JavaScript Functional Programming

(Integrative-Generative AI Edition)

Contents:

Student Price Book Center

ai

คำนำ

ในการพัฒนาโปรแกรมยุคใหม่ แนวคิดแบบ Functional Programming (FP) ได้กลับมาได้รับความนิยมอีกครั้ง โดยเฉพาะเมื่อภาษาอย่าง JavaScript ได้กลายเป็นภาษาหลักในโลกของ Web และ Software Development การเขียนโปรแกรมแบบ FP นั้นเน้นเรื่อง “ความบริสุทธิ์” ของฟังก์ชัน (Pure Functions), การไม่เปลี่ยนแปลงข้อมูล (Immutability) และการหลีกเลี่ยงผลข้างเคียง (Side Effects) ซึ่งช่วยให้โค้ดมีความอ่านง่าย ทดสอบง่าย และบำรุงรักษาได้ในระยะยาว

หนังสือเล่มนี้ถูกออกแบบมาเพื่อนำผู้อ่านเข้าสู่โลกของ Functional Programming ด้วย JavaScript โดยเริ่มตั้งแต่ **ระดับเริ่มต้น (Naive)** ที่เน้นปูพื้นฐานของภาษา JavaScript และแนวคิด FP อย่างเป็นระบบ พร้อมด้วยตัวอย่างฟังก์ชัน การจัดการข้อมูล และเทคนิคการเขียนโปรแกรมเชิงฟังก์ชันแบบค่อยเป็นค่อยไป จนกระทั่งพัฒนาความเข้าใจสู่ **ระดับมืออาชีพ (Professional)** ที่ครอบคลุมแนวคิดขั้นสูง เช่น Currying, Functors, Monads, และการออกแบบโปรเจกต์จริงด้วย FP
ผู้อ่านจะได้เรียนรู้ทั้งในเชิงทฤษฎีและภาคปฏิบัติ ด้วยตัวอย่างโค้ดจริง การทดสอบแบบ Functional, การจัดการ Asynchronous Functions แบบ FP, รวมถึงการใช้ไลบรารีที่ได้รับความนิยมอย่าง Ramda, Lodash/fp และ Sanctuary ซึ่งจะช่วยให้โค้ดของคุณมีความสะอาด (clean), เข้าใจง่าย และปลอดภัยมากขึ้น

นอกจากนี้ หนังสือยังรวมถึงการวิเคราะห์ปัญหาและการออกแบบแอปพลิเคชันแบบ Functional ผ่านกรณีศึกษา เช่นการสร้าง Todo List แบบ FP, การจัดการ State ด้วยแนวคิด Pure Function และแนวทางปฏิบัติที่ดีที่สุด (Best Practices) สำหรับนักพัฒนาที่ต้องการยกระดับความสามารถไปสู่ระดับมืออาชีพ

หวังว่าผู้อ่านจะสามารถนำแนวคิดและเทคนิคที่ได้จากหนังสือเล่มนี้ไปต่อยอดกับโปรเจกต์จริง หรือใช้ในการพัฒนาระบบ Software ที่ซับซ้อนอย่างยั่งยืน และหากคุณกำลังมองหาวิธีคิดใหม่ในการเขียนโค้ด JavaScript — Functional Programming อาจเป็นกุญแจสำคัญที่จะเปลี่ยนแนวทางการพัฒนาของคุณตลอดไป

ด้วยความปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 1

พื้นฐาน JavaScript และแนวคิด Functional Programming

(JavaScript and FP Basic)

เนื้อหา

บทนำบทที่ 1: พื้นฐาน JavaScript และแนวคิด Functional Programming

การเขียนโปรแกรมในภาษา JavaScript ถือเป็นทักษะที่จำเป็นในยุคปัจจุบัน โดยเฉพาะอย่างยิ่งเมื่อ JavaScript ได้กลายเป็นภาษาเอกของเว็บแอปพลิเคชัน การเริ่มต้นเรียนรู้ JavaScript อย่างถูกต้องจึงเป็นก้าวสำคัญที่จะนำไปสู่การพัฒนาโปรแกรมที่มีคุณภาพ ในบทนี้ ผู้อ่านจะได้ทำความรู้จักกับโครงสร้างพื้นฐานของ JavaScript โดยเฉพาะเรื่องฟังก์ชันซึ่งเป็นหัวใจสำคัญของการเขียนโปรแกรมแบบ Functional Programming

ฟังก์ชันใน JavaScript มีหลายรูปแบบ เช่น การประกาศแบบธรรมดา (Function Declaration), การกำหนดแบบนิพจน์ (Function Expression), และการใช้ลูกศร (Arrow Function) ซึ่งแต่ละรูปแบบมีคุณสมบัติและบริบทการทำงานที่แตกต่างกัน การเข้าใจกลไกของฟังก์ชันเหล่านี้อย่างชัดเจนจึงเป็นพื้นฐานที่ขาดไม่ได้ก่อนจะเข้าสู่แนวคิด FP

Functional Programming หรือ FP เป็นแนวทางในการเขียนโปรแกรมที่เน้นให้ “ฟังก์ชัน” เป็นศูนย์กลางของการคำนวณ โดยทุกสิ่งถูกมองเป็นการแปลงค่าจากอินพุตสู่เอาต์พุต โดยไม่มีการเปลี่ยนแปลงสถานะภายใน (State) หรือมีผลข้างเคียง (Side Effects) แนวคิดนี้แตกต่างจากการเขียนโปรแกรมเชิงขั้นตอน (Procedural) ที่เน้นการสั่งงานตามลำดับคำสั่ง และแตกต่างจากแนวคิดเชิงวัตถุ (OOP) ที่เน้นการรวมข้อมูลกับพฤติกรรมไว้ในวัตถุ

หนึ่งในแนวคิดหลักของ FP คือ **Pure Function** หรือฟังก์ชันที่ให้ผลลัพธ์เหมือนเดิมทุกครั้งเมื่อป้อนอินพุตเดียวกัน โดยไม่มีการเปลี่ยนแปลงค่าหรือสถานะภายนอก ฟังก์ชันประเภทนี้ง่ายต่อการทดสอบและปรับปรุง ซึ่งช่วยลดข้อผิดพลาดและเพิ่มความน่าเชื่อถือของระบบที่เขียนขึ้น
อีกหนึ่งหลักการที่สำคัญคือ **Immutability** หรือการไม่เปลี่ยนแปลงค่าของข้อมูลดั้งเดิม โดยใน FP ข้อมูลจะถูกสร้างใหม่ทุกครั้งที่มีการเปลี่ยนแปลง แทนที่จะไปแก้ไขค่าที่มีอยู่ ซึ่งทำให้การวิเคราะห์พฤติกรรมของโปรแกรมทำได้ง่ายขึ้น และลดความซับซ้อนที่เกิดจากการเปลี่ยนแปลงสถานะของตัวแปร

Side Effects หรือผลข้างเคียง คือสิ่งที่เกิดขึ้นนอกเหนือจากการคำนวณของฟังก์ชัน เช่น การแก้ไขตัวแปรนอกฟังก์ชัน, การเขียนไฟล์, หรือการเรียก HTTP request ซึ่ง FP มองว่าผลข้างเคียงเหล่านี้ควรถูกควบคุมหรือแยกออกจากส่วนหลักของโปรแกรมเพื่อความง่ายในการทดสอบและปรับปรุง บทนี้จึงถือเป็นรากฐานของแนวทาง Functional Programming ด้วย JavaScript โดยเนื้อหาทั้งหมดจะช่วยให้ผู้อ่านมีความเข้าใจที่มั่นคงเกี่ยวกับโครงสร้างฟังก์ชัน พฤติกรรมของโค้ดแบบ pure และการจัดการข้อมูลแบบ immutable ซึ่งจะถูกต่อยอดในบทถัดไปสู่การเขียนโปรแกรมที่ยืดหยุ่น ปลอดภัย และมีประสิทธิภาพยิ่งขึ้น

พื้นฐาน JavaScript และแนวคิด Functional Programming

⚡1 □ JavaScript เบื้องต้น

JavaScript เป็นภาษาโปรแกรม **แบบไดนามิก (Dynamic)** ที่นิยมใช้ทั้งบนเว็บ (ฝั่ง **Client**) และฝั่ง **Server** (ผ่าน Node.js)
มีลักษณะสำคัญคือ:

- เป็น **Interpreted Language** (รันตรง ๆ ใน Interpreter)
- มี **First-Class Functions** ⟶ ฟังก์ชันเป็นค่าชนิดหนึ่งที่เก็บเป็นตัวแปรได้
- รองรับทั้ง **Imperative** (คำสั่งตรง ๆ), **Object-oriented** (เชิงอ็อบเจกต์), และ **Functional** (เชิงฟังก์ชัน)

□ 2□□ ฟังก์ชันใน JavaScript

2.1 □ ฟังก์ชันแบบ Declaration

```
function add(a, b) {
  return a + b;
}
```

- ประกาศตรง ๆ
- ถูก hoist ขึ้นมาด้านบน (เรียกใช้ได้แม้ก่อนประกาศ)

2.2 ⚡□ ฟังก์ชันแบบ Expression

```
const multiply = function(a, b) {
```

```
  return a * b;
};
```

- กำหนดเป็นค่าของตัวแปร
- ไม่ hoist
- มีประโยชน์เมื่อต้องการส่งเป็น callback

2.3 ➡ ฟังก์ชันแบบ Arrow Functions

```
const subtract = (a, b) => a - b;


const complexOperation = (a, b) => {
  const result = a * b + 10;
  return result;
};
```

- สั้น กระชับ
- มี this เป็น lexical context (ใช้ค่าของ this จาก scope ด้านนอก)

---

3 แนวคิด Functional Programming คืออะไร

เป็นรูปแบบการเขียนโปรแกรมที่ให้ **"ฟังก์ชัน" เป็นศูนย์กลาง**

(Function-Centered Programming)

หลักการสำคัญ:

ใช้ **Pure Functions** (ฟังก์ชันบริสุทธิ์)

หลีกเลี่ยง **Side Effects** (ผลข้างเคียง)

ยึด **Immutability** (ข้อมูลต้องไม่แก้ไข แต่ต้องแทนค่าด้วยข้อมูลใหม่)

---

4 ความแตกต่าง Procedural vs OOP vs FP

รูปแบบ	แนวคิดหลัก	ตัวอย่าง
Procedural	โครงสร้างตามลำดับคำสั่ง (Statement)	if/else, for()
OOP	จัดโครงสร้างด้วย **คลาส, อ็อบเจกต์, เมธอด**	class Person { ... }
Functional	จัดโครงสร้างด้วย **ฟังก์ชัน**เป็นหน่วยงาน	const add = (a, b) => a + b

ตัวอย่าง

```
// Procedural
let total = 0;
for (let i = 1; i <= 3; i++) {
```

```
  total += i;
}

// OOP
class Calculator {
  constructor() {
    this.total = 0;
  }
  add(value) {
    this.total += value;
  }
}
const calc = new Calculator();
calc.add(1);
calc.add(2);

// Functional
const numbers = [1, 2, 3];
const totalFP = numbers.reduce((sum, n) => sum + n, 0);
```

□ 5□ Pure Functions — ฟังก์ชันบริสุทธิ์

คุณสมบัติ:

□ ให้ผลลัพธ์เดียวกันเสมอเมื่อ input เดิม

□ ไม่มี **side effects** (อ่าน/เขียนค่าตัวแปรภายนอก)

ตัวอย่าง

```
// □ Pure
const add = (a, b) => a + b;

// □ Impure
let counter = 0;
const increment = () => {
  counter++; // side effect
  return counter;
};
```

♻ 6 Immutability — ข้อมูลต้องไม่เปลี่ยนแปลง

ตัวอย่าง

```
// ❌ Mutable
const person = { name: "Alice", age: 25 };
person.age = 26; // แก้โดยตรง

// ✅ Immutable
const person2 = { name: "Alice", age: 25 };
const updatedPerson = { ...person2, age: 26 };
```

⚡ 7 Side Effects คืออะไร และทำไมต้องหลีกเลี่ยง

Side Effect

คือการที่ฟังก์ชัน **แก้ไขค่าตัวแปรภายนอก**, **อ่านไฟล์**, **แก้ DOM**, **เขียนลง console**, **ส่ง request**, ฯลฯ

ตัวอย่าง

```
let total = 0;

function addNumber(n) {
  total += n; // ❌ side effect
}
```

ทำไมต้องหลีกเลี่ยง

- ทำให้โค้ด **ทดสอบยาก**
- จัดการ state ยาก
- ผลลัพธ์ฟังก์ชัน **คาดเดาไม่ได้**

⚡ สรุป

หัวข้อ	สรุป
JavaScript	เป็นภาษาหลักบนเว็บ มีทั้ง OOP, Procedural และ FP
ฟังก์ชัน	มีทั้ง Declaration, Expression และ Arrow Functions
FP คือ	แนวคิดเขียนโค้ดแบบ Pure Functions และหลีกเลี่ยง Side Effects
Procedural vs OOP vs	Procedural = คำสั่งเป็นลำดับ, OOP = จัดกลุ่มเป็น Objects, FP = จัดเป็น

หัวข้อ	สรุป
FP	Functions
Pure Functions	ให้ผลลัพธ์เดิมจาก input เดิม ไม่มี Side Effect
Immutability	ข้อมูลต้องไม่แก้ไขตรง ๆ แต่ต้อง **แทนที่** ด้วยค่าใหม่
Side Effects	ผลลัพธ์ข้างเคียง (อ่าน/เขียนค่าตัวแปรภายนอก) ควรหลีกเลี่ยง

แนวคิดเชิงลึก FP

⚡ 1 ฟังก์ชันเป็น First-Class Citizens

JavaScript treats functions as **values**:

- ส่งเป็น parameter ให้ฟังก์ชันอื่นได้
- คืนค่ากลับเป็นฟังก์ชันได้
- เก็บไว้ในตัวแปรได้

 นี่คือหัวใจของ FP — เพราะฟังก์ชันเป็นค่า เราใช้เป็น “building block” ในการจัดการตรรกะได้

```
// Example: Higher-Order Function
const applyTwice = (fn, value) => fn(fn(value));

// Usage
const double = x => x * 2;

console.log(applyTwice(double, 5)); // 20
```

♻ 2 Pure Functions

Insight: Pure Functions เป็นรากฐานของ FP เพราะ:

- **Reusability** — นำไปใช้ซ้ำได้โดยไม่ต้องกังวล state
- **Testability** — ทดสอบได้ง่ายเพราะไม่มี state แฝง
- **Referential Transparency** — แทนค่าฟังก์ชันด้วยค่ากลับของมันได้

 Pure = f(a) => b เสมอ

ตัวอย่าง Pure

```
const add = (a, b) => a + b;

add(1, 2); // 3 (ค่าตายตัว), ไม่มี Side Effect
```

ตัวอย่าง Impure

```
let counter = 0;

function increment() {
  counter++;
  return counter; // แต่ counter เปลี่ยนค่าได้ตลอด
}
```

⚡ **3 Immutability**

ใน FP:

- ข้อมูลเป็นสิ่งต้อง “รักษา” (Preserve) ไม่แก้ไข
- ทุกครั้งต้อง **สร้างใหม่** ไม่แก้ค่าต้นฉบับ
- ช่วยหลีกเลี่ยงข้อบกพร่องจาก state ที่แชร์ (Shared State)

ตัวอย่าง

```
const person = { name: "Alice", age: 25 };
const olderPerson = { ...person, age: person.age + 1 };
```

olderPerson เป็นอ็อบเจกต์ใหม่

person ไม่ถูกแก้ไข

⚔ **4 Side Effects**

สิ่งที่เป็น Side Effect:

- เขียน/อ่านไฟล์
- ดึงข้อมูลจาก API
- แก้ DOM
- แก้ค่าตัวแปรนอกฟังก์ชัน
- แสดง console.log() (ถึงจะเล็ก แต่เป็น Side Effect)

ผลเสีย

- ฟังก์ชันต้องรู้บริบทภายนอก
- ยากต่อการทดสอบและ debug
- ผลลัพธ์ขึ้นอยู่กับ state แวดล้อม

Functional Pattern: แยก Pure Logic (Core Logic) กับ Side Effect (I/O) ให้ชัดเจน

➡ โครงสร้างทั่วไป:

```
Pure Logic ---> Returns Result
Side Effect Logic ---> Uses Result
```

□ 5 □ Functional Composition

คือแนวคิด “**ต่อฟังก์ชัน**” เพื่อแก้ปัญหาทีละขั้น

- เล็ก ๆ
- Pure
- มี input ➔ output ชัดเจน

ตัวอย่าง

```
// ฟังก์ชันเล็ก ๆ
const add = x => x + 1;
const multiply = x => x * 2;

const compose = (f, g) => x => f(g(x));

// ใช้ compose
const addThenMultiply = compose(multiply, add);
console.log(addThenMultiply(3)); // ผลลัพธ์ = multiply(add(3)) = multiply(4) = 8
```

□ ยืดหยุ่น ➔ สร้างใหม่ได้ตลอด

□ ทดสอบเป็นส่วน ๆ ได้

□ อ่านเข้าใจเป็นลำดับได้

⚡□ 6 □ Declarative vs Imperative

Imperative

บอกว่า “ต้องทำอย่างไร”

(Step by step)

```
let total = 0;
for (let i = 0; i < 5; i++) {
  total += i;
}
```

Declarative

บอกว่า “ต้องการอะไร”

(What)

```
const total = [0, 1, 2, 3, 4].reduce((sum, i) => sum + i, 0);
```

Declarative Style = FP Style

□ เข้าใจเจตนาได้ชัดเจน

- โค้ดสั้น
- โอกาสเกิดข้อผิดพลาดน้อย

⚡ 7 สรุปภาพใหญ่

หัวข้อ	FP Insight
Pure Functions	ฟังก์ชันต้องให้ค่ากลับเท่ากันเสมอและไม่มี Side Effect
Immutability	หลีกเลี่ยงการแก้ค่าเดิม สร้างใหม่แทน
Side Effect	เลี่ยงและควบคุมให้น้อยที่สุด
Higher-Order Functions	ใช้ฟังก์ชันเป็นค่า จัดการตรรกะได้หลากหลาย
Composition	ต่อฟังก์ชันเล็ก ๆ เป็นฟังก์ชันใหญ่
Declarative	บอก **What** มากกว่า **How**
ผลลัพธ์	โค้ดอ่านง่าย ทดสอบได้ เป็นโมดูล และทนทานต่อ bug

⚡ สรุปสุดท้าย

- **JavaScript + FP** = โค้ดที่เข้าใจง่าย, บำรุงรักษาง่าย, ปลอดภัย
- หัวใจ = Pure Functions + Immutability + Composition
- หลีกเลี่ยง Side Effect เพื่อให้ควบคุมได้และทดสอบได้
- เปลี่ยนจาก “ต้องทำอย่างไร” เป็น “ต้องการอะไร” ➔ ยั่งยืนและอ่านง่าย

อธิบาย JavaScript เพิ่มเติม

⚡ 1 JavaScript คืออะไร?

- เป็น **ภาษาโปรแกรม** ที่เดิมถูกออกแบบให้ทำงานบน **เว็บเบราว์เซอร์** เพื่อควบคุมและโต้ตอบกับผู้ใช้
- ปัจจุบันเป็น **General Purpose Language** ใช้ทั้งบนเซิร์ฟเวอร์ (Node.js), Mobile App (React Native), หรือ AI/IoT
- เป็น **Dynamic Language**:
  - ไม่มีต้องประกาศชนิดตัวแปรชัดเจน
  - ตัวแปรยืดหยุ่นและเป็นค่าใด ๆ ได้

⚡ 2 พื้นฐานของ JavaScript

☐ ตัวแปร

มี 3 แบบ:

- var (เก่า): มี scope เป็น function
- let (ใหม่): มี scope เป็น block (นิยมใช้มากกว่า var)
- const: ค่าต้อง **ถูกตั้งและแก้ไขค่าเดิมไม่ได้**

```
let name = "Alice";       // เปลี่ยนค่าได้
const pi = 3.1415926535; // เปลี่ยนค่าไม่ได้
```

☐ ชนิดข้อมูล (Data Types)

JavaScript มีชนิดข้อมูลดังนี้:

- **Primitive**: Number, String, Boolean, null, undefined, Symbol, BigInt
- **Object**: Object, Array, Function

ตัวอย่าง

```
let num = 42;             // Number
let str = "Hello";        // String
let isActive = true;      // Boolean
let data = null;          // null
let person = { name: "A" }; // Object
let arr = [1, 2, 3];      // Array
let sayHello = function() { console.log("Hello"); }; // Function
```

☐ โครงสร้างควบคุม

Conditional

```
if (age >= 18) {
  console.log("ผู้ใหญ่");
} else {
  console.log("เด็ก");
}
```

Loop

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

☐ ฟังก์ชัน

1️⃣ Declaration

```
function add(a, b) {
  return a + b;
}
```

2️⃣ Expression

```
const multiply = function(a, b) {
  return a * b;
};
```

3️⃣ Arrow Function

```
const subtract = (a, b) => a - b;
```

⚡ 3️⃣ แนวคิดสำคัญของ JavaScript

🔹 เป็นภาษา Event-Driven

- JavaScript ตอบสนองต่อ **event** เช่น การคลิก, การเลื่อนเมาส์
- รอและตอบสนองโดยใช้ **callback**, **promise**, และ **async/await**

🔹 เป็น Single-Thread

- มี **event loop** จัดลำดับงาน
- จัดการงานแบบ **asynchronous** ได้โดย callback queue

🔹 มี Prototype-based Inheritance

- วัตถุ (object) สืบทอดคุณสมบัติผ่าน **prototype** (ไม่มี class จริง ๆ จน ES6)

🔹 เป็น Functional-first

- ฟังก์ชันเป็น **first-class citizen** (ส่งเป็น parameter, return เป็นค่ากลับได้)

⚡ 4️⃣ ตัวอย่างจริง

โค้ดแบบ Procedural

```
let nums = [1, 2, 3];
let total = 0;

for (let i = 0; i < nums.length; i++) {
  total += nums[i];
}
console.log(total); // 6
```

โค้ดแบบ Functional

```
let nums = [1, 2, 3];
let total = nums.reduce((sum, n) => sum + n, 0);
console.log(total); // 6
```

- แม้ JavaScript จะเป็น multi-paradigm แต่รองรับ Functional Programming ได้ดีมาก

⚡ 5. สรุป

- JavaScript เป็นภาษาที่ยืดหยุ่นสูง: ใช้ได้ทั้ง **Procedural**, **OOP**, **Functional**
- ตัวแปรเป็น dynamic typing (ไม่ต้องประกาศชนิด) แต่ต้องเข้าใจกฎ scoping (let, const, var)
- ฟังก์ชันเป็นหัวใจของ JavaScript และเป็น “ค่าหนึ่ง” ที่ใช้เป็นอาร์กิวเมนต์หรือค่ากลับได้
- มี **event loop** จัดการ asynchronous programming

ฟังก์ชันใน JavaScript

ฟังก์ชันใน JavaScript ทั้ง 3 รูปแบบที่ใช้บ่อย:

⚡ 1. Function Declaration

- ประกาศฟังก์ชันแบบ “ตรง ๆ”
- ถูก Hoist ขึ้นไปไว้บนสุดของ Scope จึง **เรียกใช้ได้ แม้จะอยู่ก่อนการประกาศ**

โครงสร้าง

```
function add(a, b) {
  return a + b;
}
```

ตัวอย่าง

```
// เรียกใช้ได้ แม้จะอยู่บนสุด
console.log(add(2, 3)); // 5

function add(a, b) {
  return a + b;
}
```

⚡ 2. Function Expression

- ประกาศเป็นค่าตัวแปร
- ไม่มี Hoist ตัวฟังก์ชันเอง
- **ต้องประกาศก่อน** จึงจะเรียกใช้ได้

โครงสร้าง

```
const multiply = function(a, b) {
  return a * b;
};
```

ตัวอย่าง

```
// console.log(multiply(2, 3)); //  Error: multiply is not defined
const multiply = function(a, b) {
  return a * b;
};
console.log(multiply(2, 3)); //  6
```

---

 3 Arrow Functions

 เป็น **Syntax แบบใหม่** ตั้งแต่ ES6

 สั้น กระชับ

 ไม่มี this ของตัวเอง แต่ใช้ this จาก scope ด้านนอก (lexical this)

 เหมาะเป็น callback / method ขนาดเล็ก

โครงสร้าง

```
const subtract = (a, b) => {
  return a - b;
};
```

ตัวอย่างแบบย่อ

```
const subtract = (a, b) => a - b;

console.log(subtract(5, 3)); // 2
```

แบบย่อสุด ๆ

- ถ้า **มี parameter ตัวเดียว** และ **return ค่าตรง ๆ**

```
const square = x => x * x;

console.log(square(4)); // 16
```

⚡ **สรุปและเทียบ 3 รูปแบบ**

ประเภท	ตัวอย่าง	Hoist	มี this ของตัวเอง?
Declaration	function add(a, b) {...}		

ประเภท	ตัวอย่าง	Hoist	มี this ของตัวเอง?
Expression	const multiply = function(a,b){...}	□	□
Arrow Function	const subtract = (a,b) => a-b	□	□ (ใช้ this จาก Scope นอก)

⚡□ คำแนะนำการเลือกใช้

□ **Function Declaration**

- ดีเมื่อต้องใช้เป็น “ชื่อฟังก์ชัน” และต้องการ hoist
- เหมาะเป็นฟังก์ชันทั่วไปของโปรแกรม

□ **Function Expression**

- ดีเมื่อใช้เป็น callback หรือเป็น method ใน object
- ยืดหยุ่นเพราะเก็บเป็นค่าตัวแปรได้

□ **Arrow Functions**

- ดีเมื่อใช้เป็น callback เล็ก ๆ หรือใน FP-style (map, filter, reduce)
- ชัดเจน สั้น แต่ต้องเข้าใจกลไก this

⚡□ ตัวอย่างเลือกใช้จริง

```
// Declaration: ฟังก์ชันทั่วไป
function greet(name) {
  return `Hello, ${name}!`;
}

// Expression: ฟังก์ชันเก็บในตัวแปร
const isEven = function(n) {
  return n % 2 === 0;
};

// Arrow: ฟังก์ชัน callback
const doubled = [1, 2, 3].map(x => x * 2);

console.log(greet("Alice"));     // Hello, Alice!
console.log(isEven(4));          // true
console.log(doubled);            // [2, 4, 6]
```

แนวคิด Functional Programming (FP) คืออะไร?

□ **Functional Programming (การเขียนโปรแกรมเชิงฟังก์ชัน)**

คือ **รูปแบบ (Paradigm)** ของการเขียนโค้ดโดยเน้นใช้ **ฟังก์ชันเป็นศูนย์กลาง**

และให้ **ข้อมูลไหลผ่านฟังก์ชัน** เพื่อสร้างผลลัพธ์ใหม่โดยหลีกเลี่ยง **side effects** และ **การแก้ไขค่าต้นฉบับ**

□ **หลักการสำคัญของ FP**

□ **1 □ Pure Functions (ฟังก์ชันบริสุทธิ์):**

- ผลลัพธ์ต้องขึ้นอยู่กับ input เท่านั้น
- ไม่มี Side Effect (อ่าน/เขียนค่าตัวแปรภายนอก)

```
// □ Pure
const add = (a, b) => a + b;

// □ Not Pure (ใช้ค่าภายนอกและแก้ค่า global)
let counter = 0;
const addImpure = (a) => {
  counter += a;
  return counter;
};
```

□ **2 □ Immutability (ค่าที่แก้ไขไม่ได้):**

- ข้อมูลต้องไม่แก้ไขตรง ๆ แต่ต้อง **สร้างใหม่**
- ป้องกันข้อผิดพลาดจากการแก้ค่าต้นฉบับ

```
// □ Immutable
const person = { name: "Alice", age: 25 };
const olderPerson = { ...person, age: person.age + 1 };

// □ Mutable
person.age = 30;
```

□ **3 □ Functions as First-Class Citizens:**

- ฟังก์ชันเป็น **ค่าชนิดหนึ่ง** — ถูกส่งเป็น argument, return value, และเก็บไว้ในตัวแปรได้

```
// ฟังก์ชันเป็นค่าชนิดหนึ่ง
```

```
const multiply = (a, b) => a * b;

function operate(a, b, func) {
  return func(a, b);
}
operate(2, 3, multiply); // 6
```

---

4 Higher-Order Functions (HOF):

- ฟังก์ชันที่ **รับฟังก์ชันเป็นพารามิเตอร์** หรือ **return ฟังก์ชันเป็นค่ากลับ**

```
// map เป็น Higher-Order Function
const nums = [1, 2, 3];
const doubled = nums.map(x => x * 2);
```

---

5 Composition (การต่อฟังก์ชัน):

- ประกอบฟังก์ชันเล็ก ๆ เป็นฟังก์ชันใหญ่
- “อ่านค่าผ่านท่อ” (flow) ของฟังก์ชันโดยตรง

```
const add1 = x => x + 1;
const multiply2 = x => x * 2;

const compose = (f, g) => x => f(g(x));

// Usage
const add1ThenMultiply2 = compose(multiply2, add1);
add1ThenMultiply2(5); // ผลลัพธ์ = multiply2(add1(5)) = 12
```

⚔ FP vs แนวคิดทั่วไป

หัวข้อ	Functional Programming	Procedural / OOP
State	ไม่แก้ state เดิม (Immutable)	แก้ state โดยตรง
Functions	Pure Functions	มี Side Effects บ่อย
Modularity	ประกอบฟังก์ชันเล็ก ๆ (compose)	มักอาศัยคลาสและอ็อบเจกต์
Testing	ทดสอบง่าย (no side effects)	ทดสอบต้องควบคุม state
Concurrency	ปลอดภัย (state ไม่ถูกแก้)	มี race condition ได้

⚡ **ตัวอย่าง FP ใน JavaScript**

```
const numbers = [1, 2, 3, 4, 5];
const isEven = n => n % 2 === 0;
const double = n => n * 2;

const result = numbers.filter(isEven).map(double);

console.log(result); // [4, 8]
```

อ่านเป็น Flow:

“เลือกเลขคู่ ➔ คูณสอง”
โดยที่:

- **filter** และ **map** เป็น HOF
- ฟังก์ชัน isEven และ double เป็น Pure
- numbers ไม่ถูกแก้ไข แต่เป็น Immutable

⚡ **สรุป**

FP คือ: แนวคิดการเขียนโปรแกรมโดยใช้ฟังก์ชันเป็นหัวใจ เน้น:

- Pure Functions
- Immutability
- Higher-Order Functions
- Composition

ข้อดี:

- โค้ดอ่านง่าย
- ทดสอบได้
- จัดการ state ได้ดี
- ปลอดภัยต่อ concurrency

ความแตกต่างระหว่าง Procedural, OOP, และ Functional Programming (FP)

⚡ **1 Procedural Programming (เชิงกระบวนการ)**

แนวคิดหลัก

- เขียนโปรแกรมเป็นชุดคำสั่ง (Procedure) ที่ทำงานตามลำดับ
- โฟกัสที่ **ขั้นตอน (Step-by-step)** ในการแก้ปัญหา
- ใช้ตัวแปรและฟังก์ชันเพื่อจัดการข้อมูลและคำสั่ง
- เน้นการแบ่งโปรแกรมเป็น **ฟังก์ชัน/โปรซีเดอร์ย่อย** เพื่อใช้งานซ้ำ

ตัวอย่าง

```
// คำนวณผลรวมเลขในอาเรย์ แบบ Procedural
let numbers = [1, 2, 3, 4, 5];
let total = 0;

for (let i = 0; i < numbers.length; i++) {
  total += numbers[i];
}
console.log(total); // 15
```

⚡ **2 Object-Oriented Programming (OOP, เชิงวัตถุ)**

แนวคิดหลัก

- จัดกลุ่มข้อมูล (State) และพฤติกรรม (Behavior) เข้าไว้ด้วยกันใน **อ็อบเจกต์ (Objects)**
- ใช้ **คลาส (Classes)** เพื่อเป็นแม่แบบสร้างอ็อบเจกต์
- เน้น **Encapsulation**, **Inheritance**, และ **Polymorphism** เพื่อจัดการความซับซ้อน
- สื่อสารผ่านการส่งข้อความ (method calls) ระหว่างอ็อบเจกต์

ตัวอย่าง

```
class Calculator {
  constructor() {
    this.total = 0;
  }

  add(n) {
    this.total += n;
  }

  getTotal() {
    return this.total;
  }
}
```

```
const calc = new Calculator();
calc.add(5);
calc.add(10);
console.log(calc.getTotal()); // 15
```

⚡ 3 Functional Programming (FP, เชิงฟังก์ชัน)

แนวคิดหลัก

- เขียนโปรแกรมโดยใช้ **ฟังก์ชันบริสุทธิ์ (Pure Functions)** เป็นหน่วยย่อย
- หลีกเลี่ยงการแก้ไขค่าตัวแปรภายนอก (Immutability) และ Side Effects
- เน้นการ **ประกอบ (Composition)** ฟังก์ชันเล็ก ๆ เพื่อแก้ปัญหาใหญ่
- ฟังก์ชันเป็น **First-Class Citizens** ส่งผ่านและคืนค่าได้เหมือนตัวแปรทั่วไป

ตัวอย่าง

```
const numbers = [1, 2, 3, 4, 5];

const sum = (a, b) => a + b;
const total = numbers.reduce(sum, 0);

console.log(total); // 15
```

สรุปความแตกต่างหลัก ๆ

หัวข้อ	Procedural	Object-Oriented Programming (OOP)	Functional Programming (FP)
หลักการ	ลำดับขั้นตอนทำงานเป็นชุดคำสั่ง	ใช้อ็อบเจกต์เก็บข้อมูลและพฤติกรรมร่วมกัน	ใช้ฟังก์ชันบริสุทธิ์และหลีกเลี่ยงการแก้ไขข้อมูล
การจัดการข้อมูล	ตัวแปรทั่วไป, มักเปลี่ยนค่าได้	ข้อมูลอยู่ในอ็อบเจกต์ (state)	ข้อมูลไม่เปลี่ยนแปลง (immutable)
ฟังก์ชัน/เมธอด	ฟังก์ชันเป็นชุดคำสั่ง	เมธอดเป็นฟังก์ชันของอ็อบเจกต์	ฟังก์ชันบริสุทธิ์, higher-order functions
Side Effects	มีได้ง่าย, ปรับเปลี่ยนสถานะได้	มีได้ง่าย, สถานะของอ็อบเจกต์เปลี่ยนแปลงได้	หลีกเลี่ยง Side Effects
Modularity	แบ่งเป็นฟังก์ชัน/โปร	แบ่งตามคลาสและอ็อบเจกต์	แบ่งตามฟังก์ชันเล็ก ๆ และการ

หัวข้อ	Procedural	Object-Oriented Programming (OOP)	Functional Programming (FP)
	ซีเดอร์		ประกอบฟังก์ชัน
การทดสอบ (Testing)	ทดสอบได้แต่ซับซ้อน	ทดสอบง่ายขึ้นถ้าตัด state ออก	ทดสอบง่าย เพราะ Pure Functions ไม่มี side effects
ตัวอย่างเครื่องมือ	C, Basic, Pascal	Java, C++, Python (OOP style)	Haskell, Lisp, หรือ JavaScript (FP style)

□ ตัวอย่างเปรียบเทียบ: คำนวณผลรวมเลข 1 ถึง 5

แบบ Procedural	แบบ OOP	แบบ Functional
``js	```js	```js
let total = 0;	class Sum {	const numbers = [1, 2, 3, 4, 5];
for(let i=1;i<=5;i++)	constructor() {	const total =
total += i;	this.total = 0;	numbers.reduce((a, b) => a + b, 0);
}	}	console.log(total);
console.log(total);	add(n) { this.total += n; }	```
	getTotal() { return this.total; }	
	}	
	const s = new Sum();	
	for(let i=1;i<=5;i++) {	
	s.add(i);	
	}	
	console.log(s.getTotal());	
	``	

⚡□ สรุป

- **Procedural** เหมาะกับงานที่เรียงขั้นตอนชัดเจน
- **OOP** เหมาะกับระบบที่มีข้อมูลซับซ้อน และต้องจัดการ state ผ่านอ็อบเจกต์
- **FP** เหมาะกับงานที่เน้นความปลอดภัยของข้อมูล ทดสอบง่าย และทำงานกับข้อมูลที่เปลี่ยนแปลงบ่อยโดยไม่แก้ไขต้นฉบับ

เจาะลึกเรื่อง Pure Functions หรือ ฟังก์ชันบริสุทธิ์

⚡ Pure Functions คืออะไร?

Pure Function (ฟังก์ชันบริสุทธิ์) คือฟังก์ชันที่:

1. **ผลลัพธ์ขึ้นอยู่กับค่า input เท่านั้น** — เมื่อใส่ input เดิม ฟังก์ชันจะคืนค่าเดิมเสมอ
2. **ไม่มีผลข้างเคียง (No Side Effects)** — ฟังก์ชันจะไม่แก้ไขหรือเปลี่ยนแปลงสถานะภายนอก (เช่น ตัวแปรภายนอก, database, ไฟล์, DOM, หรือ console)

ตัวอย่าง Pure Function

```
const add = (a, b) => a + b;

console.log(add(2, 3)); // 5
console.log(add(2, 3)); // 5 (ผลลัพธ์เหมือนเดิมทุกครั้ง)
```

- add คืนค่าเดียวกันเสมอเมื่อรับ 2 และ 3 เป็น input
- ไม่แก้ไขตัวแปรใด ๆ นอกฟังก์ชัน

ตัวอย่างที่ไม่ใช่ Pure Function (มี Side Effect)

```
let counter = 0;

function increment() {
  counter += 1; // เปลี่ยนแปลงตัวแปรภายนอก (side effect)
  return counter;
}

console.log(increment()); // 1
console.log(increment()); // 2 (ผลลัพธ์เปลี่ยนเพราะ state ภายนอกเปลี่ยน)
```

- ฟังก์ชัน increment เปลี่ยนแปลงตัวแปร counter ภายนอก
- ผลลัพธ์ไม่แน่นอนขึ้นกับสถานะภายนอก ทำให้ฟังก์ชันทดสอบยาก

ทำไม Pure Functions ถึงสำคัญ?

- **ทำนายผลลัพธ์ได้ง่าย** เพราะ input เดิม ⟶ output เดิมเสมอ
- **ทดสอบง่าย** เนื่องจากไม่ต้องตั้งค่าหรือจำลองสถานะภายนอก

- **อ่านและเข้าใจง่าย** เพราะไม่มีผลกระทบต่อส่วนอื่นของโปรแกรม
- **ช่วยให้โปรแกรมปลอดภัยและไม่มีบั๊กจาก state ที่ถูกแก้ไขโดยไม่ตั้งใจ**
- **ช่วยให้สามารถทำงานแบบขนาน (parallel/concurrent) ได้อย่างปลอดภัย**

□ เทคนิคสร้าง Pure Functions

- รับค่า **parameter** แล้วประมวลผลเฉพาะข้อมูลภายใน
- ไม่แก้ไขตัวแปรหรือวัตถุที่ส่งมา (ทำงานแบบ **immutable**)
- คืนค่าผลลัพธ์ใหม่ (สร้างข้อมูลใหม่) แทนการแก้ไขข้อมูลเดิม
- หลีกเลี่ยงการใช้ I/O, การเปลี่ยนแปลง global state ภายในฟังก์ชัน

□ เปรียบเทียบ

ฟังก์ชันบริสุทธิ์ (Pure)	ฟังก์ชันที่มีผลข้างเคียง (Impure)
รับ input เดิม ⟶ คืน output เดิม	คืนผลต่างกันขึ้นกับสถานะภายนอกหรือเวลา
ไม่มีการแก้ไขตัวแปรภายนอก	เปลี่ยนแปลงตัวแปรหรือข้อมูลภายนอก
ไม่ส่งผลกระทบต่อโลกภายนอก	แก้ไข database, console, DOM, file เป็นต้น
ทดสอบง่าย	ทดสอบยาก ต้องจำลองสถานะหรือ mock

⚡□ ตัวอย่าง Pure Function ขั้นสูง

```
// ฟังก์ชันที่คำนวณค่า factorial แบบ pure
const factorial = n => {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
};

console.log(factorial(5)); // 120
```

- factorial(5) จะคืนค่า 120 เสมอ ไม่มีการแก้ไขตัวแปรภายนอก

Immutability — ข้อมูลที่ไม่เปลี่ยนแปลง

⚡□ Immutability คืออะไร?

Immutability หมายถึง **ข้อมูลที่ไม่สามารถเปลี่ยนแปลงได้หลังจากถูกสร้างขึ้น** — เมื่อข้อมูลถูกสร้างมาแล้ว จะไม่มีการแก้ไขข้อมูลเดิม แต่ถ้าต้องการเปลี่ยนแปลง จะสร้างข้อมูลใหม่ขึ้นมาแทน

ทำไม Immutability สำคัญใน Functional Programming?

- **ป้องกัน side effects จากการแก้ไขข้อมูลกลาง**
  เมื่อข้อมูลไม่เปลี่ยนแปลง โปรแกรมจะคาดเดาได้ง่ายขึ้น เพราะไม่มีใครไปแก้ข้อมูลในที่อื่น
- **ช่วยให้โค้ดปลอดภัยกับ concurrency และ parallel processing**
  เพราะไม่มีการแก้ไขข้อมูลกลางทำให้หลีกเลี่ยงปัญหา race condition
- **ช่วยให้การดีบักและทดสอบง่ายขึ้น**
  ข้อมูลไม่เปลี่ยนแปลงตลอดการทำงาน จึงสามารถตรวจสอบสถานะเดิมได้เสมอ
- **สนับสนุนการเขียนโปรแกรมแบบ Pure Functions**
  เพราะ pure functions ไม่แก้ข้อมูลเดิม แต่สร้างค่าผลลัพธ์ใหม่

ตัวอย่าง Immutability ใน JavaScript

ตัวแปรแบบ Primitive

- **Primitive Types (เช่น Number, String, Boolean)** เป็น immutable อยู่แล้ว

```
let str = "hello";
let newStr = str.toUpperCase();

console.log(str);    // "hello" (ไม่เปลี่ยนแปลง)
console.log(newStr); // "HELLO" (สร้างค่าใหม่)
```

Object และ Array (Mutable โดยปกติ)

โดยปกติ Object และ Array ใน JavaScript เป็น **mutable** — แก้ไขค่าภายในได้ เช่น

```
const arr = [1, 2, 3];
arr.push(4);
console.log(arr); // [1, 2, 3, 4] — เปลี่ยนแปลงได้
```

วิธีทำให้ Object/Array เป็น Immutable

1 ใช้ Spread Operator สร้างข้อมูลใหม่แทนแก้ไขเดิม

```
const person = { name: "Alice", age: 25 };

// สร้างอ็อบเจ็กต์ใหม่ที่แก้ไข age แต่ไม่แก้ไขต้นฉบับ
const olderPerson = { ...person, age: 26 };

console.log(person.age);      // 25 (ไม่เปลี่ยน)
```

```
console.log(olderPerson.age); // 26 (อ็อบเจ็กต์ใหม่)
```

2 ใช้ Array methods ที่ไม่แก้ไขเดิม เช่น map, filter, concat

```
const nums = [1, 2, 3];

// สร้างอาเรย์ใหม่ที่เพิ่มค่า 1 ทุกตัว
const incremented = nums.map(x => x + 1);

console.log(nums);        // [1, 2, 3]
console.log(incremented); // [2, 3, 4]
```

3 ใช้ Object.freeze() เพื่อแช่แข็ง Object (ทำให้แก้ไขไม่ได้)

```
const obj = { a: 1 };
Object.freeze(obj);

obj.a = 2; // ไม่ได้ผล (ใน strict mode จะ error)
console.log(obj.a); // 1
```

แต่ข้อจำกัดคือ freeze แบบตื้น (shallow) ไม่ได้แช่แข็ง nested objects ลึก ๆ

⚡ **Immutability กับ Performance**

- สร้างข้อมูลใหม่อาจดูเหมือนใช้หน่วยความจำมากกว่า
- แต่ด้วยเทคนิคเช่น **Structural Sharing** (ในไลบรารี FP อย่าง Immutable.js หรือ immer) จะช่วยลดการคัดลอกซ้ำซ้อน

สรุป

จุดเด่นของ Immutability	อธิบาย
ข้อมูลไม่เปลี่ยนแปลง	ข้อมูลเดิมถูกเก็บไว้อย่างปลอดภัย
ลดข้อผิดพลาดจากการแก้ไขข้อมูล	ไม่มีใครแก้ข้อมูลที่ใช้ร่วมกันโดยไม่ตั้งใจ
ง่ายต่อการดีบักและทดสอบ	สถานะข้อมูลแน่นอนและไม่เปลี่ยนแปลง
สนับสนุน Pure Functions	สร้างผลลัพธ์ใหม่โดยไม่เปลี่ยนข้อมูลเดิม

Side Effects

⚡ **Side Effects คืออะไร?**

Side Effects (ผลข้างเคียง) คือ สิ่งที่เกิดขึ้นนอกเหนือจากการคืนค่าของฟังก์ชัน เช่น

- การเปลี่ยนแปลงตัวแปรหรือสถานะภายนอกฟังก์ชัน (global variables, object properties)
- การเขียนหรืออ่านไฟล์
- การส่งข้อมูลออกสู่หน้าจอ (เช่น console.log, การแก้ไข DOM ในเว็บ)
- การเรียกใช้งาน API หรือฐานข้อมูล
- การโยกย้ายข้อมูลไปยังระบบอื่น เช่น network requests

ตัวอย่าง Side Effects

```
let count = 0;

function increment() {
  count += 1;        // เปลี่ยนแปลงตัวแปรภายนอก = side effect
  console.log(count); // แสดงผลลัพธ์ = side effect
}
```

⚡ **ทำไมต้องหลีกเลี่ยง Side Effects?**

1 ทำให้ฟังก์ชัน ไม่บริสุทธิ์ (Impure)

- ผลลัพธ์อาจเปลี่ยนแปลงขึ้นกับสถานะภายนอก
- ยากต่อการทำนายและทดสอบ

2 เพิ่มความซับซ้อนและบั๊ก

- ถ้าหลายส่วนของโปรแกรมเปลี่ยนแปลงสถานะเดียวกัน อาจเกิดความขัดแย้ง (Race Condition)
- ยากที่จะระบุว่าเกิดบั๊กจากจุดไหน เพราะผลกระทบกระจายหลายที่

3 ลดความยืดหยุ่นและความสามารถในการใช้งานซ้ำ

- ฟังก์ชันที่มี Side Effects มักจะต้องทำงานร่วมกับบริบทเฉพาะ ทำให้ยากจะนำไปใช้ในที่อื่น

⚡ **วิธีการจัดการ Side Effects ใน Functional Programming**

แยกส่วนฟังก์ชันบริสุทธิ์กับ Side Effects

- ฟังก์ชัน Pure ทำงานคำนวณผลลัพธ์
- ฟังก์ชันอื่นจัดการกับ Side Effects เช่น การแสดงผลหรือเรียก API

ใช้เทคนิค Lazy Evaluation หรือ Monads (ในภาษา FP บางภาษา) เพื่อควบคุม side effects

---

⚡ **ตัวอย่างแยก Side Effects ออกจาก Pure Function**

```
// Pure function: คำนวณผลลัพธ์อย่างเดียว
const sum = (a, b) => a + b;

// Impure function: แสดงผล (side effect)
const printResult = result => console.log("Result:", result);

// นำมารวมกันอย่างชัดเจน
const a = 2, b = 3;
const result = sum(a, b);
printResult(result);
```

☐ **สรุป**

ประเด็น	อธิบาย
Side Effects คือ	การเปลี่ยนแปลงสถานะภายนอกหรือผลลัพธ์ที่ไม่ใช่ค่ากลับของฟังก์ชัน
ปัญหาที่เกิดจาก Side Effects	ฟังก์ชันทำนายผลไม่ได้, ยากทดสอบ, บั๊กง่ายเกิดขึ้น
การหลีกเลี่ยง Side Effects	แยก Pure Functions กับ I/O, ควบคุมการเปลี่ยนแปลงสถานะให้ชัดเจน

สรุป

บทที่ 1 จะพาผู้อ่านทำความรู้จักกับพื้นฐานของ JavaScript โดยเฉพาะในส่วนของฟังก์ชันซึ่งเป็นหัวใจของแนวคิด Functional Programming (FP) ผ่านการเรียนรู้รูปแบบของฟังก์ชัน เช่น Declaration, Expression และ Arrow Functions พร้อมทั้งปูแนวคิดหลักของ FP ที่เน้นการเขียนฟังก์ชันแบบบริสุทธิ์ (Pure Functions), การไม่เปลี่ยนแปลงข้อมูล (Immutability) และการหลีกเลี่ยงผลข้างเคียง (Side Effects) ซึ่งต่างจากการเขียนโปรแกรมเชิงขั้นตอนและเชิงวัตถุอย่างชัดเจน โดยบทนี้จะวางรากฐานความเข้าใจในโครงสร้างและพฤติกรรมของโค้ดแบบ Functional ซึ่งจะเป็นพื้นฐานสำคัญในการพัฒนาโปรแกรมอย่างยั่งยืนในบทถัดไป

บทที่ 2
ฟังก์ชันขั้นพื้นฐานและการใช้งานใน FP
(Primary Function and FP)

เนื้อหา

บทนำบทที่ 2: ฟังก์ชันขั้นพื้นฐานและการใช้งานใน Functional Programming

หลังจากผู้อ่านได้ทำความเข้าใจพื้นฐานของ JavaScript และแนวคิดของ Functional Programming ในบทก่อนแล้ว บทนี้จะพาเจาะลึกสู่หัวใจสำคัญของ FP นั่นคือ “ฟังก์ชัน” ไม่ใช่เพียงแค่เครื่องมือสำหรับห่อหุ้มโค้ดซ้ำ ๆ เท่านั้น แต่ใน FP ฟังก์ชันคือค่าประเภทหนึ่งที่สามารถจัดการได้เหมือนข้อมูลอื่น ๆ เราเรียกคุณสมบัตินี้ว่า **First-class Functions** ซึ่งหมายความว่า ฟังก์ชันสามารถเก็บไว้ในตัวแปร ส่งเป็นอาร์กิวเมนต์ หรือคืนค่าจากฟังก์ชันอื่นได้

แนวคิดของ **Higher-order Functions** จึงตามมาทันทีเมื่อฟังก์ชันสามารถถูกส่งเข้าและส่งออกได้ ฟังก์ชันลักษณะนี้จะรับฟังก์ชันอื่นเป็นอาร์กิวเมนต์ หรือส่งฟังก์ชันอื่นกลับมาเป็นผลลัพธ์ การเข้าใจ Higher-order Functions คือก้าวสำคัญที่ทำให้เราสามารถเขียนโค้ดที่ยืดหยุ่น ซ้ำใช้ได้ และลดการทำงานที่ซ้ำซ้อนในระบบขนาดใหญ่

หัวข้อที่สัมพันธ์ใกล้ชิดกับ Higher-order Functions คือ **Closures** ซึ่งคือการที่ฟังก์ชันสามารถจดจำค่าหรือบริบทของตัวแปรภายนอกในขณะที่มันถูกสร้างขึ้น แม้ว่าจะถูกเรียกใช้นอกขอบเขตเดิม

แล้วก็ตาม ความสามารถนี้ทำให้สามารถ “ซ่อนข้อมูล” หรือสร้างพฤติกรรมแบบส่วนตัวของฟังก์ชันได้ ซึ่งมีประโยชน์อย่างมากในการออกแบบโค้ดที่ปลอดภัยและแยก concerns อย่างชัดเจน

นอกจากการออกแบบฟังก์ชันแล้ว อีกเทคนิคที่สำคัญใน Functional Programming คือ **Recursion** หรือการที่ฟังก์ชันสามารถเรียกตัวเองได้ ซึ่งในหลาย ๆ กรณีจะถูกใช้แทนการวนลูป (loop) โดยเฉพาะเมื่อเราต้องการรักษาความเป็น immutable และหลีกเลี่ยงการใช้ state ภายนอก การใช้ Recursion อย่างถูกต้องจะช่วยให้โค้ดมีลักษณะ declarative มากขึ้น และทำให้ตรรกะของโปรแกรมชัดเจนขึ้น

อีกหนึ่งหัวใจของ FP คือ **Function Composition** หรือการนำฟังก์ชันหลายตัวมาต่อกันเป็นท่อของกระบวนการทำงาน (pipeline) เพื่อเปลี่ยนข้อมูลทีละขั้น โดยไม่ต้องเขียนโค้ดแบบ imperative ที่ยุ่งยาก Function Composition ทำให้โค้ดอ่านง่าย ร้อยเรียงเป็นลำดับตรรกะที่เห็นได้ชัด และสามารถนำกลับมาใช้ใหม่ได้อย่างยืดหยุ่น

เมื่อแนวคิดเหล่านี้ถูกรวมกัน — จากฟังก์ชันที่เป็น First-class, การใช้ Higher-order และ Closures, ไปจนถึงการออกแบบ Recursion และการเชื่อมโยงฟังก์ชันแบบ Composition — เราจะได้เครื่องมือที่ทรงพลังในการเขียนโปรแกรมแบบ Functional ด้วย JavaScript ไม่เพียงแต่เพื่อความ “สะอาด” ของโค้ด แต่ยังเพื่อความสามารถในการแยกส่วน (modularity), ความทดสอบได้ (testability), และความสามารถในการขยาย (scalability)

บทนี้จึงมีบทบาทสำคัญอย่างยิ่งในการเปลี่ยนวิธีคิดเกี่ยวกับการเขียนโค้ด JavaScript จากมุมมองเชิงคำสั่ง (imperative) ไปสู่แนวคิดเชิงฟังก์ชัน (functional) อย่างแท้จริง พร้อมด้วยตัวอย่างและการฝึกปฏิบัติที่จะช่วยให้ผู้อ่านเข้าใจหลักการเหล่านี้อย่างลึกซึ้งและสามารถนำไปใช้ได้จริงในการพัฒนาโปรแกรม

แนวคิดพื้นฐาน

---

1 First-class Functions (ฟังก์ชันเป็นค่า)

ความหมาย

ใน JavaScript ฟังก์ชันถือเป็น **ค่าชนิดหนึ่ง (First-class citizen)** หมายความว่า ฟังก์ชันสามารถ:

- ถูกเก็บในตัวแปร
- ถูกส่งผ่านเป็นอาร์กิวเมนต์ให้ฟังก์ชันอื่น
- ถูกคืนค่าจากฟังก์ชัน
- มี property และ method ได้ (เพราะฟังก์ชันคือ Object แบบพิเศษ)

ตัวอย่าง

```
// เก็บฟังก์ชันในตัวแปร
const greet = function(name) {
  return `Hello, ${name}!`;
```

```
};

// ส่งฟังก์ชันเป็น argument
function callFn(fn, value) {
  return fn(value);
}

console.log(callFn(greet, "Alice")); // Hello, Alice!

// คืนฟังก์ชันจากฟังก์ชันอื่น
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = multiplier(2);
console.log(double(5)); // 10
```

---

2 Higher-order Functions (HOF)

ความหมาย

ฟังก์ชันที่ **รับฟังก์ชันอื่นเป็นอาร์กิวเมนต์** หรือ **คืนฟังก์ชันเป็นผลลัพธ์** เรียกว่า Higher-order Functions

ตัวอย่าง

```
// รับฟังก์ชันเป็น argument (callback)
const nums = [1, 2, 3];
const doubled = nums.map(x => x * 2);

console.log(doubled); // [2, 4, 6]

// คืนฟังก์ชันเป็นผลลัพธ์
function makeAdder(x) {
  return function(y) {
    return x + y;
```

```
  };
}

const add5 = makeAdder(5);
console.log(add5(3)); // 8
```

ประโยชน์

- ช่วยให้เขียนโค้ดซ้ำใช้งานใหม่ได้
- สร้าง abstraction (เช่น map, filter, reduce)
- ทำงานกับ callback และ asynchronous ได้ดี

---

3 Closures (การปิดทับค่าและซ่อนข้อมูล)

ความหมาย

Closure คือ ฟังก์ชันที่ “จดจำ” scope ของตัวแปรที่อยู่ในบริบท (lexical environment) ของมัน แม้ฟังก์ชันนั้นจะถูกเรียกใช้ภายนอก scope เดิม

ตัวอย่าง

```
function counter() {
  let count = 0;
  return function() {
    count += 1;
    return count;
  };
}

const increment = counter();
console.log(increment()); // 1
console.log(increment()); // 2
console.log(increment()); // 3
```

อธิบาย:

ฟังก์ชันที่คืนค่ามายังคงจดจำตัวแปร count แม้ counter() จะทำงานจบไปแล้ว — ทำให้สามารถเก็บสถานะได้แบบ “ซ่อนข้อมูล” หรือ Encapsulation

---

4 Recursion (ฟังก์ชันเรียกตัวเอง)

ความหมาย

การเรียกฟังก์ชันด้วยตัวมันเองเป็นกระบวนการแก้ปัญหาโดยแบ่งย่อยปัญหาใหญ่ให้เล็กลงจนถึงฐาน (base case)

ตัวอย่าง (ฟังก์ชันหาค่า factorial)

```
function factorial(n) {
  if (n <= 1) return 1;         // base case
  return n * factorial(n - 1); // recursive case
}

console.log(factorial(5)); // 120
```

ประโยชน์ใน FP

- ใช้แทนการวนลูปแบบ imperative
- สอดคล้องกับการทำงานของฟังก์ชันบริสุทธิ์และ immutability

---

5 Function Composition (การต่อฟังก์ชัน)

ความหมาย

การ **นำฟังก์ชันหลาย ๆ ตัวมาต่อกัน** โดยเอาผลลัพธ์ของฟังก์ชันตัวหนึ่งเป็นอินพุตให้ฟังก์ชันถัดไป

ตัวอย่าง

```
const add1 = x => x + 1;
const double = x => x * 2;

// ฟังก์ชันช่วย compose
const compose = (f, g) => x => f(g(x));

// ใช้งาน
const add1ThenDouble = compose(double, add1);

console.log(add1ThenDouble(3)); // (3 + 1) * 2 = 8
```

สรุป

หัวข้อ	ความหมาย	ตัวอย่างการใช้งานหลัก
First-class Functions	ฟังก์ชันเป็นค่า สามารถส่งและคืนได้	เก็บฟังก์ชันในตัวแปร, ส่งเป็นอาร์กิวเมนต์
Higher-order	ฟังก์ชันที่รับหรือคืนฟังก์ชัน	map, filter, reduce, ฟังก์ชันสร้างฟังก์ชัน

หัวข้อ	ความหมาย	ตัวอย่างการใช้งานหลัก
Functions		
Closures	ฟังก์ชันที่จดจำค่าจาก scope เดิม	ซ่อนข้อมูล (Encapsulation), สร้างสถานะภายใน
Recursion	ฟังก์ชันเรียกตัวเอง	แทนลูป, แก้ปัญหาย่อย (Divide & Conquer)
Function Composition	ต่อฟังก์ชันหลายตัวเป็นชุดเดียว	รวมฟังก์ชันเพื่อสร้าง pipeline ของการประมวลผล

First-class Functions

⚡ **First-class Functions คืออะไร?**

ในภาษาโปรแกรมที่สนับสนุนแนวคิด **First-class functions** ฟังก์ชันจะถูกปฏิบัติเสมือนเป็น **"ค่าชนิดหนึ่ง"** (เช่น ตัวเลข, สตริง, อ็อบเจ็กต์) ซึ่งหมายความว่า:

- สามารถ **เก็บในตัวแปร** ได้
- สามารถ **ส่งผ่านเป็นอาร์กิวเมนต์** ให้กับฟังก์ชันอื่นได้
- สามารถ **คืนค่าจากฟังก์ชัน** ได้
- สามารถ **เก็บในโครงสร้างข้อมูล** เช่น อาเรย์ หรืออ็อบเจกต์ได้

ตัวอย่างใน JavaScript

1. เก็บฟังก์ชันในตัวแปร

```
const greet = function(name) {
  return `Hello, ${name}!`;
};
console.log(greet("Alice")); // Hello, Alice!
```

2. ส่งฟังก์ชันเป็น argument ให้ฟังก์ชันอื่น

```
function callFn(fn, value) {
  return fn(value);
}

console.log(callFn(greet, "Bob")); // Hello, Bob!
```

3. คืนค่าฟังก์ชันจากฟังก์ชัน

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = multiplier(2);
console.log(double(5)); // 10
```

4. เก็บฟังก์ชันในโครงสร้างข้อมูล

```
const operations = [
  x => x + 1,
  x => x * 2,
  x => x ** 2
];

console.log(operations ); // 6 (ฟังก์ชันตัวที่สองคือคูณสอง)
```

☐ **ทำไม First-class Functions สำคัญ?**

- **ความยืดหยุ่นสูง:** ฟังก์ชันสามารถถูกใช้ในรูปแบบต่าง ๆ ได้อย่างง่ายดาย
- **แนวคิด Functional Programming:** ยึดหลักการนี้เพื่อใช้ฟังก์ชันเป็นหน่วยย่อยในการสร้างโปรแกรม
- **ง่ายต่อการเขียนโค้ดที่ reusable และ modular:** สามารถสร้าง Higher-order Functions ได้สะดวก
- **การเขียนโค้ดแบบ declarative:** ใช้ฟังก์ชันประกอบและส่งผ่านกันแทนการเขียนลูปและเงื่อนไขยาว ๆ

☐ **สรุป**

ความสามารถของ First-class Functions	อธิบาย	ตัวอย่างสั้น ๆ
เก็บในตัวแปรได้	ฟังก์ชันเป็นค่าชนิดหนึ่ง	const f = () => {};
ส่งเป็นอาร์กิวเมนต์ได้	รับฟังก์ชันเป็นพารามิเตอร์	arr.map(x => x * 2)
คืนค่าฟังก์ชันได้	ฟังก์ชันส่งกลับจาก	const makeAdder = x => y => x

ความสามารถของ First-class Functions	อธิบาย	ตัวอย่างสั้น ๆ
	ฟังก์ชัน	+ y
เก็บในโครงสร้างข้อมูลได้	เก็บใน array หรือ object	const funcs = [f1, f2];

First-class Functions (ฟังก์ชันเป็นค่า) — เชิงลึก

1. ความหมายเชิงลึก

First-class functions คือคุณสมบัติของภาษาที่ทำให้ฟังก์ชันถูกปฏิบัติเหมือนกับค่าทั่วไป (เช่น ตัวเลข, สตริง, อ็อบเจกต์) ซึ่งหมายความว่า:

- ฟังก์ชันเป็น **อ็อบเจกต์ชนิดหนึ่ง** (function objects) ที่มี property และ method ของตัวเองได้
- ฟังก์ชันสามารถ **สร้างใหม่, ส่งผ่าน, คืนค่า, และเก็บไว้ได้** เหมือนค่าปกติ
- ส่งผลให้เราสามารถใช้ฟังก์ชันเป็นหน่วยนามธรรม (abstraction) ที่ยืดหยุ่นและทรงพลังมาก

2. ผลกระทบเชิงภาษาและการใช้งาน

ฟังก์ชันคือ Object ใน JavaScript

- ฟังก์ชันใน JavaScript คือ **Callable Object** มีคุณสมบัติและสามารถเก็บข้อมูลได้

```
function foo() {}
console.log(typeof foo);            // "function"
console.log(typeof foo === "object"); // false (แต่ foo เป็น Object ในแง่ของ JS object model)
console.log(foo instanceof Function); // true
console.log(foo instanceof Object);   // true
```

- ฟังก์ชันสามารถมี property ได้

```
function sayHi() {}
sayHi.greeting = "Hello";
console.log(sayHi.greeting); // "Hello"
```

3. การส่งผ่านและคืนค่าฟังก์ชัน

ส่งผ่านฟังก์ชันเป็น argument

- ช่วยให้ทำงานกับ callback, event handlers, และ asynchronous programming

```
function greet(name) {
  return `Hello, ${name}`;
```

```
}

function processUserInput(callback) {
  let name = "Alice";
  console.log(callback(name));
}

processUserInput(greet); // Hello, Alice
```

คืนค่าฟังก์ชัน (Function Factory)

- สร้างฟังก์ชันแบบไดนามิก

```
function makeMultiplier(multiplier) {
  return function(x) {
    return x * multiplier;
  };
}

const double = makeMultiplier(2);
const triple = makeMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

---

4. ความสำคัญกับ Functional Programming

- **Higher-Order Functions** ต้องใช้ first-class functions เพราะต้องรับหรือคืนฟังก์ชัน
- ทำให้สามารถเขียนโค้ดแบบ Declarative, Modular, และ Reusable ได้
- ช่วยในการสร้าง **Closures** และ **Currying**

---

5. ตัวอย่างขั้นสูง: เก็บฟังก์ชันในโครงสร้างข้อมูล

```
const operations = {
  add: (a, b) => a + b,
  multiply: (a, b) => a * b
};

console.log(operations.add(2,3));        // 5
```

```
console.log(operations['multiply'](2,3)); // 6
```

---

6. ฟังก์ชัน vs Method

- ฟังก์ชัน (Function): เป็นค่าที่สามารถส่งต่อได้
- เมธอด (Method): ฟังก์ชันที่เป็น property ของ object

```
const obj = {
  x: 10,
  getX: function() { return this.x; }
};

const method = obj.getX;
console.log(obj.getX()); // 10
console.log(method());   // undefined เพราะ this ไม่ชี้ obj ในที่นี้
```

---

7. ข้อควรระวัง

- **Context (this)** ในฟังก์ชันธรรมดาแตกต่างกับ Arrow function
- การส่งผ่านฟังก์ชันต้องระวังการ binding context ให้ถูกต้อง
- ฟังก์ชันที่เก็บในตัวแปรและส่งผ่าน สามารถทำให้เกิดการใช้หน่วยความจำมาก หากมี closure ขนาดใหญ่

---

8. ความสัมพันธ์กับ Concept อื่น ๆ

Concept	ความเกี่ยวข้องกับ First-class Functions
Closures	ฟังก์ชันที่จดจำ scope เมื่อตัวมันถูกส่งผ่านและเรียกใช้
Higher-order Functions	ฟังก์ชันที่รับหรือคืนฟังก์ชัน ต้องใช้ first-class
Currying	ฟังก์ชันที่คืนฟังก์ชันตาม argument ที่รับมา
Function Composition	การใช้ฟังก์ชันที่คืนฟังก์ชันมาต่อกัน

---

9. สรุปเชิงลึก

- JavaScript ฟังก์ชันคือ **first-class citizens** ที่เป็น Callable Objects
- ฟังก์ชันสามารถถูกจัดการเหมือนค่าธรรมดาในทุกแง่มุม
- คุณสมบัตินี้ทำให้ FP ใน JS มีความยืดหยุ่นสูง
- การใช้ฟังก์ชันเป็นค่าเปิดโอกาสให้สร้างโค้ดที่ modular, reusable และ declarative
- ต้องระวังเรื่อง context (this) เมื่อส่งผ่านฟังก์ชัน

นี่คือตัวอย่างโปรแกรม 5 ตัวอย่างที่แสดงการใช้ **First-class Functions** ใน JavaScript พร้อมคำอธิบายโค้ดและผลลัพธ์จากการรัน

ตัวอย่างที่ 1: เก็บฟังก์ชันในตัวแปร และเรียกใช้

```
// เก็บฟังก์ชันในตัวแปร greet
const greet = function(name) {
  return `Hello, ${name}!`;
};

console.log(greet("Alice"));
```

คำอธิบาย:

ฟังก์ชันถูกเก็บในตัวแปร greet และสามารถเรียกใช้งานผ่านตัวแปรนั้นได้เหมือนกับชื่อฟังก์ชันปกติ

ผลลัพธ์:

```
Hello, Alice!
```

ตัวอย่างที่ 2: ส่งฟังก์ชันเป็น argument ให้ฟังก์ชันอื่น (Callback)

```
function processUserInput(callback) {
  const name = "Bob";
  return callback(name);
}

const greet = name => `Hi, ${name}!`;

console.log(processUserInput(greet));
```

คำอธิบาย:

ฟังก์ชัน processUserInput รับฟังก์ชัน callback เป็น argument และเรียกใช้มัน โดยส่งค่า name เข้าไป

ผลลัพธ์:

```
Hi, Bob!
```

ตัวอย่างที่ 3: คืนค่าฟังก์ชันจากฟังก์ชัน (Function Factory)

```
function makeMultiplier(factor) {
  return function(number) {
```

```
    return number * factor;
  };
}

const double = makeMultiplier(2);
const triple = makeMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

คำอธิบาย:

makeMultiplier คืนฟังก์ชันใหม่ที่ใช้ค่าคูณตาม factor ที่กำหนด ทำให้เราได้ฟังก์ชัน double และ triple ที่คูณเลขด้วย 2 และ 3 ตามลำดับ

ผลลัพธ์:

10
15

ตัวอย่างที่ 4: เก็บฟังก์ชันในอาเรย์ และวนลูปเรียกใช้

```
const operations = [
  x => x + 1,
  x => x * 2,
  x => x ** 2
];

const input = 3;
const results = operations.map(fn => fn(input));

console.log(results);
```

คำอธิบาย:

อาเรย์ operations เก็บฟังก์ชันสามตัว เราใช้ map เรียกแต่ละฟังก์ชันกับค่า input และเก็บผลลัพธ์ไว้ใน results

ผลลัพธ์:

[4, 6, 9]

ตัวอย่างที่ 5: ใช้ฟังก์ชันแบบ Higher-order เพื่อกรองข้อมูล

```
function filterArray(arr, predicate) {
  const result = [];
  for (const item of arr) {
    if (predicate(item)) {
      result.push(item);
    }
  }
  return result;
}

const numbers = [1, 2, 3, 4, 5, 6];

// ฟังก์ชัน predicate กรองเลขคู่
const isEven = num => num % 2 === 0;

const evens = filterArray(numbers, isEven);
console.log(evens);
```

คำอธิบาย:

filterArray เป็น higher-order function ที่รับฟังก์ชัน predicate เพื่อกำหนดเงื่อนไขการกรองข้อมูล ในที่นี้กรองเลขคู่

ผลลัพธ์:

[2, 4, 6]

นี่คือตัวอย่างโปรแกรมแนวประยุกต์ใช้ **First-class Functions** ใน JavaScript จำนวน 5 โปรแกรม พร้อมคำอธิบายและผลการรัน เพื่อให้เห็นภาพการนำไปใช้งานจริง

ตัวอย่างที่ 1: ระบบจัดการ Event Listener (Callback)

```
// ฟังก์ชันที่รับ callback เพื่อเรียกใช้เมื่อเกิด event
function addEventListener(eventName, callback) {
  console.log(`Listening for ${eventName} event...`);
  // จำลอง event เกิดขึ้น
  setTimeout(() => {
    console.log(`${eventName} event triggered!`);
    callback();
```

```
  }, 1000);
}

// ฟังก์ชัน callback ที่จะถูกเรียกเมื่อ event เกิดขึ้น
function onClick() {
  console.log("Button clicked!");
}

// เรียกใช้
addEventListener("click", onClick);
```

คำอธิบาย:

จำลองระบบ event listener ที่รับฟังก์ชัน callback เพื่อเรียกใช้เมื่อ event เกิดขึ้น การส่งฟังก์ชันเป็น argument ช่วยให้ยืดหยุ่นและตอบสนองเหตุการณ์ได้

ผลลัพธ์ (หลัง 1 วิ):

```
Listening for click event...
click event triggered!
Button clicked!
```

ตัวอย่างที่ 2: ฟังก์ชันสร้าง Validator แบบกำหนดเอง

```
// ฟังก์ชันสร้างฟังก์ชันตรวจสอบความยาวข้อความ
function createLengthValidator(minLength) {
  return function(text) {
    return text.length >= minLength;
  };
}

const isLongEnough = createLengthValidator(5);

console.log(isLongEnough("hello"));    // true
console.log(isLongEnough("hi"));       // false
```

คำอธิบาย:

ใช้ฟังก์ชันที่คืนฟังก์ชัน (function factory) เพื่อสร้างตัวตรวจสอบความยาวข้อความแบบ dynamic ได้ตามที่ต้องการ

ผลลัพธ์: