

พื้นฐาน ภาษา Typescript

ภาษาทางอินเทอร์เน็ตใหม่ ที่เรียกว่า Typescript (TS) นี้ กำเนิดขึ้นจาก คณะทำงานของบริษัทไมโครซอฟท์ ที่ต้องการใช้ทดแทนภาษา JavaScript ที่พวกเขาและเราพบว่า เป็นภาษาที่สร้างปัญหาความผิดพลาดในการทำงานได้ง่ายๆ เหตุผลหลักคือ ความไม่มีมาตรฐาน ที่พัฒนาขึ้นมาหลายรุ่น และตรวจหาความผิดพลาดขณะเขียนโปรแกรมได้ยาก จะมาทราบความผิดพลาดอีกทีเมื่อ ทดลองให้โปรแกรมทำงานไปแล้ว

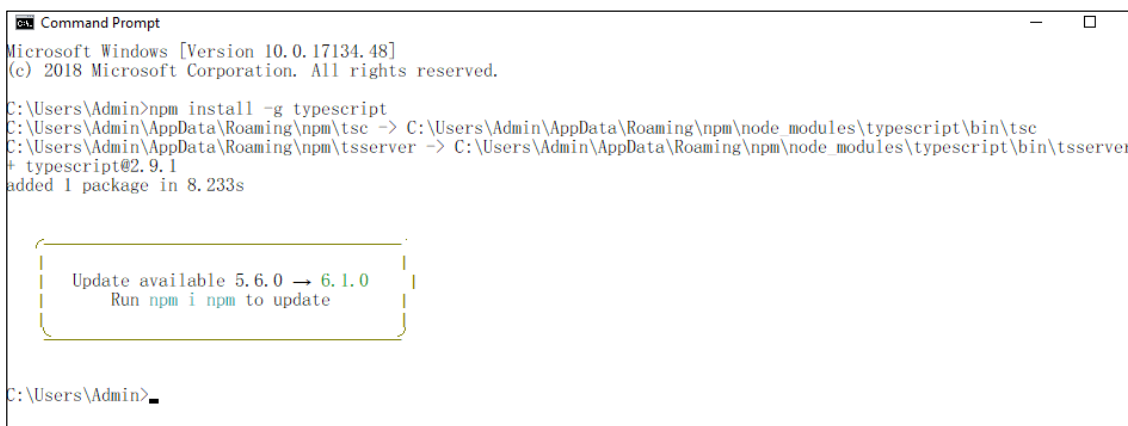
การใช้งานภาษา TS นี้ จะถูกแปลความเป็นภาษา JavaScript (JS) เพื่อให้เบราว์เซอร์ทำงานได้ ดังนั้นแล้ว TS จึงมีความเข้ากันได้กับ JS เป็นอย่างดี การสร้างภาษา TS ให้มีการตรวจสอบความผิดพลาดได้ขณะเขียนโปรแกรม จำเป็นที่จะต้องให้ภาษาตรวจสอบชนิดข้อมูลได้ (Strong type) และมีความเป็นออบเจกต์อย่างสมบูรณ์ และที่สำคัญ TS อยู่ในประเภท โอเพนซอร์ส (Apache license) ทำงานได้กับระบบปฏิบัติการหลักๆ

ติดตั้ง TS

TS ติดตั้งผ่าน Node Package Manager (npm) ซึ่งหมายความว่า จะต้องติดตั้ง NodeJs มาก่อน จึงจะมี npm ได้ โดยติดตั้ง NodeJs จากเว็บอย่างเป็นทางการที่ <https://nodejs.org>

เมื่อติดตั้ง NodeJs แล้ว ให้ติดตั้ง TS ผ่าน npm โดยการพิมพ์ผ่าน Command Line Interface (CLI) โดยเพิ่ม -g เพื่อติดตั้งในระดับ global ดังนี้

```
> npm install -g typescript
```



```
Command Prompt
Microsoft Windows [Version 10.0.17134.48]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Admin>npm install -g typescript
C:\Users\Admin\AppData\Roaming\npm\tsc -> C:\Users\Admin\AppData\Roaming\npm\node_modules\typescript\bin\tsc
C:\Users\Admin\AppData\Roaming\npm\tsserver -> C:\Users\Admin\AppData\Roaming\npm\node_modules\typescript\bin\tsserver
+ typescript@2.9.1
added 1 package in 8.233s

Update available 5.6.0 -> 6.1.0
Run npm i npm to update

C:\Users\Admin>
```

รูป 1 . การติดตั้ง TS ผ่าน npm

หลังจากติดตั้งเสร็จแล้ว ให้ทดสอบรุ่นที่ได้ติดตั้ง และทดสอบการทำงานของ TS ด้วย การสร้างไฟล์ ชื่อ hello.ts ดังเขียนเป็นภาษา TS ได้ว่า

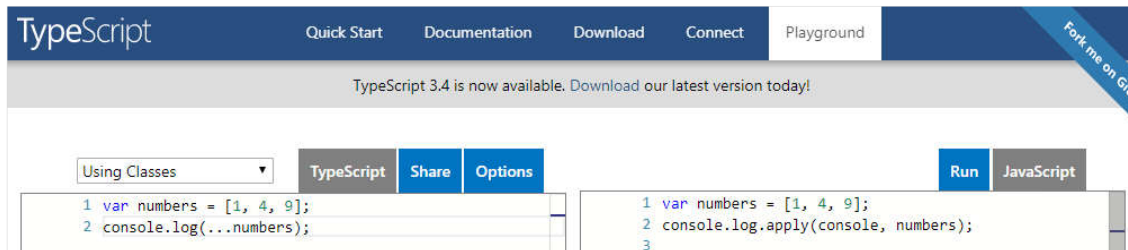
```
> tsc --version
> var name : string = 'Pol';
```

ให้บันทึกไฟล์ที่ไดเรกทอรีใดที่หนึ่ง ที่เช่น D:\TS\hello.ts ต่อมาทดสอบการทำงานด้วยคำสั่งต่อไปนี้ โดยไปยัง ไดเรกทอรีนี้ ให้เปิด CLI แล้วพิมพ์:

```
> tsc hello.ts
```

หากไม่มีอะไรผิดพลาด TS จะคอมไพล์ไปเป็นภาษา JavaScript ได้เป็นไฟล์ hello.js ซึ่งอยู่ในไดเรกทอรีเดียวกับไฟล์ hello.ts

นอกจากติดตั้ง TS บนเครื่องแล้ว ยังมีอีกวิธีที่ไม่ต้องติดตั้งบนเครื่องคือ ให้ทดสอบ TS ผ่านเว็บไซต์ <https://www.typescriptlang.org/play/>



รูป 2 . TS Playground

โปรแกรมแรก

จากไฟล์ ที่คอมไพล์ผ่านมาได้ ในชื่อ hello.js ก่อนหน้านี้นี้มีตัวแปรเพียงตัวเดียวชื่อ name ตัวแปรนี้จะนำมาอ้างอิงผ่านไฟล์ html ได้ โดยการนำเข้ามาผ่าน อีลิเมนต์ <script> ดังตัวอย่างต่อไปนี้

Code 1. index.html

```
<script src='hello.js'></script>
<script>
    document.write("Hello, "+ name);
</script>
```

เมื่อบันทึกไฟล์นี้ในชื่อ index.html แล้วให้เปิดทดสอบผ่านเบราว์เซอร์ (เช่น Google Chrome) จะปรากฏข้อความทักทาย ตามตัวแปรที่เขียนไว้ในไฟล์ hello.ts

ไทป์ (Type)

ความเป็นไทป์ หรือชนิดข้อมูลเป็นคุณสมบัติสำคัญของ TS ในขณะที่ JS ไม่มีการกำหนดไทป์ขณะเขียนโปรแกรม ถึงแม้จะมี ไทป์ก็ตาม แต่อย่างไรก็ตาม TS สามารถกำหนดไทป์ ได้แบบได้เดียวกับ JS เมื่อยังไม่ทราบไทป์แน่นอนขณะประกาศตัวแปรที่ยังไม่ใช้งาน

ไทป์พื้นฐาน คือ boolean, number, string, Array, void, และ enum ไทป์เหล่านี้เป็นไทป์ย่อยของ ไทป์สูงสุดคือ any ให้สังเกตว่า มีเพียงไทป์ Array ที่ใช้อักษรพิมพ์ใหญ่นำหน้า

ตาราง 1 ไทป์พื้นฐาน

ไทป์	ความหมาย	ตัวอย่าง
boolean	ค่าจริง (true) หรือเท็จ (false)	var isOK: Boolean = false;
number	ค่าตัวเลข ซึ่งหมายถึงตัวเลขทศนิยม (float)	var width: number = 6;
string	ตัวอักษร จะใช้ " หรือ ' คล่อมอักษร ก็ได้	var name: string = "pol ";

Array	อาร์เรย์ ที่บอกชนิดไทป์ หรือประกาศให้มีไทป์เหมือนกัน	<pre>var list: Array<number> = [1,2,3]; var items: number[] = [1,2,3];</pre>
enum	เป็นไทป์ที่กำหนดค่าได้เอง มักใช้เป็นประเภท	<pre>enum Color {Red, Green, Blue}; var red: Color = Color.Red;</pre>
tuple	มีลักษณะคล้ายอาร์เรย์ที่มีขนาดแน่นอน การใช้ งานเหมือนอาร์เรย์	<pre>let x: [string, number]; x = ['hello', 100]; console.log(x[0]); //hello</pre>
void	ไม่มีไทป์ ใช้กับฟังก์ชัน ที่ไม่คืนค่าไทป์ใด	<pre>function myfunc() : void { }</pre>
any	เป็นไทป์ใดๆ แม้กระทั่งประกาศเป็นอาร์เรย์ให้มี ไทป์ใดๆ	<pre>var unknow: any; var unknows:any[] = [1, true, "x"];</pre>
null	หมายถึงค่าที่ยังไม่หนดไทป์ (undefined)แน่นอน	<pre>var value: any = null;</pre>
object	ชนิดนี้เป็นชนิดเดียวกับ JSON	<pre>let obj = {id:1, name:"pol"};</pre>

การประกาศตัวแปรและขอบเขตตัวแปร

เมื่อเริ่มใช้งาน การประกาศตัวแปรเป็นสิ่งเริ่มต้นให้โปรแกรมมีความหมาย คีย์เวิร์ด ที่ระบุประกาศตัวแปรคือ var, let, const สำหรับการประกาศ const จะทำให้ตัวแปรมีค่าคงที่ หรือเปลี่ยนแปลงไม่ได้

การประกาศ var, let, const ตัวแปรจะมีขอบเขต หรือการเข้าถึงได้เป็นระดับ ในหลักการทั่วไป คือการเข้าถึงได้ก่อน เป็นไปตามกรอบขอบเขตที่ใกล้สุด และกรอบขอบเขตนอกไปสู่กรอบขอบเขตใน หรือกรอบขอบเขตนอกกระจายไปสู่ขอบเขตใน ทำให้กรอบขอบเขตในจะมองเห็นกรอบขอบเขตนอก ในทางตรงกันข้ามกรอบขอบเขตนอก จะมองไม่เห็นกรอบขอบเขตใน

ตาราง 2 เปรียบเทียบ TS กับ JS (หลังคอมไพล์ TS ไปเป็น JS)

hello.ts	hello.js
<pre>var pol:string = "pol";//global var function greeting():void{ var word: string = "Hello";//local var alert(word+", "+ pol); }</pre>	<pre>var pol = "pol"; //global var function greeting() { var word = "Hello"; //local var alert(word + ", " + pol); }</pre>

จากตัวอย่าง ในตาราง 2 ตัวแปร pol ประกาศให้มีขอบเขตนอกสุด ทำให้ มองเห็นได้ภายในฟังก์ชัน greet() แต่ตัวแปร word ประกาศในฟังก์ชัน จะมองไม่เห็นจากนอกฟังก์ชัน

ตาราง 3 เปรียบเทียบ การใช้ var และ let

var	let : ERROR
<pre>function variable() { var a = 1; if (true) { var b = a + 2; } var c = b + 1; return c; }</pre>	<pre>function variable() { let a = 1; if (true) { let b = a + 2; } let c = b + 1;//ERROR return c; }</pre>

สำหรับความแตกต่างของ var, let, และ const ที่สำคัญคือ ระดับขอบเขต แยกเป็นสองกลุ่ม คือ กลุ่ม var มีขอบเขตภายในฟังก์ชัน ซึ่งภายในฟังก์ชันไม่ว่าอยู่ส่วนใดสามารถเข้าถึงตัวแปรได้หมด และ กลุ่ม let, const มีขอบเขตตามหลักการทั่วไป (ภายในเห็นภายนอก ภายนอกไม่เห็นภายใน)

จากตาราง 3 ตัวแปร var c (ตารางสีเขียว) สามารถมองเห็นกันหมดทั่วตัวภายในฟังก์ชัน ดังตัวแปร c สามารถเห็นตัวแปร b ซึ่งอยู่ในขอบเขตในของ if ในทางกลับกันของตัวแปร let c (ตารางสีชมพู) ไม่สามารถมองเห็นตัวแปร b ได้ และบรรทัด let c = b + 1 จึงผิดพลาด

เปลี่ยนแปลงไพบี

ไพบีหนึ่งสามารถเปลี่ยนแปลงไปเป็นไพบีอื่นได้ ดูได้จากตัวอย่างการแปลงแต่ละชนิดต่อไปนี้

```
var str: string = String(42);
console.log(typeof(str)); //string
var num: number = Number(str);
console.log(typeof(num));
var bool: boolean = Boolean(0);
console.log(typeof(bool)); //Boolean
```

ในการแปลงไพบีให้สังเกตว่า ใช้ชื่อคลาสซึ่งเป็นอักษรพิมพ์ใหญ่ แต่ไพบีใช้อักษรพิมพ์เล็ก

ไพบีรวม (Union types)

TS อนุญาตให้ประกาศ ไพบีรวมกันหลายแบบได้ด้วยการใช้เครื่องหมายหรือ (|) เหมือนกับการประกาศอาร์เรย์ที่มีไพบีเป็นไพบีใดๆ ตัวอย่างเช่น

```
var code_id : number[] | string;
code_id = "code 1";
code_id = [10001, 10002];
```

จากตัวอย่างนี้จะเป็นว่า เป็นการประกาศให้มีไพบี เป็นอาร์เรย์ประเภทตัวเลข หรือ มีไพบีเป็นอักษรก็ได้ แต่จะมีเป็นสองไพบีที่ระบุได้เท่านั้น

ไพบีชื่อเล่น

ในบางครั้งเราอาจกำหนดไพบีในชื่ออื่นๆ ที่ไม่มีอยู่จริงในชื่อไพบีของ TS แต่ใช้เป็นชื่อเล่น หรือชื่อที่เรากำหนดได้เอง ด้วยการใช้คีย์เวิร์ดว่า **type** ซึ่งจะให้เราให้ความหมายได้ตรงตามความต้องการ ดังตัวอย่างการประกาศต่อไปนี้

```
type myInt = number;
type myFloat = number;
type myChar = string;
type myString = string;
type myStringArray = Array<string>;
```

ตัวดำเนินการทางคณิตศาสตร์

ตัวดำเนินการที่ TS สนับสนุนเป็นกับหลายภาษา เช่น Java, JavaScript, C# ตารางต่อไปนี้แสดงตัวดำเนินการ ในตัวอย่างให้ สมมุติว่ามีตัวแปร a และ b มีค่าเป็น 20, และ 10

ตาราง 4 ตัวดำเนินการทางคณิตศาสตร์ และตัวอย่าง โดยให้ $a = 20$ และ $b = 10$

ตัวดำเนินการ	ความหมาย	ตัวอย่าง
+	การบวก	$a + b$ ให้ผลเป็น 30
-	การลบ	$a - b$ ให้ผลเป็น 10
*	การคูณ	$a * b$ ให้ผลเป็น 200
/	การหาร	a / b ให้ผลเป็น 2
%	โมดูลอ เป็นการหารที่เร้าแต่เศษ	$a \% b$ ให้ผลเป็น 10
++	เพิ่มทีละหนึ่งค่า	$a++$ ให้ผลเป็น 21
--	ลดทีละหนึ่งค่า	$a--$ ให้ผลเป็น 19

ตัวดำเนินการทางคณิตศาสตร์ และให้ค่า

การให้ตัวแปรมีค่าเป็นอะไร หรือมอบหมายค่าเป็นอะไร (assignment) เช่น `var name: string = 'pol'` เป็นการกำหนดค่า ตัวแปร `name` ให้มีค่าเป็น `'pol'` ซึ่งเป็นความรู้แรกๆ ที่ทราบกันดีแล้ว แต่มีตัวดำเนินการให้ค่าอื่นๆ ที่มีร่วมกัน ตัวดำเนินการทางคณิตศาสตร์ ที่อยู่ก่อนตัวดำเนินการให้ค่า ดังอธิบายด้วยตารางต่อไปนี้

ตาราง 5 ตัวดำเนินการให้ค่า และตัวอย่าง

ตัวดำเนินการ	ความหมาย	ตัวอย่าง
$+=$	การบวก และ ให้ค่า	$a += b$ เท่ากับ $a = a + b$
$-=$	การลบ และ ให้ค่า	$a -= b$ เท่ากับ $a = a - b$
$*=$	การคูณ และ ให้ค่า	$a *= b$ เท่ากับ $a = a * b$
$/=$	การหาร และ ให้ค่า	$a /= b$ เท่ากับ $a = a / b$
$\% =$	การโมดูลอ และ ให้ค่า	$a \% = b$ เท่ากับ $a = a \% b$

ดำเนินการเปรียบเทียบและลอจิก

ในการเปรียบเทียบสิ่งใด กับสิ่งใด ผลลัพธ์ย่อมได้ คือ จริง (true) หรือไม่ก็ เท็จ (false) จะไม่มีคำตอบ ไม่จริง และไม่เท็จ หรือตอบว่าไม่รู้ ไม่ได้ เช่น $1 > 2$ ใช่ไหม คำตอบที่ถูกต้องคือ ผิด หากเราถามว่า $1 > A$ ใช่ไหม คำถามนี้ ผิด จึงตอบไม่ได้ว่า จริงหรือเท็จ เพราะของสองสิ่งนี้เทียบกันไม่ได้ ในทางโปรแกรมคอมพิวเตอร์ของสองสิ่งที่จะเปรียบเทียบกัน ได้เท่านั้นจึงจะมีคำตอบ กรณีที่ของเปรียบเทียบไม่ได้ โปรแกรมจะแจ้งความผิดพลาด หากใช้ตัวเขียนที่มีสภาพล้อมติๆ ตั้งแต่เขียนโปรแกรม (write time error) หรือไม่แจ้งความผิดพลาดขณะคอมไพล์โปรแกรม (compile time error) กรณีของ TS จะทราบทันทีเพราะไทป์ บางชนิดเปรียบเทียบไม่ได้ แต่โปรแกรมภาษาประเภทไม่ระบุไทป์ เช่น JS โปรแกรมจะผิดพลาดตอนทำงาน (runtime error)

ตาราง 6 ตัวดำเนินการเปรียบเทียบ และตัวอย่าง โดยให้ $a = 1$ และ $b = 2$

ตัวดำเนินการ	ความหมาย	ตัวอย่าง
$==$	ค่าเท่ากันหรือไม่	$a == b$ ให้ผลเป็น เท็จ
$===$	ค่าและไทป์ เท่ากันหรือไม่	$a === b$ ให้ผลเป็น เท็จ

!=	ไม่เท่ากัน ใช่ไหม	a != b ให้ผลเป็น จริง
>	มากกว่า ใช่ไหม	a > b ให้ผลเป็น แท้
>=	มากกว่า หรือเท่ากัน ใช่ไหม	a >= b ให้ผลเป็น แท้
<	น้อยกว่า ใช่ไหม	a < b ให้ผลเป็น จริง
<=	น้อยกว่า หรือเท่ากัน ใช่ไหม	a <= b ให้ผลเป็น จริง
&&	แทน และ (AND)	(a>0 && a<0) ให้ผลเป็นแท้
	แทน หรือ (OR)	(a>0 && a<0) ให้ผลเป็นจริง
!	แทน ไม่ (NOT)	!(a>0) ให้ผลเป็นแท้

คำสั่งเพื่อการควบคุม (Flow control)

ในทุกๆ ภาษาเขียนโปรแกรมคอมพิวเตอร์ จะมีคำสั่งควบคุม เพื่อสั่งให้โปรแกรมคอมพิวเตอร์ทำงานในรูปแบบโครงสร้าง เช่น โครงสร้าง ถ้า (a > b) ให้ a += 1 ซึ่งเป็นโครงสร้างคำสั่ง if ในบางภาษาโครงสร้างเพื่อการควบคุม มีจำกัด เช่น ภาษาไพธอน (Python) มีเพียงโครงสร้างคำสั่ง if, while, และ for เท่านั้น ซึ่งก็เป็นโครงสร้างคำสั่งพื้นฐานที่เพียงพอต่อการเขียนโปรแกรมให้คอมพิวเตอร์ให้ทำงานแล้ว แต่สำหรับ TS มีมากกว่านั้นมาก ดังอธิบายได้เป็นสองกลุ่ม คือ กลุ่ม ถ้า และกลุ่มวนซ้ำ กลุ่มละ 4 คำสั่งควบคุม ซึ่งเราอาจใช้บ่อยเพียงคำสั่งแรกๆ เท่านั้น ดังจะเห็นได้ว่า ภาษาไพธอน ก็ไม่จำเป็นต้องมีคำสั่งเพื่อการควบคุมมากมาย ก็ทำงานได้ดีเช่นกัน

ตาราง 7 กลุ่มคำสั่งโครงสร้าง ถ้า

คำสั่งควบคุม	คำอธิบาย	ตัวอย่าง
if	ถ้า	if(true) alert("I am valid");
if ... else	ถ้า แล้ว	if(false) alert("I am invalid"); else alert("I am valid")
if... else if ... else	ถ้า แล้วถ้า แล้ว	if(false) alert("I am invalid"); else if(fale) alert("I am invalid"); else alert("I am valid");
?	ถ้า อย่างย่อ ถ้าจริงจะคืนค่าแรก ถ้าเท็จจะคืนค่าหลัง	alert((true)? "I am valid" : "I am invalid");
switch	ถ้า แล้วเลือก ตามกรณี (case) ซึ่งแต่ละกรณี ของตัวแปร เช่น เป็น 0, 1, และ ไม่เข้ากรณีใด (default) ที่น่าสังเกตว่าแต่ละกรณีจะมีคำสั่ง break เพื่อหยุดไม่ไปกรณีถัดไป	var integer : number = 0; switch(integer){ case 0: alert("It is zero"); break; case 1: alert("It is one"); break; default: alert("It is non-zero, and non-one"); }

ตาราง 8 กลุ่มคำสั่งโครงสร้าง วนซ้ำ

คำสั่งควบคุม	คำอธิบาย	ตัวอย่าง
while	ทำซ้ำ while ไปเรื่อยๆ จนกว่าจะผิดเงื่อนไข	var integer : number = 0; while(integer < 10) { integer += 1; console.log(integer); }

	ให้ระวังว่าต้องให้มีเงื่อนไข มิฉะนั้นจะทำซ้ำไม่รู้จบ	
do ... while	ทำซ้ำ do .. while เป็นการทำงานก่อนแล้ว ค่อยตรวจสอบเงื่อนไข ดังนั้นแล้วจะทำอย่างน้อยหนึ่งครั้ง	<pre>do{ integer += 1; console.log(integer); }while(integer <5);</pre>
for	ทำซ้ำ for เป็นการทำซ้ำ โดยมีเงื่อนไข	<pre>var int:number[] = [1,2,3,4]; for(var i : number = 0; i < int.length; i++){ console.log(i); }</pre>
for ... in	ทำซ้ำ for each เป็นการทำซ้ำ แต่ละค่าของชุดข้อมูล แบบมีคีย์ (อาร์เรย์ใช้เครื่องหมาย [])	<pre>var object:any ={'id':101,'name': 'pol'}; for(var key in object){ console.log(key + ':' + object[key]); } let list = [4, 5, 6]; for (let i in list) { console.log(i); // "0", "1", "2", }</pre>
for.. of	ทำซ้ำ for each แบบอ่านเฉพาะค่า (ไม่อ่านคีย์) ซึ่งเมื่อเปรียบเทียบกับ for ... in ที่อ่านเฉพาะคีย์ หรือเลขดัชนี แสดงลำดับ	<pre>let list = [4, 5, 6]; for (let i of list) { console.log(i); // "4", "5", "6" }</pre>

ฟังก์ชัน

การสร้างฟังก์ชันของ TS ทำได้คล้ายกับ JS คือ ทำแบบประกาศฟังก์ชันที่มีชื่อ กับไม่ต้องมีชื่อ (anonymous) ลองพิจารณาสองฟังก์ชันในแต่ละแบบต่อไปนี้ ซึ่งทำงานได้คล้ายๆ กัน

ให้ทำการคอมไพล์ index.ts ซึ่งจะได้ ดังไฟล์ index.js ซึ่งนำไปทดสอบกับไฟล์ index.html

Code 2. index.ts

```
//name function
function greeting(name: string): string{
  if(name) return "Hi " + name;
  else return "Hi";
}
//anonymous function
var greetingMorning = function(name:string):string{
  if(name) return "Good morning " + name;
  else return "Good morning";
}
```

Code 3. index.js

```
//name function
function greeting(name) {
  if (name)
    return "Hi " + name;
```

```

        else
            return "Hi";
    }
    //anonymous function
    var greetingMorning = function (name) {
        if (name)
            return "Good morning " + name;
        else
            return "Good morning";
    };

```

Code 4. index.html

```

<script src='function.js'></script>
<script>
alert(greeting("pol"));
alert(greetingMorning("pol"));
</script>

```

จากตัวอย่าง การสร้างฟังก์ชันของ TS สามารถที่สร้างกำหนดให้ไปได้โดยตรงให้กับตัวแปรเข้า โดยไม่ต้องการเขียน var นำหน้า และในบางครั้ง เราไม่จำเป็นต้องเขียนไปสำหรับการคืนค่า ก็ได้ TS จะอนุมานไปจากคำสั่งการคืนค่าได้เอง เช่น เราเขียนใหม่โดยมีต้องระบุไปการคืนค่าได้ดังนี้

Code 5. index.ts

```

//name function
function greeting(name: string){
    if(name) return "Hi " + name;
    else return "Hi";
}
//anonymous function
var greetingMorning = function(name:string){
    if(name) return "Good morning " + name;
    else return "Good morning";
}

```

ตัวแปรเข้าแบบทางเลือก

ในภาษาอื่นๆ สามารถที่กำหนดให้ตัวแปร มีหรือไม่มีก็ได้ เช่น ฟังก์ชัน greeting () ต้องการให้มีตัวแปรเข้า ตัวแรกเป็น string และตัวแปรที่สองเป็น string เช่นกัน แต่ตัวแปรที่สองถ้าในการใช้งานไม่ใส่ให้ถือว่า ตัวแปรที่สองนี้ไม่ (undefined) การกำกับว่าตัวแปรนี้เป็นทางเลือกใช้เครื่องหมายคำถาม (?) ต่อท้ายตัวแปร การตรวจสอบว่ามีตัวแปรทางเลือกหรือไม่ใช้เพียงใช้เงื่อนไขตรวจสอบจริง/เท็จ ดังตัวอย่างต่อไปนี้

Code 6. index.ts

```

function greeting(name: string, lname?:string){
    if(lname) return "Hello " + lname ;
    else return "Hi " + name;
}

console.log(greeting("Tee", "L.");//output: Hello L.
console.log(greeting("Tee"))//output: Hi Tee.

```


แลมบ์ดา

ยังมีรูปแบบการเขียนฟังก์ชันแบบไม่มีชื่ออีกวิธีหนึ่งเรียกว่า แลมบ์ดา (Lambda) หรือเขียนแนวเชิงฟังก์ชัน ซึ่งวิธีการนี้เป็นการเขียนโปรแกรมที่เลียนแบบ การทำฟังก์ชันทางคณิตศาสตร์ เช่น

```
var helloLambda = ():string =>{return "Hello, lambda";};
```

จากตัวอย่างนี้จะเห็นว่า ใช้เครื่องหมาย => เพื่อบอกว่าเป็นแลมบ์ดา โดยเขียนต่อท้าย วงเล็บหรือโทบ์ทันที่ (ห้ามขึ้นบรรทัดใหม่) ชื่อฟังก์ชันไม่ต้องระบุ หากมีตัวแปรให้ใส่ในวงเล็บ และหากไม่เขียนโทบ์ทันที่จะคืนก็จะเขียนใหม่ได้ว่า

```
var helloLambda = ()=>{return "Hello, lambda";};
```

แต่เนื่องจากมีคำสั่งเดียวคือ ให้คืนค่า จึงไม่จำเป็นต้องใส่คีย์เวิร์ด return เพราะถือว่า จะเป็นการคืนค่าอยู่แล้ว และอีกอย่างเมื่อมีเพียงคำสั่งเดียวก็ไม่จำเป็นต้องใส่เครื่องหมายปีกกาก็ได้ ดังเขียนใหม่เป็น

```
var helloLambda = ()=>"Hello, lambda";
```

ซึ่งจะเหมือนกับ การเขียน JS ต่อไปนี้

```
var helloLambda = function () { return "Hello, lambda"; };
```

การเรียกใช้ เพียงเรียกชื่อตัวแปร helloLambda() ฟังก์ชันก็จะทำงาน ซึ่งจะเหมือนกับ การเขียนแบบไม่ต้องกำหนดค่าตัวแปร var วิธีการนี้จะถือว่าเป็นการเรียกใช้ทันที ต่อไปนี้

```
((())=>"Hello, lambda")();
```

หรือจะใช้ ชิดล่าง แทนการไม่ใส่ตัวแปรก็ได้

```
(_ =>"Hello, lambda")();
```

ทดสอบอีกครั้งผ่าน console.log()

```
console.log(((())=>"Hello, lambda")());
```

การประกาศแลมบ์ดา จะมีเนื้อหาในฟังก์ชัน หรือในแลมบ์ดา ว่าให้ทำอะไร แต่ยังมีฟังก์ชันบางตัวไม่ได้บอกเนื้อหาว่าให้ทำอะไร เช่น การประกาศฟังก์ชันในอินเทอร์เฟซ ตามปกติ จะประกาศเพียง:

```
log(arg:any):boolean;
```

ฟังก์ชัน log() รับตัวแปรใด ๆ และคืนค่าจริงหรือเท็จ (boolean) แต่นั่นไม่ได้เป็นแลมบ์ดา ถ้าต้องเขียนเป็นแลมบ์ดา ต้องประกาศใหม่คือ

```
log:(arg:any)=>boolean;
```

แลมบ์ดา ไม่ได้คืนค่าเสมอไป อาจเป็นการทำงานเฉพาะบางอย่างที่ไม่คืนค่า แสดงว่าต้องใช้ปีกกาคล่อมคำสั่ง เช่น ต้องการ ลบค่าจากอาร์เรย์ ที่ตำแหน่งแรก ไปหนึ่งตำแหน่ง

Code 7.

```
var users = [
  {id: 1, name: 'Hydrogen'},
  {id: 2, name: 'Helium'},
  {id: 3, name: 'Lithium'},
];

users.forEach(u => {
  if (u.id==1) {
    users.splice(0,1);
    console.log("splice is not slice");
  }
});
```

ตัวแปร users เป็นอาร์เรย์ ที่ประกอบด้วย 3 ออบเจกต์ ตำแหน่งแรกแทนด้วยดัชนี (index) ศูนย์ ใช้การวนซ้ำ forEach() ภายในอาร์เรย์ และภายในอาร์เรย์เขียนเป็นแลมบ์ดา ที่ตรวจสอบ id ที่เป็น ให้ลบออกไป ต่อด้วยเขียนที่ console.log ให้แสดงว่าได้ลบแล้ว สำหรับหัวข้ออาร์เรย์จะได้อธิบายในหัวข้อถัดไป

อาร์เรย์

การใช้งานอาร์เรย์ มีลักษณะยืดหยุ่น มีขนาดเพิ่มขึ้นได้ภายหลัง จะใช้แบบมีไทม์ หรือ ไม่ระบุไทม์ก็ได้ (แบบไม่ระบุไทม์เรียกอีกอย่างว่า ทูเพิล (tuple) มีลำดับดัชนีเริ่มต้นที่ศูนย์

การประกาศอาร์เรย์ใช้ เครื่องหมาย [] หรือสร้างจากออบเจกต์ Array() การหาขนาดของอาร์เรย์ ใช้ คุณสมบัติ length สำหรับอาร์เรย์แบบมีคีย์ (Associative array) ใช้เครื่องหมาย { } ดังที่เห็นมาก่อนในตัวอย่างการวนซ้ำมาแล้ว

ตัวอย่างต่อไปนี้ เป็นการสร้างอาร์เรย์พร้อม ๆ กับใส่ค่าในอาร์เรย์ ในลักษณะต่าง ๆ เช่น การสร้างอาร์เรย์จาก คลาส Array() พร้อม ๆ กับใส่ค่าเป็นตัวแปรเข้า การสร้างอาร์เรย์จากเครื่องหมาย [] การสร้างอาร์เรย์แบบมีคีย์จาก เครื่องหมาย { }

Code 8. array.ts

```
//declare and initial values
var books = new Array('Java', 'C#', 'VB');
var fruit:string[ ] = ['apple', 'orienne', 'banana'];
var computer = {1:'apple',2: 'ibm',3:'acer'};
var anyObj = [1, 'ABC', books];
```

ตาราง 9 คุณสมบัติและฟังก์ชันของอาร์เรย์

คำสั่งของอาร์เรย์	คำอธิบาย	ตัวอย่าง
length	เป็นการหาขนาดของอาร์เรย์	console.log(books.length); //3
pop()	เป็นการออกรายการสุดท้าย ออกเป็น	books.pop(); //remove VB
concat()	เป็นการต่ออาร์เรย์กับอาร์เรย์	[1,2,3].concat([4,5,6])

Array.isArray()	ตรวจสอบว่าเป็นอาร์เรย์หรือไม่	Array.isArray([1,3]) //output: true
push()	การเพิ่มไปยังรายการสุดท้าย	books.push("C++");//add C++
join()	จะคืนค่าผลการรวมรายการทั้งหมดของอาร์เรย์ การรวมอาจใช้ตัวแปรร่วมด้วย	console.log(books.join("-")); //output //Java-C#-C++
splice(x,y)	เป็นการเลื่อนอาร์เรย์ตั้งแต่ตำแหน่ง 2 (x) ไปจำนวน 1 (y)รายการ ซึ่งคืนออบเจกต์ใหม่ที่เลื่อนออกไป	var rem = anyObj.splice(2,1); console.log(rem); //output: book
array.forEach(function())	เป็นคำสั่งวนซ้ำภายในอาร์เรย์ โดยมีฟังก์ชันเป็นตัวจัดการกับรายการ	anyObj.forEach(item =>{ console.log(item); }); //output: 1 'ABC' anyObj.forEach((item, k) =>{ console.log(k); } } //output: return index
array.filter(function())	ใช้สำหรับกรองข้อมูลเพื่อการค้นหา	array.filter(i=>i==3);
array.map(function())	แปลงค่าของแต่ละค่าในอาร์เรย์	[1,2,3].map(t=>t+1); //output:2,3,3
array.reduce(function())	ดำเนินการสะสมค่า (acc) กับตัววิ่ง(i) ในอาร์เรย์	[1, 2, 3].reduce((acc, i)=> acc + i); //output: 6
find(function())	ใช้สำหรับเพื่อการค้นหา	array.find(i=>i==3);
sort(function())	ใช้สำหรับการเรียง โดยต้องใส่ฟังก์ชันเป็นตัวแปรเข้าเพื่อเลือกจะเรียงค่า	let score:number[] = [3,1,4]; score.sort((a,b)=>a- b).forEach(i=>console.log(i));

ฟังก์ชันที่น่าจะได้ใช้บ่อยคือ filter, find และ forEach ซึ่งเป็นเครื่องมือสำคัญในการดำเนินการภายในอาร์เรย์ ซึ่งต้องใส่ตัวแปรเป็นฟังก์ชัน หรือจะเขียนเป็นแลมบ์ดา เช่น การค้นหาข้อมูลในอาร์เรย์ ด้วย filter ดังตัวอย่างต่อไปนี้ ใช้การสืบค้นหาเฉพาะข้อมูลที่เป็นเลข 3

Code 9.

```
const numbers: number[] = [1, 3, 5];
let num = numbers.filter(i => i == 3);
console.log(num[0]); //print 3
```

จากตัวอย่างนี้ให้สังเกตว่า ผลลัพธ์ของการสืบค้นอยู่ในรูปอาร์เรย์ เช่นเดียวกับข้อมูลเดิม เพราะผลการสืบค้นอาจได้ข้อมูลหลายตัวได้

แต่ถ้าใช้ฟังก์ชัน `find()` จะสะดวกกว่าเพราะจะคืนค่า ค่าเดียว ไม่ต้องเข้าสู่ลำดับของอาร์เรย์อีกครั้ง ดังตัวอย่างต่อไปนี้ ให้ผลลัพธ์เหมือนกับการใช้ `filter()`

Code 10.

```
const numbers: number[] = [1, 3, 5];
let num = numbers.find(i => i == 3);
console.log(num); // print 3
```

สำหรับ `forEach` มีจะใช้งานได้ดี โดยทำงานร่วมกับแลมบ์ดา แต่มีสิ่ง 3 สิ่งที่เราควรคำนึงคือ ไม่สามารถใช้คำสั่ง `return`, `break`, และ `continue` ได้ ดังนั้นถ้าต้องการใช้ใน 3 คำสั่งนี้ ควรกลับไปใช้แบบ `for` ธรรมดาแทน¹

อาร์เรย์เจเนอริก (generic) คือการอาร์เรย์ให้รับค่าได้เฉพาะใดก็ได้ไปหนึ่งเท่านั้น การประกาศอาร์เรย์หากเราใช้

```
user: User[ ];
```

ซึ่งอาจมองว่าเป็นค่าเจเนอริกแล้ว แต่แท้จริงแล้ว การประกาศแบบนี้ยังไม่สร้างค่าอะไร การเพิ่มอาร์เรย์ด้วยฟังก์ชัน `push()` จะทำให้ทำงานผิดพลาด เพราะยังไม่สร้างเป็นออบเจกต์อาร์เรย์ วิธีการแก้คือ ประกาศผ่านออบเจกต์ `Array<User>()` ดังเช่น:

```
user:User[] = new Array<User>[]
```

ฟังก์ชันที่ไม่ค่อยได้ใช้แต่มีประโยชน์ในการแปลงค่าของอาร์เรย์ คือ `map` และ `reduce` เพราะช่วยลดขั้นตอนการใช้คำสั่งวนซ้ำของ `for` หรือ `forEach` ได้มาก เช่น เราต้องการสร้างข้อมูลที่นับความถี่ ให้ใช้ `reduce` จะช่วยได้มาก หรือการใช้ `map` เพื่อแปลงข้อมูลจาก JSON ไปเป็น CSV

Code 11.

```
const users = [
  { name: 'Nick', age: 20 },
  { name: 'John', age: 40 },
  { name: 'Jame', age: 30 },
  { name: 'Poly', age: 40 }
];

let groupByAge = users.reduce((acc, i) => {
  acc[i.age] = acc[i.age] + 1 || 1;
  return acc;
}, {});
console.log(groupByAge);
//output:{ "20": 1, "30": 1, "40": 2 }

let csv = users.map(({name, age}) => `\n ${name}, ${age}`).join('');
console.log(csv);
/* output
```

¹ได้มาจาก (March 20, 2020)

<https://medium.com/front-end-weekly/3-things-you-didnt-know-about-the-foreach-loop-in-js-ff02cec465b1>

```
Nick, 20
John, 40
Jame, 30
Poly, 40
*/
```

รวมอาร์เรย์ด้วย 3 จุด

เครื่องหมายจุด ... มีพบในหลาย ๆ โปรแกรมภาษาอื่น ๆ ใน JavaScript ก็มีความหมายเดียวกับ TS คือเป็นการอ้างอิงอาร์เรย์และทำการรวมกับอาร์เรย์อื่น ๆ ถือเป็นเครื่องมืออำนวยความสะดวกอย่างหนึ่ง

Code 12. array3dot.ts

```
function printNums(nums: number[]){
    nums.forEach(i => console.log(i));
}
var num1:number[] = [1,2,3];
var num2:number[] = [...num1, 4, 5, 6];
printNums(num2);
//print: 1,2,3,4,5,6
```

การต่ออาร์เรย์กับอาร์เรย์มีความหมายเหมือนกับการใช้ฟังก์ชัน concat() เช่น:

```
var num2:number[] = num1.concat([4, 5, 6]);
```

คลาส (Class)

คลาสของ TS สร้างบนพื้นฐาน ECMAScript ซึ่งเป็น JS รุ่นใหม่ ปัจจุบันเป็นรุ่นที่ 8 (June, 2017) ทำให้เขียนโปรแกรมในแนววัตถุ (Object) ได้อย่างสมบูรณ์ และมีเบราว์เซอร์รองรับการทำงานทั่วไป

ตัวอย่างต่อไปนี้ เป็นการสร้างคลาส Book เก็บเป็นไฟล์ชื่อ book.js ต่อมาเมื่อคอมไพล์จะได้ไฟล์ชื่อ book.js และทดสอบสร้างวัตถุ ด้วยไฟล์ index.html และข้อมูลดูได้ด้วย ฟังก์ชัน info()

Code 13. book.ts

```
class Book{
    title:string;
    price:number;
    constructor(title:string, price:number){
        this.title = title;
        this.price = price;
    }
    info(){
        return this.title+":"+this.price;
    }
}
```

การสร้างออบเจกต์ของ TS

```
let book = new Book("TS", 200);
let book2: Book = new Book("JS", 150);
```

การสร้างออบเจกต์ของ TS จะต้องมียีวีร์รต์ let เพื่อบอกว่าเป็นตัวแปร และไม่จำเป็นต้องระบุไทป์ (Type)

จากตัวอย่างการสร้างคลาสนี้ การอ้างอิงถึงสมาชิก ให้ใช้ this เพื่อแยกความแตกต่างกับตัวแปรเข้าของฟังก์ชัน ว่าเป็นคนละตัวกัน

นอกจากนี้ไม่มีการใช้ คีย์เวิร์ด private และ public เหมือนอย่างภาษาเขียนโปรแกรมเชิงวัตถุอื่นๆ เพราะ JS มีแนวคิดเรื่องขอบเขตการเข้าถึงในลักษณะที่ต่างออกไป แต่โดยทั่วไป การทำให้สมาชิกมีการเข้าถึงในลักษณะ public ใช้คำว่า this อ้างอิงได้ และให้สมาชิกมีการเข้าถึงในลักษณะ private ให้ประกาศตัวแปรเป็น var ในตัวอย่างนี้เลือก ให้สมาชิกทุกตัวไม่มีการประกาศอะไรนำหน้า ซึ่งจะหมายถึงเป็นสมาชิกประเภท public หรือเป็นคำปริยายถ้าไม่ระบุจะถือว่าสมาชิกนั้นเป็น public

Code 14. book.js

```
var Book = /** @class */ (function () {  
    function Book(title, price) {  
        this.title = title;  
        this.price = price;  
    }  
    Book.prototype.info = function () {  
        return this.title + ":" + this.price;  
    };  
    return Book;  
})();
```

Code 15. index.html

```
<script src='book.js'></script>  
<script>  
    var book = new Book('C#', 200);  
    alert(book.info());  
</script>
```

สร้างสมาชิกในคลาสด้วยคอนสตรักเตอร์

จากคลาส Book ที่ผ่านมาสร้างสมาชิกด้วยการประกาศ สมาชิกแบบที่เคยทำกับภาษาเชิงวัตถุอื่น ๆ สำหรับ TS มีวิธีที่ทำได้สั้นกว่า ด้วยการใส่สมาชิกลงในตัวแปรของคอนสตรักเตอร์ แต่ต้องนำหน้าด้วยคีย์เวิร์ด ว่า private หรือ public ขึ้นอยู่ว่า ต้องการให้สมาชิกเป็นประเภทใด และที่สำคัญ ถือว่าเป็นการสร้างออบเจกต์ ของสมาชิกขึ้นมาพร้อมๆ กับ คลาสหลัก วิธีการแบบนี้ใช้แนวคิด Dependency Inject ที่ลดขั้นตอนการสร้างสมาชิก ดังนั้นแล้ว คลาส Book จึงเขียนใหม่ได้ว่า

Code 16. book.ts

```
class Book{  
    constructor(public title:string, public price:number){ }  
    info(){  
        return this.title+":"+this.price;  
    }  
}
```

ด้วยการเขียนแบบนี้ แต่ยังคงให้อ้างอิงตัวแปรในฟังก์ชันอื่นได้ด้วยคีย์เวิร์ด this อยู่เช่นเดิม (ต้องมีคีย์เวิร์ดในคอนสตรักเตอร์ว่า ว่า private หรือ public) ดังพบในฟังก์ชัน info() วิธีการสร้างสมาชิกแบบนี้ พบมากในการเขียนโปรแกรมของ Angular

getter/setter

TS ไม่ยอมพลาดที่จะมี getter/setter ในแบบของ .NET อย่างที่มีใน C# ด้วยการใช้คีย์เวิร์ด get และ set (ไม่มีวงเล็บ) นำหน้าฟังก์ชัน (มีวงเล็บ) เพื่ออ่านค่า และกำหนดค่า

ตัวอย่างต่อไปนี้ กำหนดให้ `_title` มีค่าเป็น private เพื่อไม่ให้มองเห็นจากภายนอก แต่กำหนดให้มองเห็นได้ผ่าน getter/setter เท่านั้น ด้วยฟังก์ชัน `title()` ในการเรียกใช้งาน สามารถอ่านค่าและกำหนดค่า ทำได้โดยตรงผ่าน ชื่อ `title` (สังเกตว่าไม่มีวงเล็บ)

Code 17. book.ts

```
class Book{
  constructor(private _title:string, public price:number){ }
  info(){
    return this._title+":"+this.price;
  }
  get title() {
    return this._title;
  }
  set title(title: string) {
    this._title = title;
  }
}
let book = new Book("C#", 2000);
book.title = "TS";
alert(book.title);
```

สมาชิกประเภท static

สมาชิกใดของคลาสที่ประกาศเป็น สเตติก (static) ถือว่าเป็นสมาชิกของคลาส ไม่ใช่เป็นสมาชิกของออบเจกต์ สมาชิกใดที่เป็นของออบเจกต์ จะใช้คีย์เวิร์ดเรียกว่า `this` นำหน้า (ถ้าเรียกอยู่ภายในคลาส) หรือ ชื่อออบเจกต์นำหน้า (ถ้าเรียนจากนอกคลาส) แต่ตัวแปรของคลาสใช้ชื่อคลาสเรียก

จากตัวอย่างต่อไปนี้ ใช้ สมาชิก `nextId` เป็นค่า สเตติก การเรียกภายในคลาส หรือภายนอกคลาส ใช้ชื่อคลาสนำหน้า เช่น `Book.nextId` ในขณะที่สมาชิกที่ไม่เป็น สเตติก การเรียนภายในคลาสใช้ `this` นำหน้า หรือภายนอกคลาส ใช้ชื่อออบเจกต์นำหน้า

Code 18. book.ts

```
class Book{
  private static nextId: number = 0;
  public id: number;
  constructor(public title: string, public price: number) {
    this.id = Book.nextId;
    Book.nextId++;
  }
  info(){
    return Book.nextId+":"+ this.title+":"+this.price;
  }
}
```

```

}
let book1 = new Book("JS", 200);
console.log(book1.info()); //output: 1:JS:200
let book2 = new Book("TS", 230);
console.log(book2.info()); //output: 2:TS:230

```

คลาสมีการสืบทอด

คุณสมบัติสืบทอด เป็นความสามารถอย่างหนึ่งในการเขียนโปรแกรมเชิงวัตถุ TS ใช้การสืบทอดตามแนวทางของภาษาเขียนโปรแกรมอื่น ด้วยการใช้คีย์เวิร์ด `extends` ดังนั้นสมาชิกใดของคลาส ที่การเข้าถึงได้ที่ไม่ใช่ `private` ซึ่งคีย์เวิร์ด `public` หรือ `protected` ก็ส่งต่อคุณสมบัติได้ไปยังคลาสที่สืบทอด หรือคลาสลูกได้

สิ่งคัญที่ตามมาของการสืบทอด ที่ควรทำความเข้าใจ คือคุณสมบัติของ คอนสตรัคเตอร์ (constructor) และ โอเวอร์ไรต์ (override) ในคุณสมบัติแรก คอนสตรัคเตอร์ใช้ในการเกิดของวัตถุ หากคลาสฐาน หรือคลาสแม่ มีลักษณะการเกิดของวัตถุ ที่ไม่ธรรมดา หรือมีตัวแปรเข้าในคอนสตรัคเตอร์ (อย่างเช่น คลาส Book) คลาสที่สืบทอดต้องเกิดตามอย่างที่คลาสฐานเกิดด้วย หรืออย่างน้อยตามคลาสฐานเกิด หากต้องการเกิดในลักษณะที่ต่างกับคลาสฐาน เช่น ต้องการมีตัวแปรเข้าเกิดจากคลาสฐาน เช่น มีตัวแปร 3 ตัว การเกิดของคลาสที่สืบทอดจำต้องให้มีการเกิดแบบคลาสฐานก่อน โดยการเรียกคีย์เวิร์ด `super()` พร้อมกับใส่ตัวแปรตามคลาสฐาน แล้วค่อยเกิดตามของตัวเองต้องการเกิด

คุณสมบัติการทำโอเวอร์ไรต์ คือการสร้างความสามารถที่เขียนทับความสามารถของคลาสฐานที่สืบทอดมา เช่น คลาสฐานมีฟังก์ชัน `info()` ที่แสดงข้อมูลของออบเจกต์มี แต่คลาสที่สืบทอด อาจต้องแสดงข้อมูลที่ต่างกับคลาสฐาน จึงเขียนการแสดงผลข้อมูลขึ้นมาใหม่ บนฟังก์ชันชื่อเดิมที่ชื่อ `info()`

ตัวอย่างต่อไปนี้อธิบายการทำงานของการทำงานการสืบทอด และการทำโอเวอร์ไรต์ ทั้งในคอนสตรัคเตอร์ และฟังก์ชัน `info()`

Code 19. book.ts

```

class Book {
  title:string;
  price:number;
  constructor(title:string, price:number){
    this.title = title;
    this.price = price;
  }
  info(){
    return this.title+":"+this.price;
  }
}
class ComputerBook extends Book{
  constructor(public title:string,
    public price:number,
    public keywords:string[]){
    super(title, price);
  }
  info(){
    let keywords:string="[";
    for(let i:number=0;i<this.keywords.length;i++){
      if(i<this.keywords.length-1)
        keywords+= this.keywords[i]+",";
      else keywords +=this.keywords[i]+ "]";
    }
  }
}

```



```

        return this.title+":"+this.price+keywords;
    }
}

```

Code 20. index.html

```

<script src='book.js'></script>
<script>
    var book = new Book('Thailand', 300);
    document.write(book.info()+"<br>");
    //output: Thailand:300

    var computerBook = new ComputerBook("C#",200,
                                          ['programming', '.net']);
    document.write(computerBook.info());
    //output: C#:200[programming,.net]
</script>

```

จากตัวอย่างนี้ คอนสตรักเตอร์ของคลาสฐาน (Book) มีตัวแปรสองตัว แต่คลาสที่สืบทอด (ComputerBook) มีตัวแปรเข้าสามตัว ซึ่งต่างจากคลาสฐาน จึงจำเป็นต้องให้คลาสฐานเกิดก่อน คือ ใช้ super(title, price) ก่อนที่จะมีคำสั่งใดๆ ในคอนสตรักเตอร์ ส่วนฟังก์ชัน info() ในคลาสที่สืบทอดก็ถือเป็นการทำโอเวอร์ไรด์ เพราะมีการแสดงที่เพิ่มมากขึ้นจากคลาสฐาน โดยส่วนที่เพิ่มนี้คือ การอ่านค่าทั้งหมดในอาร์เรย์ keywords

คลาส/เมธอด abstract

มีคลาสบางประเภทที่ไม่ต้องการให้สร้างเป็นออบเจกต์ แต่มีเพื่อให้มีกาสืบทอดเป็นหลัก คลาสประเภทนี้คือคลาสที่ประกาศหน้าคลาสว่า abstract คลาสประเภทนี้จะประกอบไปด้วยเมธอด หรือฟังก์ชัน ที่เป็น abstract ได้ โดยมีเหตุผลก็เพื่อบังคับให้สร้างมาตรฐานการฟังก์ชัน (ฟังก์ชันที่ประกาศ abstract)

ฟังก์ชันที่ประกาศเป็น abstract จะมีเพียงส่วนประกาศเป็นฟังก์ชัน เท่านั้น ไม่ส่วนคำสั่งในภายในฟังก์ชัน (ภายในปีกกา) กลายเป็นฟังก์ชันไม่พร้อมทำงาน จึงทำให้คลาสนี้ไม่พร้อมสร้างเป็นออบเจกต์ไปด้วย นอกเสียจากว่าฟังก์ชันถึงเขียนคำสั่งภายในเสียก่อน ซึ่งทำได้ผ่านการสืบทอด

Code 21. book.ts

```

abstract class BookBase{
    constructor(public title: string, public price: number) {
    }
    abstract info(): string;
}
class Book extends BookBase{
    info(): string{
        return this.title+":"+this.price;
    }
}

```

การสืบทอดจาก BookBase ยังคงคือเวิร์ด extends เหมือนสืบทอดคลาสทั่วไป แต่การสืบทอดนี้จะต้องเขียนคำสั่งให้กับฟังก์ชัน info ซึ่งที่ประกาศเป็น abstract ให้สมบูรณ์

อินเทอร์เฟซ (Interface)

อินเทอร์เฟซเปรียบได้ดังเป็น ชุดของฟังก์ชันที่ยังไม่สมบูรณ์ กล่าวคือมีแต่ชื่อฟังก์ชัน แต่ไม่มีรายละเอียดการทำงาน ดังนั้นการนำไปใช้งานจะต้อง เขียนเพิ่มเติมให้สมบูรณ์ นอกจากนี้อินเทอร์เฟซ ยังสามารถใช้ข้อมูลทั่วไป ที่ไม่ใช่ฟังก์ชันได้ เหมือนกับคลาส

การนำไปใช้งาน จะใช้คีย์เวิร์ด ว่า implements ซึ่งมีความหมายคล้ายกับการสืบทอด คลาสที่ต้องการนำชุดฟังก์ชันไปใช้งาน จะต้องเขียนฟังก์ชันต่อให้สมบูรณ์

ตัวอย่างต่อไปนี้ คลาส Book ที่นำอินเทอร์เฟซ ชื่อ Logger ไปใช้งานต่อ โดยทำการเขียนฟังก์ชัน log() ต่อให้สมบูรณ์ ในคลาสตนเอง

Code 22. Book.ts

```
interface Logger{
    log(arg:any):void;
}
class Book implements Logger{
    title:string;
    price:number;
    constructor(title:string, price:number){
        this.title = title;
        this.price = price;
    }
    info(){
        return this.title+":"+this.price;
    }
    log(agr:any){
        console.log(agr);
    }
}
```

อินเทอร์เฟซในแบบ JSON

ที่กล่าวมาหมดของอินเทอร์เฟซนั้นเป็นแนวคิดทั่วไปในภาษาทั่วไป แต่ไม่ใช่กับภาษา TS เพราะอินเทอร์เฟซของ TS ไม่จำเป็นต้องมีฟังก์ชันเลยก็ได้ เช่น มีเพียงคุณสมบัติค่าข้อมูลอย่างเดียว ในลักษณะเดียวกับข้อมูล JSON และไม่จำเป็นต้องการสืบทอดด้วยการอิมพลิเมนต์ (implements) ก่อนการใช้งาน สำหรับการใช้งานจริงของอินเทอร์เฟซของ TS ก็ใช้ในลักษณะพิเศษนี้

ที่นี้มาดูตัวอย่างการใช้งาน สมมุติให้เราต้องการใช้งานอินเทอร์เฟซแบบด่วน ๆ (เหมือนสร้างออบเจกต์ JSON) ซึ่งไม่ต้องมีการประกาศอินเทอร์เฟซอย่างเป็นทางการ

Code 23.

```
function printName(nameObj: { name: string }){
    console.log(nameObj.name);
}
let myObj = { name: "Tee", "age": 50 };
printName(myObj);
//output: 50
```

จากตัวอย่างนี้ ส่วนไหนคืออินเทอร์เฟซ อย่างที่กล่าวมาว่า เป็นการสร้างอินเทอร์เฟซแบบด่วน ๆ จึงไม่มีการประกาศอย่างเป็นทางการ อินเทอร์เฟซในที่นี้ก็คือออบเจกต์ JSON ดังนั้น ตัวแปรเข้า nameObj ของฟังก์ชัน printName() ก็คืออินเทอร์เฟซ ที่มีสมาชิกตัวเดียวคือ name

สำหรับการสร้างออบเจกต์ myObj ถือเป็นการสร้างออบเจกต์ในแบบอินเทอร์เฟซเช่นกัน และนำไปใช้เป็นตัวแปรของฟังก์ชัน printName() แต่ให้สังเกตว่า ตัวแปรเข้าที่นิยามในฟังก์ชันนี้ เป็นอินเทอร์เฟซที่มีสมาชิกตัวเดียว แต่ myObj มีสมาชิกสองตัว ถึงแม้จะไม่เท่ากัน แต่คอมไพเลอร์ของ TS ยอมรับได้ขอให้สมาชิกได้อย่างน้อยหนึ่งตัว

กรณีที่ต้องการสร้างอย่างเป็นทางการ ก็ทำได้ แต่จะไม่เหมือนกับอินเทอร์เฟซในภาษาเชิงวัตถุทั่วไป เพราะไม่มีฟังก์ชันภายในอินเทอร์เฟซ มีเพียงสมาชิกข้อมูล

Code 24.

```
interface NameInterface{
    name: string;
}
function printName(nameObj: NameInterface){
    console.log(nameObj.name);
}
```

จะเห็นว่าตัวอย่างนี้เขียนอินเทอร์เฟซให้เสียเวลา แต่ถ้ามีการใช้ซ้ำ ๆ และมีสมาชิกจำนวนมาก การเขียนอินเทอร์เฟซก็เป็นเหตุผลที่ดี

อินเทอร์เฟซมีสมาชิกแบบทางเลือกแบบใส่ขาด

อีกลักษณะพิเศษของอินเทอร์เฟซในแบบ TS คือสามารถให้สมาชิกมีครบ หรือไม่ครบ ตามนิยามอินเทอร์เฟซก็ได้ ดังคือเห็นตัวอย่างก่อนหน้านี้แล้ว ในการสร้างออบเจกต์อินเทอร์เฟซ ที่เป็นตัวแปรเข้าของฟังก์ชัน printName() โดยนิยามให้มีเพียงหนึ่งสมาชิกในอินเทอร์เฟซ แต่ตอนใช้งานกลับใส่สมาชิกมาสองตัว แบบนี้เรียกว่าใส่เกินได้ แต่ไม่รับส่วนที่เกิน

การสร้างสมาชิกแบบทางเลือกใช้เครื่องหมายคำถาม(?) หรือปรีศน์ การสร้างสมาชิกแบบทางเลือก ช่วยเพิ่มทางเลือกที่ไม่จำเป็นต้องสร้างออบเจกต์ให้มีสมาชิกครบ แต่ชื่อสมาชิกต้องตรงกับชื่อที่สร้างในอินเทอร์เฟซ

Code 25.

```
interface Student {
    firstName?: string;
    lastName?: string;
}
function printStudent(student: Student){
    if(student.firstName)
        console.log('firstName:' + student.firstName);
    if(student.lastName)
        console.log('lastName:' + student.lastName);
}
let st: Student = {firstName:'pol'};
printStudent(st);
//output: "firstName:pol"
```

จากตัวอย่างนี้ จะเห็นว่าไม่จำเป็นต้องสร้างออบเจกต์ student ให้มีสมาชิกครบตามนิยามอินเทอร์เฟซ Student หรือไม่สร้างเป็นออบเจกต์ไม่มีสมาชิกเลยก็ตาม เช่น

```
let st: Student = { };
```

แล้วเติมสมาชิกภายหลัง

```
st.firstName = 'mon';
```

แต่จะสร้างในชื่อที่ไม่มีในนิยามอินเทอร์เฟซไม่ได้

```
st.age = 50;
```

```
//ERROR: Property 'age' does not exist on type 'Student'.
```

อินเทอร์เฟซมีสมาชิกแบบทางเลือกแบบใส่เกิน

ดูภาษา TS พยายามจะสร้างทางเลือกให้กว้างขวาง เหนือจะไม่มีความอะไรในการสร้างสมาชิก ที่ผ่านมามีทางเลือกให้สร้างเฉพาะที่มีนิยามในอินเทอร์เฟซ แต่ตอนสร้างออบเจกต์ใส่ไม่ครบได้ มาตอนนี้สามารถสร้างจะออบเจกต์ให้มีสมาชิกเกินกว่านิยามได้

วิธีการคือสร้างสร้างสมาชิกแบบอาร์เรย์ วิธีการนี้พบได้ในหลายภาษา แต่ถ้าเป็นภาษาที่แข็งแกร่งในชนิดตัวแปร หรือ ไทป์ แล้ว ยังทำไม่ได้ดีเท่าภาษา TS เพราะ TS มีไทป์ any ซึ่งเป็นไทป์ใด ๆ

Code 26.

```
interface Student {  
    firstName?: string;  
    lastName?: string;  
    [propName: string]: any;  
}  
  
function printStudent(st: Student){  
    console.log(st);  
}  
  
let student = {firstName:'pol', nickName1:'tee', nickName2:'pol'};  
printStudent(student);
```

จากตัวอย่างนี้จะเห็นแล้วสามารถเพิ่มสมาชิกได้เรื่อย ๆ ในไทป์ใด ๆ การพิมพ์ค่าเพื่ออ่านค่าก็พิมพ์ทั้งหมด จะใช้คำสั่งวนซ้ำเพื่ออ่านไม่ได้ในสมาชิกแบบทางเลือกนี้ จึงต้องทราบชื่อสมาชิก

อินเทอร์เฟซในรูปแบบดิชันนารี (Dictionary)

การสร้างสมาชิกแบบให้เพิ่มค่าสมาชิกได้เรื่อย ๆ ดูมีประโยชน์ นอกจากเพิ่มสมาชิกได้ แต่มีปัญหาในการอ่านต้องทราบชื่อสมาชิก แต่เราสามารถสร้างในรูปแบบดิชันนารี ที่สามารถค่าส่งวนซ้ำได้โดยไม่จำเป็นต้องทราบชื่อสมาชิก

Code 27.

```
interface JsonObj{  
    label: string;  
    data: number[];  
}  
  
interface TemplateIndex{
```

```

        [index: number]: JsonObject;
        length: number;
    }

    let t1: JsonObject = {label: 'cigarat', data:[1, 2, 3]};
    let t2: JsonObject = {label: 'car', data:[1, 2, 3]};
    let myTemplate: TemplateIndex = [ t1, t2 ];

    console.log(myTemplate.length); //print 2
    for(let i=0; i < myTemplate.length; i++) {
        console.log(myTemplate[i]);
    }

```

จากตัวอย่างนี้ใช้คำสั่งวนซ้ำ for กับ อินเทอร์เน็ต TemplateIndex ได้ โดยใช้ค่าขนาด (length) แทนขนาดของอาร์เรย์ ซึ่งนิยามชื่อ length ไว้ด้วย

ชนิดข้อมูลเจนเนอริก (Generic Type)

ในบางครั้งชนิดข้อมูลต้องการให้เปลี่ยนแปลงได้หลายอย่างขึ้นอยู่กับสถานการณ์การนำไปใช้งาน เช่น เมื่อต้องการให้อาร์เรย์ตัวหนึ่งเก็บค่าตัวเลข (number) ได้อย่างเดียว ทำให้เราสามารถที่จะดำเนินการในรูปตัวเลขได้ หรือถ้าต้องการให้อาร์เรย์นั้นเก็บได้เฉพาะตัวอักษร (string) ซึ่งก็ทำให้เราสามารถดำเนินการในรูปแบบอักษรได้ แต่จะเกิดอะไรขึ้นเมื่อเรากำหนดให้ข้อมูลเป็นชนิดอะไรก็ได้ (any) และคิดว่าทุกตัวเป็นตัวเลข ใช้วิธีการดำเนินการกับตัวเลข ลองพิจารณาตัวอย่างต่อไปนี้

Code 28.

```

var score: any = [1, "T", 2];
var sum: number = 0;
for (let s of score){
    sum += s;
}
console.log(sum); //output: 1T2

```

ในตัวอย่างนี้ความตั้งใจต้องการให้เกิดการบวกเลขรวมของทั้งอาร์เรย์ แต่มีตัวหนึ่งที่ไม่ใช่ชนิดตัวเลข ผลการทำงานกลายเป็นการต่ออักษร เพราะถือว่าการดำเนินการแบบอักษรแทน ด้วยเหตุนี้ นี่จึงเป็นเหตุผลหนึ่งที่ต้องจำกัดชนิดข้อมูลให้อยู่ในชนิดเดียวทั้งหมด การทำให้ชนิดข้อมูลเป็นเจนเนอริก คือการทำให้เป็นชนิดใดชนิดหนึ่งเท่านั้นแต่เปลี่ยนแปลงได้ตอนสร้างเป็นออบเจกต์

อย่างตัวอย่างที่ผ่านมา แทนที่จะใช้ชนิดเป็น any แต่ให้ระบุชนิดตั้งแต่สร้างอาร์เรย์ เป็น number

```

var numbers2: number[] = [1, 2, 3];
var numbers1: Array<number> = [1, 2, 3];

```

การใช้ Array<number> นี้ถือเป็นตัวอย่างการสร้างอาร์เรย์แบบเจนเนอริก โดยใช้เครื่องหมาย <T> โดย T แทนไทป์ (Type)

จากตัวอย่างที่ผ่านมาอาจมองไม่ค่อยเห็นประโยชน์ของชนิดเจนเนอริก แต่ถ้าต้องการให้มีที่เก็บของอย่างหนึ่ง (สมมุติว่าชื่อ shelves) เก็บรายการที่ยังไม่ได้รับทุนในต้นแบบ (สมมุติว่าเก็บชนิด T, โดย T เป็นชื่อสมมุติแทน Type) แต่จะระบุภายหลังเมื่อสร้างที่เก็บของ

Code 29.

```
function shelves<T>(id:number,items: T[]){
  return { id: id, items: items };
}
```

ฟังก์ชันนี้จะเห็นว่า ใช้ชนิด T เป็นชนิดที่จะอยู่ในฟังก์ชันนี้ โดยมีตัวแปรเข้าสองตัวคือ id และ items โดยตัวแปรหลังนี้ แทนอาร์เรย์ของ T ภายในฟังก์ชันนี้จะคืนค่าออบเจกต์ ของ id และ items (การคืนค่าแบบนี้ถือว่า มีชนิดเป็น any ได้)

ต่อมาสร้างคลาส Book แทนต้นแบบออบเจกต์หนังสือ และเรียกใช้ฟังก์ชัน selves<Book> โดยส่งค่า T เป็น Book ณ ขณะเรียกใช้ฟังก์ชัน

Code 30.

```
class Book{
  constructor(public title: string, public price: number) {}
  info(): string { return this.title + ":"+this.price;}
}
var books = shelves<Book>(1, [
  new Book("C#", 2000),
  new Book("VB", 150),
  new Book("Java", 230)
]);
console.log(books.id);
let sort_books = books.items.sort((a, b) => a.price - b.price);
for (let book of books.items) console.log(book.info());
```

ฟังก์ชัน shelves ต้องการตัวแปรเข้าสองตัวตามนิยามฟังก์ชันนี้ ตัวแปรเลขเป็นตัวเลข ส่วนตัวแปรที่สองเป็นอาร์เรย์ของ Book ซึ่งตรงกับ T[] ที่ได้นิยามไว้แล้ว

ต่อมาต้องการแสดงผลผ่าน console.log() การแสดงผลแรกแสดงเพียง id ส่วนการแสดงผลที่สอง แสดงข้อมูลของแต่ละรายการหนังสือผ่านฟังก์ชัน info() ซึ่งได้ทำการเรียงหนังสือตามราคา ผลลัพธ์การแสดงผลคือ :

```
1
VB:150
Java:230
C#:2000
```

ตัวอย่างที่ผ่านมา เราล่องหน้าแล้วผ่านการสร้าง books ให้ชนิดเป็น Book ทำให้เราดำเนินการข้อมูลในออบเจกต์ books ได้ ไม่ว่าจะเป็นการเรียง การแสดงผลให้เป็นเช่นไร

ลองพิจารณาอีกตัวอย่างในการสร้างออบเจกต์ selves ตัวใหม่กับ ที่เก็บเงินเนอริกคลาส จากเดิมที่ใช้คลาสทั่วไป

Code 31.

```
class Tool<T,U>{
  constructor(public title: T, public size: U) { }
}
```

คลาส `Tool<T,U>` นี้ใช้ชนิด `T` และ `U` เพื่อแทนสมาชิก `title` และ `size` ดังนั้นการสร้างการออบเจ็กต์ `Too` จึงต้องระบุสองชนิดว่าเป็นอะไร เช่น ต้องการให้ตัวแรกเป็น `string` และ ตัวที่สองเป็น `number` จะเขียนได้ว่า

```
var tool = new Tool<string, number>("screwdriver", 2)
```

เมื่อต้องการนำออบเจ็กต์ `tool` เข้าไปเก็บในออบเจ็กต์ `shelves` ซึ่งเป็นเจนเนอริกฟังก์ชัน ที่เคยสร้างก่อนหน้านี้ ก็จะสามารถเก็บได้ เพราะใช้ `T` แทนด้วย `Tool<>` ได้

Code 32.

```
var tools = shelves<Tool<string, number>>(1, [  
  new Tool("screwdriver", 2),  
  new Tool("screwdriver", 3),  
  new Tool("screwdriver", 3)  
]);
```

จากตัวอย่างนี้จะเห็นแล้วว่า การสร้างชนิดเป็นเจนเนอริกสามารถเก็บชนิด `T` ที่เปลี่ยนแปลงได้ตามสถานการณ์การใช้งาน ซึ่งต่อไปก็สามารถดำเนินกับ ชนิดข้อมูลที่เปลี่ยนได้เอง เพราะทราบล่วงหน้าแล้วแทน `T` ด้วยอะไร

เจนเนอริกคอลเล็กชัน (Generic collection)

คอลเล็กชันในที่นี้หมายถึงที่เก็บข้อมูล เหมือนกับที่เก็บในลักษณะอาร์เรย์ แต่มีลักษณะพิเศษกว่านั้นคือ เก็บข้อมูลที่เป็นเจนเนอริก คอลเล็กชันที่เก็บข้อมูลเจนเนอริกมีด้วยกันสองคลาสคือ `Set<T>` ซึ่งเก็บข้อมูลที่ไม่ซ้ำกัน และ `Map<K, V>` ซึ่งเก็บข้อมูลในค่าเป็นคู่กันโดยมี `K` เป็นคีย์และ `V` เป็นค่าของคีย์ เหมือนในลักษณะดัชนีนาฬิกา

ตาราง 10 ชื่อคุณสมบัติและฟังก์ชันของ `Set<T>` และ `Map(K, V)`

คลาส	ชื่อ	ความหมาย
Set<T>	size	ขนาด
	add(value)	เพิ่ม value
	values()	อ่านได้ในรูปแบบอาร์เรย์
Map<K, V>	size	ขนาด
	get(key)	อ่านตามค่า key
	set(key, value)	เพิ่ม key และ value

การใช้งานของคอลเล็กชันประเภทนี้ต้องสร้างออบเจ็กต์ก่อนซึ่งระบุไว้ว่าต้องการไทป์อะไร เมื่อได้ออบเจ็กต์แล้วก็สามารถนำไปดำเนินการต่อตามชื่อฟังก์ชันหรือชื่อคุณสมบัติ เช่น การสร้างออบเจ็กต์ `set` แล้วทำการเพิ่มค่าทีละค่า การเพิ่มค่าซ้ำจะทับค่าเดิม

```
let set = new Set<String>();  
set.add('pol').add('mon').add('tree').forEach(i=>console.log(i));
```

สำหรับตัวอย่างการใช้ Map<> นี้ยกตัวอย่างเป็นคลาส CollectionMap<> โดยมีคีย์เป็นค่าเดียวในไทป์ K ส่วนค่าตามคีย์ อยู่ในไทป์ V[] ซึ่งเป็นอาร์เรย์ของไทป์ V ณ ขณะนิยามคลาส ใช้ไทป์ที่ยังไม่ระบุไทป์อะไรที่แน่นอน จนกว่าจะสร้างเป็นออบเจกต์

Code 33.

```
class CollectionMap<K, V>{
  constructor(public dictionary: Map<K, V[]>){ }
  add(name: K, values: V[]): CollectionMap<K, V> {
    this.dictionary.set(name, values);
    return this;
  }
}

let collection = new CollectionMap(new Map<string, String[]>());
collection.add('cpu', ['amd', 'intel'])
           .add('os', ['unix', 'windows', 'linux'])
           .dictionary
           .forEach((v, k)=>console.log(`${k}: [${v}]`));
/*output:
cpu: [amd,intel]
os: [unix,windows,linux]
*/
```

เนมสเปส (Namespaces)

การบริหารคลาส ให้มีการจัดเก็บในลักษณะคล้ายๆ ห้องเก็บคลาส ต่างๆ อย่างเป็นระเบียบ TS ใช้ เนมสเปส ซึ่งรู้จักในรูปแบบโมดูลภายใน (internal module)

การประกาศเนมสเปสใช้คีย์เวิร์ด namespace เพื่อระบุชื่อเนมสเปส และ export ประกาศการใช้ให้คลาสอื่นรู้จักในชื่อเนมสเปสนี้

Code 34. book.ts

```
namespace MyBook{
  interface Logger{
    log(arg:any):void;
  }
  export
  class Book implements Logger{
    title:string;
    price:number;
    constructor(title:string, price:number){
      this.title = title;
      this.price = price;
    }
    info(){
      return this.title+":"+this.price;
    }
    log(agr:any){
      console.log(agr);
    }
  }
}
```



```
}  
}
```

สำหรับการใช้งานจะต่างจากเดิมเล็กน้อย คือ เราจะต้องใช้ ชื่อเนมสเปสขึ้นหน้าชื่อห้องก่อน เช่น `MyBook.Book()` ดังเช่นตัวอย่างต่อไปนี้

Code 35. index.html

```
<script src='book.js'></script>  
<script>  
  var book = new MyBook.Book('C#', 200);  
  alert(book.info());  
  book.log('hello');  
</script>
```

ด้วยวิธีการนี้ ทำให้เราแยกคลาสที่เหมือนกันได้ เพราะอยู่คนละเนมสเปส ซึ่งจำเป็นมากเมื่อโปรแกรมมีขนาดใหญ่ ใหม่ มีหลายคลาสที่อาจซ้ำกันได้ แต่อยู่คนละส่วนงานกัน

นอกจากนี้เรายังจะสามารถสร้างเนมสเปสที่ซ้อนกันได้ ทำให้การเรียกใช้ ใช้จุดคั่นแทนแต่ละชั้นของเนมสเปส เช่น ดังตัวอย่างต่อไปนี้

Code 36. app.ts

```
namespace app{  
  interface Logger{  
    log(arg:any):void;  
  }  
  export  
  namespace models{  
    export  
    class Book implements Logger{  
      title:string;  
      price:number;  
      constructor(title:string, price:number){  
        this.title = title;  
        this.price = price;  
      }  
      info(){  
        return this.title+":"+this.price;  
      }  
      log(agr:any){  
        console.log(agr);  
      }  
    }  
  }  
}
```

การเขียนซ้อนกันของเนมสเปส อีกวิธีหนึ่งใช้ จุดเชื่อมระหว่างสองเนมสเปสได้เลย เพื่อลดความซ้ำซ้อนในการอ่าน และจำนวนปีกกา เช่น

```
namespace app.models{  
  //..  
}
```

และตัวอย่างเรียกใช้งานก็ยังเหมือนกันทั้งสองวิธีการเขียน ดังเขียนได้ว่า :

Code 37. index.html

```
<script src='app.js'></script>
<script>
  var book = new app.models.Book('C#', 200);
  alert(book.info());
  book.log('hello');
</script>
```

โมดูล (Module)

แนวคิดการสร้างไฟล์ ให้เป็นโมดูล ซึ่งถือเป็นโมดูลภายนอก (external module) หรือหนึ่งไฟล์เป็นหนึ่งโมดูล แต่ที่ต่างกับเนมสเปซคือ โมดูล (ภายนอก) จะสามารถทำการ import โดยการใช้ HTML เช่น มีแหล่งไฟล์สองไฟล์ คือ book_module.js ทำหน้าที่เป็น แบบจำลองข้อมูล หรือต้องการให้เป็นโมดูลหนึ่งที่เกี่ยวข้องกับการทำงานกับ book และ main.js ที่เป็นไฟล์หลักที่ใช้ในการเรียกใช้งานโมดูล

Code 38. index.html

```
<!-- DOCTYPE html -->
<script type="module" src="book_module.js"> </script>
<script type='module' src='main.js'></script>
```

Code 39. main.js

```
import {Book} from './book_module.js';
var book = new Book('C#', 200);
console.log('test module');
console.log(book.info());
book.log('hello');
```

การเขียนใช้งานคลาสระหว่างโมดูล ใช้คำสำคัญ import เช่น ไฟล์ หรือโมดูล main.js เรียกใช้ book_module.js การใช้คำสั่ง import { ชื่อคลาส หรือ ชื่อตัวแปรที่มีการ export } และตามด้วยชื่อไฟล์ ดังเห็นในตัวอย่างนี้

จาก ไฟล์ JS ข้างต้น มาจากการแปลงจากไฟล์ TS ที่เขียนแบบ โมดูล (ไม่มีเนมสเปซ) ใช้การ export ชื่อคลาส หรือ ชื่อตัวแปรอื่น ๆ ที่ต้องการให้โมดูลอื่นเรียกใช้ได้

Code 40. book_module.ts

```
interface Logger{
  log(arg:any):void;
}
class Book implements Logger{
  title:string;
  price:number;
  constructor(title:string, price:number){
    this.title = title;
    this.price = price;
  }
  info(){
    return this.title+": "+this.price;
  }
}
```

```

    }
    log(agr:any){
        console.log(agr);
    }
}
export {Book}

```

หลังจากถูกสร้างเป็นไฟล์ JS ซึ่งต้องดัดแปลงบางส่วน ให้มีลักษณะดังต่อไปนี้

Code 41. book_module.js

```

var Book = /** @class */ (function () {
    function Book(title, price) {
        this.title = title;
        this.price = price;
    }
    Book.prototype.info = function () {
        return this.title + ":" + this.price;
    };
    Book.prototype.log = function (agr) {
        console.log(agr);
    };
    return Book;
})();
export {Book};

```

การเขียนแบบโมดูล(ภายนอก) นิยมเขียนกันมากกว่าการใช้แบบเนมสเปส เพราะการแยกไฟล์ทำให้เป็นอิสระต่อกัน ชัดเจน มากกว่าการแยกโมดูลภายในของเนมสเปส และป้องกันความสับสนของการใช้โมดูลแบบใดกันแน่

สรุป

การใช้ TS แทนการเขียน JS ถือว่าจะช่วยโปรแกรมมิ่งถูกเกณฑ์ที่เป็นระเบียบแบบแผนมากกว่า JS เพราะ JS เองมีหลายรุ่น ช่วยลดความไม่เข้ากันกับเบราว์เซอร์รุ่นต่างๆ ได้ดี ในบทนี้เป็นอธิบายการใช้ TS อย่างย่อซึ่งก็เพียงพอที่นำไปใช้งานขั้นพื้นฐานได้ ไม่ว่าจะเป็นนำไปใช้เขียนกับ Angular ซึ่งเป็น เฟรมเวิร์กเว็บประยุกต์ตัวหนึ่งที่ได้รับคามนิยม เนื้อหาการสร้างฟังก์ชัน การสร้างคลาส เป็นเรื่องสำคัญที่พบใน Angular เพราะทุกคอมโพเน้นท์ของ Angular เป็นคลาสทั้งสิ้น การสร้างตัวแปรในคอนสตรัคเตอร์ก็เป็นเรื่องใหม่ที่ต่างกับภาษาอื่น นอกจากนี้ยังอธิบายถึงการสืบทอดทั้งที่มาจากคลาส และอินเทอร์เฟส ซึ่งเป็นคุณสมบัติที่มีในการเขียนโปรแกรมเชิงวัตถุ การใช้งานอาร์เรย์ก็มีฟังก์ชันอำนวยความสะดวกและใช้งานง่าย การใส่ตัวแปรเข้าของฟังก์ชันเป็นแลมบ์ดา ก็เพื่อเรื่องทั่วไปที่นิยมใช้งานกัน ความสามารถที่จะรับค่าชนิดข้อมูลได้แตกต่างกันตามการสร้างออบเจกต์ ด้วยการใส่คุณสมบัติ เจนเนอริก ก็ช่วยสร้างเป็นไดนามิกของออบเจกต์ได้ และการใช้งานโมดูลก็เป็นส่วนสำคัญ เพราะแนวโน้มการเขียนโปรแกรมเน้นการใช้งานเป็นโมดูล มากกว่าที่จะเป็น เนมสเปส ดังพบใน Angular

คำถามทบทวน

1. ไทป์เป็น null มีอีกชื่อแทนกันได้อะไร
2. การสร้างไทป์ขึ้นมาเอง ใช้ศัพท์เวิร์ดอะไร
3. อะไรคือความแตกต่างระหว่างการประกาศตัวแปรที่เป็น let กับ var
4. ตัวดำเนินการ === มีความหมายอย่างไร

- เมื่อใดที่ใช้คำสั่งแบบ for .. in
- เหตุใดตัวแปรแบบทางเลือกต้องอยู่เป็นตัวสุดท้ายเสมอ
- การสร้างแลมบ์ดา ถ้ามีคำสั่งเพียงบรรทัดเดียวภายใต้แลมบ์ดา จำเป็นต้องมีคีย์เวิร์ด return หรือไม่ และเมื่อใดจำเป็นต้องมีคีย์เวิร์ด return
- การสร้างคลาสให้มีสมาชิกเพิ่มทันทีในวงเล็บของคอนสตรักเตอร์ มีข้อดีอะไร และถ้าไม่มีคีย์เวิร์ด public หรือ private จะเกิดอะไรขึ้น
- ข้อดีของการสร้างข้อมูลในรูปแบบเจนเนอริกคืออะไร
- อะไรคือความแตกต่างระหว่าง Set<> กับ Map<>

แบบฝึกหัด

- จงเขียน แสดงข้อมูล (value) ทั้งหมดของ ตัวแปร anyObj

```
var books = new Array('Java', 'C#', 'VB');
var anyObj = [1, 'ABC', books];
```
- จงเขียน แสดงข้อมูล (value) ทั้งหมดของ ตัวแปร anyObj จากโจทย์ข้อ 1 ในรูป แลมบ์ดา
- จงสร้างคลาส Books มีสมาชิก id, title, price
- จงสร้างอาร์เรย์ ชื่อ books เพื่อเก็บออบเจกต์ Book 3 ตัว โดยมีข้อมูล id, name, price กำหนดค่าของข้อมูลคือ (1, Java, 200), (2, C#, 199), (3, VB, 300) ตามลำดับ แล้วทำการวนซ้ำเพื่ออ่านข้อมูลในอาร์เรย์ทั้งหมด
- จากข้อ 4 ให้ทำการสืบค้นเฉพาะหนังสือที่มีราคา มากกว่า 200 บาท
- จากเจนเนอริกคลาส ต่อไปนี้ให้แปลงเป็น เจนเนอริกฟังก์ชัน

```
class Tool<T,U>{
    constructor(public title: T, public size: U) { }
}
```
- สร้างคลาส EnglishBook ที่สืบทอดจากคลาส Book โดยทั้งสองคลาสนี้อยู่คนละไฟล์กัน กำหนดสมาชิกได้เอง แต่มีอย่างน้อยคลาสละสองสมาชิก และทดสอบเรียกใช้งานทั้งสองคลาส ผ่านไฟล์ html

อ้างอิง

- Code Compiled: Arrays in TypeScript. (2018 Oct., 14). <http://www.codecompiled.com/arrays-in-typescriptJansen>
- R. H. (2015). *Learning TypeScript*. Packt Publishing Ltd.
- Ken Dale: TypeScript Constructor Assignment. (2018 Oct., 14). <https://kendaleiv.com/typescript-constructor-assignment-public-and-private-keywords>
- TypeScript: Document. (2018 Oct., 3). <https://www.typescriptlang.org/docs/home.html>