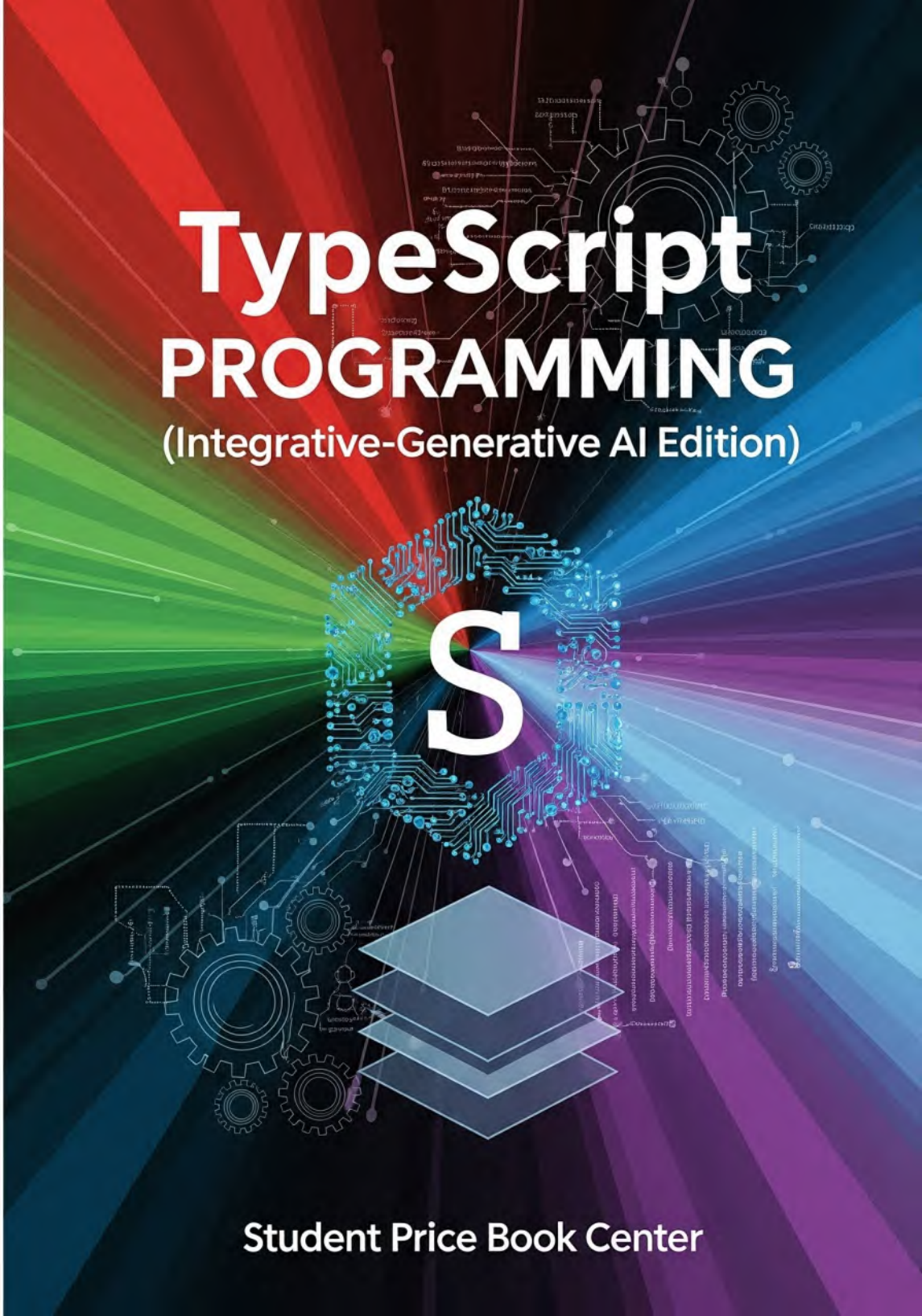
TypeScript PROGRAMMING

(Integrative-Generative AI Edition)



Student Price Book Center

คำนำ

ในโลกยุคปัจจุบันที่ซอฟต์แวร์มีบทบาทสำคัญต่อธุรกิจและชีวิตประจำวัน ความคาดหวังต่อคุณภาพ ความปลอดภัย และความสามารถในการบำรุงรักษาของซอฟต์แวร์จึงสูงขึ้นเรื่อยๆ ภาษา JavaScript ซึ่งเป็นหัวใจหลักของการพัฒนาเว็บและแอปพลิเคชันในทศวรรษที่ผ่านมา ได้พิสูจน์แล้วว่ามีความยืดหยุ่นและทรงพลัง แต่ในขณะเดียวกันก็มีข้อจำกัดบางประการ โดยเฉพาะอย่างยิ่งในโครงการขนาดใหญ่ที่ต้องการระบบชนิดข้อมูลที่เข้มงวดและการตรวจสอบข้อผิดพลาดที่ชัดเจน ปัญหาเหล่านี้ได้กลายเป็นแรงบันดาลใจในการสร้าง **TypeScript** ภาษาที่พัฒนาขึ้นเพื่อเพิ่มศักยภาพของ JavaScript ให้สามารถพัฒนาโปรแกรมได้อย่างมั่นใจ ปลอดภัย และง่ายต่อการบำรุงรักษา

หนังสือ *TypeScript Programming* เล่มนี้ถูกเขียนขึ้นด้วยความตั้งใจที่จะเป็นคู่มือแบบครบวงจร ทั้งสำหรับผู้ที่เริ่มต้นศึกษา TypeScript และนักพัฒนามืออาชีพที่ต้องการทำความเข้าใจคุณสมบัติขั้นสูง โดยเนื้อหาได้จัดเรียงเป็นลำดับขั้น เริ่มตั้งแต่บทนำที่อธิบายพื้นฐาน ความแตกต่างระหว่าง JavaScript และ TypeScript การติดตั้ง การตั้งค่า และการเริ่มต้นใช้งาน จนกระทั่งถึงเรื่องที่ซับซ้อน เช่น Generics, Advanced Type Features, Decorators, และการออกแบบระบบขนาดใหญ่ให้มีความยืดหยุ่นสูง บทต่างๆ ในหนังสือจะค่อยๆ ปูพื้นฐานแล้วต่อยอดไปสู่แนวทางการเขียนโปรแกรมที่มีประสิทธิภาพและมีมาตรฐานระดับมืออาชีพ

ใน **บทที่ 1** ผู้อ่านจะได้รู้จักที่มาและแนวคิดสำคัญของ TypeScript รวมถึงเหตุผลว่าทำไมองค์กรซอฟต์แวร์ทั่วโลกจึงเลือกนำมาใช้แทน JavaScript ในโครงการที่ต้องการคุณภาพและความน่าเชื่อถือ เนื้อหาจะพาไปทำความเข้าใจตั้งแต่การติดตั้ง การใช้เครื่องมืออย่าง tsc และ ts-node ไปจนถึงโครงสร้างไฟล์การตั้งค่า tsconfig.json ที่ถือเป็นหัวใจสำคัญของโปรเจกต์ TypeScript

ถัดไป **บทที่ 2–5** จะอธิบายรายละเอียดของชนิดข้อมูล ตัวแปร ฟังก์ชัน Array Tuple และ Object Type ที่เป็นรากฐานของการพัฒนาโปรแกรม ไม่เพียงแค่สอนวิธีเขียนโค้ด แต่ยังอธิบายที่มา เหตุผล และข้อควรระวังในการใช้งาน เพื่อให้ผู้อ่านเข้าใจหลักการอย่างแท้จริง

เมื่อผู้อ่านมีพื้นฐานแน่นแล้ว **บทที่ 6–8** จะพาเข้าสู่โลกของ *Interface*, *Type Alias*, *Class* และ *Inheritance* ซึ่งเป็นหัวใจของแนวคิดเชิงวัตถุ (OOP) ใน TypeScript หนังสือจะอธิบายอย่างละเอียดว่าการกำหนดรูปแบบข้อมูล การใช้ Access Modifiers และ Abstract Class ช่วยเพิ่มความปลอดภัยและความเป็นระบบในซอฟต์แวร์อย่างไร

ในส่วนของ **บทที่ 9–14** หนังสือจะพาผู้อ่านทำความเข้าใจฟีเจอร์ที่ทรงพลัง เช่น *Enum*, *Literal Types*, *Generics*, *Union* และ *Intersection Types*, *Advanced Type Features*, และ *Utility Types* เนื้อหาชุดนี้จะช่วยให้โค้ดของคุณยืดหยุ่น สามารถ reuse ได้ และยังคงตรวจสอบชนิดข้อมูลได้อย่างเข้มงวด ทำให้ซอฟต์แวร์ของคุณปลอดภัยต่อข้อผิดพลาดที่อาจเกิดขึ้น

ต่อมา **บทที่ 15–17** จะพาผู้อ่านไปเรียนรู้แนวคิดขั้นสูง เช่น *Decorators*, *Mixins* และ *Advanced Type Narrowing* ซึ่งจะทำให้คุณสามารถสร้างโครงสร้างระบบที่ซับซ้อนและมีความสามารถในการต่อขยายได้โดยไม่สูญเสียความชัดเจนของโครงสร้างโปรแกรม ตัวอย่างเช่นการประยุกต์ใช้

Mixins เพื่อเลียนแบบ Multiple Inheritance หรือการใช้ User-Defined Type Guard เพื่อตรวจสอบชนิดข้อมูลที่ซับซ้อน

สำหรับ **บทที่ 18** หนังสือได้เตรียมเนื้อหาเรื่อง *Error Handling แบบ Type-Safe* ซึ่งมีความสำคัญในระบบโปรแกรมที่ต้องรับประกันความถูกต้องของข้อมูลและการทำงานที่ปลอดภัย พร้อมแนะนำ Pattern ที่ทันสมัย เช่น Result หรือ Either Pattern ซึ่งช่วยให้จัดการข้อผิดพลาดได้เป็นระบบมากขึ้น

ใน **บทที่ 19** ผู้อ่านจะได้เห็นวิธีประยุกต์ใช้ TypeScript ร่วมกับ Framework ยอดนิยม ไม่ว่าจะเป็น React, Angular, Node.js หรือ NestJS โดยมีตัวอย่างและแนวปฏิบัติที่สามารถนำไปใช้พัฒนาโปรเจกต์จริงได้ทันที และสุดท้าย **บทที่ 20** จะสรุป Best Practices และแนวทางการออกแบบโปรเจกต์ตั้งแต่การตั้งค่า tsconfig การเปิด Strict Mode การใช้เครื่องมือ lint และ formatter การแยก Layer ของระบบ และการจัดการ Dependency Injection อย่างเป็นระบบ

จุดเด่นสำคัญของหนังสือเล่มนี้คือการนำเสนอเนื้อหา *เชิงลึก* ควบคู่กับ *ตัวอย่างโค้ดจริง* และ *แนวคิดเบื้องหลัง* ทุกบทมีตัวอย่างการใช้งานจริงที่คุณสามารถคัดลอกไปทดลองได้ทันที ไม่ว่าจะเป็นโค้ดพื้นฐานสำหรับผู้เริ่มต้น หรือเทคนิคขั้นสูงที่ใช้ในระบบโปรแกรมระดับองค์กร หนังสือยังให้คำอธิบายเชิงเปรียบเทียบระหว่างแนวทางหลายๆ แบบ พร้อมข้อดีข้อเสีย เพื่อให้คุณเข้าใจบริบทการตัดสินใจเลือกใช้วิธีการแต่ละแบบ

ผู้เขียนหวังว่าหนังสือ *TypeScript Programming* จะเป็นเครื่องมืออันทรงคุณค่าในการพัฒนาทักษะของคุณ ไม่ว่าคุณจะเป็นนักพัฒนาที่ต้องการเสริมความรู้ใหม่ๆ หรือสถาปนิกซอฟต์แวร์ที่มองหาแนวทางการออกแบบระบบที่แข็งแกร่งและมีประสิทธิภาพ หนังสือเล่มนี้จะช่วยเปิดมุมมองใหม่ๆ ทำให้คุณเห็นพลังที่แท้จริงของ TypeScript และมั่นใจในการสร้างซอฟต์แวร์ที่ยั่งยืนและมีคุณภาพ

ท้ายที่สุดนี้ ขอขอบคุณผู้อ่านทุกท่านที่ให้ความสนใจและเลือกหนังสือเล่มนี้เป็นส่วนหนึ่งของการเรียนรู้ ขอให้การเดินทางในโลกของ TypeScript ของคุณเต็มไปด้วยแรงบันดาลใจ ความเข้าใจที่ลึกซึ้ง และความสำเร็จในการสร้างซอฟต์แวร์ที่ทรงพลัง

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 1 รู้จักกับ TypeScript (Introduction to TypeScript) 1

- รู้จักกับ TypeScript
- รู้จักกับ TypeScript (รายละเอียดเชิงลึก)
- TypeScript คืออะไร
- เปรียบเทียบ TypeScript กับ JavaScript
- ข้อดีของการใช้ TypeScript (Advantages of TypeScript)
- การติดตั้ง TypeScript
- เริ่มต้นใช้งาน TypeScript: ติดตั้งเครื่องมือ + ตัวอย่างโปรแกรมพร้อมผลรัน
- การใช้ tsc และ ts-node ใน TypeScript
- tsconfig.json คืออะไร?

บทที่ 2 ตัวแปรและชนิดข้อมูลพื้นฐาน (Variables and Basic Data Types)..... 28

- ตัวแปรและชนิดข้อมูลพื้นฐานใน TypeScript
- ตัวแปรและชนิดข้อมูลพื้นฐาน (เชิงลึก)
- let, const, var และ Scope ใน TypeScript
- string, number, boolean ใน TypeScript
- null, undefined, any, unknown, never ใน TypeScript

บทที่ 3 ฟังก์ชันพื้นฐาน (Basic Functions)..... 64

- ฟังก์ชันพื้นฐานใน TypeScript
- ฟังก์ชันพื้นฐานใน TypeScript — รายละเอียดเชิงลึก
- การกำหนด Parameter Type
- Optional Parameters คืออะไร?
- Default Parameters (พารามิเตอร์ที่มีค่าเริ่มต้น) ใน TypeScript
- Rest Parameters (พารามิเตอร์จำนวนไม่จำกัด) ใน TypeScript
- Arrow Functions และ Function Expressions ใน TypeScript

บทที่ 4 Array และ Tuple (Array and Tuple)..... 101

- Array และ Tuple

●การประกาศ Array ใน TypeScript ทั้งแบบ number[] และ Array<string>
●รายละเอียดเชิงลึกของ Tuple ใน TypeScript
●การเข้าถึงและจัดการสมาชิกใน Array และ Tuple
บทที่ 5 Object Type พื้นฐาน (Object Type) 124
●Object Type พื้นฐาน
●Object Type พื้นฐาน (TypeScript) เพิ่มเติม
●การกำหนดรูปแบบอ็อบเจกต์ (Defining Object Types)
●Optional Property ใน TypeScript
•Readonly Property ใน TypeScript
บทที่ 6 Interface และ Type Alias (Interface and Type Alias) 154
●Interface และ Type Alias
●Interface และ Type Alias (รายละเอียดเชิงลึก)
●การประกาศ Interface ใน TypeScript
●การใช้ type กับ Union และ Intersection ใน TypeScript
●ความแตกต่างและการเลือกใช้ Interface กับ Type Alias ใน TypeScript
บทที่ 7 Class และ OOP (Class and OOP) 187
●Class และ OOP ใน TypeScript
●Class และ OOP ใน TypeScript (เชิงลึก)
●การสร้างคลาสและ Constructor ใน TypeScript
●Property และ Method ใน TypeScript
●Access Modifiers ใน TypeScript
●การใช้ readonly ใน TypeScript
●this และ Context ใน TypeScript
บทที่ 8 Inheritance และ Abstract Class (Inheritance and Abstract Class) 230
●Inheritance และ Abstract Class
●Inheritance และ Abstract Class — รายละเอียดเชิงลึก
●การสืบทอดคลาส (Inheritance) ใน TypeScript
●extends และ super ใน TypeScript
●การใช้ Abstract Class และ Implements Interface ใน TypeScript

บทที่ 9 Enum และ Literal Types (Enum and Literal Types) 267
- Enum และ Literal Types
- Enum และ Literal Types — รายละเอียดเชิงลึก
- Enum แบบตัวเลข (Numeric Enum)
- Literal Types ใน TypeScript — 'small' | 'medium' | 'large'
- การใช้ Literal Types ร่วมกับ Union Types และการ Validate ค่า

บทที่ 10 Generics (Generics) 303
- Generics ใน TypeScript
- รายละเอียดเชิงลึกเรื่อง Generics ใน TypeScript
- ฟังก์ชัน Generic: <T>(arg: T): T
- คลาส Generic: class Box<T>
- การใช้ constraint: <T extends U>
- ประโยชน์ในการเขียนโค้ดที่ reusable ด้วย Generics

บทที่ 11 Union, Intersection, Type Guard (Union, Intersection, Type Guard) 339
- Union, Intersection, Type Guard
- Union, Intersection, Type Guard — รายละเอียดเชิงลึก
- Union Type: string | number
- Intersection Types: { a: number } & { b: string }
- Type Guards: typeof, instanceof, in
- User-Defined Type Guard ใน TypeScript

บทที่ 12 Modules, Import/Export (Modules, Import/Export) 378
- Modules, Import/Export
- Modules, Import/Export — รายละเอียดเชิงลึก
- การใช้ export และ import ใน TypeScript
- การแบ่งไฟล์และจัดระเบียบโปรเจกต์ใน TypeScript
- Named Export vs Default Export ใน TypeScript
- Path Alias ใน tsconfig.json

บทที่ 13 Advanced Type Features (Advanced Type Features) 417
- Advanced Type Features

- รายละเอียดเชิงลึก — Advanced Type Features ใน TypeScript
- Mapped Types: { [K in keyof T]: T[K] }
- Conditional Types: T extends U ? X : Y
- Template Literal Types: `ID_${string}`
- Recursive Types ใน TypeScript

บทที่ 14 Utility Types (Utility Types) 449

- Utility Types ใน TypeScript
- Utility Types — รายละเอียดเชิงลึก
- Partial<T>, Required<T>, และ Readonly<T>
- Pick<T, K> และ Omit<T, K> ใน TypeScript
- การสร้าง Utility Types เอง (Custom Utility Types)

บทที่ 15 Decorators (Experimental) (Decorators (Experimental)) 482

- Decorators (Experimental)
- รายละเอียดเชิงลึก: Decorators ใน TypeScript
- การเปิดใช้ Decorators ใน tsconfig.json
- Class, Method, Property Decorators ใน TypeScript
- การใช้ Reflect Metadata ใน TypeScript
- การใช้ Decorators ร่วมกับ Dependency Injection (DI) เช่น NestJS

บทที่ 16 Mixins (Mixins) 536

- Mixins ใน TypeScript
- Mixins ใน TypeScript — รายละเอียดเชิงลึก
- เทคนิค Multiple Inheritance ด้วย Mixins ใน TypeScript
- การแยกพฤติกรรมเป็นหลายคลาสย่อย (Decomposing Behavior into Smaller Classes) ใน TypeScript ด้วย Mixins
- การประยุกต์ใช้ Mixins ร่วมกับ Interface ใน TypeScript

บทที่ 17 Advanced Type Narrowing (Advanced Type Narrowing) 582

- Advanced Type Narrowing ใน TypeScript
- Advanced Type Narrowing (การจำกัดชนิดข้อมูลขั้นสูง) ใน TypeScript
- Discriminated Unions (Tagged Union) ใน TypeScript

●switch-case กับ Literal Type ใน TypeScript
•User Defined Type Guard ที่ซับซ้อน ใน TypeScript
บทที่ 18 Error Handling แบบ Type-Safe (Error Handling Type-Safe) 623
●Error Handling แบบ Type-Safe ใน TypeScript
●Error Handling แบบ Type-Safe ใน TypeScript (รายละเอียดเชิงลึก)
●try/catch/finally ใน TypeScript
●การสร้าง Custom Error Class ใน TypeScript
●การใช้งานกับ Result / Either Pattern ใน TypeScript
บทที่ 19 การใช้ TypeScript กับ Framework (TypeScript and Framework) 657
●การใช้ TypeScript กับ Framework
●การใช้ TypeScript กับ Framework (รายละเอียดเชิงลึก)
●React + TypeScript
●Angular + TypeScript (ใช้ TS โดย default)
●Node.js + Express + TypeScript
●NestJS (Framework ที่สร้างด้วย TypeScript)
บทที่ 20 Best Practices & Project Architecture (Best Practices & Project Architecture)
..... 712
●Best Practices & Project Architecture
●Best Practices & Project Architecture (เชิงลึก)
●การตั้งค่า tsconfig.json ให้เหมาะกับ Production
● Strict Mode และ noImplicitAny ใน TypeScript
● การใช้ ESLint + Prettier ร่วมกับ TypeScript อย่างมีประสิทธิภาพ
● การแยก Layer: Services, Repositories, Models
●การจัดการ Dependency ด้วย DI Container

บรรณานุกรม 763

บทที่ 1
รู้จักกับ TypeScript
(Introduction to TypeScript)

เนื้อหา

- รู้จักกับ TypeScript
- รู้จักกับ TypeScript (รายละเอียดเชิงลึก)
- TypeScript คืออะไร
- เปรียบเทียบ TypeScript กับ JavaScript
- ข้อดีของการใช้ TypeScript (Advantages of TypeScript)
- การติดตั้ง TypeScript
- เริ่มต้นใช้งาน TypeScript: ติดตั้งเครื่องมือ + ตัวอย่างโปรแกรมพร้อมผลรัน
- การใช้ tsc และ ts-node ใน TypeScript
- tsconfig.json คืออะไร?

บทนำ

ในยุคที่การพัฒนาเว็บแอปพลิเคชันและซอฟต์แวร์ฝั่งไคลเอนต์เติบโตอย่างรวดเร็ว JavaScript ยังคงเป็นภาษาหลักที่นักพัฒนาทั่วโลกใช้กันอย่างแพร่หลาย อย่างไรก็ตาม ข้อจำกัดหลายประการของ JavaScript โดยเฉพาะในเรื่องระบบประเภทข้อมูล (type system) ทำให้นักพัฒนาเริ่มมองหาเครื่องมือที่สามารถช่วยเพิ่มประสิทธิภาพและความปลอดภัยในการพัฒนา TypeScript จึงถือกำเนิดขึ้นในฐานะภาษาที่ต่อยอดจาก JavaScript โดยมีระบบ type แบบ static ที่ช่วยลดข้อผิดพลาดในระดับ compile-time และเพิ่มความสามารถในการจัดการโค้ดขนาดใหญ่

TypeScript เป็นภาษาที่พัฒนาโดย Microsoft โดยมีเป้าหมายเพื่อเติมเต็มข้อบกพร่องของ JavaScript โดยเฉพาะอย่างยิ่งในแง่ของการเขียนโปรแกรมเชิงวัตถุ การควบคุมชนิดข้อมูล และการสนับสนุนการพัฒนาแอปพลิเคชันขนาดใหญ่ จุดเด่นของ TypeScript คือการที่มันสามารถคอมไพล์กลับเป็น JavaScript ซึ่งทำให้สามารถใช้งานได้บนแพลตฟอร์มหรือเบราว์เซอร์ใดๆ ที่รองรับ JavaScript โดยไม่ต้องติดตั้งอะไรเพิ่มเติมในฝั่งผู้ใช้

การเปรียบเทียบระหว่าง TypeScript กับ JavaScript เผยให้เห็นถึงข้อดีหลายประการของ TypeScript เช่น ความสามารถในการตรวจจับข้อผิดพลาดในขณะเขียนโค้ด (early error detection), การใช้ interface และ generic เพื่อเขียนโค้ดที่ยืดหยุ่นและปลอดภัยยิ่งขึ้น ตลอดจนความสามารถในการ

ใช้ฟีเจอร์ของ ECMAScript เวอร์ชันใหม่ก่อนที่เบราว์เซอร์จะรองรับอย่างเต็มรูปแบบ ทั้งนี้ TypeScript ไม่ได้มาแทนที่ JavaScript แต่กลับขยายขีดความสามารถของมัน

ข้อดีสำคัญอีกประการของ TypeScript คือการสนับสนุนเครื่องมือพัฒนา (IDE) อย่างสมบูรณ์แบบ เช่น Visual Studio Code ซึ่งสามารถตรวจสอบ type, แนะนำโค้ด, และจัดการ refactor ได้อย่างชาญฉลาด ส่งผลให้กระบวนการพัฒนามีประสิทธิภาพและรวดเร็วขึ้น โดยเฉพาะในโปรเจกต์ขนาดใหญ่ที่มีหลายไฟล์และทีมพัฒนาหลายคน การใช้ TypeScript ช่วยลดปัญหาการสื่อสารทางเทคนิค และเพิ่มความสามารถในการอ่านและบำรุงรักษาโค้ดในระยะยาว

ในการเริ่มต้นใช้งาน TypeScript นักพัฒนาสามารถติดตั้งได้ง่ายผ่านคำสั่ง npm install -g typescript ซึ่งจะติดตั้งตัวคอมไพเลอร์ tsc สำหรับแปลงไฟล์ .ts เป็น .js และยังสามารถใช้ ts-node เพื่อรันไฟล์ TypeScript ได้โดยตรงในสภาพแวดล้อม Node.js โดยไม่ต้องคอมไพล์ล่วงหน้า นอกจากนี้การสร้างไฟล์ tsconfig.json ยังช่วยกำหนดค่าการคอมไพล์และโครงสร้างของโปรเจกต์ได้อย่างเป็นระบบ

ไฟล์ tsconfig.json เป็นหัวใจสำคัญในการจัดการโปรเจกต์ TypeScript ซึ่งช่วยให้นักพัฒนาสามารถควบคุมพฤติกรรมของคอมไพเลอร์ เช่น การกำหนดโฟลเดอร์ต้นทางและปลายทางของไฟล์, การเลือกใช้ ECMAScript target version, และการเปิดใช้งานฟีเจอร์ strict typing ต่างๆ การตั้งค่าอย่างเหมาะสมในไฟล์นี้ส่งผลโดยตรงต่อคุณภาพของโค้ดและความง่ายในการดูแลโปรเจกต์ในระยะยาว

หนังสือเล่มนี้เริ่มต้นด้วยบทที่ 1: รู้จักกับ TypeScript เพื่อปูพื้นฐานความเข้าใจสำหรับผู้อ่านทุกระดับ โดยครอบคลุมตั้งแต่คำอธิบายพื้นฐานเกี่ยวกับภาษานี้ เปรียบเทียบกับ JavaScript อธิบายข้อดีและการใช้งานจริง ไปจนถึงการติดตั้งเครื่องมือและการตั้งค่าพื้นฐานที่จำเป็นสำหรับเริ่มต้นใช้งาน TypeScript อย่างมีประสิทธิภาพ ซึ่งจะวางรากฐานสำคัญก่อนเข้าสู่เนื้อหาเชิงลึกในบทต่อๆ ไป ทั้งในด้านการเขียนโปรแกรม การใช้งาน framework และการจัดการโปรเจกต์ขนาดใหญ่ในระดับมืออาชีพ.

รู้จักกับ TypeScript

1.1 TypeScript คืออะไร

TypeScript คือภาษาโปรแกรมที่ถูกพัฒนาโดย Microsoft เป็น **Superset ของ JavaScript** (หมายถึงทุกอย่างใน JavaScript ใช้ได้ใน TypeScript) แต่เพิ่มคุณสมบัติเด่นคือ **Static Type Checking** หรือระบบการตรวจสอบชนิดของข้อมูลตั้งแต่ตอนคอมไพล์

TypeScript ถูกออกแบบมาเพื่อช่วยให้โค้ด JavaScript:

- **มีความชัดเจน (Explicit)**
- **ลดข้อผิดพลาด**
- **ดูแลรักษาได้ง่ายโดยเฉพาะในโปรเจกต์ใหญ่**

1.2 เปรียบเทียบกับ JavaScript

ด้าน	JavaScript	TypeScript

ด้าน	JavaScript	TypeScript
ประเภทภาษา	Dynamic Typing	Static Typing
การตรวจ error	ตอนรันโปรแกรม (Runtime)	ตอนคอมไพล์ (Compile-time)
ใช้ในโปรเจกต์ใหญ่	ไม่เหมาะเท่าไร	เหมาะมาก
รองรับ OOP เต็มรูปแบบ	จำกัด	รองรับ class, interface, generic ฯลฯ
การใช้กับ IDE	จำกัด	รองรับ IntelliSense เต็มที่

ตัวอย่าง:

```
// JavaScript: ไม่มีการระบุชนิดข้อมูล
function greet(name) {
  return "Hello " + name;
}
// TypeScript: ระบุชนิดข้อมูลชัดเจน
function greet(name: string): string {
  return `Hello ${name}`;
}
```

1.3 ข้อดีของการใช้ TypeScript

- ตรวจสอบข้อผิดพลาดตั้งแต่ก่อนรัน
- ช่วยให้ IDE อย่าง VS Code ให้คำแนะนำที่แม่นยำ
- โค้ดชัดเจนและอ่านง่าย (self-documenting)
- รองรับแนวทางการเขียนโปรแกรมเชิงวัตถุ (OOP)
- เหมาะกับโปรเจกต์ที่มีหลายคนพัฒนา
- ทำงานร่วมกับ JavaScript ได้ 100%

1.4 การติดตั้ง TypeScript

ขั้นตอนการติดตั้งผ่าน npm:

```
npm install -g typescript
```

คำสั่งนี้จะติดตั้งคำสั่ง tsc (TypeScript Compiler) ให้ใช้งานได้ทั่วเครื่อง

1.5 ใช้ tsc และ ts-node

✴ **tsc (TypeScript Compiler)**

ใช้สำหรับ **แปลงไฟล์ .ts เป็น .js**

tsc hello.ts

จะได้ไฟล์ hello.js ซึ่งสามารถรันด้วย Node.js ได้

✶ **ts-node (TypeScript Node Executor)**

หากต้องการรันไฟล์ .ts ได้ทันทีโดยไม่ต้องคอมไพล์:

npm install -g ts-node

แล้วใช้:

ts-node hello.ts

1.6 สร้างและใช้งาน tsconfig.json

tsconfig.json คือไฟล์ที่ใช้กำหนดการตั้งค่าของโปรเจกต์ TypeScript เช่น:

- เวอร์ชัน JavaScript เป้าหมาย (ES5, ES6, ES2020)
- โฟลเดอร์ต้นทาง/ปลายทาง
- ตัวเลือก strict mode

วิธีสร้าง:

tsc --init

จะได้ไฟล์ tsconfig.json พร้อม default config

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "outDir": "dist",
    "rootDir": "src"
  },
  "include": ["src"],
  "exclude": ["node_modules"]
}
```

การคอมไพล์โปรเจกต์ทั้งหมด:

tsc

(เมื่อมี tsconfig.json แล้ว ไม่ต้องระบุไฟล์ใด ๆ)

รู้จักกับ TypeScript (รายละเอียดเชิงลึก)

1.1 TypeScript คืออะไร (เชิงลึก)

TypeScript เป็น **ภาษาการเขียนโปรแกรมที่ขยายจาก JavaScript** โดยเพิ่มระบบ **Static Type System** และฟีเจอร์ของภาษาโปรแกรมระดับสูง เช่น:

- Type Annotation (name: string)
- Interface, Enum, Tuple
- Generic (คล้ายกับใน Java/C++)
- Decorator (experimental)
- Module System (ESM/CJS)
- Type Narrowing และ Type Guard

ประโยชน์หลักคือ: ช่วยจับข้อผิดพลาดก่อนการรัน (early error detection), เพิ่มประสิทธิภาพการ refactor, และเสริมความเข้าใจให้กับ IDE

TypeScript ไม่ได้แทนที่ JavaScript

แต่เป็น “Layer บน JavaScript” ที่ **Compile ไปเป็น JavaScript** ที่รันบนเบราว์เซอร์หรือ Node.js ได้

1.2 เปรียบเทียบกับ JavaScript (เชิงลึก)

ด้าน Static Type

Aspect	JavaScript	TypeScript
Typing	Dynamic	Static (optional)
Error Detection	Runtime only	Compile-time checking
Tooling	Limited	Excellent (VS Code, tsserver)
Refactoring	เสี่ยง	ปลอดภัยกว่า
Code Base Growth	ลำบากในระยะยาว	รองรับดี (scalability)

ตัวอย่างที่ชัดเจน:

```
// JavaScript
function getLength(s) {
  return s.length; // ถ้า s เป็น null จะ error
}
// TypeScript
function getLength(s: string | null): number {
  if (s === null) return 0;
  return s.length;
}
```

TypeScript บังคับให้คุณจัดการทุกกรณี (null safety)

☐ 1.3 ข้อดีของการใช้ TypeScript (เชิงลึก)

☐ 1. ป้องกันบั๊กที่มักพบใน JavaScript

TypeScript ตรวจสอบ type ต่าง ๆ เช่น undefined, null, หรือ any ช่วยป้องกัน runtime error เช่น:

```
const user: { name: string, age: number } = { name: "John", age: "20" };
// Error: Type 'string' is not assignable to type 'number'
```

☐ 2. ช่วยในการ Refactor โค้ด

เมื่อเปลี่ยนชื่อ property, function, หรือ type ต่าง ๆ – TypeScript จะช่วยตรวจสอบว่าโค้ดทุกจุดยังทำงานถูกต้องหรือไม่

☐ 3. IDE ที่ทรงพลัง

TypeScript มาพร้อมกับ tsserver ซึ่งทำงานร่วมกับ VS Code ทำให้ได้ประโยชน์จาก:

- Autocomplete ที่แม่นยำ
- Inline error check
- Jump to definition / symbol
- Rename symbol อย่างปลอดภัย

☐ 4. เหมาะกับทีมและโปรเจกต์ขนาดใหญ่

- บังคับ structure ของโค้ดได้ด้วย interface
- ป้องกัน contract mismatch (API, DTO, etc.)
- แยก layer ได้ชัดเจน

☐ 1.4 การติดตั้ง TypeScript (เชิงลึก)

☐ ติดตั้ง global (ใช้ได้ทั้งเครื่อง)

```
npm install -g typescript
```

จะได้คำสั่ง tsc ใน PATH

ตรวจสอบเวอร์ชัน:

```
tsc --version
```

☐ ติดตั้งในโฟลเดอร์โปรเจกต์

```
npm install --save-dev typescript
```

ใช้ร่วมกับโปรเจกต์ Node.js ได้อย่างเป็นระบบ (ไม่รบกวน global)

☐ 1.5 ใช้งาน tsc และ ts-node (เชิงลึก)

☐ tsc คือ TypeScript Compiler

ทำหน้าที่:

- ตรวจสอบประเภทข้อมูล (type checking)
- แปลง .ts เป็น .js
- อ้างอิงจาก tsconfig.json

```
tsc path/to/file.ts      # แบบ single file
tsc                      # แบบทั้งโปรเจกต์ (มี tsconfig)
```

ts-node: สำหรับพัฒนาแบบเร็ว

ไม่ต้องคอมไพล์ก่อนรัน:

```
npx ts-node index.ts
```

เหมาะกับ dev mode / CLI tool / script ทดสอบ

เบื้องหลังใช้ typescript + ts-node/register + node runtime

ความต่าง:

เครื่องมือ	คอมไพล์ไฟล์	รันจริง	ใช้ใน dev
tsc			
ts-node	(ใน memory)		

1.6 การสร้างและใช้งาน tsconfig.json (เชิงลึก)

ไฟล์ tsconfig.json เป็นหัวใจของโปรเจกต์ TypeScript บอก compiler ว่า:

- จะ compile อะไร
- ตั้งค่าการทำงานอย่างไร
- จะ output ไปไหน

คำสั่งเริ่มต้น:

```
tsc --init
```

จะได้โครงสร้างเช่น:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true
  },
  "include": ["src"],
```

```
  "exclude": ["node_modules"]
}
```

คำอธิบาย:

ตัวเลือก	ความหมาย
target	เวอร์ชัน JavaScript ที่ output (ES5, ES6, ES2020, etc.)
module	ระบบ module (commonjs, esnext)
outDir	โฟลเดอร์ผลลัพธ์
rootDir	ต้นทางของไฟล์ TypeScript
strict	เปิด strict mode (เช่น noImplicitAny, strictNullChecks)
esModuleInterop	ให้ import style แบบ ES module ได้ใน commonjs
include, exclude	ระบุไฟล์ที่รวม/ยกเว้นในการคอมไพล์

ตัวอย่างโครงสร้างโปรเจกต์

```
my-ts-app/
├── src/
│   └── index.ts
├── dist/
│   └── index.js (output)
├── tsconfig.json
└── package.json
```

✳ □ สรุปท้ายบท

หัวข้อ	เนื้อหา
TypeScript คือ	ภาษาที่เพิ่ม type-checking บน JavaScript
ต่างจาก JS อย่างไร	ตรวจ error ตั้งแต่ compile-time, IDE ดีกว่า
ข้อดี	ป้องกันบั๊ก, เหมาะกับทีม, ใช้กับโปรเจกต์ใหญ่
การติดตั้ง	npm install -g typescript
คอมไพล์	tsc, หรือรันตรงด้วย ts-node
tsconfig.json	ควบคุมการคอมไพล์ทั้งโปรเจกต์

TypeScript คืออะไร

TypeScript คืออะไร (What is TypeScript)

TypeScript คือ **ภาษาการเขียนโปรแกรมที่ถูกออกแบบมาเพื่อขยายความสามารถของ JavaScript** โดยเฉพาะอย่างยิ่งในด้าน **Static Typing**, **การรองรับ OOP**, และ **เครื่องมือพัฒนา (IDE Support)** ที่มีประสิทธิภาพ

TypeScript ถูกพัฒนาโดย Microsoft และเปิดตัวครั้งแรกในปี 2012 ปัจจุบันเป็นหนึ่งในภาษายอดนิยมในโลก JavaScript/Node.js โดยเฉพาะกับโปรเจกต์ที่ต้องการความมั่นคงและสามารถดูแลระยะยาว

จุดเด่นหลักของ TypeScript

1. Static Type System

คุณสามารถระบุประเภทข้อมูล (type) ของตัวแปร ฟังก์ชัน และอ็อบเจกต์ได้ล่วงหน้า เช่น:

```
let name: string = "Alice";
function square(x: number): number {
  return x * x;
}
```

เมื่อเขียนผิด เช่นส่ง string เข้า square() ที่รับ number – TypeScript จะตรวจพบก่อนรันทันที

2. Superset ของ JavaScript

TypeScript คือ **Superset ของ JavaScript** หมายความว่า:

- ทุกโค้ด JavaScript ที่ถูกต้อง = เป็น TypeScript ที่ถูกต้องด้วย
- เพิ่มความสามารถ “บน” JavaScript โดยไม่เปลี่ยนแปลงพฤติกรรมพื้นฐาน

เช่น:

```
// JavaScript
const user = { name: "John", age: 30 };
// TypeScript (เทียบเท่า, แต่สามารถเพิ่ม type ได้)
const user: { name: string; age: number } = { name: "John", age: 30 };
```

3. Compile-time Error Checking

TypeScript ไม่รันโค้ดโดยตรง แต่ต้อง **คอมไพล์ (compile)** ไปเป็น JavaScript ก่อน

ระหว่างคอมไพล์ TypeScript จะ:

- ตรวจจับข้อผิดพลาดเกี่ยวกับชนิดข้อมูล
- แจ้งเตือนทันทีโดยไม่ต้องรันจริง

4. ทำงานได้กับทุก JS Project

คุณสามารถ:

- เพิ่ม .ts หรือ .d.ts เข้าไปในโปรเจกต์ JavaScript เดิมได้ทันที
- ทำงานร่วมกับ library JS เช่น React, Vue, Express, Lodash ฯลฯ ได้โดยใช้ **type declaration file (@types/...)**

---

5. □ เครื่องมือพัฒนาอัจฉริยะ (Powerful IDE Support)

เมื่อใช้ VS Code หรือ IDE ที่รองรับ TypeScript:

- โค้ดจะมี autocomplete, refactor อย่างปลอดภัย
- แสดง error ทันทีที่เขียนผิด
- สามารถดู type ของค่าทุกตัวแบบ inline

เช่น:

```
const point = { x: 10, y: 20 };
point.z = 5; // □ Property 'z' does not exist on type ...
```

□ เป้าหมายของ TypeScript

เป้าหมายหลัก	คำอธิบาย
□ ความปลอดภัยของโค้ด	ตรวจสอบความถูกต้องของ type ตั้งแต่แรก
□ ความเข้าใจในโค้ด	ทำให้คนอื่นอ่านโค้ดเข้าใจแม้ไม่เขียนเอง
□ รองรับ refactor	เปลี่ยนชื่อ, โครงสร้าง, หรือ logic ได้โดยมั่นใจ
□ ทำงานร่วมกันในทีม	ป้องกันการส่งค่าผิด type หรือ contract API
□ ใช้ร่วมกับ JS เดิมได้	ไม่ต้องเขียนใหม่ทั้งหมด

□ ความเข้าใจผิดที่พบบ่อย

ความเข้าใจผิด	ความจริง
TypeScript = ภาษาใหม่	จริง ๆ แล้วเป็นการ “เพิ่มความสามารถให้ JavaScript”
ต้องใช้เฉพาะกับโปรเจกต์ใหม่	TypeScript ใช้กับโค้ด JavaScript เดิมได้
เขียนยากกว่า JS	อาจซับซ้อนในตอนแรก แต่ปลอดภัยและช่วยให้โค้ดดูดีระยะยาว
ต้องใช้กับ Framework เท่านั้น	ใช้กับ Node.js, CLI, หรือ browser script ได้ทั้งหมด

✳ □ สรุปสั้น: TypeScript คือ...

"ภาษาที่ช่วยให้ JavaScript ฉลาดขึ้น ปลอดภัยขึ้น และดูแลได้ง่ายขึ้น"
โดยเพิ่มระบบ type, ตรวจ error ตั้งแต่ก่อนรัน, และเสริมเครื่องมือพัฒนาแบบที่ JavaScript ธรรมดาทำไม่ได้

เปรียบเทียบ TypeScript กับ JavaScript

□ ความสัมพันธ์โดยภาพรวม

- **JavaScript (JS)** เป็น **ภาษาต้นทาง** ที่เบราว์เซอร์และ Node.js เข้าใจโดยตรง
- **TypeScript (TS)** เป็น **ภาษาที่ถูกคอมไพล์ (compiled language)** ที่แปลงเป็น JavaScript ก่อนรัน

□ ทุกไฟล์ TypeScript จะต้องถูกแปลงเป็น JavaScript เพื่อใช้งานจริง

□ TypeScript คือ **Superset ของ JavaScript** ⟶ ทุก JavaScript ที่ถูกต้อง = ถูกต้องใน TypeScript

---

□ เปรียบเทียบโดยสรุป

ด้านเปรียบเทียบ	JavaScript	TypeScript
ประเภทภาษา	Dynamic (ไม่มีชนิดข้อมูลแน่นอน)	Static (ระบุชนิดข้อมูลได้ชัดเจน)
ตรวจข้อผิดพลาด	ตอนรัน (Runtime Errors)	ตอนคอมไพล์ (Compile-time Errors)
ระบบชนิดข้อมูล (Type)	ไม่มีในตัวภาษา	มีระบบ Type Annotation / Inference
การใช้งานใน IDE	Autocomplete พื้นฐาน	Autocomplete + Type Checking + Jump to Def.
รองรับ OOP เต็มรูปแบบ	จำกัด	รองรับ Class, Interface, Access Modifier
ขนาดโปรเจกต์ที่เหมาะสม	เล็ก-กลาง	กลาง-ใหญ่, ทำงานร่วมกันหลายคน
ความง่ายในการเริ่มต้น	ง่าย, เริ่มเขียนได้เลย	ต้องติดตั้งและเรียนรู้เพิ่มเล็กน้อย
ความปลอดภัยของโค้ด	ต่ำ (พลาดง่าย)	สูง (แจ้งเตือนตั้งแต่ต้น)
ทำงานร่วมกับ JS เดิม	–	ทำงานร่วมได้ 100%

□ เปรียบเทียบผ่านตัวอย่าง

□ **ตัวอย่าง 1: ไม่มีชนิดข้อมูล**

JavaScript:

```
function greet(name) {
  return "Hello, " + name.toUpperCase();
}

greet(123); // □ ไม่มี error ตอนเขียน แต่ crash ตอนรัน
```

TypeScript:

```
function greet(name: string): string {
  return "Hello, " + name.toUpperCase();
}

greet(123); // □ Compile-time error: Argument of type 'number' is not assignable to 'string'
```

□ ตัวอย่าง 2: Object Structure

JavaScript:

```
const user = { name: "Alice" };
console.log(user.age); // undefined, ไม่มี warning
```

TypeScript:

```
type User = { name: string; age: number };
const user: User = { name: "Alice" }; // □ Error: Property 'age' is missing
```

□ ตัวอย่าง 3: IDE Autocomplete

JavaScript:

```
let person = { name: "John", age: 30 };
// IDE เดา type ได้บางส่วน (limited intellisense)
```

TypeScript:

```
type Person = { name: string; age: number };
let person: Person = { name: "John", age: 30 };
// IDE แสดง autocomplete, ป้องกัน property ผิด
```

□ ประโยชน์ที่ JavaScript ไม่มีแต่ TypeScript มี

Feature	อธิบาย
□ Type Annotation	กำหนดว่า x: number, y: boolean, z: string[] ได้

Feature	อธิบาย
☐ Interface / Type Alias	สร้างรูปแบบของ object และ enforce ได้
☐ Enum	นิยามค่าคงที่ในแบบที่ JavaScript ไม่มี
☐ Generics	ทำฟังก์ชัน/class ที่รองรับหลายชนิดข้อมูลอย่างปลอดภัย
☐ Type Inference	ไม่ต้องประกาศ type ทุกครั้ง แต่ TS เดาได้ฉลาด
☐ Custom Types	เช่น union, intersection, literal types (`"asc"

☐ ทำไมคนยังใช้ JavaScript?

เหตุผล	คำอธิบาย
☐ เริ่มต้นง่าย	ไม่ต้องติดตั้งอะไรเลย
☐ ทำงานได้ทุกที่	ทุกเบราว์เซอร์, ทุก runtime
☐ Community ใหญ่	มีไลบรารีทุกอย่าง
☐ ใช้ในโค้ดที่ไม่ต้องการ type safety	เช่นโค้ดสั้น ๆ, script ชั่วคราว

☐ แต่เมื่อโปรเจกต์เริ่มใหญ่ขึ้น หรือมีหลายคนพัฒนา ⟶ **TypeScript คือทางเลือกที่ดีกว่า**

☐ สรุปความแตกต่างแบบภาพรวม

ประเด็นหลัก	JavaScript	TypeScript
Type Safety	☐ ไม่มี	☐ มี
ความปลอดภัย	ต่ำ	สูง
ความเร็วเริ่มต้น	เร็วมาก	ช้ากว่าเล็กน้อย
รองรับโปรเจกต์ใหญ่	จำกัด	เหมาะสมมาก

✳ ☐ บทสรุปสั้น:

TypeScript = JavaScript + ความปลอดภัย + ความแม่นยำ + IDE อัจฉริยะ

เหมาะกับโปรเจกต์ที่ต้องการ **ควบคุมคุณภาพโค้ด**, **ทำงานเป็นทีม**, และ **ขยายในระยะยาว**

ข้อดีของการใช้ TypeScript (Advantages of TypeScript)

☐ 1. ตรวจจับข้อผิดพลาดได้ตั้งแต่ก่อนรัน (Early Error Detection)

☐ **อธิบาย:**

TypeScript ใช้ **Static Type Checking** ⟶ ตรวจสอบชนิดของตัวแปร ฟังก์ชัน และ object ต่าง ๆ **ตอนคอมไพล์** ไม่ใช่ตอนรันจริง (runtime)

```
function add(x: number, y: number): number {
  return x + y;
}

add("10", 5); // ❌ Error: Argument of type 'string' is not assignable to parameter of type 'number'
```

✅ **ผลลัพธ์:**

- ลดโอกาสเจอบั๊กขณะรันจริง
- ค้นพบข้อผิดพลาดเร็วขึ้น (เร็วกว่า runtime error ของ JS)

✅ 2. ทำงานกับ IDE ได้ยอดเยี่ยม (Intelligent IDE Support)

✅ **VS Code และ IDE อื่น ๆ จะมี:**

- Autocomplete ที่แม่นยำ (based on types)
- ตรวจ error ทันทีที่พิมพ์
- Jump to definition / Go to type / Rename symbol
- Inline documentation / parameter hints

```
interface User {
  id: number;
  name: string;
}

const user: User = { id: 1, name: "Alice" };
user. // ⟵ แสดง suggestion ครบ
```

✅ **ผลลัพธ์:**

- เขียนโค้ดได้เร็วขึ้น
- ลดการพิมพ์ผิด
- เข้าใจโค้ดได้แม้ไม่เขียนเอง

✅ 3. บังคับโครงสร้างโค้ดให้ชัดเจน (Code Contracts & Design)

ใช้ interface, type, enum, class ในการกำหนดข้อตกลงระหว่างไฟล์/module/function:

```
interface Product {
  id: number;
```

```
  title: string;
  price: number;
}

function print(product: Product) {
  console.log(product.title);
}
```

ประโยชน์:

- เพิ่มความสอดคล้องในทีม
- ทำให้โค้ด **self-documenting** (อ่านรู้เลยว่า type อะไร)
- ป้องกันการใช้ผิดโครงสร้าง (invalid API data)

---

4. เหมาะสำหรับโปรเจกต์ขนาดใหญ่ (Scalability)

เหตุผล:

- แยกโค้ดเป็น module ได้ชัด
- จัดการ dependency ได้ดี
- รองรับการ refactor ใหญ่ ๆ ได้อย่างปลอดภัย
- ใช้ร่วมกับระบบ build tools ได้ดี (Webpack, Vite, tsup)

ผลลัพธ์:

- พัฒนา/ดูแลได้ง่ายแม้โค้ดมีหลายหมื่นบรรทัด
- เหมาะกับองค์กร, team-based development

---

5. ปลอดภัยจากค่าที่ไม่คาดคิด (Type Safety & Null Safety)

ตัวอย่าง:

```
function length(s: string | null): number {
  return s.length; //  Error
}
```

TypeScript บังคับให้คุณตรวจ null ก่อนใช้งาน:

```
function length(s: string | null): number {
  if (s === null) return 0;
  return s.length;
}
```

ผลลัพธ์:

- ป้องกันบั๊กที่เกิดจาก undefined หรือ null ที่ JS มองข้าม

- ช่วยเพิ่มความแม่นยำโดยอัตโนมัติ

□ 6. รองรับฟีเจอร์ของภาษาโปรแกรมสมัยใหม่

TypeScript รองรับ:

- **Class**, **Inheritance**, **Access Modifier**
- **Generics** (เหมือน Java/C#)
- **Union Type**, **Intersection Type**
- **Decorator** (experimental)
- **Type Guard**, **Type Narrowing**

```
function print<T>(value: T): void {
  console.log(value);
}
```

สิ่งเหล่านี้ไม่มีใน JavaScript ⟶ ทำให้โค้ดมีพลังและยืดหยุ่นมากขึ้น

□ 7. ทำงานร่วมกับ JavaScript เดิมได้ 100%

- สามารถเพิ่ม TypeScript เข้ากับโปรเจกต์ JavaScript เดิมทีละส่วนได้ (migration)
- ใช้ .d.ts (declaration file) หรือ @types/... สำหรับ library JS
- แปลง .ts เป็น .js โดยอัตโนมัติในขั้นตอน build

tsc yourfile.ts ⟶ yourfile.js

□ 8. มี Ecosystem ที่แข็งแรง

ด้าน	ข้อมูล
ไลบรารี JS	รองรับทุกตัว เช่น React, Express, Vue
Framework	React + TS, Next.js, NestJS, Angular
Community	ใหญ่, เติบโตเร็ว, มี lib @types เต็มไปหมด
Tooling	VS Code, Vite, ESLint, TypeDoc

□ 9. สนับสนุนการเขียนโค้ดเชิงสัญญะ (Declarative / Defensive Programming)

- เพิ่มความชัดเจนของ input/output
- ป้องกัน logic ผิดด้วย type

```
type SortOrder = "asc" | "desc";
function sortBy(field: string, order: SortOrder) {}
sortBy("price", "ascending"); // □ Error
```

✳ □ สรุปสั้น: ข้อดีของ TypeScript

ข้อดี	อธิบายสั้น
□ Type Safety	ป้องกันบั๊กจากชนิดข้อมูล
□ IDE Friendly	Smart IntelliSense
□ Strong Structure	โค้ดอ่านง่าย ดูแลง่าย
□ Refactor Friendly	เปลี่ยนโค้ดโดยไม่พังระบบอื่น
□ ใช้ได้กับทุก JS lib	มี @types รองรับ
□ รองรับ OOP & Generics	เหมาะกับโปรเจกต์ซับซ้อน
□ เหมาะกับทีมพัฒนา	ทำงานร่วมกันได้แบบมีมาตรฐาน

การติดตั้ง TypeScript

ผ่านคำสั่ง npm install -g typescript (และแบบอื่น ๆ)

□ 1. ติดตั้งแบบ Global

□ **คำสั่ง:**

npm install -g typescript

□ **ความหมาย:**

- ติดตั้ง TypeScript CLI (tsc) ให้ใช้งานได้ **ทุกที่ในเครื่อง**
- เหมาะสำหรับ: ผู้ที่ต้องการ compile โค้ด .ts ด้วยตนเอง

□ **ตรวจสอบว่า tsc ติดตั้งสำเร็จ:**

tsc --version

เช่น:

$ tsc --version

5.4.3

□ 2. ติดตั้งแบบ Local ในโปรเจกต์

□ **คำสั่ง:**

npm install --save-dev typescript

□ **ความหมาย:**

- ติดตั้ง TypeScript เฉพาะภายในโปรเจกต์นั้น ๆ
- ไม่กระทบระบบ global

- นิยมใช้ในระบบ build (เช่น Vite, Webpack, tsup)

☐ **สร้างไฟล์ config:**

npx tsc --init

☐ 3. ใช้ tsc (TypeScript Compiler)

☐ **Compile แบบ manual:**

tsc hello.ts

จะได้ไฟล์:

hello.js

ถ้าไม่มี tsconfig.json ⟶ tsc จะใช้ default option

ถ้ามี tsconfig.json ⟶ tsc จะคอมไพล์โปรเจกต์ทั้งชุด

☐ 4. ติดตั้ง ts-node สำหรับรันไฟล์ .ts ตรง ๆ

☐ **คำสั่ง:**

npm install -g ts-node

หรือแบบ local:

npm install --save-dev ts-node

☐ **ใช้งาน:**

ts-node hello.ts

เหมาะกับการพัฒนา CLI, script หรือไฟล์ที่ไม่ต้องการ build ล่วงหน้า

☐ 5. ติดตั้งเสริมแนะนำ (Optional)

Package	ใช้ทำอะไร
@types/node	ใช้เมื่อเขียน TypeScript ร่วมกับ Node.js
typescript	ตัวคอมไพเลอร์หลัก
ts-node	รันไฟล์ .ts ได้ทันที
tsconfig.json	กำหนด config ของ TypeScript

☐ โครงสร้างโปรเจกต์ TypeScript พื้นฐาน

my-ts-app/

```
├── src/
│   └── index.ts
```

```
├── dist/
│   └── index.js (output)
├── package.json
└── tsconfig.json
```

src คือไฟล์ต้นฉบับ

dist คือโฟลเดอร์ไฟล์ .js ที่คอมไพล์แล้ว

✳ □ สรุป

วิธีติดตั้ง	คำสั่ง	เหมาะกับ
Global	npm install -g typescript	ใช้ tsc ได้ทุกที่
Local	npm install --save-dev typescript	โปรเจกต์ที่ใช้ build tools
ts-node	npm install -g ts-node	รัน .ts ตรง ๆ แบบ dev

เริ่มต้นใช้งาน TypeScript: ติดตั้งเครื่องมือ + ตัวอย่างโปรแกรมพร้อมผลรัน

□ ขั้นตอนที่ 1: ติดตั้งเครื่องมือ (IDE + Node.js + TypeScript)

□ 1.1 ติดตั้ง VS Code (IDE แนะนำ)

- ดาวน์โหลดได้ที่: https://code.visualstudio.com/
- ติดตั้งส่วนเสริมแนะนำ:
  - □ ESLint
  - □ Prettier
  - □ TypeScript Hero หรือ TypeScript Essentials *(optional)*

□ 1.2 ติดตั้ง Node.js

- ดาวน์โหลด: https://nodejs.org/
- ตรวจสอบว่า node & npm พร้อมใช้:

```
node -v
npm -v
```

□ 1.3 ติดตั้ง TypeScript แบบ global

```
npm install -g typescript ts-node
```

ตรวจสอบเวอร์ชัน:

tsc -v

ts-node -v

□ ขั้นตอนที่ 2: สร้างโปรเจกต์ TypeScript

mkdir my-ts-app

cd my-ts-app

npm init -y

ติดตั้ง TypeScript สำหรับโปรเจกต์ (local):

npm install --save-dev typescript ts-node @types/node

สร้าง config:

npx tsc --init

□ ได้ไฟล์ tsconfig.json

□ ขั้นตอนที่ 3: เขียนโปรแกรมตัวอย่าง

สร้างโฟลเดอร์ src/:

mkdir src

สร้างไฟล์ src/index.ts แล้วใส่โค้ดนี้:

```
// src/index.ts

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

const result = greet("TypeScript");
console.log(result);
```

⚙□ ขั้นตอนที่ 4: คอมไพล์และรัน

□ **แบบ compile ⟶ run**

1. คอมไพล์:

npx tsc

จะได้ไฟล์: dist/index.js (ถ้า config set "outDir": "dist")

2. รัน:

node dist/index.js

□ **ผลลัพธ์:**

Hello, TypeScript!

□ แบบรันโดยตรงด้วย ts-node

npx ts-node src/index.ts

□ ผลลัพธ์:

Hello, TypeScript!

□ ตัวอย่างโครงสร้างโปรเจกต์

my-ts-app/

```
├── node_modules/
├── src/
│   └── index.ts
├── tsconfig.json
├── package.json
└── dist/
    └── index.js (เมื่อคอมไพล์แล้ว)
```

✳□ สรุปขั้นตอน

ขั้นตอน	รายละเอียด
ติดตั้ง	VS Code, Node.js, TypeScript
เริ่มโปรเจกต์	npm init, ติดตั้ง TS, สร้าง tsconfig.json
เขียนโค้ด	ในไฟล์ .ts เช่น src/index.ts
คอมไพล์	npx tsc หรือ npx ts-node
ดูผลลัพธ์	บน console: Hello, TypeScript!

การใช้ tsc และ ts-node ใน TypeScript

□ tsc (TypeScript Compiler)

tsc คือคำสั่งหลักของ TypeScript ใช้สำหรับ:

- □ ตรวจสอบชนิดข้อมูล (type checking)
- □ แปลงไฟล์ .ts ให้กลายเป็น .js
- □ ควบคุมผ่าน tsconfig.json

ตัวอย่างการใช้งาน

ไฟล์ *hello.ts*

```
function sayHi(name: string): string {
  return `Hello, ${name}`;
}

console.log(sayHi("World"));
```

คำสั่ง Compile:

```
tsc hello.ts
```

➡ ได้ไฟล์ hello.js:

```
function sayHi(name) {
    return "Hello, " + name;
}
console.log(sayHi("World"));
```

รันด้วย Node:

```
node hello.js
```

ผลลัพธ์:

```
Hello, World
```

---

tsc แบบโปรเจกต์ (ใช้ tsconfig.json)

หากมีหลายไฟล์ และใช้ tsconfig.json แล้ว ให้ compile ทั้งโปรเจกต์ได้โดย:

```
tsc
```

ตัวอย่าง tsconfig.json (ย่อ):

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true
  },
  "include": ["src"]
}
```

จะคอมไพล์ src/**/*.ts ไปยัง dist/**/*.js

☐ ts-node (TypeScript + Node.js)

ts-node คือเครื่องมือที่ใช้ **รันไฟล์ .ts ได้โดยตรง**

(ไม่ต้องคอมไพล์เป็น .js ล่วงหน้า)

☐ ติดตั้ง:

npm install -g ts-node

หรือแบบโปรเจกต์:

npm install --save-dev ts-node

☐ ใช้งาน:

ts-node hello.ts

☐ ผลลัพธ์:

Hello, World

ts-node รันแบบ in-memory compile ⟶ เหมาะกับ dev/testing

☐ เปรียบเทียบ tsc vs ts-node

ประเด็น	tsc	ts-node
คอมไพล์หรือรัน	คอมไพล์ .ts เป็น .js	รัน .ts ตรง ๆ เลย
ความเร็ว	เร็ว (เฉพาะ compile)	ช้ากว่าเล็กน้อย (compile + run)
เหมาะกับ	production, build system	development, script, CLI
ใช้กับ tsconfig?	☐ รองรับเต็มรูปแบบ	☐ รองรับเต็มรูปแบบ
เก็บไฟล์ .js	☐ (มี output)	☐ (ไม่สร้างไฟล์ใหม่)

☐ ตัวอย่างใช้งานจริง (Project Mode)

☐ โครงสร้าง:

my-ts-app/

```
├── src/
│   └── index.ts
├── dist/
└── tsconfig.json
```

☐ เขียน src/index.ts

function sum(a: number, b: number): number {

```
  return a + b;
}

console.log("Result =", sum(2, 3));
```

☐ **แบบ compile ⟶ run:**

```
npx tsc        # คอมไพล์ทุกไฟล์ใน src
node dist/index.js
```

☐ ผลลัพธ์:

```
Result = 5
```

☐ **แบบ run โดยตรง:**

```
npx ts-node src/index.ts
```

☐ ผลลัพธ์:

```
Result = 5
```

☐ สรุปคำสั่งสำคัญ

คำสั่ง	ใช้ทำอะไร
tsc file.ts	Compile ไฟล์เดียว
tsc	Compile ทั้งโปรเจกต์
tsc --watch	Compile อัตโนมัติเมื่อไฟล์เปลี่ยน
ts-node file.ts	รัน .ts ทันที (เหมาะกับ dev)

---

tsconfig.json คืออะไร?

tsconfig.json คือไฟล์ config สำหรับบอก tsc (TypeScript Compiler) ว่าจะ:

- คอมไพล์ไฟล์ใด
- ตั้งค่าคอมไพล์แบบไหน (เช่นเป็น ES5 หรือ ESNext)
- จะเอา output ไปไว้ที่ไหน

ถ้าคุณใช้ TypeScript ในโปรเจกต์ระดับจริง **ควรมี tsconfig.json เสมอ**

☐ วิธีสร้าง **tsconfig.json**

```
npx tsc --init
```

ระบบจะสร้างไฟล์ tsconfig.json พร้อม option พื้นฐาน เช่น:

```
{
```

```
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./dist",
    "strict": true,
    "esModuleInterop": true
  },
  "include": ["src"],
  "exclude": ["node_modules"]
}
```

☐ อธิบายส่วนสำคัญของ compilerOptions

Option	คำอธิบาย
"target"	ระบุว่า JS ที่ output จะอยู่ในระดับ ES อะไร (ES5, ES2020, ESNext)
"module"	ระบบโมดูลที่ใช้ เช่น commonjs, ESNext, AMD, UMD
"rootDir"	โฟลเดอร์ต้นทางของ TypeScript (ไฟล์ .ts ทั้งหมดอยู่ที่นี่)
"outDir"	โฟลเดอร์ปลายทางของไฟล์ที่คอมไพล์แล้ว (.js)
"strict"	เปิดระบบตรวจ type แบบเข้มงวดทั้งหมด (strictNullChecks, noImplicitAny)
"esModuleInterop"	ให้รองรับ import/export ระหว่าง CommonJS กับ ES Module

☐ ตัวอย่าง tsconfig.json แบบสมบูรณ์

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "CommonJS",
    "lib": ["ES2020"],
    "outDir": "dist",
    "rootDir": "src",
    "strict": true,
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "skipLibCheck": true
```

```
  },
  "include": ["src"],
  "exclude": ["node_modules", "dist"]
}
```

□ โครงสร้างโปรเจกต์ที่ดี

```
my-ts-app/
├── src/
│   ├── index.ts
│   └── utils.ts
├── dist/
│   └── (ไฟล์ .js output)
├── tsconfig.json
├── package.json
```

□ การคอมไพล์ด้วย tsconfig.json

เมื่อมี tsconfig.json แล้ว คุณสามารถ:

npx tsc

ระบบจะคอมไพล์ไฟล์ทั้งหมดตาม config

ผลลัพธ์จะไปที่ dist/

□ ใช้กับ ts-node ก็ได้

npx ts-node src/index.ts

ts-node จะอ่าน tsconfig.json อัตโนมัติ

□ ตัวอย่างโค้ดทดสอบ

□ src/index.ts

```
import { double } from "./utils";

console.log("Double 5 =", double(5));
```

□ src/utils.ts

```
export function double(x: number): number {
  return x * 2;
}
```

☐ Compile:

npx tsc

☐ Output:

dist/index.js

dist/utils.js

☐ ผลลัพธ์:

node dist/index.js

Output: Double 5 = 10

☐ คำแนะนำการใช้งาน

สถานการณ์	Option ที่แนะนำใน tsconfig.json
Web frontend	"module": "ESNext", "target": "ES6"
Node.js backend	"module": "CommonJS", "target": "ES2020"
รองรับ import/export	"esModuleInterop": true
ใช้งาน lib สมัยใหม่	"lib": ["ES2020", "DOM"]

✳☐ สรุป

หัวข้อ	สาระสำคัญ
สร้าง tsconfig.json	ใช้คำสั่ง npx tsc --init
ใช้ควบคุม tsc	คอมไพล์ไฟล์ทั้งหมดแบบควบคุมได้
ช่วยจัดโครงสร้าง	แยก src/ และ dist/ อย่างชัดเจน
ใช้ร่วมกับ ts-node ได้	อ่าน config จากไฟล์เดียวกัน

สรุป

TypeScript คือภาษาที่ต่อยอดจาก JavaScript โดยเพิ่มระบบประเภทข้อมูลแบบ static และฟีเจอร์ที่ช่วยให้การพัฒนาแอปพลิเคชันมีความปลอดภัยและเป็นระบบมากขึ้น โดยเฉพาะในโปรเจกต์ขนาดใหญ่ บทแรกของหนังสือเล่มนี้จะปูพื้นฐานให้ผู้อ่านเข้าใจว่า TypeScript คืออะไร มีข้อดีอย่างไรเมื่อเทียบกับ JavaScript รวมถึงวิธีการติดตั้ง การใช้คำสั่ง tsc และ ts-node ตลอดจนการตั้งค่าโปรเจกต์ด้วยไฟล์ tsconfig.json เพื่อเตรียมความพร้อมก่อนเข้าสู่เนื้อหาเชิงลึกในบทต่อไป.

บทที่ 2
ตัวแปรและชนิดข้อมูลพื้นฐาน
(Variables and Basic Data Types)

เนื้อหา

- ตัวแปรและชนิดข้อมูลพื้นฐานใน TypeScript
- ตัวแปรและชนิดข้อมูลพื้นฐาน (เชิงลึก)
- let, const, var และ Scope ใน TypeScript
- string, number, boolean ใน TypeScript
- null, undefined, any, unknown, never ใน TypeScript

บทนำ: ตัวแปรและชนิดข้อมูลพื้นฐาน

ในการเริ่มต้นเขียนโปรแกรมด้วยภาษา TypeScript การเข้าใจเรื่องตัวแปรและชนิดข้อมูลพื้นฐานเป็นสิ่งสำคัญอย่างยิ่ง เนื่องจากเป็นรากฐานของการเขียนโค้ดที่ถูกต้อง มีประสิทธิภาพ และสามารถตรวจสอบข้อผิดพลาดได้ตั้งแต่ขั้นตอนการเขียน การนิยามตัวแปรใน TypeScript ไม่ได้ต่างจาก JavaScript มากนัก แต่มีการเพิ่มเติมระบบชนิดข้อมูลที่ช่วยให้โค้ดมีความชัดเจนและปลอดภัยยิ่งขึ้น

ในบทนี้ ผู้อ่านจะได้เรียนรู้วิธีการประกาศตัวแปรด้วยคำสำคัญ let, const และ var ซึ่งแม้จะดูคล้ายกันแต่กลับมีพฤติกรรมแตกต่างกันอย่างมีนัยสำคัญ โดยเฉพาะในเรื่องของขอบเขตการมองเห็นของตัวแปร (scope) และการเปลี่ยนค่าในระหว่างการทำงาน การเข้าใจว่าเมื่อใดควรใช้ let หรือ const แทน var เป็นหัวใจสำคัญของการเขียนโค้ดที่ปลอดภัยและมีโครงสร้าง

นอกจากการประกาศตัวแปรแล้ว TypeScript ยังช่วยให้นักพัฒนาสามารถระบุชนิดข้อมูลของตัวแปรได้อย่างชัดเจน เช่น string สำหรับข้อความ, number สำหรับตัวเลข และ boolean สำหรับค่าความจริง การกำหนดชนิดข้อมูลช่วยลดความผิดพลาดที่อาจเกิดขึ้นในระหว่างการทำงาน และยังช่วยให้โปรแกรมสามารถคาดเดาพฤติกรรมของข้อมูลได้ล่วงหน้า

TypeScript ยังมีการจัดการค่าพิเศษ เช่น null และ undefined ซึ่งมักเป็นแหล่งที่มาของข้อผิดพลาดใน JavaScript การทำความเข้าใจค่าทั้งสองนี้จะช่วยให้นักพัฒนาสามารถเขียนโค้ดที่จัดการกรณีข้อมูลว่างหรือไม่มีค่าได้อย่างถูกต้อง นอกจากนี้ยังสามารถกำหนดตัวแปรให้รับค่าทั้งแบบมีข้อมูลหรือไม่มีข้อมูลด้วย union types เช่น string | null เพื่อเพิ่มความยืดหยุ่นในการเขียนโปรแกรม

อีกหนึ่งจุดเด่นของ TypeScript คือชนิดข้อมูลแบบยืดหยุ่น เช่น any, unknown และ never ซึ่งมีบทบาทเฉพาะในกรณีที่ต้องการเปิดให้ตัวแปรสามารถเก็บค่าหลายชนิด, ยังไม่ทราบชนิดแน่ชัด,

หรือไม่ควรมีค่าใดๆ เลย การเลือกใช้ชนิดข้อมูลเหล่านี้อย่างเหมาะสมสามารถเพิ่มความปลอดภัยของโค้ดและลดปัญหาที่เกิดจากการใช้งานข้อมูลอย่างไม่ระมัดระวัง

ในขณะที่ any เปิดกว้างให้กับตัวแปรสามารถเก็บค่าอะไรก็ได้ แต่ก็แฝงมาด้วยความเสี่ยงในการเกิดข้อผิดพลาดโดยไม่ถูกตรวจสอบ ในทางตรงกันข้าม unknown ให้ความปลอดภัยมากกว่าเพราะต้องมีการตรวจสอบก่อนใช้งาน ส่วน never ใช้กับฟังก์ชันที่ไม่มีวันส่งค่ากลับ เช่น การโยนข้อผิดพลาดหรือการวนลูปไม่รู้จบ ซึ่งเป็นรายละเอียดที่แม้จะเฉพาะทางแต่ก็มีประโยชน์ในโค้ดระดับมืออาชีพ

บทที่ 2 นี้จึงมีเป้าหมายในการปูพื้นฐานความเข้าใจเกี่ยวกับตัวแปร การกำหนดชนิดข้อมูล และการเลือกใช้ชนิดพิเศษต่างๆ อย่างถูกต้อง เพื่อให้นักพัฒนาสามารถเขียนโค้ดที่ปลอดภัย ยืดหยุ่น และสามารถดูแลรักษาได้ดีในระยะยาว อันเป็นก้าวสำคัญก่อนเข้าสู่หัวข้อที่ซับซ้อนขึ้นในบทถัดไป.

ตัวแปรและชนิดข้อมูลพื้นฐานใน TypeScript

---

1. ตัวแปรและ Scope: let, const, var

□ **var**

- ประกาศตัวแปรในแบบ **function-scoped** (ขอบเขตในฟังก์ชัน)
- มีการ **hoisting** (ยกตัวแปรขึ้นไปก่อนรัน)
- อาจเกิดปัญหาเมื่อใช้ใน block เช่น if, for เพราะไม่มี scope แบบ block

```
function testVar() {
  if (true) {
    var x = 10;
  }
  console.log(x); // 10 — x อยู่ใน scope function แม้ประกาศใน if-block
}
```

□ **let**

- ประกาศตัวแปรในแบบ **block-scoped** (ขอบเขตในวงเล็บ {})
- ไม่ hoisting แบบ var
- ปลอดภัยกว่า ใช้แทน var ในโค้ดใหม่

```
function testLet() {
  if (true) {
    let y = 20;
  }
  // console.log(y); // □ Error: y ไม่อยู่ใน scope นี้
}
```

□ **const**

- ประกาศตัวแปรที่ไม่สามารถ reassigned (เปลี่ยนค่าได้)
- ต้องกำหนดค่าเริ่มต้นตอนประกาศเสมอ
- **ไม่ใช่ immutable object** (ค่าภายใน object ยังเปลี่ยนได้)

```
const z = 30;
// z = 40; // □ Error: Assignment to constant variable.

const arr = [1, 2, 3];
arr.push(4); // □ สามารถแก้ไขเนื้อหาภายในได้
```

2. ชนิดข้อมูลพื้นฐาน (Primitive Types)

□ **string**

เก็บข้อความ

```
let name: string = "Alice";
```

□ **number**

เก็บตัวเลข (ทั้งจำนวนเต็มและทศนิยม)

```
let age: number = 25;
let price: number = 12.5;
```

□ **boolean**

เก็บค่าความจริง true หรือ false

```
let isActive: boolean = true;
```

3. Special Types: null, undefined, any, unknown, never

□ **null และ undefined**

- null คือค่าที่ตั้งใจว่า “ไม่มีค่า”
- undefined คือค่าที่ยังไม่ถูกกำหนด

```
let n: null = null;
let u: undefined = undefined;
```

ใน strict mode (default ใน TypeScript) ต้องระบุชนิดให้ชัดเจนว่าตัวแปรรับ null หรือ undefined ได้หรือไม่ เช่น

```
let maybeNumber: number | null = null;  // union type รับได้ทั้ง number หรือ null
```

□ **any**

- ชนิดข้อมูลที่ **ปิดการตรวจสอบชนิดทั้งหมด**
- ใช้ในกรณีที่ยังไม่รู้ชนิด หรือโค้ด legacy

```
let anything: any = 123;
anything = "hello";
anything = true;
```

⚠ ควรหลีกเลี่ยง any เพราะจะเสียประโยชน์จาก TypeScript ในการตรวจจับข้อผิดพลาด

---

unknown

- ชนิดข้อมูลที่ปลอดภัยกว่า any
- ตัวแปร unknown ต้องถูกตรวจสอบชนิดก่อนนำไปใช้

```
let value: unknown = "Hello";

if (typeof value === "string") {
  console.log(value.toUpperCase()); // ปลอดภัย
}
```

---

never

- ชนิดที่ไม่เคยเกิดขึ้นจริง (เช่นฟังก์ชันที่ไม่คืนค่า หรือ throw error เสมอ)

```
function throwError(message: string): never {
  throw new Error(message);
}
```

ตัวอย่างโค้ดรวม

```
let a: number = 10;
let b: string = "TypeScript";
const c: boolean = true;

let d: null = null;
let e: undefined = undefined;

let f: any = 123;
f = "now a string";

let g: unknown = "could be anything";
```

```
if (typeof g === "string") {
  console.log(g.length);
}

function fail(): never {
  throw new Error("This function never returns");
}
```

สรุปตารางชนิดข้อมูล

ชนิดข้อมูล	ความหมาย	ตัวอย่าง	หมายเหตุ
string	ข้อความ	"hello", 'world'	
number	ตัวเลข	123, 3.14	รวม integer และ float
boolean	ค่าความจริง	true, false	
null	ไม่มีค่า (ตั้งใจว่าง)	null	ต้องระบุ union ใน strict mode
undefined	ค่าที่ยังไม่กำหนด	undefined	เช่นตัวแปรที่ยังไม่ได้กำหนดค่า
any	ปิดตรวจสอบ type	any	ใช้เฉพาะกรณีพิเศษ
unknown	ชนิดไม่รู้ ต้องตรวจสอบก่อนใช้	unknown	ปลอดภัยกว่า any
never	ไม่มีค่าเกิดขึ้นจริง	ฟังก์ชันที่โยน Error หรือไม่จบ	ใช้กับฟังก์ชันที่ไม่คืนค่า

ตัวแปรและชนิดข้อมูลพื้นฐาน (เชิงลึก)

1. ตัวแปรและ Scope: var, let, const

1.1 var

- ประกาศตัวแปรแบบ **function-scoped**
  หมายความว่า ตัวแปรที่ประกาศด้วย var จะอยู่ใน scope ของฟังก์ชันที่มันถูกประกาศ
- มีการ **hoisting** (ยกการประกาศตัวแปรขึ้นก่อนการรันโค้ด) แต่ค่าเริ่มต้นจะเป็น undefined ก่อนถูกกำหนดค่า

- ไม่เหมาะกับโค้ดสมัยใหม่ เพราะทำให้เกิดบั๊กจาก scope ที่ไม่ชัดเจน เช่น การใช้ตัวแปรใน loop หรือ block

```
function exampleVar() {
  console.log(x); // undefined (hoisted)
  var x = 10;
  if (true) {
    var x = 20; // ตัวแปร x นี้คือ x เดิม ไม่ใช่ตัวแปรใหม่
    console.log(x); // 20
  }
  console.log(x); // 20 เพราะถูกเปลี่ยนค่าภายใน if block
}
```

ผล: การประกาศ var ใน block ไม่ได้สร้าง scope ใหม่ ทำให้โค้ดสับสนและเกิดบั๊กง่าย

1.2 let

- ประกาศตัวแปรแบบ **block-scoped** (ขอบเขตใน {})
- ไม่มี hoisting แบบ var (แต่มี hoisting แบบ Temporal Dead Zone — TDZ คือยังใช้ตัวแปรก่อนประกาศไม่ได้)
- เหมาะสำหรับใช้งานแทน var ในทุกกรณี เพราะปลอดภัยและคาดเดาพฤติกรรมได้

```
function exampleLet() {
  // console.log(y); // □ Error: Cannot access 'y' before initialization (TDZ)
  let y = 10;
  if (true) {
    let y = 20; // y ตัวใหม่คนละตัวกับด้านบน
    console.log(y); // 20
  }
  console.log(y); // 10
}
```

1.3 const

- ประกาศค่าคงที่ ไม่สามารถ reassigned ค่าใหม่ได้
- ใช้สำหรับค่าที่ไม่เปลี่ยนแปลงตลอดโปรแกรม เช่น ค่าคงที่, การกำหนด reference ของ object หรือ array
- **อย่างไรก็ตาม** ตัวแปรที่เป็น object หรือ array ที่ประกาศด้วย const สามารถเปลี่ยนแปลง property หรือสมาชิกได้ (mutable)

```
const PI = 3.14159;
// PI = 3.14; // □ Error: Assignment to constant variable.

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

2. ชนิดข้อมูลพื้นฐานใน TypeScript

TypeScript เป็น **Static Typed Language** หมายความว่า ตัวแปรต้องมีชนิดข้อมูลแน่นอนในช่วงเวลาคอมไพล์ (compile-time) ซึ่งช่วยลดบั๊กและทำให้ IDE ช่วยตรวจสอบโค้ดได้ดีกว่า JavaScript

2.1 string

- เก็บข้อความแบบ Unicode
- ใช้ได้ทั้ง "double quotes", 'single quotes', หรือ Template Strings `backticks`

```
let firstName: string = "Alice";
let greeting: string = `Hello, ${firstName}!`;
```

2.2 number

- เก็บตัวเลขทั้ง integer และ floating-point
- ไม่มีแยกระหว่าง int กับ float แบบบางภาษา

```
let age: number = 30;
let pi: number = 3.1415;
```

2.3 boolean

- เก็บค่าความจริง true หรือ false

```
let isLoggedIn: boolean = true;
let isLoading: boolean = false;
```

3. ชนิดข้อมูลพิเศษ: null, undefined, any, unknown, never

3.1 null และ undefined

- **null** คือ ตัวแทนของ “ไม่มีค่า” แบบเจาะจง (explicitly empty)
- **undefined** คือ ตัวแทนของค่าที่ยังไม่ถูกกำหนด

```
let a: null = null;
```

```
let b: undefined = undefined;
```

Strict Null Checking: TypeScript มี option strictNullChecks (เปิดเป็นค่า default)

ซึ่งหมายความว่าคุณต้องระบุชนิดให้ชัดเจนว่าตัวแปรจะรับค่า null หรือ undefined ได้ด้วย เช่น

```
let maybeNumber: number | null = null;  // union type
```

ถ้าไม่ระบุ union นี้ แล้วพยายามกำหนด null ให้ตัวแปร number จะเกิด error

---

3.2 any

- คือชนิดข้อมูล “ปิดการตรวจสอบชนิด” (opt-out from type checking)
- เหมาะกับโค้ด legacy หรือสถานการณ์ที่ไม่รู้ชนิดล่วงหน้า
- **แต่ควรหลีกเลี่ยง เพราะทำให้สูญเสียความปลอดภัยของ type**

```
let anything: any = 123;
anything = "Hello";
anything = false;
```

3.3 unknown

- คล้ายกับ any แต่ปลอดภัยกว่า
- ต้องตรวจสอบชนิดก่อนใช้งานจริง
- เป็นวิธีที่แนะนำเมื่อชนิดข้อมูลไม่ทราบล่วงหน้า

```
let value: unknown = 10;
value = "A string";

if (typeof value === "string") {
  console.log(value.toUpperCase()); // ปลอดภัย
}
```

3.4 never

- หมายถึง “ไม่มีค่าที่เป็นไปได้”
- ใช้กับฟังก์ชันที่ไม่มีทาง return (เช่น throw error หรือตัวลูปไม่มีที่สิ้นสุด)
- ช่วยให้ TypeScript รู้ว่าโค้ดไม่สามารถไปถึงจุดหลัง never ได้

```
function fail(msg: string): never {
  throw new Error(msg);
}

function infiniteLoop(): never {
```

```
  while (true) {}
}
```

---

4. Temporal Dead Zone (TDZ)

- เกิดกับ let และ const
- หมายความว่าก่อนถึงจุดประกาศตัวแปรในโค้ด จะไม่สามารถเข้าถึงตัวแปรนั้นได้ (ต่างจาก var ที่ hoisting เป็น undefined)
- ช่วยลดบั๊กจากการใช้ตัวแปรก่อนประกาศ

```
console.log(x); // □ Error: Cannot access 'x' before initialization
let x = 5;
```

---

5. Union Types และ Type Narrowing (ต่อยอด)

- Union Type คือการระบุว่าตัวแปรรับได้หลายชนิด

```
let id: number | string;
id = 123;
id = "abc";
```

- การใช้ typeof หรือ instanceof เพื่อตรวจชนิดและแคบ type (type narrowing)

```
function printId(id: number | string) {
  if (typeof id === "string") {
    console.log(id.toUpperCase());
  } else {
    console.log(id);
  }
}
```

---

6. ตัวอย่างโค้ดเต็ม (เชิงลึก)

```
function processValue(value: number | null | undefined) {
  if (value === null) {
    console.log("Value is null");
  } else if (value === undefined) {
    console.log("Value is undefined");
  } else {
    console.log("Value is number:", value);
  }
```

```
}

let data: unknown = "Hello";

if (typeof data === "string") {
  console.log(data.toLowerCase());
} else {
  console.log("Not a string");
}

const neverReturns = (msg: string): never => {
  throw new Error(msg);
};

try {
  neverReturns("Error happened!");
} catch (e) {
  console.error(e);
}
```

7. สรุปข้อแนะนำ

ข้อแนะนำ	รายละเอียด
ใช้ let และ const แทน var	ปลอดภัยกว่าและมี scope ชัดเจน
ใช้ const สำหรับค่าที่ไม่เปลี่ยนแปลง	ลดบั๊กจากการ reassignment
ระบุชนิดตัวแปรให้ชัดเจน	เพื่อช่วยตรวจจับบั๊กตั้งแต่คอมไพล์
หลีกเลี่ยง any	ใช้ unknown หรือ union types แทน
ใช้ strictNullChecks	ป้องกันการใช้ null/undefined ผิดพลาด
ใช้ type narrowing	ตรวจสอบชนิดก่อนใช้งาน

let, const, var และ Scope ใน TypeScript

1. var

- ประกาศตัวแปรแบบ **function-scoped**
  ตัวแปรที่ประกาศด้วย var จะมีขอบเขตอยู่ในฟังก์ชันที่ประกาศเท่านั้น ไม่ได้อยู่แค่ในบล็อก {}
- มีการ **hoisting** — การยกการประกาศตัวแปรขึ้นไปด้านบนของฟังก์ชัน (แต่ค่าจะยังไม่ถูกกำหนด)
- ข้อเสีย: อาจทำให้เกิดบั๊กเพราะตัวแปรมีขอบเขตกว้างเกินไป และ hoisting ทำให้เข้าใจผิดได้

ตัวอย่าง var กับ hoisting และ scope

```
function testVar() {
  console.log(x); // undefined (hoisting)
  var x = 10;

  if (true) {
    var x = 20;  // ตัวแปร x ตัวเดียวกับด้านบน
    console.log(x); // 20
  }

  console.log(x); // 20 — เพราะ x ถูกเปลี่ยนใน if-block
}

testVar();
```

ผลลัพธ์:

```
undefined
20
20
```

---

2. let

- ประกาศตัวแปรแบบ **block-scoped**
  หมายความว่าตัวแปรจะมีขอบเขตอยู่ภายใน {} ที่ประกาศ เช่น ใน if, for, while, หรือฟังก์ชัน
- ไม่มี hoisting แบบ var — ตัวแปรจะอยู่ใน **Temporal Dead Zone (TDZ)** จนกว่าจะถึงบรรทัดที่ประกาศ
- ปลอดภัยกว่าและแนะนำให้ใช้แทน var ในโค้ดสมัยใหม่

ตัวอย่าง let กับ scope

```
function testLet() {
  // console.log(y); // □ Error: Cannot access 'y' before initialization
```

```
  let y = 10;

  if (true) {
    let y = 20; // ตัวแปรใหม่คนละตัวกับข้างบน
    console.log(y); // 20
  }

  console.log(y); // 10
}

testLet();
```

ผลลัพธ์:

```
20
10
```

---

3. const

- ประกาศค่าคงที่ (constant) — **ไม่สามารถเปลี่ยนค่าใหม่ได้** (reassignment forbidden)
- ต้องกำหนดค่าเริ่มต้นตอนประกาศเท่านั้น
- const มี **block scope** เช่นเดียวกับ let
- **แต่ค่าภายในอ็อบเจ็กต์หรืออาเรย์ที่ประกาศด้วย const ยังเปลี่ยนแปลงได้** (mutable reference)

ตัวอย่าง const

```
const PI = 3.14159;
// PI = 3.14; // □ Error: Assignment to constant variable

const arr = [1, 2, 3];
arr.push(4); // □ สามารถเปลี่ยนเนื้อหาภายในอาเรย์ได้

const obj = { name: "Alice" };
obj.name = "Bob"; // □ เปลี่ยน property ได้
```

---

4. Scope คืออะไร?

Scope คือขอบเขตที่ตัวแปรหรือฟังก์ชันสามารถเข้าถึงได้

ตัวแปร	Scope Type	ขอบเขต	Hoisting

ตัวแปร	Scope Type	ขอบเขต	Hoisting
var	Function Scope	ในฟังก์ชันที่ประกาศ	มี hoisting
let	Block Scope	ในบล็อก {} ที่ประกาศ	มี hoisting แบบ TDZ
const	Block Scope	ในบล็อก {} ที่ประกาศ	มี hoisting แบบ TDZ

---

5. Temporal Dead Zone (TDZ)

- เกิดกับตัวแปร let และ const
- ตัวแปรจะไม่สามารถถูกเข้าถึงได้ตั้งแต่ก่อนบรรทัดประกาศจนถึงบรรทัดนั้นจริง ๆ
- ช่วยป้องกันการใช้ตัวแปรก่อนประกาศ ทำให้โค้ดปลอดภัยขึ้น

```
console.log(a); // □ Error: Cannot access 'a' before initialization
let a = 5;
```

สรุปข้อแตกต่าง

จุดเปรียบเทียบ	var	let	const
ขอบเขต (Scope)	Function-scoped	Block-scoped	Block-scoped
Hoisting	มี hoisting (ค่าเป็น undefined)	Hoisting แบบ TDZ (ไม่เข้าถึงก่อนประกาศ)	Hoisting แบบ TDZ
เปลี่ยนค่าได้ไหม	เปลี่ยนได้	เปลี่ยนได้	ไม่สามารถเปลี่ยนค่าใหม่ได้
เหมาะกับการใช้งาน	ไม่แนะนำสำหรับโค้ดใหม่	แนะนำสำหรับตัวแปรที่เปลี่ยนค่า	แนะนำสำหรับค่าคงที่

ตัวอย่างโปรแกรมแบบเต็มไฟล์ หรือการใช้งานจริงพร้อมผลลัพธ์

2 กลุ่มหลัก คือ

- **โปรแกรมพื้นฐาน 3 ตัว** ที่เน้นแสดงความแตกต่างของ var, let, const และ Scope
- **โปรแกรมแนวประยุกต์ 3 ตัว** ที่ใช้ความรู้เหล่านี้ร่วมกับฟังก์ชัน และโครงสร้างควบคุมต่าง ๆ

โปรแกรมพื้นฐาน 3 ตัว: let, const, var และ Scope

ตัวอย่างที่ 1: var กับ Scope และ Hoisting

โครงสร้างไฟล์

basic-var/

└── index.ts

โค้ด index.ts

```
function varScopeExample() {
  console.log("Before declaration:", x); // undefined (hoisting)
  var x = 5;

  if (true) {
    var x = 10;  // Same variable as above, reassign
    console.log("Inside block:", x);     // 10
  }

  console.log("After block:", x);        // 10 (changed inside block)
}

varScopeExample();
```

คำอธิบายโค้ด

- var x ถูก hoisted ขึ้นไปก่อนรัน ทำให้ log แรกแสดงเป็น undefined ไม่ใช่ error
- การประกาศ var ใน block (if) ใช้ตัวแปรเดียวกับนอก block ทำให้ค่าถูกเปลี่ยน
- สรุปคือ var ไม่มี block scope แต่มี function scope

ผลการรัน

```
Before declaration: undefined
Inside block: 10
After block: 10
```

ตัวอย่างที่ 2: let กับ Block Scope

โครงสร้างไฟล์

```
basic-let/
└── index.ts
```

โค้ด index.ts

```
function letScopeExample() {
  // console.log(y); // Error: Cannot access 'y' before initialization
  let y = 5;

  if (true) {
```

```
    let y = 10; // Different variable in block scope
    console.log("Inside block:", y);  // 10
  }

  console.log("Outside block:", y);   // 5
}

letScopeExample();
```

คำอธิบายโค้ด

- let y ไม่สามารถเข้าถึงก่อนประกาศ (Temporal Dead Zone)
- ตัวแปร y ใน block เป็นตัวแปรคนละตัวกับด้านนอก
- ทำให้ค่าในบล็อกและนอกบล็อกแยกกันอย่างชัดเจน

ผลการรัน

```
Inside block: 10
Outside block: 5
```

ตัวอย่างที่ 3: const กับค่าคงที่และ mutable object

โครงสร้างไฟล์

```
basic-const/
└── index.ts
```

โค้ด index.ts

```
function constExample() {
  const PI = 3.14;
  // PI = 3.1415; // Error: Assignment to constant variable

  const arr = [1, 2, 3];
  arr.push(4); // Allowed - modifies array contents
  console.log("Array after push:", arr);

  const obj = { name: "Alice" };
  obj.name = "Bob"; // Allowed - modifies object property
  console.log("Object after modification:", obj);
}
```

```
constExample();
```

คำอธิบายโค้ด

- const PI ไม่สามารถ reassigned ได้
- แต่ const กับ object/array สามารถเปลี่ยนแปลงข้อมูลภายในได้
- การใช้ const ช่วยป้องกันการเปลี่ยน reference แบบไม่ตั้งใจ

ผลการรัน

```
Array after push: [ 1, 2, 3, 4 ]
Object after modification: { name: 'Bob' }
```

โปรแกรมแนวประยุกต์ 3 ตัว: ใช้ let, const, var กับฟังก์ชันและโครงสร้างควบคุม

ตัวอย่างที่ 4: นับเลขด้วย let ในลูป

โครงสร้างไฟล์

```
app-count-let/
└── index.ts
```

โค้ด index.ts

```
function countWithLet() {
  for (let i = 0; i < 3; i++) {
    setTimeout(() => {
      console.log(`let loop: ${i}`);
    }, 100 * i);
  }
}

countWithLet();
```

คำอธิบายโค้ด

- ใช้ let i ทำให้แต่ละ iteration มี scope ของตัวเอง
- เวลาใช้กับ callback อย่าง setTimeout ค่า i ในแต่ละ callback จะแยกกันไม่ซ้ำกัน

ผลการรัน (เวลารันทีละ 100ms)

```
let loop: 0
let loop: 1
let loop: 2
```

ตัวอย่างที่ 5: นับเลขด้วย var ในลูป (บั๊ก)

โครงสร้างไฟล์

app-count-var/
└── index.ts

โค้ด index.ts

```
function countWithVar() {
  for (var i = 0; i < 3; i++) {
    setTimeout(() => {
      console.log(`var loop: ${i}`);
    }, 100 * i);
  }
}

countWithVar();
```

คำอธิบายโค้ด

- var i เป็นตัวแปรเดียวสำหรับทุก iteration
- เมื่อ callback setTimeout รัน ค่า i จะเป็น 3 ทั้งหมด (หลัง loop จบ)
- เป็นบั๊กคลาสสิกที่เกิดจาก var scope ไม่ใช่ block scoped

ผลการรัน

```
var loop: 3
var loop: 3
var loop: 3
```

ตัวอย่างที่ 6: การใช้ const กับ object ในฟังก์ชัน

โครงสร้างไฟล์

app-const-object/
└── index.ts

โค้ด index.ts

```
function createUser(name: string) {
  const user = {
    name,
    createdAt: new Date(),
  };
```

```
  // user = {}; // Error: Assignment to constant variable

  user.name = user.name.toUpperCase(); // Allowed to modify properties
  return user;
}

console.log(createUser("alice"));
```

คำอธิบายโค้ด

- user ประกาศด้วย const หมายถึงไม่สามารถ reassigned
- แต่สามารถแก้ไข property ภายในได้
- เหมาะสำหรับเก็บข้อมูลคงที่ที่อาจมี property เปลี่ยนแปลง

ผลการรัน

```
{ name: 'ALICE', createdAt: 2025-06-28Txx:xx:xx.xxxZ }
```

สรุป

โปรแกรม	เนื้อหา	เทคนิคหลัก
1	var และ hoisting	Function scope, hoisting
2	let และ block scope	Block scope, TDZ
3	const กับค่าคงที่และ mutable object	Reassignment forbidden, mutable properties
4	ใช้ let ใน loop กับ setTimeout	Block scope, callback closure
5	ใช้ var ใน loop กับ setTimeout (บั๊ก)	Function scope, closure บั๊ก
6	const กับ object และ property mutation	Const reference, mutable object

---

string, number, boolean ใน TypeScript

---

1. คำอธิบายเชิงลึก

☐ **string**

- เก็บข้อมูลตัวอักษรหรือข้อความ
- รองรับการประกาศด้วย single quotes '...', double quotes "..." และ template literals `...`
- Template literals ช่วยให้แทรกตัวแปรหรือ expression ได้ง่ายด้วย ${}

☐ **number**

- เก็บข้อมูลตัวเลขทั้ง integer และ float

- ไม่มีชนิดแยกระหว่าง integer กับ float
- รองรับเลขฐานต่าง ๆ เช่น ทศนิยม (decimal), ฐานสิบหก (hex), ฐานแปด (octal), และฐานสอง (binary)

☐ **boolean**

- เก็บค่าความจริงสองค่า คือ true หรือ false
- ใช้สำหรับสถานะหรือเงื่อนไขทางตรรกะ

---

2. ตัวอย่างโปรแกรมแบบเต็มไฟล์

ตัวอย่างที่ 1: การใช้ string แบบต่าง ๆ

โครงสร้างโปรเจกต์

```
string-example/
└── index.ts
```

*โค้ด **index.ts***

```
function greet(name: string): string {
  // ใช้ template literal
  return `Hello, ${name}! Welcome to TypeScript.`;
}

const userName: string = "Alice";

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

คำอธิบาย

- ฟังก์ชัน greet รับ parameter name เป็น string
- ใช้ template literal แทรกตัวแปร name ลงในข้อความ
- ตัวแปร userName ประกาศชนิด string และส่งเข้า greet

ผลการรัน

```
Hello, Alice! Welcome to TypeScript.
```

ตัวอย่างที่ 2: การใช้ number กับเลขฐานต่าง ๆ และคำนวณ

โครงสร้างโปรเจกต์

```
number-example/
└── index.ts
```

โค้ด index.ts

```
function calculateArea(radius: number): number {
  const pi: number = 3.14159;
  return pi * radius * radius;
}

const r1: number = 5;
const r2: number = 10;

console.log(`Area of circle with radius ${r1} is ${calculateArea(r1)}`);
console.log(`Area of circle with radius ${r2} is ${calculateArea(r2)}`);

// แสดงเลขฐานต่าง ๆ
const hex: number = 0xff;     // 255 decimal
const binary: number = 0b1010;  // 10 decimal
const octal: number = 0o744; // 484 decimal

console.log(`Hexadecimal 0xff is ${hex}`);
console.log(`Binary 0b1010 is ${binary}`);
console.log(`Octal 0o744 is ${octal}`);
```

คำอธิบาย

- ฟังก์ชัน calculateArea รับ radius เป็น number และคืนค่าเป็น number
- แสดงการใช้เลขฐานต่าง ๆ ใน TypeScript
- ใช้ template literals แสดงผล

ผลการรัน

```
Area of circle with radius 5 is 78.53975
Area of circle with radius 10 is 314.159
Hexadecimal 0xff is 255
Binary 0b1010 is 10
Octal 0o744 is 484
```

ตัวอย่างที่ 3: การใช้ boolean กับเงื่อนไข

โครงสร้างโปรเจกต์

```
boolean-example/
```

└── index.ts

*โค้ด **index.ts***

```
function canVote(age: number): boolean {
  return age >= 18;
}

const personAge1: number = 17;
const personAge2: number = 21;

console.log(`Can person aged ${personAge1} vote? ${canVote(personAge1)}`);
console.log(`Can person aged ${personAge2} vote? ${canVote(personAge2)}`);
```

คำอธิบาย

- ฟังก์ชัน canVote คืนค่า true ถ้าอายุไม่ต่ำกว่า 18
- แสดงผล boolean ในข้อความด้วย template literals

ผลการรัน

```
Can person aged 17 vote? false
Can person aged 21 vote? true
```

---

3 โปรแกรมพื้นฐานและ 3 โปรแกรมแนวประยุกต์ ที่ใช้ string, number, และ boolean พร้อมโครงสร้างไฟล์ คำอธิบาย และผลรันครบถ้วน

โปรแกรมพื้นฐาน 3 ตัว: string, number, boolean

ตัวอย่างที่ 1: ฟังก์ชันทักทายด้วย string และ template literals

โครงสร้างไฟล์

```
basic-greet/
└── index.ts
```

โค้ด index.ts

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

const userName: string = "Bob";
```

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

คำอธิบาย

- รับชื่อเป็น string แล้วคืนข้อความทักทายโดยใช้ template literals
- ตัวแปร userName ประกาศแบบ string และส่งเข้า greet

ผลการรัน

```
Hello, Bob!
```

ตัวอย่างที่ 2: คำนวณพื้นที่วงกลมด้วย number

โครงสร้างไฟล์

```
basic-area/
└── index.ts
```

โค้ด index.ts

```
function circleArea(radius: number): number {
  const PI = 3.14159;
  return PI * radius * radius;
}

const r = 7;

console.log(`Area of circle with radius ${r} is ${circleArea(r)}`);
```

คำอธิบาย

- ฟังก์ชันรับรัศมี (number) คืนพื้นที่วงกลม
- ใช้ constant PI เป็น number
- แสดงผลลัพธ์ด้วย template literals

ผลการรัน

```
Area of circle with radius 7 is 153.93811
```

ตัวอย่างที่ 3: เช็คสถานะ boolean ด้วยฟังก์ชัน

โครงสร้างไฟล์

```
basic-vote/
└── index.ts
```

โค้ด index.ts

```
function canVote(age: number): boolean {
  return age >= 18;
```

```
}

const age1 = 16;
const age2 = 22;

console.log(`Age ${age1} can vote? ${canVote(age1)}`);
console.log(`Age ${age2} can vote? ${canVote(age2)}`);
```

คำอธิบาย

- ฟังก์ชันคืนค่า boolean แสดงว่าอายุมากกว่าเท่ากับ 18 หรือไม่
- แสดงผลคำตอบเป็น string ด้วย template literals

ผลการรัน

```
Age 16 can vote? false
Age 22 can vote? true
```

โปรแกรมแนวประยุกต์ 3 ตัว: ใช้ string, number, boolean รวมโครงสร้างโปรเจกต์

ตัวอย่างที่ 4: ฟอร์มสมัครสมาชิก ตรวจสอบความถูกต้องเบื้องต้น

โครงสร้างไฟล์

```
app-registration/
└── index.ts
```

โค้ด index.ts

```
interface User {
  username: string;
  age: number;
  email: string;
}

function isValidEmail(email: string): boolean {
  return email.includes("@") && email.includes(".");
}

function registerUser(user: User): string {
  if (user.username.length < 3) {
    return "Username must be at least 3 characters.";
```

```
  }
  if (!isValidEmail(user.email)) {
    return "Invalid email format.";
  }
  if (user.age < 13) {
    return "User must be at least 13 years old.";
  }
  return `User ${user.username} registered successfully!`;
}

const newUser: User = {
  username: "Alice123",
  age: 20,
  email: "alice@example.com",
};

console.log(registerUser(newUser));
```

คำอธิบาย

- ใช้ interface กำหนดโครงสร้างข้อมูล User
- ฟังก์ชันตรวจสอบความถูกต้องของ email (boolean), อายุ (number), ชื่อ (string)
- คืนข้อความสถานะการสมัครสมาชิก

ผลการรัน

```
User Alice123 registered successfully!
```

ตัวอย่างที่ 5: โปรแกรมแปลงหน่วยอุณหภูมิ (Celsius ⇄ Fahrenheit)

โครงสร้างไฟล์

```
app-temp-converter/
└── index.ts
```

โค้ด index.ts

```
function celsiusToFahrenheit(celsius: number): number {
  return (celsius * 9) / 5 + 32;
}

function fahrenheitToCelsius(fahrenheit: number): number {
```

```
  return ((fahrenheit - 32) * 5) / 9;
}

const tempC = 25;
const tempF = 77;

console.log(`${tempC}°C = ${celsiusToFahrenheit(tempC).toFixed(2)}°F`);
console.log(`${tempF}°F = ${fahrenheitToCelsius(tempF).toFixed(2)}°C`);
```

คำอธิบาย

- ฟังก์ชันแปลงหน่วยรับและคืนค่าเป็น number
- ใช้ .toFixed(2) กำหนดทศนิยม 2 ตำแหน่ง
- แสดงผลด้วย template literals

ผลการรัน

```
25°C = 77.00°F
77°F = 25.00°C
```

ตัวอย่างที่ 6: เกมทายเลขง่าย ๆ ใช้ boolean เช็คคำตอบ

โครงสร้างไฟล์

```
app-guess-number/
└── index.ts
```

โค้ด index.ts

```
function guessNumber(secret: number, guess: number): string {
  if (guess === secret) {
    return "Correct! □";
  } else if (guess < secret) {
    return "Too low.";
  } else {
    return "Too high.";
  }
}

const secretNumber = 42;
const userGuess = 30;
```

```
console.log(`Guess: ${userGuess} -> ${guessNumber(secretNumber, userGuess)}`);
```

คำอธิบาย

- ฟังก์ชันเปรียบเทียบ guess กับ secret คืนข้อความ string
- ใช้ boolean expression (===, <, >)
- แสดงผลลัพธ์ที่เหมาะสม

ผลการรัน

```
Guess: 30 -> Too low.
```

---

null, undefined, any, unknown, never ใน TypeScript

---

1. null และ undefined

- null หมายถึง “ไม่มีค่า” ที่ระบุชัดเจน
- undefined หมายถึง “ยังไม่ได้กำหนดค่า” หรือ “ไม่มีค่าเริ่มต้น”
- ใน TypeScript (เมื่อเปิด strictNullChecks) ตัวแปรชนิดปกติจะไม่สามารถรับ null หรือ undefined ได้ถ้าไม่ระบุไว้ด้วย union type เช่น string | null
- ช่วยป้องกันบั๊กจากการใช้ค่าที่ว่างเปล่าโดยไม่ตั้งใจ

ตัวอย่างโปรแกรม null และ undefined

```
function greet(name: string | null) {
  if (name === null) {
    console.log("Hello, Guest!");
  } else {
    console.log(`Hello, ${name}!`);
  }
}

let myName: string | null = null;
greet(myName); // Hello, Guest!

myName = "Alice";
greet(myName); // Hello, Alice!

let value: undefined = undefined;
console.log("Value is", value);
```

2. any

- ชนิดข้อมูลแบบ “ใดก็ได้”
- ปิดการตรวจสอบชนิดข้อมูล (opt-out)
- ใช้งานง่ายแต่เสี่ยงเกิดบั๊กจากการใช้ข้อมูลผิดชนิด
- ควรหลีกเลี่ยงการใช้ any หากเป็นไปได้

ตัวอย่างโปรแกรม any

```
let data: any = 10;
data = "hello";
data = { x: 100 };

console.log(data.myProperty); // ไม่มี error ตอนคอมไพล์ แต่ runtime อาจ error
```

3. unknown

- คล้าย any แต่ปลอดภัยกว่า
- ต้องตรวจสอบชนิดก่อนนำไปใช้จริง
- เป็นทางเลือกที่ดีเมื่อข้อมูลเข้ามาจากภายนอก (เช่น API)

ตัวอย่างโปรแกรม unknown

```
let input: unknown = 5;

if (typeof input === "number") {
  console.log(input + 10); // ปลอดภัย
} else {
  console.log("Input is not a number");
}
```

4. never

- หมายถึง “ไม่มีค่าเลย” หรือ “ไม่มีทางเกิดขึ้น”
- ใช้กับฟังก์ชันที่ไม่ return ค่า เช่น ฟังก์ชันที่โยน error หรือทำลูปไม่มีที่สิ้นสุด

ตัวอย่างโปรแกรม never

```
function throwError(message: string): never {
```

```
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {}
}
```

ตัวอย่างโปรแกรมเต็มไฟล์ 3 ตัวอย่าง

ตัวอย่างที่ 1: ใช้ null และ undefined

```
// file: null-undefined.ts

function printLength(str: string | null | undefined) {
  if (str === null) {
    console.log("Input is null");
  } else if (str === undefined) {
    console.log("Input is undefined");
  } else {
    console.log("Length is", str.length);
  }
}

printLength(null);
printLength(undefined);
printLength("Hello, TypeScript!");
```

ผลการรัน

```
Input is null
Input is undefined
Length is 17
```

ตัวอย่างที่ 2: ใช้ any และ unknown

```
// file: any-unknown.ts

function processAny(data: any) {
```

```
  console.log("Any data:", data);
  // ไม่ปลอดภัย อาจเกิด runtime error
  console.log(data.someProperty);
}

function processUnknown(data: unknown) {
  if (typeof data === "object" && data !== null && "someProperty" in data) {
    // TypeScript รู้ว่า data มี someProperty
    console.log("Unknown data someProperty:", (data as any).someProperty);
  } else {
    console.log("Unknown data has no someProperty");
  }
}

processAny({ someProperty: 123 });
processAny("string");

processUnknown({ someProperty: 456 });
processUnknown(100);
```

ผลการรัน

```
Any data: { someProperty: 123 }
123
Any data: string
undefined
Unknown data someProperty: 456
Unknown data has no someProperty
```

ตัวอย่างที่ 3: ฟังก์ชันที่คืน never

```
// file: never-example.ts

function fail(message: string): never {
  throw new Error(message);
}
```

```
function testFail(condition: boolean) {
  if (condition) {
    fail("This is an error");
  }
  console.log("Condition is false");
}

try {
  testFail(true);
} catch (e) {
  console.error("Caught error:", e.message);
}

testFail(false);
```

ผลการรัน

```
Caught error: This is an error
Condition is false
```

ตัวอย่างโปรแกรมพื้นฐาน 3 ตัว และตัวอย่างโปรแกรมแนวประยุกต์ 3 ตัว ที่เน้นการใช้ **null, undefined, any, unknown, never** พร้อมโครงสร้างไฟล์ คำอธิบาย และผลการรัน

โปรแกรมพื้นฐาน 3 ตัว: null, undefined, any, unknown, never

ตัวอย่างที่ 1: null และ undefined เบื้องต้น

โครงสร้างไฟล์

```
basic-null-undefined/
└── index.ts
```

โค้ด index.ts

```
function checkValue(value: string | null | undefined): void {
  if (value === null) {
    console.log("Value is null");
  } else if (value === undefined) {
    console.log("Value is undefined");
  } else {
```

```
    console.log(`Value length is ${value.length}`);
  }
}

checkValue(null);
checkValue(undefined);
checkValue("Hello, TypeScript!");
```

คำอธิบาย

- ฟังก์ชันรับค่าที่อาจเป็น string, null, หรือ undefined
- เช็คเงื่อนไขแยกตามค่า และแสดงข้อความตามชนิดค่าที่รับมา

ผลการรัน

```
Value is null
Value is undefined
Value length is 17
```

ตัวอย่างที่ 2: any กับ unknown การใช้งานและความแตกต่าง

โครงสร้างไฟล์

```
basic-any-unknown/
└── index.ts
```

โค้ด index.ts

```
function handleAny(input: any): void {
  console.log("Any input:", input);
  // เข้าถึง property โดยไม่ตรวจสอบ
  console.log("Accessing property 'foo':", input.foo);
}

function handleUnknown(input: unknown): void {
  if (typeof input === "object" && input !== null && "foo" in input) {
    // ต้องแคสต์เป็น any หรือชนิดที่เหมาะสมก่อนใช้
    console.log("Unknown input foo:", (input as any).foo);
  } else {
    console.log("Unknown input has no property 'foo'");
  }
}
```

```
handleAny({ foo: 123 });
handleAny("hello");

handleUnknown({ foo: 456 });
handleUnknown(789);
```

คำอธิบาย

- any ปิดการตรวจสอบชนิด ใช้งานสะดวกแต่เสี่ยง runtime error
- unknown ต้องตรวจสอบชนิดก่อนใช้ข้อมูลอย่างปลอดภัย

ผลการรัน

```
Any input: { foo: 123 }
Accessing property 'foo': 123
Any input: hello
Accessing property 'foo': undefined
Unknown input foo: 456
Unknown input has no property 'foo'
```

ตัวอย่างที่ 3: never กับฟังก์ชันที่โยน error

โครงสร้างไฟล์

```
basic-never/
└── index.ts
```

โค้ด index.ts

```
function throwError(message: string): never {
  throw new Error(message);
}

function testCondition(cond: boolean): void {
  if (!cond) {
    throwError("Condition failed!");
  }
  console.log("Condition passed.");
}

try {
```

```
  testCondition(false);
} catch (e) {
  console.error("Caught error:", (e as Error).message);
}

testCondition(true);
```

คำอธิบาย

- ฟังก์ชัน throwError มีชนิดคืนเป็น never เพราะไม่เคยคืนค่า
- ใช้ในกรณีที่เกิด exception หรือโปรแกรมหยุดทำงาน

ผลการรัน

```
Caught error: Condition failed!
Condition passed.
```

โปรแกรมแนวประยุกต์ 3 ตัว: ใช้งานจริงกับ null, undefined, any, unknown, never

---

ตัวอย่างที่ 4: ฟังก์ชันอ่านข้อมูลและตรวจสอบ null กับ undefined

โครงสร้างไฟล์

```
app-null-check/
└── index.ts
```

โค้ด index.ts

```
interface Product {
  id: number;
  name: string;
  description?: string | null;
}

function printProductDescription(product: Product): void {
  if (product.description === undefined) {
    console.log("Description is undefined.");
  } else if (product.description === null) {
    console.log("Description is null.");
  } else {
    console.log(`Description: ${product.description}`);
  }
```

```
}

const product1: Product = { id: 1, name: "Pen", description: null };
const product2: Product = { id: 2, name: "Notebook" };
const product3: Product = { id: 3, name: "Pencil", description: "A graphite pencil." };

printProductDescription(product1);
printProductDescription(product2);
printProductDescription(product3);
```

คำอธิบาย

- ใช้ optional property description ที่อาจเป็น string, null, หรือ undefined
- ตรวจสอบค่าและแสดงผลข้อความตามสถานะ

ผลการรัน

```
Description is null.
Description is undefined.
Description: A graphite pencil.
```

ตัวอย่างที่ 5: ฟังก์ชันรับข้อมูล unknown และตรวจสอบก่อนใช้งาน

โครงสร้างไฟล์

```
app-unknown-check/
└── index.ts
```

โค้ด index.ts

```
function processInput(input: unknown): void {
  if (typeof input === "string") {
    console.log("Input is a string:", input.toUpperCase());
  } else if (typeof input === "number") {
    console.log("Input is a number:", input * 2);
  } else if (Array.isArray(input)) {
    console.log("Input is an array of length:", input.length);
  } else {
    console.log("Input type is unknown.");
  }
}
```

```
processInput("hello");
processInput(123);
processInput([1, 2, 3]);
processInput({ key: "value" });
```

คำอธิบาย

- ใช้ unknown เพื่อรับข้อมูลไม่แน่นอน
- ตรวจสอบชนิดก่อนใช้งานทุกครั้ง เพื่อความปลอดภัย

ผลการรัน

```
Input is a string: HELLO
Input is a number: 246
Input is an array of length: 3
Input type is unknown.
```

ตัวอย่างที่ 6: ฟังก์ชันที่ใช้ never กับการจัดการข้อผิดพลาดในระบบ

โครงสร้างไฟล์

```
app-error-handler/
└── index.ts
```

โค้ด index.ts

```
class AppError extends Error {
  constructor(message: string) {
    super(message);
    this.name = "AppError";
  }
}

function fatalError(message: string): never {
  throw new AppError(message);
}

function runApp(shouldFail: boolean): void {
  if (shouldFail) {
    fatalError("Fatal error occurred!");
  }
  console.log("Application is running.");
```

```
}

try {
  runApp(true);
} catch (e) {
  if (e instanceof AppError) {
    console.error("AppError caught:", e.message);
  } else {
    console.error("Unknown error:", e);
  }
}

runApp(false);
```

คำอธิบาย

- กำหนดคลาส error ใหม่ AppError
- ฟังก์ชัน fatalError คืนค่า never เพราะหยุดโปรแกรมโดยโยน error
- ฟังก์ชัน runApp ใช้ fatalError เมื่อต้องการหยุดโปรแกรม
- จัดการ error ด้วย try/catch และตรวจสอบชนิดของ error

ผลการรัน

```
AppError caught: Fatal error occurred!
Application is running.
```

สรุป

บทที่ 2 ของหนังสือจะปูพื้นฐานความเข้าใจเกี่ยวกับตัวแปรและชนิดข้อมูลใน TypeScript โดยอธิบายความแตกต่างของ let, const, และ var พร้อมแนวคิดเรื่อง Scope การกำหนดชนิดข้อมูลพื้นฐาน เช่น string, number, boolean ตลอดจนการจัดการค่าพิเศษอย่าง null และ undefined นอกจากนี้ยังแนะนำการใช้ชนิดข้อมูลพิเศษอย่าง any, unknown และ never เพื่อเพิ่มความยืดหยุ่นและความปลอดภัยในการเขียนโค้ดอย่างมีประสิทธิภาพ.

บทที่ 3
ฟังก์ชันพื้นฐาน
(Basic Functions)

เนื้อหา

- ฟังก์ชันพื้นฐานใน TypeScript
- ฟังก์ชันพื้นฐานใน TypeScript — รายละเอียดเชิงลึก
- การกำหนด Parameter Type
- Optional Parameters คืออะไร?
- Default Parameters (พารามิเตอร์ที่มีค่าเริ่มต้น) ใน TypeScript
- Rest Parameters (พารามิเตอร์จำนวนไม่จำกัด) ใน TypeScript
- Arrow Functions และ Function Expressions ใน TypeScript

บทนำ: ฟังก์ชันพื้นฐาน

ฟังก์ชันถือเป็นองค์ประกอบหลักของการเขียนโปรแกรมในทุกภาษา และใน TypeScript ฟังก์ชันไม่ได้เป็นเพียงแค่ชุดของคำสั่งที่สามารถเรียกใช้ซ้ำได้เท่านั้น แต่ยังถูกออกแบบให้สามารถกำหนดประเภทของข้อมูลที่รับเข้า (parameter) และส่งกลับ (return type) ได้อย่างชัดเจน ซึ่งเป็นประโยชน์อย่างยิ่งในการตรวจสอบความถูกต้องของโค้ดตั้งแต่ขั้นตอนการเขียน ช่วยลดข้อผิดพลาดที่อาจเกิดขึ้นในขณะรันโปรแกรม

ในบทนี้ ผู้อ่านจะได้เรียนรู้วิธีการประกาศและเรียกใช้งานฟังก์ชันโดยระบุชนิดของ parameter และค่าที่ฟังก์ชันส่งกลับอย่างชัดเจน เช่น function greet(name: string): string ซึ่งช่วยให้โปรแกรมทำงานอย่างถูกต้องและมีความปลอดภัยมากยิ่งขึ้น การกำหนดประเภทข้อมูลที่ชัดเจนยังช่วยให้ IDE อย่าง Visual Studio Code สามารถแนะนำโค้ด (auto-completion) ได้แม่นยำขึ้น

นอกจาก parameter ปกติแล้ว TypeScript ยังรองรับ **optional parameters** ซึ่งเปิดโอกาสให้สามารถส่งค่าหรือไม่ส่งค่าบาง parameter ก็ได้ โดยการใส่เครื่องหมาย ? หลังชื่อพารามิเตอร์ เช่น function greet(name?: string) การใช้ optional parameters อย่างเหมาะสมทำให้ฟังก์ชันมีความยืดหยุ่น และสามารถนำไปใช้ในสถานการณ์ที่มีข้อมูลไม่ครบถ้วนได้ดี

อีกความสามารถที่น่าสนใจคือ **default parameters** ซึ่งช่วยให้สามารถกำหนดค่าตั้งต้นให้กับ parameter ได้ในกรณีที่ผู้เรียกใช้งานไม่ได้ส่งค่าเข้ามา ตัวอย่างเช่น function greet(name: string = "Guest") การใช้ default parameter ทำให้ฟังก์ชันมีความสามารถในการจัดการข้อมูลได้เองในระดับหนึ่ง ลดภาระในการตรวจสอบค่าที่ผู้ใช้ส่งเข้ามา

สำหรับกรณีที่ไม่สามารถคาดเดาจำนวน parameter ได้ล่วงหน้า TypeScript ก็มี **rest parameters** ซึ่งช่วยให้สามารถรับค่าหลายๆ ค่าได้ในรูปแบบของ array โดยใช้เครื่องหมาย ... เช่น function sum(...numbers: number[]) คุณสมบัตินี้เหมาะสำหรับฟังก์ชันที่ต้องจัดการข้อมูลจำนวนมากหรือจำนวนไม่แน่นอน

นอกจากรูปแบบการเขียนแบบเดิมแล้ว TypeScript ยังสนับสนุน **arrow functions** ซึ่งเป็นรูปแบบการเขียนฟังก์ชันที่กระชับ และมักนิยมใช้ในโค้ดยุคใหม่ โดยเฉพาะเมื่อต้องการเขียนฟังก์ชันแบบ inline หรือใช้ร่วมกับ array methods เช่น .map(), .filter(), .reduce() นอกจากนี้ยังมี **function expressions** ซึ่งช่วยให้สามารถกำหนดฟังก์ชันเป็นค่าของตัวแปรได้ เพิ่มความยืดหยุ่นในการเขียนโปรแกรมเชิงฟังก์ชัน

บทที่ 3 นี้จึงมีเป้าหมายเพื่อให้นักพัฒนาเข้าใจและใช้ฟังก์ชันใน TypeScript ได้อย่างมีประสิทธิภาพ ทั้งในแง่ของโครงสร้าง การจัดการพารามิเตอร์ และรูปแบบการเขียนที่หลากหลาย ซึ่งจะช่วยเพิ่มความสามารถในการออกแบบโปรแกรมที่ซับซ้อนและสามารถนำไปใช้ได้จริงในสถานการณ์หลากหลาย.

ฟังก์ชันพื้นฐานใน TypeScript

1. การกำหนด Parameter และ Return Type

- TypeScript ช่วยระบุชนิดของพารามิเตอร์และชนิดผลลัพธ์ที่ฟังก์ชันคืนค่า
- ช่วยเพิ่มความปลอดภัยและความชัดเจนของโค้ด

ตัวอย่าง:

```
function add(x: number, y: number): number {
  return x + y;
}

const result = add(3, 4); // result เป็น number = 7
```

2. Optional Parameters (พารามิเตอร์ที่ไม่จำเป็น)

- พารามิเตอร์ที่ไม่จำเป็นต้องใส่ใช้เครื่องหมาย ?
- ถ้าไม่ใส่ ค่าจะเป็น undefined

ตัวอย่าง:

```
function greet(name: string, greeting?: string): string {
  if (greeting) {
    return `${greeting}, ${name}!`;
  }
```

```
  return `Hello, ${name}!`;
}

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

---

3. Default Parameters (พารามิเตอร์มีค่าเริ่มต้น)

- กำหนดค่าเริ่มต้นให้พารามิเตอร์ได้
- ถ้าไม่ได้ส่งค่ามา จะใช้ค่าเริ่มต้นนั้น

ตัวอย่าง:

```
function greet(name: string, greeting: string = "Hello"): string {
  return `${greeting}, ${name}!`;
}

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

---

4. Rest Parameters (พารามิเตอร์จำนวนไม่จำกัด)

- ใช้ ... เพื่อรับพารามิเตอร์หลายตัวเป็นอาเรย์
- เหมาะสำหรับฟังก์ชันที่รับ argument ไม่จำกัดจำนวน

ตัวอย่าง:

```
function sum(...numbers: number[]): number {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}

console.log(sum(1, 2, 3));      // 6
console.log(sum(5, 10, 15, 20));  // 50
```

---

5. Arrow Functions และ Function Expressions

- Arrow function คือฟังก์ชันที่นิยามแบบย่อด้วยเครื่องหมาย =>
- ใช้แทนฟังก์ชันแบบปกติ (function expression) ได้
- เก็บค่า this จากบริบทรอบข้าง (lexical this)
- โค้ดย่อและอ่านง่าย

ตัวอย่าง:

```
// Function expression แบบปกติ
const multiply = function(x: number, y: number): number {
  return x * y;
};

// Arrow function แบบย่อ
const multiplyArrow = (x: number, y: number): number => x * y;

console.log(multiply(3, 4));       // 12
console.log(multiplyArrow(3, 4));  // 12
```

ตัวอย่างโปรแกรมแบบเต็มไฟล์ 3 โปรแกรม (ครอบคลุมหัวข้อข้างต้น)

ตัวอย่างที่ 1: ฟังก์ชัน greet ใช้ optional และ default parameters

โครงสร้างไฟล์

```
function-basic-1/
└── index.ts
```

โค้ด index.ts

```
function greet(name: string, greeting?: string): string {
  const greetWord = greeting ?? "Hello";
  return `${greetWord}, ${name}!`;
}

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

คำอธิบาย

- greeting เป็น optional parameter
- ใช้ nullish coalescing operator (??) กำหนดค่า default ถ้า greeting เป็น undefined

---

ตัวอย่างที่ 2: ฟังก์ชันคำนวณผลรวมด้วย rest parameters และ arrow function

โครงสร้างไฟล์

```
function-basic-2/
└── index.ts
```

โค้ด index.ts

```
const sum = (...nums: number[]): number => {
  return nums.reduce((acc, cur) => acc + cur, 0);
};

console.log(sum(1, 2, 3));          // 6
console.log(sum(10, 20, 30, 40));  // 100
```

คำอธิบาย

- sum ใช้ rest parameter เพื่อรับจำนวนเลขไม่จำกัด
- ใช้ arrow function แบบย่อ

ตัวอย่างที่ 3: ฟังก์ชันคูณแบบ function expression และ arrow function

โครงสร้างไฟล์

```
function-basic-3/
└── index.ts
```

โค้ด index.ts

```
const multiply = function(x: number, y: number): number {
  return x * y;
};

const multiplyArrow = (x: number, y: number): number => x * y;

console.log(multiply(5, 6));        // 30
console.log(multiplyArrow(5, 6));  // 30
```

คำอธิบาย

- แสดง function expression กับ arrow function ที่ทำงานเหมือนกัน

ฟังก์ชันพื้นฐานใน TypeScript — รายละเอียดเชิงลึก

---

1. การกำหนด Parameter และ Return Type

- ใน TypeScript เราสามารถระบุชนิดข้อมูลของพารามิเตอร์และค่าที่ฟังก์ชันคืนค่า (return) ได้
- การระบุชนิดช่วยให้ตรวจจับข้อผิดพลาดตั้งแต่ก่อนรันจริง (compile time) เช่น ส่งพารามิเตอร์ผิดชนิด หรือคืนค่าผิดชนิด
- ฟังก์ชันที่ไม่มีการคืนค่า (void) จะใช้ : void เช่น

```
function sayHi(name: string): void {
```

```
  console.log(`Hi, ${name}`);
}
```

- ถ้าไม่ระบุ return type TypeScript จะ infer ให้จากโค้ด แต่ควรระบุชัดเจนเพื่อความปลอดภัยและ readability
- ตัวอย่างถ้าส่งพารามิเตอร์ผิดชนิด จะ error ตั้งแต่ compile เช่น

```
sayHi(123); // Error: Argument of type 'number' is not assignable to parameter of type 'string'.
```

---

2. Optional Parameters (พารามิเตอร์ที่ไม่จำเป็น)

- ใช้เครื่องหมาย ? ต่อท้ายชื่อพารามิเตอร์เพื่อระบุว่าไม่จำเป็นต้องส่งค่า
- พารามิเตอร์ optional จะเป็นชนิด union กับ undefined อัตโนมัติ เช่น greeting?: string เทียบเท่ากับ greeting: string | undefined
- ควรวางพารามิเตอร์ optional ไว้หลังพารามิเตอร์ที่จำเป็นเท่านั้น เพราะการวางก่อนอาจทำให้การเรียกฟังก์ชันสับสน

```
function greet(name: string, greeting?: string) { ... }
```

- เมื่อเรียกฟังก์ชันโดยไม่ส่งพารามิเตอร์นี้มา จะได้ค่า undefined และต้องตรวจสอบก่อนใช้ เช่น

```
if (greeting) { ... }
```

---

3. Default Parameters (พารามิเตอร์มีค่าเริ่มต้น)

- กำหนดค่าเริ่มต้นให้พารามิเตอร์ ทำให้ไม่จำเป็นต้องส่งค่ามา
- ถ้าส่งค่ามา จะใช้ค่านั้นแทนค่าเริ่มต้น
- ถ้าไม่ส่ง จะใช้ค่า default แทน
- Default parameter ยังถือเป็น optional parameter แบบมีค่า default

```
function greet(name: string, greeting: string = "Hello") { ... }
```

- ตัวอย่างการใช้:

```
greet("Alice");              // ใช้ greeting = "Hello"
greet("Bob", "Welcome");     // ใช้ greeting = "Welcome"
```

- ถ้าพารามิเตอร์ default อยู่ข้างหน้า ต้องส่งค่า undefined เพื่อให้ใช้ default เช่น

```
function test(x = 10, y: number) { ... }
test(undefined, 5); // x=10, y=5
```

---

4. Rest Parameters (พารามิเตอร์จำนวนไม่จำกัด)

- ช่วยให้ฟังก์ชันรับจำนวนอาร์กิวเมนต์ไม่จำกัดโดยเก็บในรูปอาเรย์
- ใช้ ... หน้า parameter ชื่อเดียวเท่านั้น (ต้องเป็น parameter สุดท้าย)
- ประกาศชนิดของอาเรย์ เช่น ...nums: number[]