒนาซอฟต์แวร์ส่วนใหญ่ล้วนรู้จักภาษา JavaScript ซึ่งถือว่ายอดนิยมใช้กันมากภาษาหนึ่งในโลก ถ้าศึกษาอย่างผิวเผินก็อาจคิดว่าง่ายง่าย แต่เมื่อศึกษาลงลึก ๆ แล้ว จะพบว่ามันโคตรจะอินดี้ เป็นภาษาปราบเซียนตัวหนึ่ง จนคนไม่ค่อยเข้าใจกันมากเท่าไรนัก จนหาว่าไม่ว่ามันมีความสามารถแฝงที่ซ่อนเร้นอยู่เยอะเลย
- JavaScript ไม่ใช่ภาษา Java นะครับ คนละภาษา (คนมักสับสนกัน)

**แวนน้ำ ไม่ใช่ แวนจันได**  
**JavaScript ก็ไม่ใช่ Java จันนั่น**

- คนส่วนใหญ่รู้แค่จะใช้ JavaScript ร่วมกับภาษา HTML (ปัจจุบันเวอร์ชัน HTML5.1) กับ CSS (ปัจจุบันเวอร์ชัน CSS3) เพื่อทำให้เว็บมันไดนามิก ฟุ้งฟ้าง กรู้งกิ้ง (มันดังในฝั่ง Font-end มานาน)
- แต่ปัจจุบันนี้ **JavaScript สมัยใหม่** มันก้าวหน้าไปไกลมาก ๆ เพราะสามารถทำงานอยู่ฝั่งเซิร์ฟเวอร์ได้ (Back-end) ด้วย Node.js แม้แต่เอาไปทำแอปบนโมบาย หรือแม้แต่เว็บอท ก็ยังทำได้ด้วย ....อาเยยะ



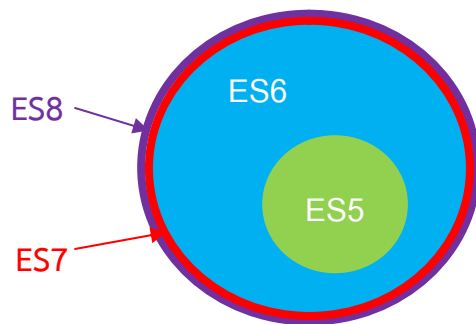
- องค์กร Ecma International (องค์กรจัดการมาตรฐานแห่งยุโรป ) เป็นผู้กำหนดมาตรฐาน JavaScript ซึ่งจะเรียกมาตรฐานนี้ว่า “ECMA-262” ส่วนตัวภาษา JavaScript นั้น ก็จะมีชื่อเรียกเต็มยศอย่างเป็นทางการว่า “ภาษา ECMAScript”

## JAVASCRIPT (ECMAScript)

- ES6 (ECMAScript 2015) เป็นมาตรฐานใหม่ล่าสุดของ JavaScript ประกาศออกมาเมื่อกลางเดือนมิถุนายนปี 2558 ซึ่งถือว่า เปลี่ยนแปลงเวอร์ชันครั้งใหญ่ที่สุดในประวัติศาสตร์ของภาษานี้ หลังจากไม่ได้เปลี่ยนมาเกือบ 6 ปี (เวอร์ชันเก่าที่เราคุ้นเคยกันดี ก็คือ ES5)



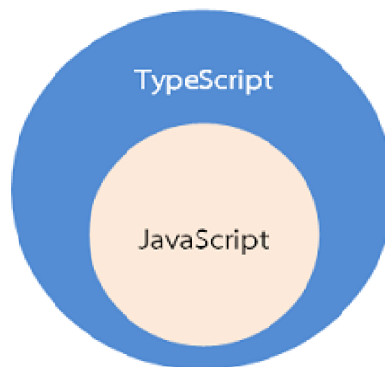
- ปีค.ศ. 2016 เวอร์ชันใหม่ ES7 (ECMAScript 2016) ก็ออกมาแหละ ส่วนปีหน้า 2017 ก็จะเป็นคิวของเวอร์ชัน ES8 (ECMAScript 2017) จะออกมาเช่นกัน
- ต้องเข้าใจอย่างนี้นะครับ เนื่อง ES6 มันใหญ่โตลงการทำงานของงานสร้างมาก คึ้นรอบปล่อยออกมาหมดทีเดียว ก็คงรอหลายชาติภพ อาจทำให้มีเสียงบ่นตามมาได้ ด้วยเหตุนี้เข้าถึงเพิ่มฟีเจอร์เล็กยิบ ๆ ย่อย ๆ มาใส่ไว้ในเวอร์ชันหลัง ๆ แทน
- โดยคาดว่าจากนี้ไป จะมีการประกาศเวอร์ชันใหม่ทุก ๆ ปี โดยให้คิดเสียว่า ES6 เหมือนโปรแกรมหลัก ส่วนเวอร์ชันที่ออกตามทีหลัง ไม่ได้ว่าจะเป็น ES7, ES8 และ ESXXXX (ถ้ามีตอนนะ) ก็คือการอัปเดตซอฟต์แวร์ อะไรประมาณนี้



- API ที่ใช้ติดต่อกับ DOM หรือใช้งานร่วมกับ HTML5.1, CSS3 ใน ES6 เขาไม่ได้เปลี่ยนแปลงอะไรเลย
- ES6,ES7,ES8 มันเป็นแค่มาตรฐานใหม่สด ๆ ชิง ๆ ดังนั้นการใช้งานโดยตรงบนเว็บเบราว์เซอร์ (ปัจจุบันที่ผมเขียนอยู่นี้) ก็ยังไม่ support ทุกฟีเจอร์ ต้องมีตัวคอมไพเลอร์ช่วยก่อน (ยังมีข้อจำกัดบางประการ) ...แต่ถ้าใครใช้ Node.js เวอร์ชัน 7 ก็รองรับ ES6 ได้ 99%

ชื่อเวอร์ชัน	ชื่อมาตรฐานเดิม	ปีที่ออก
ES6	ECMAScript 2015	2015
ES7	ECMAScript 2016	2016
ES8	ECMAScript 2017	2017
เวอร์ชันต่อไปนี้จะอัปเดตจาก ES6 ไปเรื่อยๆ (ถ้ามีตอนนะ)		

- TypeScript เป็นภาษาดัดแปลงมาจาก JavaScript โดยทั้งนี้ไวยากรณ์และพีเจอร์ต่างๆ จะมากกว่า อาจมองว่าเป็นซูเปอร์เซตของ JavaScript อีกที (แน่นอนมันครอบคลุม ES6) ซึ่งเจ้าของภาษาคือ Microsoft



- ลองมาดูความนิยมของ ES6 กัน



จากรูปเป็นผลสำรวจจปี 2016 จะเห็นว่ามาตรฐานใหม่ ES6 คนเริ่มใช้งานเยอะ ไล่จี้ JavaScript แบบเก่าติดๆ แล้ว (ES5) (ที่มา <http://stateofjs.com/2016/flavors/>)

### ข้อตกลงเวลาอ่านเอกสารชุดนี้

- จะกล่าวถึงเฉพาะพีเจอร์ที่เพิ่มเข้ามาใน ES7 และ สิ่งที่เปลี่ยนแปลงไปในเวอร์ชันดังกล่าว
- ส่วน ES8 จะรื้อให้ดูก่อนล่วงหน้า อนาคตอาจเปลี่ยนแปลงได้ เพราะเขายังไม่ประกาศออกมา
- สำหรับความรู้ JavaScript มาตรฐานใหม่ ES6 แบบเจาะลึกถึงหัวหัวใจ (เนื้อหาเยอะมาก) ท่านสามารถอ่านได้จากหนังสือที่อ้างอิง [1] เพราะความรู้ JavaScript แบบเก่า (ES5) ที่คุณรู้จัก ...นับวันใกล้หมดอายุลงเต็มทน

## ตัวอย่างการเขียน ES6 กับ ES7

ต่อไปจะแสดงการเขียน JavaScript ด้วย ES6 กับ ES7 แล้วสั่งรันผ่านทาง Node.js โดยตรง

\*\*\* ทั้งนี้ Node.js เวอร์ชัน 7 ก็รองรับ ES6 ได้ 99 %

```
class Chat{                                     // class ไวยากรณ์ใหม่ของ ES6
    constructor(message) {                     // constructor ไวยากรณ์ใหม่ของ ES6
        this.message = message;
    }
    say() {
        console.log(this.message);
    }
}
let chat = new Chat("Hello, world!");          // let ไวยากรณ์ใหม่ของ ES6
chat.say();                                    // "Hello, world!"

let array = ["A", "B", "C"];
console.log(array.includes("A"));              // true    -- เมธอดของอาร์เรย์ที่เพิ่มมาใน ES7
```

ถ้าซอร์สโค้ดดังกล่าวเซฟเป็นไฟล์เก็บไว้ที่ "C:/ES6/test.js" และมีโครงสร้างโปรเจกต์ดังนี้

```
C:/ES6/
```

```
|- test.js
```

เมื่อรันคอมมานด์ไลน์บนวินโดวส์ (ให้ Node.js มาอ่านและประมวลผลไฟล์ JavaScript) ก็จะได้ผลลัพธ์ดังภาพ

```
C:\ES6>node test.js
Hello, world!
true
```

หมายเหตุ สำหรับวิธีติดตั้งและรัน JavaScript บน Node.js สามารถอ่านเพิ่มเติมได้ที่ (เป็นเอกสาร PDF แจกฟรี)

- <http://www.patanasongsivilai.com/javascript.html>
- ขณะเดียวกัน เนื้อหาในหัวข้อถัดไปจากนี้ จะใช้ npm (ติดตั้งมาพร้อมกับ Node.js) มาประกอบการอธิบายพอควร ดังนั้นจึงแนะนำให้ท่านอ่านเอกสารแจกฟรีดังกล่าวด้วยครับ

## ตัวอย่างการเขียน ES6 กับ ES7 บนเว็บเบราว์เซอร์

เนื่องจากเว็บเบราว์เซอร์ส่วนใหญ่จะใช้งานได้กับ ES5 ด้วยเหตุนี้จึงต้องนำซอร์สโค้ดที่เขียนด้วย ES6 มาคอมไพล์ ด้วยคอมไพเลอร์ที่เรียกว่า “transpiler” เพื่อแปลงจาก ES6 ให้กลายเป็นเวอร์ชัน ES5 ที่เว็บเบราว์เซอร์ส่วนใหญ่ใช้งานได้ไปก่อน

### Traceur

โดยตัวอย่างต่อไปนี้จะแสดงการเขียน JavaScript บนเว็บเบราว์เซอร์ โดยใช้ Traceur ทำตัวเป็น transpiler (อย่าเพิ่งสนใจรายละเอียดซอร์สโค้ดที่ยกมาให้ดูนะครับ)

```
<!-- ไฟล์ index.html-->
<!DOCTYPE html>
<html>
<head>

<!-- Traceur (เป็นตัว transpiler)-->
<script src="https://google.github.io/traceur-compiler/bin/traceur.js"></script>
<script src="https://google.github.io/traceur-compiler/bin/BrowserSystem.js"></script>
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js"></script>
</head>
<body>
<h1 id="element1"></h1>
<script type="module">                                     // ต้องเขียนกำกับ type = "module"

    class Chat{                                           // class ไวยากรณ์ใหม่ของ ES6
        constructor(message) {                          // constructor ไวยากรณ์ใหม่ของ ES6
            this.message = message;
        }
        say() {
            let element = document.querySelector('#element1');
            element.innerHTML = this.message;
        }
    }

    let chat = new Chat("Hello, world!");                // let ไวยากรณ์ใหม่ของ ES6
    chat.say();

    // ตัวอย่างโค้ด ES7 ชุดนี้ยังรันได้เฉพาะบน Google Chrome

    let array = ["A", "B", "C"];                         // let ไวยากรณ์ใหม่ของ ES6
    console.log(array.includes("A"));                    // true    -- เมธอดของอาร์เรย์ที่เพิ่งเข้ามาใน ES7
</script>
</body>
</html>
```

ถ้าซอร์สโค้ดดังกล่าวเซฟเป็นไฟล์เก็บไว้ที่ “C:/ES6/index.html” เมื่อดับเบิลคลิกเปิดมันขึ้นมา ก็จะปรากฏดังรูป



หมายเหตุ ถ้าอ่านจากหนังสือ [1] วิธีการใช้ ES6 กับ ES7 จะต่างกับซอร์สโค้ดที่เห็นในตัวอย่างนี้เล็กน้อย เพราะ traceur มันมีการเปลี่ยนแปลง ด้วยการเพิ่ม BrowserSystem.js เข้าไปใน <head> ...</head> ดังนี้

```
<script src="https://google.github.io/traceur-compiler/bin/BrowserSystem.js"></script>
```

(ที่มา <https://github.com/google/traceur-compiler/wiki/Getting-Started>)

สังเกตในโค้ดตัวอย่างที่ยกมาให้ดูก่อนหน้านี้ จะต้องระบุ `<script type="module">`

แต่ถ้าจะเขียนโค้ด JavaScript แยกออกมาเป็นไฟล์ .js เช่น mylib.js ก็สามารทำได้ โดยจะมีโครงสร้างข้างล่าง

```
C:\ES6>
|-- index.html
|-- mylib.js
```

ส่วนไฟล์ mylib.js ก็หน้าตาแบบนี้ไง แค่แยกโค้ด JavaScript ออกมา

```
class Chat{                                     // class ไวยากรณ์ใหม่ของ ES6
  constructor(message) {                       // constructor ไวยากรณ์ใหม่ของ ES6
    this.message = message;
  }
  say(){
    let element = document.querySelector('#element1');
    element.innerHTML = this.message;
  }
}

let chat = new Chat("Hello, world!");          // let ไวยากรณ์ใหม่ของ ES6
chat.say();

// ตัวอย่างโค้ด ES7 ชุดนี้ยังรันได้เฉพาะบน Google Chrome
let array = ["A", "B", "C"];                   // let ไวยากรณ์ใหม่ของ ES6
console.log(array.includes("A"));               // true    -- เมธอดของอาร์เรย์ที่เพิ่มเข้ามาใน ES7
```

สามารถเขียนอ้างไฟล์ .js ได้ง่ายๆ ดังนี้ (หน้าถัดไป)



```

<!-- ไฟล์index.html-->
<!DOCTYPE html>
<html>
<head>

<!-- Traceur (ใช้เป็นตัวtranspiler)-->
<script src="https://google.github.io/traceur-compiler/bin/traceur.js"></script>
<script src="https://google.github.io/traceur-compiler/bin/BrowserSystem.js"></script>
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js"></script>

</head>
<body>
<h1 id="element1"></h1>

<script type="module">
import "./mylib.js";    // อ้างไฟล์.js
</script>

</body>
</html>

```

**หมายเหตุ** วิธีพิมพ์โค้ดไฟล์ด้วยวิธีนี้ ถ้าไปเปิดดูบน Google Chrome อาจไม่ทำงาน แต่ไม่ต้องซีเรียส เรามีทางแก้ไข แนะนำให้ไปอ่านหัวข้อถัดไปเรื่อง [Cross-origin resource sharing \(CORS\)](#)

## Babel

ต่อไปจะแสดงการเขียน JavaScript บนเว็บเบราว์เซอร์ โดยใช้ Babel ทำตัวเป็น transpiler (ผลการทำงานจะเหมือนตัวอย่างตอนใช้ Traceur)

```

<!-- ไฟล์index.html-->
<!DOCTYPE html>
<html>
<head>

<!-- Babel (ใช้เป็นตัวtranspiler)-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.14.0/babel.min.js"></script>

</head>
<body>
<h1 id="element1"></h1>
<script type="text/babel">                                // ต้องเขียนกำกับ type = "text/babel"
    class Chat{                                           // class ไวยากรณ์ใหม่ของ ES6
        constructor(message) {                            // constructor ไวยากรณ์ใหม่ของ ES6
            this.message = message;
        }
        say(){
            let element = document.querySelector('#element1');
            element.innerHTML = this.message;
        }
    }
    let chat = new Chat("Hello, world!");    // let ไวยากรณ์ใหม่ของ ES6

```

```

chat.say();

// ตัวอย่างโค้ด ES7 ชุดนี้ยังรันได้เฉพาะบน Google Chrome
let array = ["A", "B", "C"];           // let ไวยากรณ์ใหม่ของ ES6
console.log(array.includes("A"));       // true    -- เมธอดของอาร์เรย์ที่เพิ่มเข้ามาใน ES7
</script>
</body>
</html>

```

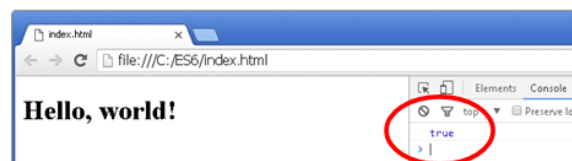
จะสมมติว่าบันทึกเป็นไฟล์ index.html โดยมีโครงสร้างโปรเจกต์ดังนี้

```

C:\ES6>
|-- index.html

```

เมื่อดับเบิลคลิกที่ไฟล์ index.html จะปรากฏตามรูป



สังเกตในโค้ดจะต้องระบุ `<script type="text/babel">` หรือเขียนเป็น `<script type="text/jsx">` ก็ได้เหมือนกัน

แต่ถ้าจะเขียนโค้ด JavaScript แยกออกมาเป็นไฟล์ .js เช่น mylib.js ก็สามารทำได้ โดยจะมีโครงสร้างข้างล่าง (ไฟล์ .js โค้ดข้างในจะหน้าตาเหมือนตอนใช้ Traceur เต็มเลย)

```

C:\ES6>
|-- index.html
|-- mylib.js

```

สามารถเขียนอ้างไฟล์ .js ได้ง่ายๆ ...ดั่งหน้าถัดไป (สังเกตโค้ดดีๆ วิธีอิมพอร์ตไฟล์ .js จะต่างกับ Traceur เล็กน้อย)

```

<!-- ไฟล์index.html-->
<!DOCTYPE html>
<html>
<head>

<!-- Babel (ใช้เป็นตัวtranspiler)-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.14.0/babel.min.js"></script>

</head>
<body>
<h1 id="element1"></h1>
<script type="text/babel" src="mylib.js"> // อ้างไฟล์.js
</script>
</body>
</html>

```

หมายเหตุ วิธีอิมพอร์ตไฟล์ด้วยวิธีนี้ ถ้าไปเปิดดูบน Google Chrome อาจไม่ทำงาน แต่ไม่ต้องซีเรียส เรามีทางแก้ไข แนะนำให้ไปอ่านหัวข้อ [Cross-origin resource sharing \(CORS\)](#)

## โหลดไฟล์ Traceur กับ Babel มาเก็บไว้ที่เครื่องแบบออฟไลน์

### Traceur แบบออฟไลน์

จากตัวอย่างก่อนๆ เวลาเขียน ES6 กับ ES7 บนเว็บเบราว์เซอร์ด้วย Traceur ผมต้องอ้างถึงไฟล์ traceur.js, BrowserSystem.js และ bootstrap.js แบบออนไลน์ แต่ถ้าจะโหลดไฟล์นี้ (ทั้งหมดที่เกี่ยวข้อง) มาเก็บไว้ที่เครื่องแบบออฟไลน์ ก็ให้ใช้คำสั่ง npm ข้างล่าง (วิธีติดตั้งและใช้งาน npm ก็ตามหนังสือข้างบนที่แจกให้อ่านฟรี)

```
C:\ES6>npm install -save traceur
```

จะเห็นไฟล์ถูกโหลดเข้ามาเก็บ ได้แก่ traceur.js กับ BrowserSystem.js

```

C:\ES6\node_modules\traceur\bin
|-- BrowserSystem.js
|-- traceur.js

```

ส่วนไฟล์ bootstrap.js ก็จะอยู่ที่

```

C:\ES6\node_modules\traceur\src
|-- bootstrap.js

```

## Babel แบบออฟไลน์

สำหรับ Babel ก็เช่นกัน สามารถโหลดไฟล์ babel.js หรือ babel.min.js มาใช้แบบออฟไลน์ (เลือกใช้ไฟล์ไหนก็ได้) ด้วยคำสั่ง npm ดังนี้

```
C:\ES6>npm install --save babel-standalone
```

จะเห็นไฟล์ถูกโหลดมาเก็บตามนี้

```
C:\ES6\node_modules\babel-standalone
|-- babel.js
|-- babel.min.js
```

หรือไปที่เว็บข้างล่างแล้วเลือกโหลดไฟล์ทั้งสองนี้ก็ได้

<https://github.com/Daniel15/babel-standalone/releases>

\*\*\* Traceur กับ Babelเท่าที่ผมลองใช้งานดู มันยังไม่นิ่งเท่าไร ถ้าจะนำมาไปใช้งานยังไง ก็ควรหมั่นอัปเดตจากทีมสร้างเขาอีกทีนะครับ ...ที่สำคัญวิธีใช้งานแต่ละเจ้า ก็ดันแตกต่างกันอีกแะ! จนหนังสือที่ผมเขียนไป ถ้าใครลองทำตาม แล้วใช้งาน ES6 ไม่ได้ เค้าขอโทษแล้วกันน้า! ยังไงเดี๋ยวขออัปเดตโค้ดล่าสุดที่นี่แล้วกันนะ

## วิธีคอมไพล์จาก ES6 ให้เป็น ES5 ด้วยมือตนเอง

### Traceur

เราสามารถใช้กระบวนการแปลงซอร์สโค้ดจาก ES6 เป็น ES5 ด้วยมือตนเอง ด้วยการเปิดคอมมานไลน์ขึ้นมา (ตัวอย่างจะใช้วินโดวส์) แล้วเรียกสคริปต์ traceur ซึ่งถ้าคุณทำตามตัวอย่างก่อนหน้านี้ ที่แนะนำวิธีโหลดไฟล์ Traceur มาเก็บแบบออฟไลน์ ด้วยคำสั่ง npm install -save traceur ก็ให้ไปที่โฟลเดอร์ ...\node\_modules\.bin จะเห็นไฟล์สคริปต์ดังนี้

```
C:\ES6\node_modules\.bin
|-- traceur
|-- traceur.cmd
```

จากไฟล์ mylib.js ในตัวอย่างก่อนหน้านี้ (โค้ด ES6)

```
C:\ES6>
|-- index.html
|-- mylib.js
```

ซึ่งเราสามารถเรียกสคริปต์ traceur ให้มาทำการคอมไพล์ mylib.js เพื่อแปลงเป็น ES5 ได้คำสั่งดังนี้

```
C:\ES6\node_modules\.bin>traceur --out ../../out/mylib.js --script ../../mylib.js
```

(ถ้าติดตั้ง Traceur ด้วยคำสั่ง `npm install -g traceur` ก็ไม่ต้อง cd มาที่ C:\ES6\node\_modules\.bin)

สำหรับไฟล์ที่ถูกแปลงเป็น ES5 จะเก็บอยู่ที่โฟลเดอร์ out\mylib.js

```
C:\ES6>
|-- index.html
|-- mylib.js
|-- out
    |-- mylib.js
```

ถ้าแอบไปเปิดไฟล์ out\mylib.js ก็ sẽเห็นว่าโค้ดถูกแปลงเป็น ES5 หน้าตาหล่อสวยดังนี้

```
var Chat = function() {
  "use strict";
  function Chat(message) {
    this.message = message;
  }
  return ($traceurRuntime.createClass)(Chat, {say: function() {
    var element = document.querySelector('#element1');
    element.innerHTML = this.message;
  }}, {});
}();
var chat = new Chat("Hello, world!");
chat.say();
var array = ["A", "B", "C"];
console.log(array.includes("A"));
```

จากตัวอย่างเดิม ก็สามารถเขียนใหม่ได้ดังนี้

```
<!-- ไฟล์index.html-->
<!DOCTYPE html>
<html>
<head>

<!-- ระบุตัวtranspiler -->
<script src="node_modules/traceur/bin/traceur-runtime.js"></script>

</head>

<body>
<h1 id="element1"></h1>

<!-- ไฟล์.js ที่ถูกแปลงเป็น ES5 -->
<script src="out/mylib.js"></script>

</body>
</html>
```

ซึ่งผลการทำงานจะเหมือนกับตัวอย่างก่อนๆ ที่ยกมา

## Babel

สำหรับ Babel ก็เช่นกัน สามารถใช้กระบวนการแปลงซอร์สโค้ดจาก ES6 ให้เป็น ES5 ด้วยมือตนเอง โดยทำตามตัวอย่างจากเว็บต้นทางผู้สร้าง เขาจะแนะนำตามนี้

```
var input = 'const getMessage = () => "Hello World";';
var output = Babel.transform(input, { presets: ['es2015'] }).code;
```

จากตัวอย่างเดิม ก็สามารถเขียนใหม่ได้ดังนี้

```
<!-- ไฟล์index.html-->
<!DOCTYPE html>
<html>
<head>

<!-- ระบุตัวtranspiler -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.14.0/babel.min.js"></script>

</head>
<body>
<h1 id="element1"></h1>
<script>                                // ไม่ต้องเขียนกำกับ type = "text/babel"

    // ใช้Template Strings ของES6 เขียนได้
    var input = `
    class Chat{                                // class ไวยากรณ์ใหม่ของES6
        constructor(message) {                // constructor ไวยากรณ์ใหม่ของES6
            this.message = message;
        }
        say(){
            let element = document.querySelector('#element1');
            element.innerHTML = this.message;
        }
    }

    let chat = new Chat("Hello, world!");    // let ไวยากรณ์ใหม่ของES6
    chat.say();

    // ตัวอย่างโค้ดES7 ชุดนี้ยังรันได้เฉพาะบน Google Chrome
    let array = ["A", "B", "C"];              // let ไวยากรณ์ใหม่ของES6
    console.log(array.includes("A"));          // true    -- เมธอดของอาร์เรย์ที่เพิ่มเข้ามาในES7
    `;

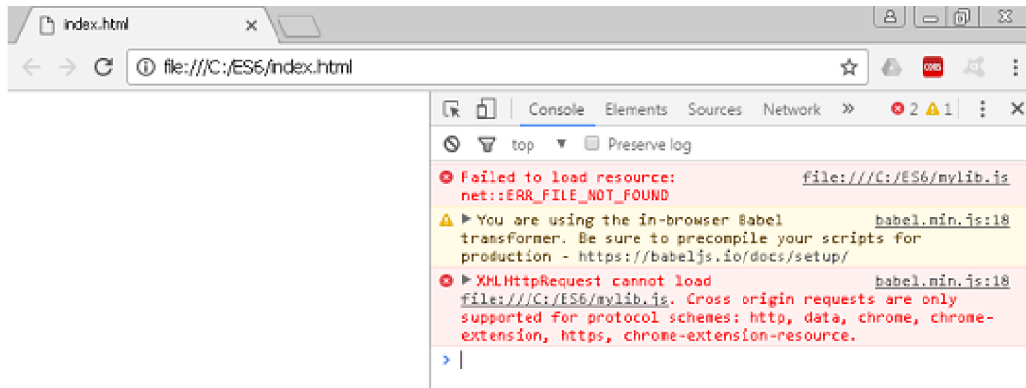
    var output = Babel.transform(input, { presets: ['es2015'] }).code; // คอมไพล์ES6 เป็นES5
    eval(output);              // ประมวลผล
</script>
</body>
</html>
```

ซึ่งผลการทำงานจะเหมือนกับตัวอย่างก่อนๆ ที่ยกมา

## Cross-origin resource sharing (CORS)

โดยปกติแล้วเว็บเพจ จะไม่สามารถแชร์ resources ข้าม domain กันได้ (เช่น ฟอนต์, ไฟล์จาวาสคริปต์ และรูปภาพ เป็นต้น) เพราะมันเป็นเรื่องของความปลอดภัย (same-origin policy)

คราวนี้ถ้าเขียน JavaScript แบบแยกไฟล์ .js แล้วอิมพอร์ตเข้ามา (จากตัวอย่างก่อนหน้านี้ ผมอิมพอร์ตไฟล์ mylib.js เข้ามา ด้วยวิธี Traceur หรือ Babel) เมื่อนำไปเปิดบน Google Chrome อาจทำงานไม่ได้ (ชวยแล้วไง!) เพราะเมื่อไปดูที่ console จะเห็นมันฟ้องเรื่อง Cross origin ดังรูป



แต่เราสามารถหลีกเลี่ยงกฎข้อนี้ได้ โดยใช้ Cross-origin resource sharing (CORS) ซึ่งเป็นกลไกอนุญาตให้ resources บนเว็บเพจ ถูกเข้าถึงจาก Domain อื่นได้

## วิธีการแก้ปัญหา

### วิธี1

สามารถทำได้ง่ายๆ เพียงแค่บอกให้เว็บเซิร์ฟเวอร์ เพิ่มค่าต่อไปนี้ลงไปใน HTTP Header ดังนี้ (วิธีกำหนดค่านี้ ต้องดูที่คู่มือของเซิร์ฟเวอร์แต่ละเจ้าเอเอง)

```
Access-Control-Allow-Origin: *
```

จริงๆ ทำแบบนี้ก็ดูไม่ปลอดภัยเท่าไร ทางที่ดีควรให้สิทธิเฉพาะ url เท่าที่จำเป็น ตัวอย่างเช่น

```
Access-Control-Allow-Origin: http://www.example.com http://test.example.com
```

(ที่มา <http://manit-tree.blogspot.com/2012/07/cross-origin-resource-sharing.html>)

## วิธีที่ 2

ถ้าเราไม่ได้เขียนเว็บ แล้วทดสอบ (Test) บนเว็บเซิร์ฟเวอร์ อาจมีทดสอบเว็บบนเครื่องตัวเองแบบ local ก็ต้องเปิด Google Chrome ด้วยท่าพิศดาร โดยปลดความปลอดภัยเรื่องนี้ออก เพื่อให้มันทำ CORS ได้

บนวินโดวส์ก็ให้ไปที่คอมมานด์ไลน์ แล้วพิมพ์คำสั่งตามนี้ เมื่อนั้น Google Chrome ก็จะมีเปิดขึ้นมา แล้วถึงเปิดไฟล์ HTML ตามที่หลัง

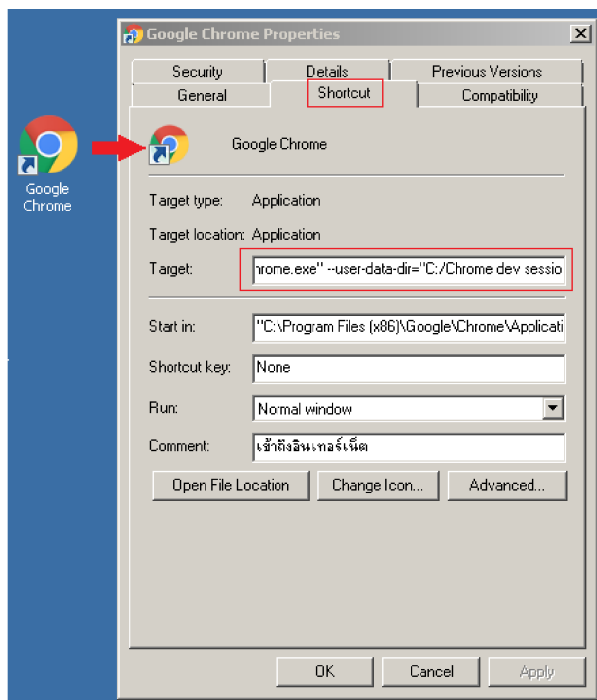
```
chrome.exe --user-data-dir="C:/Chrome dev session" --disable-web-security
```

หรือจะระบุชื่อไฟล์ HTML ให้เปิดขึ้นมาพร้อมกับ Google Chrome ก็ได้

```
chrome.exe --user-data-dir="C:/Chrome dev session" --disable-web-security  
"c:\ES6\index.html"
```

อีกวิธีหนึ่งง่ายดี ให้ไปที่ Shortcut ของ Google Chrome แล้วคลิกขวาเปิดมันขึ้นมา จากนั้นจึงเพิ่มค่าต่อไปนี้ตรงช่อง "Target:" หลังข้อความเดิม

```
--user-data-dir="C:/Chrome dev session" --disable-web-security
```



ต่อไปนี้ ก็ให้เปิดที่ Shortcut ของ Google Chrome ก่อนเสมอ แล้วหลังจากนั้น จึงเปิดไฟล์ HTML ตามที่หลัง

ส่วนบน OSX กับ Linux ผมไม่มีเครื่องลองครับ จึงไม่กล้าเขียน ลองดูเพิ่มเติมได้ที่

<http://stackoverflow.com/questions/3102819/disable-same-origin-policy-in-chrome>



### วิธีที่ 3

จากไฟล์ index.html ที่มีปัญหาเวลาเปิด Google Chrome แล้วไม่ทำงาน

```
C:\ES6>
|-- index.html
|-- mylib.js
```

ให้ลองใช้เซิร์ฟเวอร์จำลอง จาก Node.js แต่ก่อนอื่นจะให้ติดตั้งเซิร์ฟเวอร์ที่ว่า ก็คือ live-server ด้วยคำสั่ง npm ดังนี้

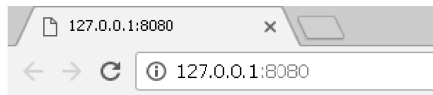
```
npm install -g live-server
```

จากนั้นก็ cd ไปที่ C:\ES6\ ต่อด้วยสั่งให้ live-server ทำการรัน index.html ด้วยคำสั่งง่ายๆ ดังนี้

```
C:\ES6>live-server
```

เมื่อนั้นเว็บเบราว์เซอร์ที่ถูกตั้งไว้เป็นดีฟอลต์ ก็จะด้งขึ้นมา และเปิดไฟล์ index.html อย่างอัตโนมัติ หรือถ้าเครื่องเรา

Google Chrome ไม่ได้ตั้งเป็นดีฟอลต์ ก็ให้กรอก url ตรงๆ เป็น <http://127.0.0.1:8080/> ตามรูป



**Hello, world!**

\*\* เสริมอีกหนึ่ง ถ้าใครใช้ Python ก็อาจใช้เซิร์ฟเวอร์จำลองได้ด้วยเช่นกัน อย่างกรณีผมใช้ Python 3 ก็จะมีพม์คำสั่งดังนี้

```
C:\ES6>python -m http.server 8080
```

จากนั้นก็เปิด Google Chrome ขึ้นมาโดยกรอก url เป็น <http://127.0.0.1:8080/>

## ฟีเจอร์ใหม่ที่เพิ่มเข้ามาใน ES7 (นิตเดี่ยวเอง)

### เพิ่มการใช้งานโอเปอเรเตอร์ยกกำลัง (Exponentiation Operator)

โอเปอเรเตอร์ยกกำลังจะใช้สัญลักษณ์เป็น **\*\*** (ดอกจันทั้งสองอันวางติดกัน) เพื่อแทนการคำนวณตัวเลขแบบยกกำลัง โดยไม่ต้องใช้เมธอด `Math.pow()` ซึ่งจะมีตัวอย่างการใช้งานดังนี้

```
let ans = 10 ** 2; // นำเลข 10 มายกกำลัง 2 ( 102 )
console.log(ans); // 100

// เปรียบเทียบใช้เมธอด Math.pow() ดังนี้
console.log(ans === Math.pow(10, 2)); // true
```

### ลำดับของโอเปอเรเตอร์ \*\*

โอเปอเรเตอร์ **\*\*** จะถือว่ามีความสำคัญสูงกว่าโอเปอเรเตอร์ทางคณิตศาสตร์ตัวอื่น ๆ

```
let ans = 3 * 10 ** 2;
console.log(ans); // 300
```

จากตัวอย่างเดิมจะเหมือนมีวงเล็บมาครอบนิพจน์  $(10 ** 2)$  ดังตัวอย่างซอร์สโค้ดข้างล่าง

```
let ans = 3 * (10 ** 2);
console.log(ans); // 300
```

### ข้อห้ามของโอเปอเรเตอร์ \*\*

โอเปอเรเตอร์ยกกำลังไม่สามารถใช้งานร่วมกับโอเปอเรเตอร์พวก unary expression เช่น `-` (เครื่องหมายลบ ไม่ใช่การลบ), `+` (เครื่องหมายบวก ไม่ใช่การบวก), `void`, `delete` และ `typeof` เป็นต้น โดยจะให้ดูตัวอย่างต่อไปนี้ประกอบ

```
let ans = -10 ** 2; // syntax error
```

ตัวอย่างที่ยกมานี้จะเกิด error เพราะตรรกนิพจน์  $-10 ** 2$  มันกำกวม เนื่องจากอาจหมายถึง

- $-(10 ** 2)$
- $(-10) ** 2$

จากตัวอย่างเดิม ถ้าลองนำวงเล็บมาครอบเพื่อกำหนดลำดับการทำงานเสียใหม่ ก็จะไม่เกิด error ดังตัวอย่างหน้าถัดไป

```
let ans = - (10 ** 2); // -100
```

จากตัวอย่างเดิมเช่นกัน ถ้าลองเปลี่ยนการครอบวงเล็บเสียใหม่ ก็จะได้ผลการทำงานที่แตกต่างกันดังนี้

```
let ans = (-10) ** 2; // 100
```

ขณะเดียวกันโอเปอเรเตอร์ยกกำลังก็มีข้อยกเว้น มันสามารถใช้ได้กับ **++** หรือ **--** (เป็น unary expression) โดยไม่ต้องใช้วงเล็บครอบ

ลองพิจารณาการใช้โอเปอเรเตอร์ยกกำลังร่วมกับโอเปอเรเตอร์ **++** ดังตัวอย่าง

```
let value1 = 9, value2 = 10;

// ใช้งานโอเปอเรเตอร์ ++ แบบ prefix
// ค่าของ value1 ถูกบวกด้วยหนึ่ง ก่อนที่จะยกกำลัง 2
console.log(++value1 ** 2); // 100
console.log(value1); // 10

// ใช้งานโอเปอเรเตอร์ ++ แบบ postfix
// หลังจากยกกำลัง 2 ไปแล้ว ค่าของ value2 จึงถูกบวกด้วยหนึ่งที่หลัง
console.log(value2++ ** 2); // 100
console.log(value2); // 11
```

ลองพิจารณาการใช้โอเปอเรเตอร์ยกกำลังร่วมกับโอเปอเรเตอร์ **--** ดังตัวอย่าง

```
let value1 = 11, value2 = 10;

// ใช้งานโอเปอเรเตอร์ -- แบบ prefix
// ค่า value1 ถูกลบด้วยหนึ่ง ก่อนที่จะยกกำลัง 2
console.log(--value1 ** 2); // 100
console.log(value1); // 10

// ใช้งานโอเปอเรเตอร์ -- แบบ postfix
// หลังจากยกกำลัง 2 ไปแล้ว ค่าของ value2 จึงถูกลบด้วยหนึ่งที่หลัง
console.log(value2-- ** 2); // 100
console.log(value2); // 9
```

## เพิ่มเมธอด Array.prototype.includes()

สำหรับ ES6 นั้น สตริงทุกตัวจะมีเมธอด includes() และเช่นเดียวกันใน ES7 ก็ได้เพิ่มเมธอดดังกล่าวให้กับอาร์เรย์ โดยมีจุดประสงค์ใช้ค้นหาข้อมูลในอาร์เรย์ ถ้าเจอข้อมูลที่ต้องการหาก็จะรีターンเป็น true ถ้าไม่เจอก็จะได้เป็น false ดังตัวอย่าง (ทำงานแบบเดียวกับ includes() ของสตริงบที่ 5 ในหนังสือ [1])

```
let array = ["A", "B", "C"]; // ประกาศอาร์เรย์

console.log(array.includes("A")); // true
console.log(array.includes("Z")); // false
```

ในตัวอย่างนี้จะค้นหาตัวอักษร "A" เจอในอาร์เรย์ แต่ไม่สามารถค้นหา "Z" พบ เพราะมันไม่มีอยู่ในอาร์เรย์

ปกติแล้วเมธอด includes() จะเริ่มค้นหาที่ตำแหน่งอินเด็กซ์เป็น 0 โดยดีฟอลต์ ดังนั้นถ้าจะเปลี่ยนตำแหน่งอินเด็กซ์ที่ใช้ค้นหา ก็สามารถทำได้ดังตัวอย่าง

```
let array = ["A", "B", "C"]; // ประกาศอาร์เรย์
// เริ่มค้นหา "B" จากอินเด็กซ์คือ 2 ซึ่งจะพบว่าไม่เจอ
console.log(array.includes("B", 2)); // false

// แต่ถ้าเปลี่ยนมาเริ่มค้นหาจากอินเด็กซ์เป็น 1 ก็จะหา "B" เจอ
console.log(array.includes("B", 1)); // true
```

ในตัวอย่างดังกล่าวจะเห็นว่าเมธอด includes รับค่าอาร์กิวเมนต์ตัวที่สอง เพื่อระบุตำแหน่งเริ่มต้นของอินเด็กซ์ที่จะใช้ค้นหาข้อมูลในอาร์เรย์

### ข้อควรระวัง includes()

เมธอด includes() จะเสมือนใช้โอเปอเรเตอร์ === เปรียบเทียบว่ามีสมาชิกที่ต้องค้นหาหรือไม่ แต่ทว่าเวลามันเห็นข้อมูลเป็น NaN ก็จะมีค่าเท่ากัน (เปรียบเทียบแล้วได้ true) ซึ่งจะแตกต่างจาก indexOf ซึ่งจะเสมือนใช้ === เช่นกัน ซึ่งเวลามันเห็น NaN จะมองว่ามีค่าต่างกัน (เปรียบเทียบแล้วได้ false) ดังตัวอย่าง

```
let array = [0, NaN, 1];

console.log(array.indexOf(NaN)); // -1 -- ไม่เจอสมาชิกที่ต้องการ
console.log(array.includes(NaN)); // true
```

แต่ถ้าข้อมูลเป็น +0 กับ -0 จะมองว่าเท่ากัน (เปรียบเทียบแล้วได้เป็น true) ทั้ง includes() กับ indexOf() ดังตัวอย่าง

```
let array = [-0, NaN, 1];

console.log(array.indexOf(+0)); // 0 -- เจอค่า -0 อยู่ในอาร์เรย์ที่ตำแหน่งอินเด็กซ์ 0
console.log(array.includes(+0)); // true
```

## TypedArray.prototype.includes()

ในอาร์เรย์ระดับบิต (TypedArray บทที่ 12 ของหนังสือ [1]) ก็จะมีเมธอด includes() ให้ใช้งานเหมือนกับอาร์เรย์ในหัวข้อก่อนหน้านี้ทุกประการเด๊ะ ดังตัวอย่าง

```
let uint8 = new Uint8Array([1, 2, 3, 4, 5]);
console.log(uint8.includes(1));           // true
console.log(uint8.includes(5));           // true
console.log(uint8.includes(10));          // false
```

## สิ่งที่เปลี่ยนแปลงไปของ ES7 เมื่อเทียบกับ ES6 (นิดเดียวเอง)

หัวข้อก่อนหน้านี้ได้กล่าวถึงฟีเจอร์ที่เพิ่มมาใหม่ใน ES7 แต่หัวข้อนี้จะกล่าวถึงฟีเจอร์ที่เปลี่ยนไปจาก ES6 ดังนี้

- 1) trap ที่เป็น enumerate() ของพร็อกซี (บทที่ 14 ของหนังสือ [1]) ถูกเอาออกไปใน ES7 เรียบร้อยแล้ว
- 2) เจอเนอเรเตอร์ (บทที่ 13 ของหนังสือ [1]) ไม่มี [[Construct]] ถ้าเรียก new จะเกิด error ขึ้นมาดังตัวอย่าง

```
function * generator() {}
let iterator = new generator(); // throws "TypeError: f is not a constructor"
```

## ฟีเจอร์ที่เพิ่มเข้ามาใน ES8 (ECMAScript 2017)

สำหรับฟีเจอร์ใหม่ ES8 หรือ ECMAScript 2017 [ที่เพิ่มเข้ามา](#) ตามแพลนที่ผมแอบไปส่องดู คาดว่าจะออกมาในกลางปี 2017 (มั้ง) ...ดังนั้นฟีเจอร์นี้ จึงเป็นสิ่งที่ยังไม่เกิดขึ้น เป็นแค่ฉบับร่าง ยังใช้งานไม่ได้นะครับ (ในช่วงที่พิมพ์เขียนอยู่ตอนนี้)

แต่เนื่องจากผมทะลึ่งอยากรู้ดี เลยไปสรุปและแปลมาจาก Dr. Axel Rauschmayer [5] ที่เขียนแนะนำไว้  
อย่างดี เลือกเอาเท่าที่สำคัญมาเขียน (ได้ผมก็ก๊อปปี้  
เกมมาอธิบายแล้วกัน บอกตามตรงเลย ไม่ได้คิดเองนะ)



**\*\*\* อนาคตจากนี้ เนื้อหาอาจเปลี่ยนแปลงได้ トラバิดที่มาตรฐานใหม่ยังไม่ถูกประกาศออกมาเต็มตัว**

### เมธอดใหม่ Object.entries() กับ Object.values()

ตอนนี้จะให้ดูเมธอด Object.entries() ก่อน ...ซึ่งจะมี signature ดังนี้

```
Object.entries(value : any) : Array<[string,any]>
```

เจ้าเมธอดนี้รับค่าออบเจกต์เป็นอินพุต แล้วรีเทิร์นผลลัพธ์ออกมาเป็นอาร์เรย์สองมิติ ...มิติแรกเก็บค่าคีย์ (เป็นสตริง)  
ส่วนมิติที่สองจะเก็บข้อมูล (อินพุตใน JavaScript จะมีโครงสร้างเป็นพร็อพเพอร์ตี้ ที่เขียนในรูปของ key กับ value)

➤ ดูตัวอย่างวิธีการใช้งาน

```
Object.entries({ one: 1, two: 2 })
```

มันจะรีเทิร์นอาร์เรย์ออกมา

```
[ [ 'one', 1 ], [ 'two', 2 ] ]
```

➤ แต่ถ้าคีย์ของอินพุตเป็น symbol ...ก็จะทำให้เมธอด Object.entries() เพิกเฉย ไม่สนใจโดยดี

```
Object.entries({ [Symbol()]: 123, foo: 'abc' });
```

มันจะรีเทิร์นอาร์เรย์ออกมา (ไม่มีสมาชิก [Symbol()]: 123)

```
[ [ 'foo', 'abc' ] ]
```

- ขณะเดียว `Object.entries()` ก็สามารนำไปใช้ในประโยคควบคู่ `for ...of` (ฟีเจอร์ใหม่ใน ES6)

```
let obj = { one: 1, two: 2 };
for (let [k,v] of Object.entries(obj)) {
  console.log(`${JSON.stringify(k)}: ${JSON.stringify(v)}`);
}
```

แสดงผลลัพธ์เป็น

```
"one": 1
"two": 2
```

- ลองพิจารณาตัวอย่างการประยุกต์ใช้งานเมธอด `Object.entries()` สร้างอ็อบเจ็กต์ `Map` (โดยไม่ต้องใช้อาร์เรย์สองมิติ)

```
let map = new Map(Object.entries({
  one: 1,
  two: 2,
}));
console.log(JSON.stringify([...map])); // [{"one",1},{"two",2}]
```

คราวนี้ลองมาดูเมธอด `Object.values()` กันบ้าง ...ซึ่งจะมี signature ดังนี้

```
Object.values(value : any) : Array<any>
```

เจ้าเมธอดนี้จะรับค่าอากิวเมนต์เป็นอ็อบเจ็กต์ แล้ววีเทิร์นผลลัพธ์ออกมาเป็นอาร์เรย์ ที่เก็บเฉพาะส่วนข้อมูลของอ็อบเจ็กต์เท่านั้น

- ลองดูตัวอย่างวิธีการใช้งาน

```
Object.values({ one: 1, two: 2 })
```

มันจะวีเทิร์นอาร์เรย์ออกมา

```
[ 1, 2 ]
```

## เมธอดใหม่ padStart กับ padEnd

ตอนนี้จะให้ดูเมธอด padStart() ก่อน ... ซึ่งจะประกาศเป็นโพรโตไทป์ที่ String ดังนี้

```
String.prototype.padStart(maxLength, fillString='')
```

เมธอดนี้เอาไว้เติมข้อความ (fillString) นำหน้าข้อความเดิม ตามความยาวที่ระบุไว้ (maxLength)

### ➤ ลองดูตัวอย่างการใช้งาน

```
'x'.padStart(5, 'ab')
```

สตริง 'x' จะเปลี่ยนไป เพราะเมธอดได้เติมคำว่า 'ab' หน้า 'x' จนข้อความมันยาว 5 ตัวอักษรพอดี ดังนี้

```
'ababx'
```

### ➤ จากตัวอย่างเดิม ถ้าระบุความยาวเป็น 4 ก็จะทำให้ข้อความ 'ab' ที่อยู่หน้า ถูกตัดออกไปบางส่วน

```
'x'.padStart(4, 'ab')
```

สตริง 'x' จะเปลี่ยนไปดังนี้

```
'abax'
```

### ➤ ลองพิจารณาอีกตัวอย่าง (fillString มีความยาวข้อความเท่ากับ maxLength)

```
'abc'.padStart(10, '0123456789')
```

สตริง 'x' จะเปลี่ยนไปดังนี้

```
'0123456abc'
```

### ➤ ถ้าสตริงมันยาวกว่าค่า maxLength ก็จะไม่เกิดขึ้น

```
'abcd'.padStart(2, '#')
```

ได้สตริงตามเดิม

```
'abcd'
```



- ถ้าไม่มีค่า fillString ...เมธอด padStart() ก็จะเติมช่องว่างนำหน้าแทน

```
'x'.padStart(3)
```

สตริง 'x' จะเปลี่ยนไปดังนี้ (มีช่องว่างนำหน้า 2 ตัว แต่เมื่อนับตัวอักษร x ก็จะมีทั้งสิ้น 3 ตัวอักษร)

```
'  x'
```

คราวนี้ลองเปลี่ยนมาดูเมธอด padEnd() ซึ่งประกาศเป็นโปรโตไทป์ที่ String บ้าง

```
String.prototype.padEnd(maxLength, fillString='')
```

เมธอดนี้จะคล้าย padStart() แต่เปลี่ยนมาเติม fillString ที่ท้ายข้อความตามความยาวที่ระบุ maxLength

- ลองพิจารณาตัวอย่างการใช้งาน

```
'x'.padEnd(5, 'ab')      // 'xabab'
'x'.padEnd(4, 'ab')      // 'xaba'
'abc'.padEnd(10, '0123456789') // 'abc0123456'
'abcd'.padEnd(2, '#')    // 'abcd'
'x'.padEnd(3)            // 'x  '
```

**ประโยชน์ของเมธอด padStart() กับ padEnd()** ก็เช่น สามารถเติมจุดทศนิยมต่อท้ายข้อความ หรือเติมคำก่อนหน้าข้อความเดิม หรือจัดข้อความให้แลดูสวยงาม ฯลฯ

## Object.getOwnPropertyDescriptors()

เมธอดใหม่นี้จะรีเทิร์น **Property descriptors** ของพร็อพเพอร์ตี้ในอ็อบเจกต์ (ตัวอธิบายคุณสมบัติพร็อพเพอร์ตี้) ลองดูตัวอย่างการใช้งาน

```
const obj = {
  [Symbol('foo')]: 123,
  get bar() { return 'abc' },
};
console.log(Object.getOwnPropertyDescriptors(obj));
```

ผลลัพธ์ที่เมธอดนี้ทำงาน จะรีเทิร์นอ็อบเจกต์ที่มีคีย์ (ชื่อ Property descriptors) กับส่วนข้อมูล ดังนี้

```
{ [Symbol('foo')]:
  { value: 123,
    writable: true,
    enumerable: true,
    configurable: true },
  bar:
    { get: [Function: bar],
      set: undefined,
      enumerable: true,
      configurable: true } }
```

เนื่องจากเมธอด `Object.assign()` ใน ES6 จะไม่ก๊อปปี้พร็อพเพอร์ตี้บางตัว (พวก non-default attributes ได้แก่ getters, setters, non-writable properties และอื่นๆ)

➤ ลองพิจารณาตัวอย่าง เมธอด `foo` ที่อยู่ในอ็อบเจกต์ `source` ซึ่งกำหนดให้เป็น `set (setter)`

```
const source = {
  set foo(value) {
    console.log(value);
  }
};
console.log(Object.getOwnPropertyDescriptor(source, 'foo'));
```

ผลลัพธ์ต่อไปนี้จะพบว่า `Object.getOwnPropertyDescriptor()` สามารถมองเห็นเมธอด `foo` ที่เป็น `setter` ได้

```
{ get: undefined,
  set: [Function: foo],
  enumerable: true,
  configurable: true }
```

มองเห็น foo

- แต่ถ้าเราใช้งานเมธอด `Object.assign()` จะไม่สามารถก็อปปี้อ็อบเจกต์ที่เป็น `set foo(value){...}` ได้เลย

```
const target1 = {};  
Object.assign(target1, source);  
console.log(Object.getOwnPropertyDescriptor(target1, 'foo'));
```

เมื่อใช้ `Object.getOwnPropertyDescriptor()` แสดงผลลัพธ์ ก็จะเห็นว่าอ็อบเจกต์ `target1` ที่สร้างจาก `Object.assign()` จะไม่มีเมธอด `foo` ที่เป็น `setter` เลย

```
{ value: undefined,  
  writable: true,  
  enumerable: true,  
  configurable: true }
```

- แต่ถ้าใช้ `Object.getOwnPropertyDescriptor()` ประยุกต์ร่วมกับ `Object.defineProperties()` เพื่อแก้ปัญหาจากการใช้เมธอด `Object.assign()` ก็จะได้โค้ดหน้าตาประมาณนี้

```
const target2 = {};  
Object.defineProperties(target2, Object.getOwnPropertyDescriptors(source));  
console.log(Object.getOwnPropertyDescriptor(target2, 'foo'));
```

เมื่อแสดงผลลัพธ์ก็จะเห็นเมธอด `foo` ที่เป็น `setter` อยู่ในอ็อบเจกต์ `target2`

```
{ get: undefined,  
  set: [Function: foo],  
  enumerable: true,  
  configurable: true }
```

มองเห็น foo

ขณะเดียวกันเราก็สามารถนำ `Object.getOwnPropertyDescriptor()` และ `Object.getPrototypeOf()` มาใช้กับ `Object.create()` เพื่อสร้างอ็อบเจกต์ใหม่ขึ้นมา (เหมือนกับการโคลนนิ่ง)

```
const clone = Object.create(Object.getPrototypeOf(obj),  
Object.getOwnPropertyDescriptors(obj));
```

ต่อไปจะให้ลองพิจารณาการสร้างอ็อบเจกต์ ด้วยการกำหนดโปรโตไทป์ (Prototype) ในหลากหลายรูปแบบ (ในตัวอย่างจะให้ prot คืออ็อบเจกต์ที่ใช้เป็นโปรโตไทป์)

วิธีที่ 1 ใช้พร็อพเพอร์ตี้ `__proto__`

```
const obj = {  
  __proto__: prot,  
  foo: 123,  
};
```

วิธีที่ 2 ใช้เมธอด `Object.create()`

```
const obj = Object.create(prot);  
obj.foo = 123;
```

วิธีที่ 3 จะนำเมธอด `Object.assign()` มาใช้ร่วมกับ `Object.create()`

```
const obj = Object.assign(  
  Object.create(prot),  
  {  
    foo: 123,  
  }  
);
```

วิธีที่ 4 จะนำเมธอด `Object.getOwnPropertyDescriptors()` มาใช้งานร่วมกับ `Object.create()`

```
const obj = Object.create(  
  prot,  
  Object.getOwnPropertyDescriptors({  
    foo: 123,  
  })  
);
```

## การใช้คอมม่า (,)

การใช้คอมม่า (,) ต่อท้ายในพารามิเตอร์ของฟังก์ชัน และตอนเรียกใช้งานฟังก์ชัน

มาตรฐาน ES8 สามารถมีคอมม่าต่อจากพารามิเตอร์ของฟังก์ชัน ในตำแหน่งท้ายสุด ดังตัวอย่าง

```
function foo(  
  param1,  
  param2,  
) {}
```

หรือจะใช้คอมม่าต่อท้ายอาทิวนต์ตัวท้ายสุด เมื่อเรียกใช้งานฟังก์ชัน ดังตัวอย่าง

```
foo(  
  'abc',  
  'def',  
);
```

การใช้คอมม่าในอ็อบเจกต์ และอาร์เรย์

ใน ES8 สามารถใช้คอมม่าต่อท้ายสมาชิกตัวสุดท้าย ตอนประกาศสมาชิกของอ็อบเจกต์ ดังตัวอย่าง

```
let obj = {  
  first: 'Jane',  
  last: 'Doe',  
};
```

สามารถใช้คอมม่าต่อท้ายสมาชิกตัวสุดท้าย ตอนประกาศอาร์เรย์ก็ได้ ดังตัวอย่าง

```
let arr = [  
  'red',  
  'green',  
  'blue',  
];
```

## Async functions

ฟังก์ชันแบบ async ถือว่าเป็นฟีเจอร์ที่โดดเด่นในเรื่องอำนวยความสะดวก เวลาเขียนโค้ดในลักษณะ Asynchronous ...

นอกเหนือจากการใช้งาน Promise (สำหรับเรื่อง Promise และการทำงานแบบ asynchronous ผมเขียนไว้ในหนังสือ [1] อย่างละเอียด)


เราสามารถประกาศฟังก์ชันแบบ async ได้หลากหลายแบบ ด้วยการใช้คีย์เวิร์ด **async** นำหน้าฟังก์ชันดังนี้

ประกาศฟังก์ชันแบบปกติ	<code>async function foo() {}</code>
ประกาศนิพจน์ฟังก์ชัน (function expressions)	<code>const foo = async function () {};</code>
ประกาศเมธอดของอ็อบเจกต์	<code>let obj = { async foo() {} }</code>
ประกาศฟังก์ชันลูกศร (Arrow function)	<code>const foo = async () =&gt; {};</code>

## ฟังก์ชันแบบ async จะรีเทิร์นเป็นพรอมิส

- ตัวอย่างต่อไปนี้ เมื่อฟังก์ชันแบบ async เวลามันทำงานเสร็จแล้ว จะรีเทิร์นพรอมิสที่มีสถานะ Fulfilled

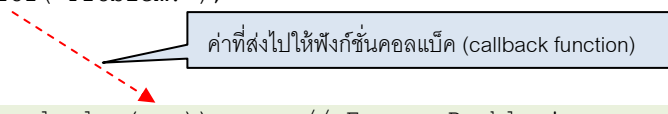
```
async function asyncFunc() {  
  return 123;  
}  
  
asyncFunc()  
  .then(x => console.log(x));           // 123
```



A red dashed arrow points from the value `123` returned by the `asyncFunc()` function to the argument `x` in the `.then(x => console.log(x));` callback function. A callout box points to this arrow with the text: "ค่าที่ส่งไปให้ฟังก์ชันคอลแบ็ค (callback function)".

- ตัวอย่างเมื่อฟังก์ชันแบบ async เกิด error ขึ้นมา จะรีเทิร์นพรอมิสที่มีสถานะ Rejected

```
async function asyncFunc() {  
  throw new Error('Problem!');  
}  
  
asyncFunc()  
  .catch(err => console.log(err));     // Error: Problem!
```



A red dashed arrow points from the `throw new Error('Problem!');` statement inside the `asyncFunc()` function to the argument `err` in the `.catch(err => console.log(err));` callback function. A callout box points to this arrow with the text: "ค่าที่ส่งไปให้ฟังก์ชันคอลแบ็ค (callback function)".

## await

คีย์เวิร์ด **await** จะใช้เป็นโอเปอเรเตอร์ อยู่ภายในฟังก์ชันแบบ **async** โดยมันจะวางไว้หน้าพรอมิส เพื่อหยุดรอให้พรอมิสทำงานแบบ **asynchronous** ให้เสร็จเสียก่อน

- ถ้าพรอมิสทำงานเสร็จ และมีสถานะเป็น **fulfilled** ก็จะทำให้ **await** เลิกการรอคอย (พร้อมส่งค่าไปให้ `then(callback(resolve))`) ... จากนั้นจึงขยับไปทำบรรทัดถัดไป
- แต่ถ้าพรอมิสเป็น **rejected** แล้วละก็ ... ตัว **await** ก็จะโยน **error** ออกมา

```
async function asyncFunc() {  
  const result = await otherAsyncFunc();  
  console.log(result);  
}
```

ตัวอย่างนี้ `otherAsyncFunc()` รีเทิร์นพรอมิสที่มีสถานะ **fulfilled**

รอทำงานต่อจากพรอมิส (ที่ทำงานเสร็จแล้ว)  
โดยเอาค่า `result` ที่พรอมิสส่งออกมา แสดงผลออกทางหน้าคอนโซล  
(`result` ไม่ใช่ผลลัพธ์จากฟังก์ชัน `otherAsyncFunc()` นะ)

โค้ดข้างต้น จริงๆ แล้วจะเหมือนเขียนแบบนี้

```
function asyncFunc() {  
  return otherAsyncFunc().  
    .then(result => {  
      console.log(result);  
    });  
}
```

- ตัวอย่างใช้ **await** จัดการเมื่อเกิด **error**

`otherAsyncFunc()` รีเทิร์นพรอมิสที่ทำงานไม่สำเร็จ (สถานะ **rejected**)

```
async function asyncFunc() {  
  try {  
    const result = await otherAsyncFunc();  
  } catch (err) {  
    console.error(err);  
  }  
}
```

จะเหมือนเขียนโค้ดดังนี้

```
function asyncFunc() {  
  return otherAsyncFunc().  
    .catch(err => {  
      console.error(err);  
    });  
}
```

- ลองพิจารณาการเขียน await แบบเรียงต่อกัน ภายในฟังก์ชันแบบ async

```
async function asyncFunc() {  
  const result1 = await otherAsyncFunc1();  
  console.log(result1);  
  const result2 = await otherAsyncFunc2();  
  console.log(result2);  
}
```

รอพรอมิสทำงานก่อน

รอพรอมิสทำงานก่อน

จะเสมือนเขียนโค้ดดังนี้

```
function asyncFunc() {  
  return otherAsyncFunc1().  
    .then(result1 => {  
      console.log(result1);  
      return otherAsyncFunc2();  
    })  
    .then(result2 => {  
      console.log(result2);  
    });  
}
```

ทำงานก่อน

ทำงานทีหลัง

- ในตัวอย่างข้างบนฟังก์ชัน otherAsyncFunc1() จะทำก่อน otherAsyncFunc2() ซึ่งทั้งคู่ทำงานเป็น asynchronous เหมือนกัน แต่ทั้งนี้เราสามารถให้ทั้งสอง ทำงานพร้อมกันเป็นคู่ขนานได้ ด้วยการประยุกต์ใช้ Promise.all() ดังตัวอย่าง

```
async function asyncFunc() {  
  const [result1, result2] = await Promise.all([  
    otherAsyncFunc1(),  
    otherAsyncFunc2(),  
  ]);  
  console.log(result1, result2);  
}
```

ทำงานคู่ขนานกัน

จะเสมือนเขียนโค้ดดังนี้

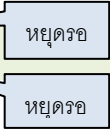
```
function asyncFunc() {  
  return Promise.all([  
    otherAsyncFunc1(),  
    otherAsyncFunc2(),  
  ])  
  .then([result1, result2] => {  
    console.log(result1, result2);  
  });  
}
```



สำหรับการทำงานของ async คล้ายๆ กับหลักการเรื่อง generator ใน ES6 มาก (เรื่อง generator ผมเขียนไว้อย่างละเอียดที่หนังสือ [1] เช่นเคย)

คราวนี้จะให้ลองดูตัวอย่างการใช้ Fetch API ซึ่งเป็น API ตัวใหม่ คล้ายๆ กับ XMLHttpRequest ...ที่เราสามารถเรียกไปยังเซิร์ฟเวอร์แบบ asynchronous ได้นั่นเอง (ในตัวอย่างด้านล่าง fetch() กับ request.text() จะรีเทิร์นพรอมิส)

```
const fetchJson = co.wrap(function* (url) { // generator function
  try {
    let request = yield fetch(url);
    let text = yield request.text();
    return JSON.parse(text);
  }
  catch (error) {
    console.log(`ERROR: ${error.stack}`);
  }
});
```



จากตัวอย่างข้างต้น ได้ใช้คีย์เวิร์ด yield ภายในฟังก์ชันเจเนอเรเตอร์ (generator function) เพื่อหยุดรอการทำงาน ซึ่งมันจะคล้ายๆ กับการเขียนโค้ดฟังก์ชันแบบ async ประมาณตัวอย่างนี้ (แต่ก็ไม่เหมือนกันซะทีเดียวนะ)

```
async function fetchJson(url) {
  try {
    let request = await fetch(url);
    let text = await request.text();
    return JSON.parse(text);
  }
  catch (error) {
    console.log(`ERROR: ${error.stack}`);
  }
}
```

ต้องทำความเข้าใจเกี่ยวกับฟังก์ชันแบบ async กันก่อน ชักนิตหนึ่ง (ที่ผ่านมา ได้กล่าวถึง await พอควรแล้ว)

- ฟังก์ชันแบบ async จะเริ่มต้นทำงาน synchronous
- แกมมันยังสร้างพรอมิสขึ้นมาด้วย (สมมติชื่อ p แล้วกันเนอะ)
- พอฟังก์ชันนี้จบการทำงาน ก็จะรีเทิร์นพรอมิส p ที่ว่าออกมา
- เวลาฟังก์ชันจบการทำงาน เพราะไปเจอะอะไรประโยค return หรือ throw (หรือเจอ await ตัวสุดท้ายก็ได้)
- แต่ทั้งนี้ ถ้าฟังก์ชัน async จบด้วยประโยค `return x` ...ค่า `x` ก็จะถูกส่งไปให้ `p.then(callback(x))`
- แต่ถ้าจบโดยเกิด `throw error` ...ค่า `error` ก็จะถูกส่งไปให้ `p.then(..., callback(error))` หรือ `p.catch(callback(error))`

ลองพิจารณาตัวอย่าง

```
async function asyncFunc() {
  console.log('asyncFunc()'); // (A)
  return 'abc';
}

asyncFunc().
then(x => console.log(`Resolved: ${x}`)); // (B)
console.log('main'); // (C)
```

ฟังก์ชันรีเทิร์นพรอมิสออกมา ส่วนค่า 'abc' จะถูกนำไปใช้  
มอนิเตอร์ในเมธอด then() ต่อไป

มอนิเตอร์เมื่อพรอมิสทำงานสำเร็จ พร้อมรับ  
ค่า 'abc' มาแสดงทีหลัง

แสดงผลเป็น

```
asyncFunc()
main
Resolved: abc
```

บรรทัด C ทำงานก่อน บรรทัด B

จากตัวอย่างข้างต้น จะเห็นว่าฟังก์ชันแบบ async เมื่อจบด้วยประโยค `return 'abc'` หลังคำว่า `return` จะตามด้วยข้อมูล ...ซึ่งจะทำให้ `asyncFunc()` รีเทิร์นเป็นพรอมิส โดยจะสมมติว่าชื่อพรอมิส `p` ...ส่วนค่า 'abc' จะส่งไปให้พารามิเตอร์ของ `p.then(callback(x))`

แต่ถ้าประโยคท้ายสุดของ `asyncFunc()` เป็น `return innerPromise` โดยที่ `innerPromise` ก็คือพรอมิส ในกรณีนี้ `asyncFunc()` จะรีเทิร์น `innerPromise` ออกมา ไม่ใช่พรอมิส `p` แบบก่อนหน้านี้แล้ว ...หรืออาจมองว่าฟังก์ชัน `asyncFunc()` ไม่ได้ห่อพรอมิส แต่จะรีเทิร์นพรอมิสที่มีอยู่แล้วในฟังก์ชันออกมาตรงๆ ก็ได้ ดังตัวอย่าง

```
async function asyncFunc() {
  return Promise.resolve(123);
}

asyncFunc().
.then(x => console.log(x)) // 123
```

ส่งพรอมิสที่เป็น Fulfilled ออกมาจาก  
ฟังก์ชัน asyncFunc()

ลองดูอีกตัวอย่าง

```
async function asyncFunc() {  
  return Promise.reject(new Error('Problem!'));  
}  
  
asyncFunc()  
  .catch(err => console.error(err)); // Error: Problem!
```

ส่งพรอมิสที่เป็น Rejected ออกมาจาก  
ฟังก์ชัน asyncFunc()

สองตัวอย่างก่อนหน้านี้ ถ้ามอง Promise.resolve() กับ Promise.reject() คือฟังก์ชัน anotherAsyncFunc() ที่รีเทิร์นพรอมิส  
สออกมา ก็จะเขียนโค้ดได้ดังนี้

```
async function asyncFunc() {  
  return anotherAsyncFunc();  
}
```

จากตัวอย่างข้างต้น ถ้าลองเขียนใหม่ให้มี await นำหน้า ...ก็จะทำให้ฟังก์ชัน asyncFunc() รีเทิร์นพรอมิส p ที่ตัวเองสร้าง  
ขึ้นมามาดังตัวอย่าง (อาจมองว่า ฟังก์ชันไม่ได้ห่อพรอมิส)

```
async function asyncFunc() {  
  return await anotherAsyncFunc();  
}
```

สุดท้ายเรื่องฟังก์ชันแบบ async ขอบข่ายแค่นี้ดีกว่า เพราะตอนนี้ผมยังไม่มีโอกาสได้ทดลอง ใช้งานจริงๆ เท่าไร มันยังเป็นแค่  
พีเจอรือนาคเท่านั้น จึงไม่อยากเขียนอะไรไปเยอะกว่านี้ เดี่ยวหน้าแตกขึ้นมา

อีกอย่างถ้าใครอ่านหัวข้อนี้แล้วมีงง ...พรอมิส (Promise) คืออะไร? ...แล้ว asynchronous คืออะไร ...มีเขียนอะไรปะ  
อ่านไม่รู้เรื่องเลย ...เค้าก็ขอโทษที่นำ ไม่ได้เกริ่นนำอะไรมาเลย มาถึงก็เอาแต่อธิบาย

เขาเป็นว่า ถ้าใครอยากรู้เรื่องที่กำลังกล่าวมา ก็ให้ไปอ่านหนังสือที่ผมเขียนได้เลย [1] ...โฆษณาขายของสักหน่อย

ถ้าไม่ได้ซื้อหนังสือผมก็ไม่เป็นไร (แต่แอบงอนนิดหนึ่ง ฮือๆ) ...ท่านสามารถอ่านเนื้อหา JavaScript ที่ผมเขียนสรุปให้  
ตั้งแต่ ES5, ES6, ES7 ตามลิงค์ข้างล่าง (เท่าที่จะมีเวลาพอเขียนให้ได้ ยังไม่เสร็จดี)

- <https://github.com/adminho/javascript/>

## อ้างอิง

- [1] หนังสือ “พัฒนาเว็บแอปพลิเคชันด้วย JavaScript” จะอธิบายถึงมาตรฐานตัวใหม่ **ECMAScript 2015** หรือเรียกสั้น ๆ ว่า **“ES6”** หรือ **“ES6 Harmony”** โดยเล่มนี้ตีพิมพ์และจัดจำหน่ายโดยซีเอ็ด



- [2] [https://developer.mozilla.org/en-US/docs/Web/JavaScript/New\\_in\\_JavaScript/ECMAScript\\_Next\\_support\\_in\\_Mozilla](https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_Next_support_in_Mozilla)
- [3] <https://github.com/nzakas/understandings6/blob/master/manuscript/B-ECMAScript-7.md>
- [4] <https://tc39.github.io/ecma262/2016/>
- [5] <http://exploringjs.com/es2016-es2017/>

## ก่อนจากกัน

เท่าที่ผมรื้อมาให้อู๋ จะเห็นว่าเจ้า ES7 มันเพิ่มมานิดเดียวเอง (ไม่กระทบอะไรเลย) ส่วน ES8 ก็เพิ่มนิดเดียวเช่นกัน แต่ที่ไฮไลต์เป็นพระเอก ก็คงเป็นเรื่องฟังก์ชันแบบ async ...แต่ที่ว่ามันเป็นเรื่องอนาคตข้างหน้า ที่ต้องรอต่อไป ที่สำคัญถึงจะประกาศ ES7, ES8 ออกมาแล้ว แต่ใช้ว่าเว็บเบราว์เซอร์ หรือ JavaScript Engine ทุกเจ้าจะรองรับได้ทันทีซะเมื่อไร กว่าจะนำไปใช้งานได้จริงก็คงอีกนานทีเดียว ...ดังนั้นถ้าคิดจะเรียนรู้ JavaScript ยุคใหม่ ก็ต้องเริ่มจาก ES6 เป็นพื้นฐานสำคัญก่อน (เพราะเปลี่ยนใหญ่สุดแล้ว เมื่อเปรียบเทียบกับ ES5)

PDF ตัวนี้ จะอยู่ที่นั่นครับ <http://www.patanasongsivilai.com/javascript.html> (ถ้าไปดาวโหลดมาจากที่อื่น อาจไม่ได้เวอร์ชันใหม่ล่าสุด) สนใจอยากติดตามเพจเกี่ยวกับไอที ก็กด Like ได้ที่ <https://www.facebook.com/programmerthai/>

เขียนโดย

แอดมินโฮ ोन้อยออก

(จตุรพัชร พัฒนทรงศิริไธ)