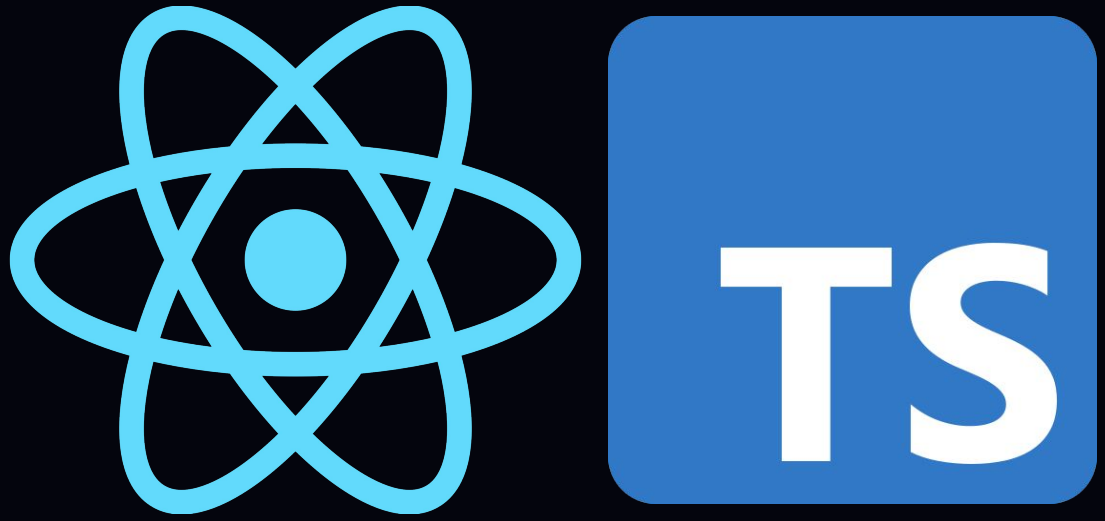
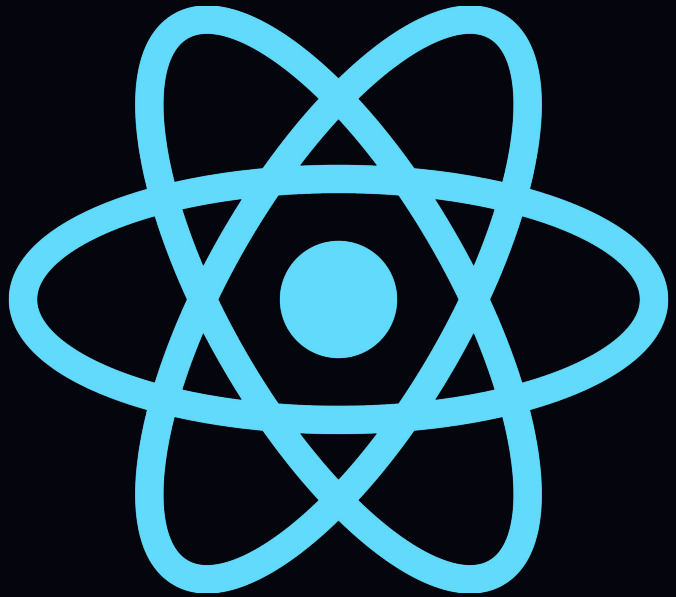


พัฒนาเว็บแอพพลิเคชั่นด้วย React & TypeScript

React คืออะไร



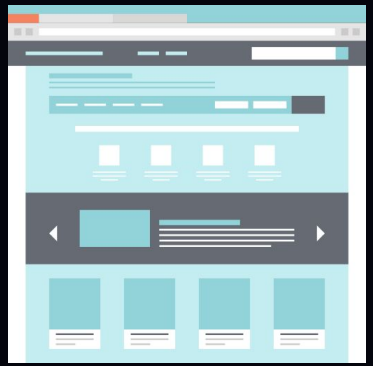
คือ ไลบรารี่ของภาษา JavaScript

ที่ใช้สำหรับสร้างส่วนติดต่อกับผู้ใช้งาน

(User Interface : UI) หรือการสร้าง

หน้าเว็บให้สวยงามและใช้งานง่าย

แนวคิดของ React


การสร้างเว็บไซต์แบบดั่งเดิม ผู้พัฒนาเว็บไซต์ต้องเขียนโค้ดหน้าเว็บทั้งหมดเก็บไว้ในไฟล์เดียว ส่งผลให้เว็บไซต์ทำงานช้าเนื่องจากต้องโหลดเนื้อหาใหม่ทั้งหมดเมื่อมีการเปลี่ยนแปลงการทำงานภายในหน้าเว็บ

แนวคิดของ React

React ถูกพัฒนาขึ้นเพื่อนำมาใช้สร้างหน้าเว็บ โดยมีแนวคิดคือการแบ่งส่วนแสดงผลออกเป็นชิ้นส่วนย่อยหลายๆส่วน โดยไม่ต้องเขียนโค้ดเก็บไว้ในไฟล์เดียวเพื่อให้ง่ายต่อการจัดการ จากนั้นจะนำส่วนย่อยดังกล่าวมาประกอบรวมกันในภายหลัง เราจะเรียกองค์ประกอบที่แบ่งออกเป็นชิ้นส่วนย่อย ๆ นี้ว่า “คอมโพเนนต์ (Component)” ซึ่งเขียนด้วยภาษา JavaScript เพื่อออกแบบและทำหน้าตาแต่ละส่วนของเว็บไซต์

แนวคิดของ React

- Component คือ ชิ้นส่วนต่างๆที่ถูกนำมาประกอบรวมกันเป็นหน้าเว็บ *(คล้ายๆ การสร้าง tag ขึ้นมาใช้เอง เช่น <Navbar/>)* และสามารถนำกลับมาใช้ซ้ำได้

- State คือ *ข้อมูลที่ถูกเก็บไว้ภายใน Component* สามารถเปลี่ยนแปลงได้ตามการกระทำของผู้ใช้งาน เช่น สถานะการล็อกอิน ข้อมูลที่กรอกในฟอร์ม จำนวนครั้งที่กดปุ่ม เป็นต้น เมื่อข้อมูล State เปลี่ยน React จะอัปเดตหน้าเว็บให้อัตโนมัติ

แนวคิดของ React

- Props (Properties) คือ ข้อมูลที่ถูกส่งจาก Component หนึ่งไปยังอีก Component หนึ่ง ทำให้แต่ละ Component สามารถแลกเปลี่ยนข้อมูลเพื่อรับค่ามาแสดงผลหรือนำมาใช้งานได้หลากหลายตามที่เราต้องการได้

แนวคิดของ React

- ใน React จะไม่เขียนคำสั่ง HTML ในไฟล์นามสกุล .html โดยตรง แต่จะเขียนในไฟล์ JavaScript แทน ซึ่งจะอาศัยสิ่งที่เรียกว่า JSX (JavaScript XML) ที่ทำให้นักพัฒนาสามารถแทรกคำสั่ง HTML เข้าไปใน JavaScript ได้

- *ดังนั้นการใช้งาน React ก็คือ การสร้างหน้าเว็บด้วยภาษา JavaScript ที่มี HTML แทรกอยู่ภายในนั่นเอง !*

JSX คืออะไร

JSX (JavaScript XML) คือ ไวยากรณ์แบบพิเศษสำหรับนำมาใช้ในการพัฒนาเว็บด้วย React ที่ช่วยให้นักพัฒนาสามารถเขียนคำสั่ง HTML แทรกเข้าไปใน JavaScript ได้

การใช้งาน JSX

```
function App(){
    return (
        //พื้นที่เขียนคำสั่ง HTML
    );
}
```

```
function App() {
  return (
      <h1>สวัสดีครับ</h1>
  );
}
```

การใช้งาน JSX

```
function App() {
  const name="ก้องรักสยาม"
  return (
        <h1>สวัสดีครับผมชื่อ : {name}</h1>
  );
}
```

ใช้ปีกกาสำหรับแทรกคำสั่ง JavaScript ลงไปใน HTML

ข้อเสียของการใช้งาน JavaScript

- **ไม่มีการตรวจสอบชนิดข้อมูล** เนื่องจาก JavaScript เป็นภาษาแบบ Dynamic Type กล่าวคือสามารถนิยามตัวแปรโดยไม่ต้องระบุชนิดข้อมูลกำกับได้

ข้อดีของ Dynamic Type คือ ทำให้การเขียนโปรแกรมมีความยืดหยุ่นสูงเพราะสามารถเปลี่ยนแปลงชนิดข้อมูลของตัวแปรได้ตลอดเวลาโดยไม่ต้องประกาศไว้ล่วงหน้า แต่ในขณะเดียวกันนั้น ก็อาจทำให้เกิดบั๊กหรือเกิดข้อผิดพลาดได้ง่าย

ข้อเสียของ JavaScript

- **Debug ยาก** การใช้งาน JavaScript นั้นนักพัฒนาจะไม่รู้เลยว่าโค้ดทำงานถูกต้องหรือไม่ เนื่องจากข้อผิดพลาดต่างๆ จะถูกพบเมื่อตอน Runtime หรือตอนรันโปรแกรมเท่านั้น ทำให้หาจุดผิดพลาดได้ยากและใช้เวลานานเพราะต้องรันโปรแกรมก่อนจึงจะทราบสาเหตุ **(ไม่มีการตรวจสอบข้อผิดพลาดก่อนรันโปรแกรมจริง)**

แก้ปัญหาดังกล่าวด้วย TypeScript

รู้จักกับ TypeScript

- TypeScript ถูกพัฒนาขึ้นโดย Microsoft เพื่อแก้ปัญหาความยุ่งยากในการพัฒนาระบบขนาดใหญ่ที่มีความซับซ้อนสูงด้วย JavaScript

- เนื่องจากในช่วงแรกนั้น JavaScript เป็นภาษาที่มีความยืดหยุ่นมาก แต่ความยืดหยุ่นดังกล่าวทำให้เกิดปัญหาตามมา เพราะการเขียนโค้ดด้วย JavaScript นั้นไม่มีการบังคับให้กำหนดชนิดข้อมูล (Data Types) ทำให้เกิดข้อผิดพลาดได้ง่ายและยากต่อการบำรุงรักษาโค้ดในระยะยาว

รู้จักกับ TypeScript

- TypeScript คือ JavaScript เวอร์ชั่นอัพเกรดที่มีการเพิ่มความสามารถในการกำหนดชนิดข้อมูลเข้าไปรวมถึงคุณสมบัติต่างๆ ที่ช่วยให้นักพัฒนาสามารถเขียนโค้ด JavaScript ให้มีโครงสร้างชัดเจน เข้าใจง่าย และมีความปลอดภัยมากขึ้น เช่น ระบบตรวจสอบชนิดข้อมูลเพื่อลดข้อผิดพลาดต่างๆ ในระหว่างการเขียนโค้ดรวมถึงรองรับการเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming)

รู้จักกับ TypeScript

- การใช้งาน TypeScript จึงเหมาะสำหรับนำไปใช้กับโปรเจกต์ขนาดใหญ่ที่มีความซับซ้อนสูงและมีทีมนักพัฒนาหลายคน ซึ่งจะช่วยให้นักพัฒนาสามารถทำงานร่วมกันได้อย่างมีประสิทธิภาพมากขึ้นและง่ายต่อการบำรุงรักษาโค้ดในระยะยาว

คุณสมบัติของ TypeScript

- Static Type เป็นภาษาที่มีระบบกำหนดและตรวจสอบชนิดข้อมูล ทำให้นักพัฒนาสามารถกำหนดชนิดข้อมูลลงในตัวแปร , ฟังก์ชั่น , Object ได้ จึงเหมาะสำหรับนำไปใช้ในโปรเจกต์ขนาดใหญ่ที่ต้องการความเสถียรและเน้นเรื่องความปลอดภัย

คุณสมบัติของ TypeScript

- **ตรวจจับข้อผิดพลาดตั้งแต่ขั้นตอนการเขียนโค้ด** ภาษา TypeScript นั้นมีความสามารถในการค้นหาข้อผิดพลาดตั้งแต่ขั้นตอนการเขียนโค้ด (Compile Time) เช่น ถ้าพิมพ์ผิดหรือเรียกฟังก์ชั่นผิดจะขึ้นแจ้งเตือนทันที ทำให้นักพัฒนาทราบปัญหาที่อาจจะเกิดขึ้นก่อนจะเริ่มต้นรันโปรแกรม

คุณสมบัติของ TypeScript

- Superset ของ JavaScript หมายถึง รองรับฟีเจอร์ใหม่ๆของ JavaScript และสามารถแปลง (Compile) เป็น JavaScript เวอร์ชั่นอื่นๆ ได้ เพื่อนำไปใช้งานร่วมกับเบราว์เซอร์หรือระบบเก่าๆ ที่ยังไม่รองรับฟีเจอร์ใหม่ (ใช้ได้ทั้งฝั่ง Front-End และ Back-End)

คุณสมบัติของ TypeScript

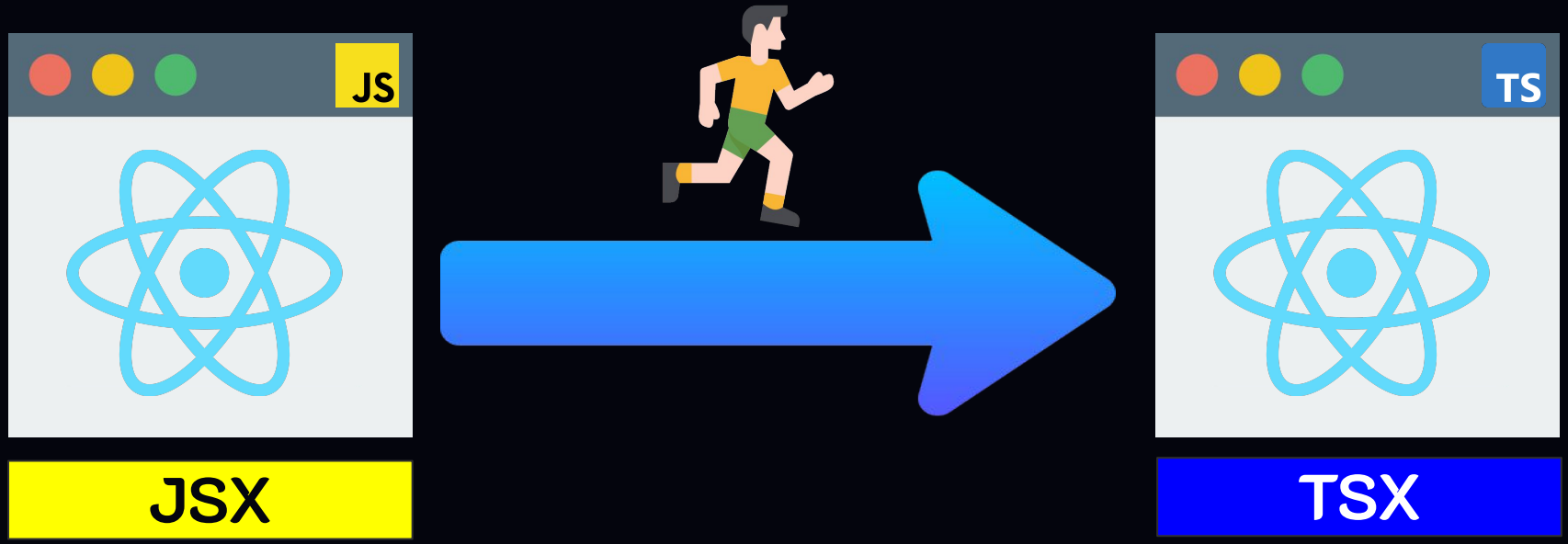
- **มีโครงสร้างที่ชัดเจน** ช่วยทำให้โค้ดมีความยืดหยุ่นและสามารถนำไปใช้ซ้ำได้ เหมาะกับโปรเจกต์ขนาดใหญ่ที่ต้องการความเป็นระเบียบและบำรุงรักษาโค้ดง่ายในระยะยาว

- **เครื่องมือสนับสนุน** เนื่องจาก TypeScript นั้นมีโครงสร้างที่ชัดเจน ทำให้ Editor หรือ IDE สามารถแนะนำคำสั่งต่างๆที่เกี่ยวข้องได้แม่นยำขึ้น (Auto Complete) อีกทั้งยังระบุจุดข้อผิดพลาดได้แบบ Real-Time ทำให้ Refactor โค้ดได้ง่ายและปลอดภัย

TypeScript Compile JavaScript

ข้อควรทราบ : การใช้งาน TypeScript นั้นไม่สามารถนำไปใช้งานบน Browser ได้โดยตรง เนื่องจาก Browser เข้าใจแค่ JavaScript นักพัฒนาต้องเขียนโปรแกรมด้วย TypeScript ก่อนและคอมไพล์เพื่อแปลงโค้ดดังกล่าวให้เป็น JavaScript จากนั้นนำผลลัพธ์ที่ได้จากการแปลงไปทำงานบน Browser อีกที

อยากนำ TypeScript ไปใช้งานร่วมกับ React ต้องทำอย่างไร ?


JS
JSX
TS
TSX

TSX คืออะไร

- TSX คือ การนำ TypeScript มาใช้งานร่วมกับ JSX (JavaScript XML) เพื่อให้นักพัฒนาสามารถสร้างโปรเจกต์ด้วย React โดยใช้ไวยากรณ์ของ JSX ให้ทำงานร่วมกับระบบตรวจสอบชนิดข้อมูลและคุณสมบัติอื่นๆของ TypeScript ได้

ประโยชน์ของ TSX

- มีระบบตรวจสอบชนิดข้อมูลและช่วยตรวจจับข้อผิดพลาดต่างๆ เพื่อลดโอกาสการเกิดบั๊กหรือข้อผิดพลาดที่อาจจะเกิดขึ้นก่อนรันโปรแกรม

- โค้ดมีโครงสร้างที่ชัดเจน อีกทั้งยังมีความยืดหยุ่นและสามารถนำไปใช้ซ้ำได้ทำให้ง่ายต่อการบำรุงรักษาโค้ดในระยะยาว

ประโยชน์ของ TSX

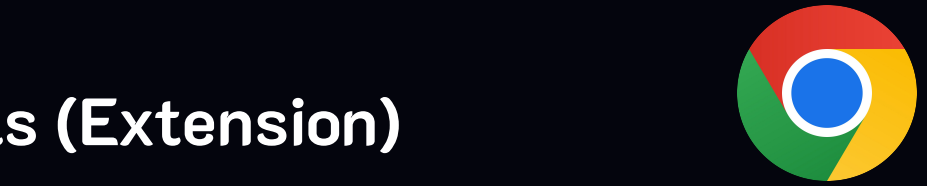
- **ทำงานเป็นทีมได้ง่ายขึ้น** สำหรับโปรเจกต์ขนาดใหญ่ที่มีนักพัฒนาหลายคน การใช้งาน TSX จะช่วยให้ทุกคนในทีมจัดการโค้ดได้ง่ายขึ้น โดยใช้ข้อกำหนดหรือโครงสร้างโค้ดที่เป็นมาตรฐานเดียวกัน เพื่อลดความผิดพลาดจากการสื่อสาร ทำให้ทุกคนในทีมทำงานร่วมกันได้อย่างมีประสิทธิภาพ

ต้องมีพื้นฐานอะไรบ้าง

- มีพื้นฐาน HTML5 , CSS3 , JavaScript
- มีพื้นฐาน TypeScript
- มีพื้นฐาน React
- มีพื้นฐานการใช้งาน Visual Studio Code

เครื่องมือพื้นฐาน

- Node.js
- Visual Studio Code
- Google Chrome
- React Developer Tools (Extension)



VSCode Extension

- Auto Rename Tag
- Color Highlight
- Prettier - Code formatter
- Error Lens *(Optional)*
- Material Icon Theme *(Optional)*

ปูพื้นฐานการใช้งาน TypeScript

รูปแบบการใช้งาน

1. ใช้งานผ่านเว็บไซต์ (TypeScript Playground)

   https://www.typescriptlang.org/play

2. ติดตั้ง TypeScript และใช้งานในเครื่องคอมพิวเตอร์

การใช้งาน TypeScript

1. ติดตั้ง Node.js : (https://nodejs.org/en)
2. ติดตั้ง TypeScript : npm install -g typescript
3. ตรวจสอบเวอร์ชั่น : tsc -v หรือ tsc - -version
4. สร้างโปรเจกต์ TypeScript : tsc - - init
5. tsc ชื่อไฟล์.ts (คอมไพล์หรือแปลง TypeScript เป็น JavaScript)

(tsc = TypeScript Compiler)

รู้จักกับไฟล์ tsconfig.json

- ไฟล์ที่เก็บการตั้งค่าการทำงานของ TypeScript Compiler เพื่อใช้สำหรับคอมไพล์หรือแปลง TypeScript ไปเป็น JavaScript โดยนักพัฒนาสามารถระบุกฎเกณฑ์หรือเงื่อนไขในการทำงานเพื่อควบคุมคุณภาพโค้ดได้

ตัวอย่างการตั้งค่า

- ต้องการคอมไพล์ไฟล์ TypeScript ใดบ้าง (กรณีมีหลายไฟล์)
- ต้องการคอมไพล์ TypeScript ให้เป็น JavaScript เวอร์ชั่นใด
- ไฟล์ที่คอมไพล์เรียบร้อยแล้วจะจัดเก็บไว้ที่ไหน
- ต้องการตรวจสอบโค้ดแบบเข้มงวดหรือไม่

ข้อดีของ tsconfig.json

- ไม่ต้องพิมพ์ Option ยาวในตอนรันคำสั่ง tsc
- ทำงานเป็นทีมได้ง่ายขึ้น เนื่องจากทุกคนในทีมใช้การตั้งค่าเดียวกันทำให้ได้ผลลัพธ์เหมือนกัน
- สามารถจัดการโปรเจ็กต์ขนาดใหญ่ได้อย่างเป็นระบบ

compilerOptions	ความหมาย
target	ต้องการให้คอมไพล์เป็น JavaScript เวอร์ชั่นใด (ES5, ES2015, ES2020, อื่นๆ)
module	ตั้งค่าระบบ Module ที่ TypeScript จะใช้คอมไพล์โค้ดออกมา ซึ่งเกี่ยวข้องกับวิธีการ import และ export โค้ดรวมถึงการจัดการ Dependencies ต่างๆ ใน JavaScript (CommonJS , AMD , UMD)
outDir	ตั้งค่าโฟลเดอร์ปลายทางสำหรับเก็บไฟล์ที่คอมไพล์แล้ว (JavaScript)
rootDir	ตั้งค่าโฟลเดอร์ต้นทางสำหรับเก็บโค้ด TypeScript

compilerOptions	ความหมาย
strict	เปิดการตรวจสอบชนิดข้อมูลแบบเข้มงวดใน TypeScript
noImplicitAny	ห้ามใช้ตัวแปรที่ไม่ระบุชนิดข้อมูล
strictNullChecks	ตรวจสอบค่า null และ undefined อย่างเข้มงวด

ข้อมูลเพิ่มเติม : https://www.typescriptlang.org/tsconfig/

ตัวอย่างการตั้งค่า	ความหมาย
target: "es2016"	กำหนดให้คอมไพล์ TypeScript ออกไปเป็น JavaScript ES2016
"module": "commonjs"	ใช้ระบบ Module ในรูปแบบ CommonJS
"outDir": "./dist/"	กำหนดให้โฟลเดอร์ที่ชื่อว่า dist เก็บกลุ่มไฟล์ที่คอมไพล์เรียบร้อยแล้ว (ไฟล์ JavaScript)
"rootDir": "./"	กำหนดให้โฟลเดอร์ปัจจุบันเก็บกลุ่มไฟล์ TypeScript (.ts)

ตัวอย่างการตั้งค่า	ความหมาย
target: "es2016"	กำหนดให้คอมไพล์ TypeScript ออกไปเป็น JavaScript ES2016
"module": "commonjs"	ใช้ระบบ Module ในรูปแบบ CommonJS
"outDir": "./dist/"	กำหนดให้โฟลเดอร์ที่ชื่อว่า dist เก็บกลุ่มไฟล์ที่คอมไพล์เรียบร้อยแล้ว (ไฟล์ JavaScript)
"rootDir": "./"	กำหนดให้โฟลเดอร์ปัจจุบันเก็บกลุ่มไฟล์ TypeScript (.ts)

ควรระบุเลขเวอร์ชั่นเป็น

เลขรุ่น หรือ เลขปี ?

TypeScript อนุญาตให้เขียนได้

ทั้งคู่เพื่อความสะดวกในการใช้งาน

มาตรฐานอย่างเป็นทางการของ

TypeScript จะแนะนำให้นักพัฒนา

ระบุเลขเวอร์ชั่นเป็น “เลขปี”

เกี่ยวกับ JavaScript และ ECMAScript

- JavaScript เป็นภาษาคอมพิวเตอร์ที่ถูกสร้างขึ้นในปี ค.ศ 1995 โดย Brendan Eich เพื่อใช้งานกับเว็บเบราเซอร์ Netscape ภายใต้ข้อตกลงกับบริษัท Sun Microsystems

Brendan Eich

เกี่ยวกับ JavaScript และ ECMAScript

- ต่อมาในปี ค.ศ 1997 ทางด้าน Netscape ได้ส่งต่อภาษา JavaScript ให้กับทางองค์กรไม่แสวงหากำไรภายใต้ชื่อ ECMA (European Computer Manufacturers Association) เพื่อกำหนดมาตรฐานของภาษานี้ให้รองรับการทำงานในทุกเบราเซอร์

- ดังนั้นเราอาจเรียกภาษา JavaScript ในอีกชื่อหนึ่งว่า ECMAScript (เอ็กมาสคริปต์) ตามชื่อองค์กรที่เป็นผู้กำกับดูแลและกำหนดมาตรฐานของภาษานี้ในปัจจุบัน

JavaScript Version (ECMAScript)	ฟีเจอร์ที่สำคัญ
ES1 *(เผยแพร่ปี 1997)*	เวอร์ชั่นแรกของ JavaScript
ES2 *(เผยแพร่ปี 1998)*	ปรับปรุงโครงสร้างภายในของเวอร์ชั่นแรก
ES3 *(เผยแพร่ปี 1999)*	Regular Expressions , Try..Catch
ES5 *(เผยแพร่ปี 2009)*	รองรับ JSON อย่างเป็นทางการ , Array Method (forEach, map, filter, reduce)

JavaScript Version (ECMAScript)	ฟีเจอร์ที่สำคัญ
ES6 / ES2015 *(เผยแพร่ปี 2015)*	**Arrow Functions , let | const , Template Literals , Destructuring Assignment** *(มีการใช้เลขเวอร์ชั่นตามเลขปี)*
ES7 / ES2016 *(เผยแพร่ปี 2016)*	**Array.prototype.includes() , Exponentiation Operator (**)**
ES8 / ES2017 *(เผยแพร่ปี 2017)*	**async/await**
ES9 / ES2018 *(เผยแพร่ปี 2018)*	**Promise.finally() , Spread operator , Rest parameters**

JavaScript Version (ECMAScript)	ฟีเจอร์ที่สำคัญ
ES10 / ES2019 *(เผยแพร่ปี 2019)*	Array.prototype.flat() , Array.prototype.flatMap()
ES11 / ES2020 *(เผยแพร่ปี 2020)*	Optional chaining (?.), Nullish coalescing (??)
ES12 / ES2021 *(เผยแพร่ปี 2021)*	Logical Assignment Operators , Numeric separators
ES13 / ES2022 *(เผยแพร่ปี 2022)*	Static class blocks , Top-level await

JavaScript Version (ECMAScript)	ฟีเจอร์ที่สำคัญ
ES14 / ES2023 *(เผยแพร่ปี 2023)*	Array methods (toSorted, toReversed) , Shebang support
ES15 / ES2024 *(เผยแพร่ปี 2024)*	Set Operations (union , intersection, difference)
ES16 / ES2025 *(เผยแพร่ปี 2025)*	Iterator , Promise.try() , Regex.Escape()
ESNext *(Coming Soon...)*	ฟีเจอร์ใหม่ล่าสุดของ JavaScript ที่ยังไม่ประกาศใช้อย่างเป็นทางการ

ตั้งแต่ปี 2015 เป็นต้นมา
จะมีการปล่อย ECMAScript เวอร์ชั่นใหม่ใน
เดือนมิถุนายนของทุกปี เพื่อให้การพัฒนา
เป็นไปอย่างต่อเนื่องและสม่ำเสมอ

สรุปแบบเข้าใจง่าย

- ECMAScript เป็นมาตรฐานของภาษา (Specification) เพื่อใช้กำหนดกฎเกณฑ์และหลักการทำงานของภาษาโปรแกรม เช่น ไวยากรณ์และฟีเจอร์พื้นฐาน ดูแลโดย ECMA International

สรุปแบบเข้าใจง่าย

JavaScript เป็นภาษาโปรแกรมที่นำเอามาตรฐานของ ECMAScript มาใช้งาน (implementation) พร้อมกับเพิ่มฟีเจอร์หรือ APIs เฉพาะสำหรับแต่ละ Environment ตัวอย่างเช่น

- Browser Environment ประกอบด้วย DOM API, Web API, BOM
- Node.js (Runtime Environment) ประกอบด้วย File System API , HTTP API

สรุปแบบเข้าใจง่าย

การทำงานของ JavaScript นั้นต้องอาศัย JavaScript Engine หรือชุดแปลคำสั่งของภาษา JavaScript ซึ่งถูกพัฒนาขึ้นโดยบริษัทหรือองค์กรต่าง ๆ เช่น

- Google V8 Engine ใช้ใน Chrome, Node.js , อื่น ๆ
- Mozilla SpiderMonkey ใช้ใน Firefox
- Apple JavaScriptCore (Nitro) ใช้ใน Safari
- Microsoft Chakra เคยใช้ใน Internet Explorer และ Edge รุ่นเก่า

Module	ความหมาย
None	ไม่มีโครงสร้างโมดูล (ไม่มีการแบ่งไฟล์เป็นโมดูล)
CommonJS	ใช้ใน Node.js เวอร์ชั่นเก่า ที่มีการใช้คำสั่ง require , module.exports / exports
ES6/ESNext *(ตั้งแต่ ES6 หรือ ES2015 ขึ้นไป)*	ใช้ใน Node.js เวอร์ชั่นใหม่ที่รองรับ ES Modules (import / export , export default)

ข้อมูลเพิ่มเติม : https://www.typescriptlang.org/tsconfig/#module

Module	ความหมาย
AMD *(Asynchronous Module Definition)*	ใช้ในระบบที่ต้องการโหลด module แบบ Asynchronous เช่น เว็บไซต์ที่ใช้ RequireJS
UMD *(Universal Module Definition)*	เหมาะสำหรับนำไปสร้าง Library ที่ทำงานได้หลายสภาพแวดล้อม (ใช้ได้ทั้ง Browser และ Node.js)
System	สำหรับการโหลดโมดูลแบบ Dynamic

ข้อมูลเพิ่มเติม : https://www.typescriptlang.org/tsconfig/#module

ชนิดข้อมูลพื้นฐาน

ชนิดข้อมูลพื้นฐาน (Primitive Types)

ชนิดข้อมูล	คำอธิบาย	ตัวอย่าง
boolean	ค่าทางตรรกศาสตร์ *(ค่าความจริง)*	• true • false
number	ตัวเลข (จำนวนเต็มและทศนิยม)	• 2025 • 3.14 • -99.99
string	ข้อความ	• “ก้องรักสยาม”

รูปแบบการประกาศใช้งาน

- Explicit Type คือ การประกาศตัวแปรหรือค่าคงที่ โดยนักพัฒนาจะต้องระบุชนิดข้อมูลกำกับอย่างชัดเจนด้วยตนเอง
- Implicit Type หรือ Type Inference คือ การประกาศตัวแปรหรือค่าคงที่*โดยไม่มีการระบุชนิดข้อมูลกำกับ*และต้องกำหนดค่าเริ่มต้นในตอนที่ประกาศใช้งานซึ่งส่งผลให้ตัวแปรหรือค่าคงที่นั้นมีชนิดข้อมูลอ้างอิงตามค่าที่กำหนดให้โดยอัตโนมัติ

Explicit Type (กำหนดชนิดข้อมูลด้วยตนเอง)

- let name: string = "KongRuksiam";
- let age: number = 30;
- let isActive: boolean = true;

Implicit Type (กำหนดชนิดข้อมูลโดยอัตโนมัติ)

- let name = "KongRuksiam"; // string
- let age = 30; // number
- let isActive = true; // boolean

ตัวแปร (Variable)

- ชื่อที่ถูกนิยามขึ้นมาเพื่อใช้เก็บค่าข้อมูลสำหรับนำไปใช้งานในโปรแกรม โดยข้อมูลประกอบด้วย ข้อความ ตัวเลข ตัวอักษร หรือผลลัพธ์จากการประมวลผล โดยข้อมูลที่เก็บในตัวแปรนั้นสามารถเปลี่ยนแปลงค่าได้

โครงสร้างคำสั่ง

- let ชื่อตัวแปร : ชนิดข้อมูล = ค่าเริ่มต้น

ตัวอย่าง

- let username : string = “kongruksiam”
- let age:number = 30;
- let isActive : boolean = false;

ตัวอย่างการใช้งาน

- let username : string = “kongruksiam”
- username=”admin” 
- let age : number = 30
- age = 20 

ค่าคงที่ (Constant)

- มีลักษณะการใช้งานคล้ายกับตัวแปร แต่ค่าคงที่จะถูกนำมาใช้สำหรับจัดเก็บข้อมูลที่*ไม่สามารถเปลี่ยนแปลงค่าได้* ตอนประกาศใช้งานจะต้องมีการกำหนดค่าเริ่มต้นเสมอ

โครงสร้างคำสั่ง

- const ชื่อค่าคงที่ : ชนิดข้อมูล = ค่าเริ่มต้น

ตัวอย่าง

- const username : string = “admin”
- const age:number = 30;
- const isActive : boolean = false;

ตัวอย่างการใช้งาน

- const username : string = “kongruksiam”
- username=”admin” ❌
- const age : number = 30
- age = 20 ❌

Watch Mode

ปัญหาที่เกิดขึ้น!!!

ต้องสั่งคอมไพล์ด้วยตนเอง

ทุกครั้งที่มีการแก้ไขโค้ด TypeScript

แก้ปัญหาดังกล่าวด้วย

Watch Mode

Watch Mode

เป็นโหมดสำหรับสั่งคอมไพล์อัตโนมัติพร้อมกับติดตามการเปลี่ยนแปลงไฟล์ TypeScript ที่อยู่ภายในโปรเจกต์ ทุกครั้งที่มีการแก้ไขโค้ดและบันทึกไฟล์จะให้คอมไพล์โค้ดใหม่ทันที (คอมไพล์แบบเรียลไทม์)

คำสั่ง : tsc -w หรือ tsc --watch

ข้อดีของ Watch Mode

- **สะดวก** ไม่จำเป็นต้องสั่งคอมไพล์ด้วยตนเองทุกครั้งที่มีการแก้ไขโค้ด
- **ประหยัดเวลา** ทำการคอมไพล์เฉพาะไฟล์ที่เปลี่ยนแปลงเท่านั้น ไม่ได้คอมไพล์ใหม่ทั้งโปรเจกต์ ทำให้นักพัฒนาเห็นผลลัพธ์ทันทีหลังแก้ไขโค้ด
- **ทำงานได้อย่างต่อเนื่อง** เหมาะกับงานด้าน Development (เขียนโค้ด , แก้ไข,ทดสอบ)

Union Type

ปัญหาที่เกิดขึ้น

- การสร้างตัวแปรขึ้นมาใช้งานในภาษา TypeScript นั้นต้องระบุชนิดข้อมูลกำกับเพื่อบอกว่าตัวแปรที่สร้างขึ้นมาสามารถจัดเก็บข้อมูลอะไรได้บ้าง

ตัวอย่าง

- let age:number; //เก็บตัวเลขเท่านั้น
- let gender:string; //เก็บข้อความเท่านั้น

Union Types

- Union Types คือ การรวมชนิดข้อมูลหลายๆประเภทเข้าด้วยกัน โดยใช้เครื่องหมาย | (pipe) เป็นตัวเชื่อม ทำให้นักพัฒนาสามารถกำหนดค่าให้กับตัวแปรโดยใช้ชนิดข้อมูลแบบใดก็ได้มากกว่าหนึ่งชนิดตามที่ระบุในรายการที่กำหนดไว้

Union Types

โครงสร้างคำสั่ง

- let ชื่อตัวแปร : type1 | type2 | typeN;

ตัวอย่าง

- let id : number | string; //ใช้ชนิดข้อมูลได้ทั้ง number หรือ string

Union Types

```
let id : number | string;
id = 1234
id = "ABC"
id = false
```

Object Type

Object Types

- คือ การกำหนดโครงสร้างและชนิดข้อมูลของ Properties ที่อยู่ภายใน Object ทำให้นักพัฒนารู้ว่าใน Object นั้นมีโครงสร้าง Property อะไรบ้างและแต่ละ Property นั้นต้องจัดเก็บข้อมูลชนิดใด

ตัวอย่างการสร้าง Object

```
let product:object
product={
    name:"เมาส์",
    price:300,
    isActive:true
}
```

ตัวอย่างการสร้าง Object

```
let product:object
product={
    name:"เมาส์",
    price:"สามร้อย",
    isActive:true
}
```

สร้าง Object + กำหนดโครงสร้าง

```
//กำหนดโครงสร้าง
let product:{
    name:string
    price:number
    isActive:boolean
}
```

1

```
//สร้าง Object
product={
    name:"เมาส์",
    price:300,
    isActive:true
}
```

2

สร้าง Object + กำหนดโครงสร้าง

```
//กำหนดโครงสร้าง
let product:{
    name:string
    price:number
    isActive:boolean
}
```

```
//สร้าง Object
product={
    name:"เมาส์",
    price:"สามร้อย",
    isActive:true
}
```

```
//เขียนรวมกัน
let product:{
    name:string
    price:number
    isActive:boolean
}={
    name:"เมาส์",
    price:300,
    isActive:true
}
```

อาเรย์ (Array)

อาร์เรย์ (Array)

อาร์เรย์ เป็นชนิดข้อมูลแบบพิเศษสำหรับใช้เก็บข้อมูลหลายๆค่าเอาไว้ด้วยกัน โดยใช้ชื่ออ้างอิงเพียงชื่อเดียวและใช้หมายเลขกำกับ (index) เพื่อเข้าถึงตำแหน่งของข้อมูลที่จัดเก็บอยู่ภายใน ทำให้นักพัฒนาสามารถจัดการกลุ่มข้อมูลได้ง่ายมากขึ้น อีกทั้งยังมีความยืดหยุ่นสามารถเพิ่มและลดขนาดได้เองอัตโนมัติตามข้อมูลที่มีอยู่

คุณสมบัติของอาร์เรย์ (Array)

- ใช้เก็บกลุ่มของข้อมูลที่มีชนิดข้อมูลเดียวกัน
- ข้อมูลที่อยู่ในอาร์เรย์จะเรียกว่าสมาชิก (Element)
- สมาชิกแต่ละรายการจะเก็บค่าข้อมูล (value) และ อินเด็กซ์ (Index)
- Index หมายถึงคีย์ของอาร์เรย์ใช้อ้างอิงตำแหน่งของ Element เริ่มต้นที่ 0
- ขนาดมีความยืดหยุ่นสามารถเพิ่ม-ลดจำนวนสมาชิกได้

อาร์เรย์ (Array)

โครงสร้างคำสั่ง

```
let arrayName:type[]
let arrayName:type[] = []
```

ตัวอย่าง

```
let data:string[]
let data:string[] = []
```

ตัวอย่างอาร์เรย์ (Array)

```
//สร้าง Array
let colors:string[]
colors=["สีแดง","สีเขียว","สีน้ำเงิน"]

//สร้าง Array พร้อมกำหนดค่าเริ่มต้น
let scores:number[] = [10,20,30]
scores.push(40)
```

ตัวอย่างอาร์เรย์ (Array)

```
//สร้าง Array
let colors:string[]
colors=["สีแดง","สีเขียว","สีน้ำเงิน"]


//สร้าง Array พร้อมกำหนดค่าเริ่มต้น
let scores:number[] = [10,20,30]
scores.push("kong")❌
```

ฟังก์ชั่น (Function)

ฟังก์ชั่น (Function)

- ชุดคำสั่งที่นำมาเขียนรวมกันเป็นกลุ่มเพื่อเรียกใช้งานตามวัตถุประสงค์ที่ต้องการและลดความซ้ำซ้อนของคำสั่งที่ใช้งานบ่อย

- ฟังก์ชั่นสามารถนำไปใช้งานได้ทุกที่และแก้ไขได้ในภายหลังทำให้โค้ดในโปรแกรมมีระเบียบและใช้งานได้สะดวกมากยิ่งขึ้น

การสร้างฟังก์ชั่น (Function)

- ฟังก์ชั่นแบบไม่มีการรับและส่งค่า (Void)
- ฟังก์ชั่นแบบมีการรับค่า (Parameter)
- ฟังก์ชั่นแบบส่งค่ากลับ (Return)
- ฟังก์ชั่นแบบมีการรับและส่งค่า (Pamameter & Return)

ฟังก์ชั่นแบบไม่มีการรับและส่งค่า

```
function ชื่อฟังก์ชั่น() {

    // คำสั่งต่างๆ

}

การเรียกใช้งานฟังก์ชั่น

ชื่อฟังก์ชั่น ();
```

```
function ชื่อฟังก์ชั่น() : void {

    // คำสั่งต่างๆ

}

การเรียกใช้งานฟังก์ชั่น

ชื่อฟังก์ชั่น ();
```

ตัวอย่าง

```
function showMessage(){
    console.log("Hello World")
}
```

หรือ

```
function showMessage():void{
    console.log("Hello World")
}
```

ฟังก์ชั่นแบบมีการรับค่า (Parameter)

```
function ชื่อฟังก์ชั่น(พารามิเตอร์ : ชนิดข้อมูล){

    // คำสั่งต่างๆ

}
```

การเรียกใช้งานฟังก์ชั่น

```
ชื่อฟังก์ชั่น (argument1,argument2,...);
```

ฟังก์ชั่นแบบมีการรับค่า (Parameter)

```
function ชื่อฟังก์ชั่น(พารามิเตอร์ : ชนิดข้อมูล) : void {

    // คำสั่งต่างๆ

}
```

การเรียกใช้งานฟังก์ชั่น

```
ชื่อฟังก์ชั่น (argument1,argument2,...);
```

ตัวอย่าง

```
function showMessage(text:string):void{
    console.log("Hello",text)
}
//เรียกใช้งานฟังก์ชั่น
showMessage("TypeScript")
showMessage(30) ❌
```

ฟังก์ชั่นแบบส่งค่ากลับ (Return)

```
function ชื่อฟังก์ชั่น() : ชนิดข้อมูล {

    return ค่าที่จะส่งออกไปทำงานด้านนอก (อ้างอิงตามชนิดข้อมูล)

}
```

การเรียกใช้งานฟังก์ชั่น

```
ตัวแปรที่รับค่าจากฟังก์ชั่น = ชื่อฟังก์ชั่น ();
```

ตัวอย่าง

```
function getLocation():string{
    return "กรุงเทพมหานคร "
}

//เรียกใช้งานฟังก์ชั่น
let address = getLocation()
```

ตัวอย่าง

```
function getLocation():string{
    return true
}
function getLocation():string{
    return 100
}
```

ฟังก์ชั่นแบบมีการรับค่าและส่งค่า

```
function ชื่อฟังก์ชั่น(พารามิเตอร์ : ชนิดข้อมูล) : ชนิดข้อมูล {

    return ค่าที่จะส่งออกไปทำงานด้านนอก (อ้างอิงตามชนิดข้อมูล)

}
```

การเรียกใช้งานฟังก์ชั่น

```
ตัวแปรที่รับค่าจากฟังก์ชั่น = ชื่อฟังก์ชั่น (argument1,argument2,...);
```

ตัวอย่าง

```
function total(x:number,y:number):number{
    return x+y
}
//เรียกใช้งานฟังก์ชั่น
let result = total(100,200)
```

Arrow Function

- เป็นรูปแบบการเขียน Function ให้มีความกระชับมากยิ่งขึ้น

โครงสร้างคำสั่ง

```
const ชื่อฟังก์ชั่น=(พารามิเตอร์ : ชนิดข้อมูล) : ชนิดข้อมูล =>{
    return ค่าที่จะส่งออกไปทำงานด้านนอก (อ้างอิงตามชนิดข้อมูล)
}
```

ตัวอย่าง

Arrow Function

```
const total=(x:number,y:number):number=>{
    return x+y
}
```

หรือเขียนแบบสั้นกระชับ (บรรทัดเดียว)

```
const total = (x: number, y: number): number => x + y
```

Type Aliases

Type Aliases

- นอกจากชนิดข้อมูลพื้นฐานที่มีอยู่ในภาษา TypeScript นักพัฒนาสามารถนิยามหรือกำหนดชนิดข้อมูลขึ้นมาใช้งานเองได้โดยใช้เครื่องมือที่เรียกว่า Type Aliases (การตั้งชื่อใหม่ให้กับชนิดข้อมูล) เพื่อให้โค้ดอ่านง่ายและมีโครงสร้างที่เป็นระเบียบมากขึ้น

Type Aliases (Primitive Types)

```
type Text = string; //ตั้งชื่อเรียกใหม่
//การใช้งาน
let username: Text = "admin";
let firstname: Text = "ก้อง";
let lastname: Text = "รักสยาม";
let address: Text = "เชียงใหม่";
```

Type Aliases (Union Types)

```
type ID = string | number
let myid:ID="xyz" //string
myid=123 //number
```

Type Aliases (Object)

โครงสร้างคำสั่ง

```
type ชื่อTypeAliases = {
    property:type
    ....
}
```

ตัวอย่าง

```
type Employee = {
    id:number
    name:string
    salary:number
}
```

Type Aliases (Object)

```
type Employee = {
     id:number
     name:string
     salary:number
}
let data:Employee={id:1,name:"ก้อง",salary:15000}
```

ข้อดีของ Type Aliases

- **ทำให้โค้ดอ่านง่าย** มีโครงสร้างที่เป็นระเบียบมากขึ้น
- **นำไปใช้ซ้ำได้** สามารถนำ Type Aliases ไปใช้งานในส่วนอื่นๆ ได้เพื่อลดการเขียนโค้ดที่ซ้ำซ้อนและช่วยจัดกลุ่มประเภทข้อมูลที่มีความเกี่ยวข้องกัน
- **แก้ไขและปรับแต่งง่าย** เมื่อเปลี่ยนแปลงโครงสร้างของ Type Aliases โค้ดที่มีการเรียกใช้งานอยู่จะถูกอัปเดตอัตโนมัติ ทำให้นักพัฒนาไม่ต้องไล่แก้ไขทีละจุด

Interface

Interface

- Interface คือ กระบวนการสร้างชนิดข้อมูล Object รูปแบบใหม่ สำหรับใช้กำหนดความสัมพันธ์ของ Property ต่างๆ ที่อยู่ภายใน Object
- เนื่องจากการใช้งาน Object นั้นมีความยืดหยุ่นอย่างมาก กล่าวคือ Object สามารถมีรูปร่างเป็นแบบใดก็ได้ มีจำนวน Property และชนิดข้อมูลแบบใดก็ได้ การใช้งาน Interface จะช่วยให้โครงสร้างของ Object มีลักษณะที่เฉพาะเจาะจงมากขึ้น

การสร้าง Interface

โครงสร้างคำสั่ง

```
interface InterfaceName{
    property:type
    property:type
}
```

ตัวอย่าง

```
interface User{
    name:string
    age:number
}
```

การใช้งาน Interface

```
interface User{
    name:string
    age:number
    gender:string
}
```

```
let data:User={
    name:"kongruksiam",
    age:30,
    gender:"ชาย"
}
```

ความแตกต่างของโครงสร้างคำสั่ง

```
interface User{
   name:string
   age:number
   gender:string
}
```

```
type User = {
    name:string
    age:number
    gender:string
}
```

ความแตกต่างของโครงสร้างคำสั่ง

```
interface User{
   name:string
   age:number
   gender:string
}
```

```
type User = {
    name:string
    age:number
    gender:string
}
```

คุณสมบัติ	Type Aliases	Interface
การใช้งาน	ใช้งานร่วมกับชนิดข้อมูลพื้นฐาน , Union Type , Literal Type เป็นหลัก	ใช้สำหรับกำหนดโครงสร้างของ Object และ Class เป็นหลัก
การประกาศซ้ำและรวมโครงสร้าง (Merging)	ไม่สามารถรวม Type ที่มีชื่อเดียวกันได้	สามารถรวม Interface ที่มีชื่อเดียวกันได้ เพื่อขยายขอบเขตของ Property
การขยาย/ต่อยอด (Extend)	ใช้ Intersection (&) เพื่อรวม type	ใช้ extends เพื่อสืบทอดคุณสมบัติ
ความยืดหยุ่น	มีความยืดหยุ่นสูง ใช้ได้กับข้อมูลทุกชนิด (ชนิดข้อมูลพื้นฐาน , Union)	จำกัดการใช้งาน เฉพาะ Object และ Class
เหมาะสำหรับ	งานที่มีโครงสร้างซับซ้อนและต้องการความยืดหยุ่นในการใช้งาน	งานที่มีการออกแบบโครงสร้าง Object ที่มีการขยายและนำไปพัฒนาต่อยอดในอนาคต

Merging

- Merging ใน TypeScript หมายถึง การประกาศโครงสร้างของ Interface ที่มีชื่อเดียวกันหลายครั้ง แล้ว TypeScript จะทำการผสาน / รวม (merge) เข้าด้วยกันให้โดยอัตโนมัติ

ตัวอย่าง Merging

```ts
interface User { 1
  id: number
  name: string
}
// ประกาศชื่อซ้ำได้
interface User { 2
  isAdmin: boolean
}
const data : User = { 3
  id: 1,
  name: "Alice",
  isAdmin: true, // รวม property
};
```

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

interface User {
  name: number  // เปลี่ยนชนิดข้อมูลไม่ได้
}
```

Literal Types

Literal Types

คือ รูปแบบการสร้าง Type ขึ้นมาใช้งานพร้อมกำหนดค่าที่แน่นอนเฉพาะเจาะจงไว้ภายใน เพื่อต้องการเก็บข้อมูลตามขอบเขตที่กำหนดไว้เท่านั้น โดยใช้งานร่วมกับเครื่องหมาย | (pipe)

โครงสร้างคำสั่ง

- let variableName : Value1 | Value2 | ValueN *(เหมือน Union Type แต่เก็บค่าคงที่)*
- type TypeName = Value1 | Value2 | ValueN

ตัวอย่าง Literal Types

ตัวอย่าง

```
let role : "Admin"|"Manager" //มี 2 ตัวเลือกให้ใช้งาน
```

```
role = "Admin"   ✅
role = "Manager" ✅
role = "Editor"  ❌
```

Literal Types + Type Aliases

ตัวอย่างเพิ่มเติม

```
type Status = "pending" | "approved" | "rejected";
type Theme = "light" | "dark";

let mytheme:Theme="light"
let mystatus:Status="pending"
```

Union Types	Literal Types
การรวมหลายชนิดข้อมูล (Data Type) เข้าด้วยกัน เช่น let age : string | number	การรวมกลุ่มค่าข้อมูล (value) ที่ชัดเจน เฉพาะเจาะจงเข้าด้วยกัน เช่น let confirm : "yes" | "no"
เก็บได้หลากหลายชนิด	เก็บข้อมูลได้หลายค่า แต่จำกัดให้เลือกเฉพาะค่าที่กำหนดไว้เท่านั้น
มีความยืดหยุ่นสูง	มีความเข้มงวดโดยบังคับให้เลือกเฉพาะ ข้อมูลตามขอบเขตที่กำหนดไว้

Type Guards

Type Guards

- คือกระบวนการที่ช่วยให้ TypeScript สามารถตรวจสอบและระบุชนิดข้อมูลได้แม่นยำขึ้นในช่วงเวลาที่โปรแกรมกำลังทำงาน
- ในบางครั้งเราสร้างตัวแปรแบบ Union Type เช่น string | number ทำให้ตัวแปรมีค่าได้หลายค่า (เป็นได้ทั้งข้อความและตัวเลข) แต่ TypeScript จะไม่รู้เลยว่าในตอนนั้นข้อมูลปัจจุบันที่อยู่ในตัวแปรเป็นชนิดใด

Type Guards

- การใช้งาน Type Guard จะเป็นการตรวจสอบชนิดข้อมูลเพื่อบอกว่าค่าข้อมูลปัจจุบันเป็นชนิดใด เมื่อทราบชนิดข้อมูลที่ชัดเจนแล้วก็สามารถเรียกใช้งาน Property หรือ Method ต่างๆที่เกี่ยวข้องกับชนิดข้อมูลนั้นๆได้เลย เช่น
  - ถ้าเป็น number ก็สามารถใช้ Property & Method จัดการตัวเลขได้
  - ถ้าเป็น String ก็สามารถใช้ Property & Method จัดการข้อความได้

ตัวอย่าง TypeOf

```
function convert(value: string | number) {
    if (typeof value === "string") {
        //แปลงเป็นตัวอักษรพิมพ์ใหญ่
        console.log(value.toUpperCase());
    } else {
        //แปลงเป็นตัวเลขทศนิยม  2 ตำแหน่ง
        console.log(value.toFixed(2));
    }
}
```

Generic

Generic

คือ กระบวนการจัดการชนิดข้อมูลในภาษา TypeScript ที่ช่วยให้นักพัฒนาสามารถจัดการชนิดข้อมูลต่างๆ ได้ โดยไม่ต้องเขียนโค้ดซ้ำซ้อนทำให้โค้ดมีความยืดหยุ่นและสามารถนำกลับมาใช้ซ้ำได้ง่ายขึ้น เหมาะสำหรับนำไปสร้างไลบรารี่หรือ Utility Functions ที่รองรับการทำงานกับข้อมูลที่มีความหลากหลาย

ไม่ใช้ Generic

```
function getItemString(arr: string[]){
    return arr[0];
}
function getItemNumber(arr: number[]){
    return arr[0];
}
function getItemBoolean(arr: boolean[]){
    return arr[0];
}
//เรียกใช้งาน
getItemString(["hello", "world"]);
getItemNumber([1, 2, 3]);
getItemBoolean([true, false]);
```

```
function getItemString(arr: string[]){
    return arr[0]; 1
}
function getItemNumber(arr: number[]){
    return arr[0]; 2
}
function getItemBoolean(arr: boolean[]){
    return arr[0]; 3
}
//เรียกใช้งาน
getItemString(["hello", "world"]);
getItemNumber([1, 2, 3]);
getItemBoolean([true, false]);
```

เขียนโค้ดซ้ำซ้อนหลายจุด

แต่ละฟังก์ชั่นต้อง
กำหนดชนิดข้อมูลแบบตายตัว
เพื่อรองรับการทำงาน

ใช้ Generic

```ts
function getItem<T>(arr: T[]){
    return arr[0];
}

//เรียกใช้งาน
getItem(["hello", "world"]);
getItem([1, 2, 3]);
getItem([true, false]);
```

ใช้ Generic

```
function getItem<T>(arr: T[]){
    return arr[0];
}

//เรียกใช้งาน
getItem(["hello", "world"]);
getItem([1, 2, 3]);
getItem([true, false]);
```

ระบุว่าเป็น Generic โดยให้ T เป็นตัวแทนของชนิดข้อมูลที่เราสนใจ

<T> คืออะไร , ทำไมต้องใช้ T

เราสามารถใช้ตัวอักษรอื่นแทน T ได้ เช่น X , Y , Z เนื่องจากตัวอักษรดังกล่าวเปรียบเสมือนกับการนิยามตัวแปรขึ้นมาใช้งาน เพื่อเป็นตัวแทนของชนิดข้อมูลหรือคลาสที่เราสนใจเท่านั้น เช่น

ตัวอย่างตัวอักษรที่นิยมใช้งาน

T - Type (ชนิดข้อมูล)

E - Element (สมาชิก)

K - Key (คีย์)

V - Value (ข้อมูล)

ใช้ Generic

```
function getItem<T>(arr: T[]){
    return arr[0];
}

//เรียกใช้งาน
getItem(["hello", "world"]);
getItem([1, 2, 3]);
getItem([true, false]);
```

ระบุว่าเป็น Generic โดยให้ T เป็นตัวแทนของชนิดข้อมูลที่เราสนใจ

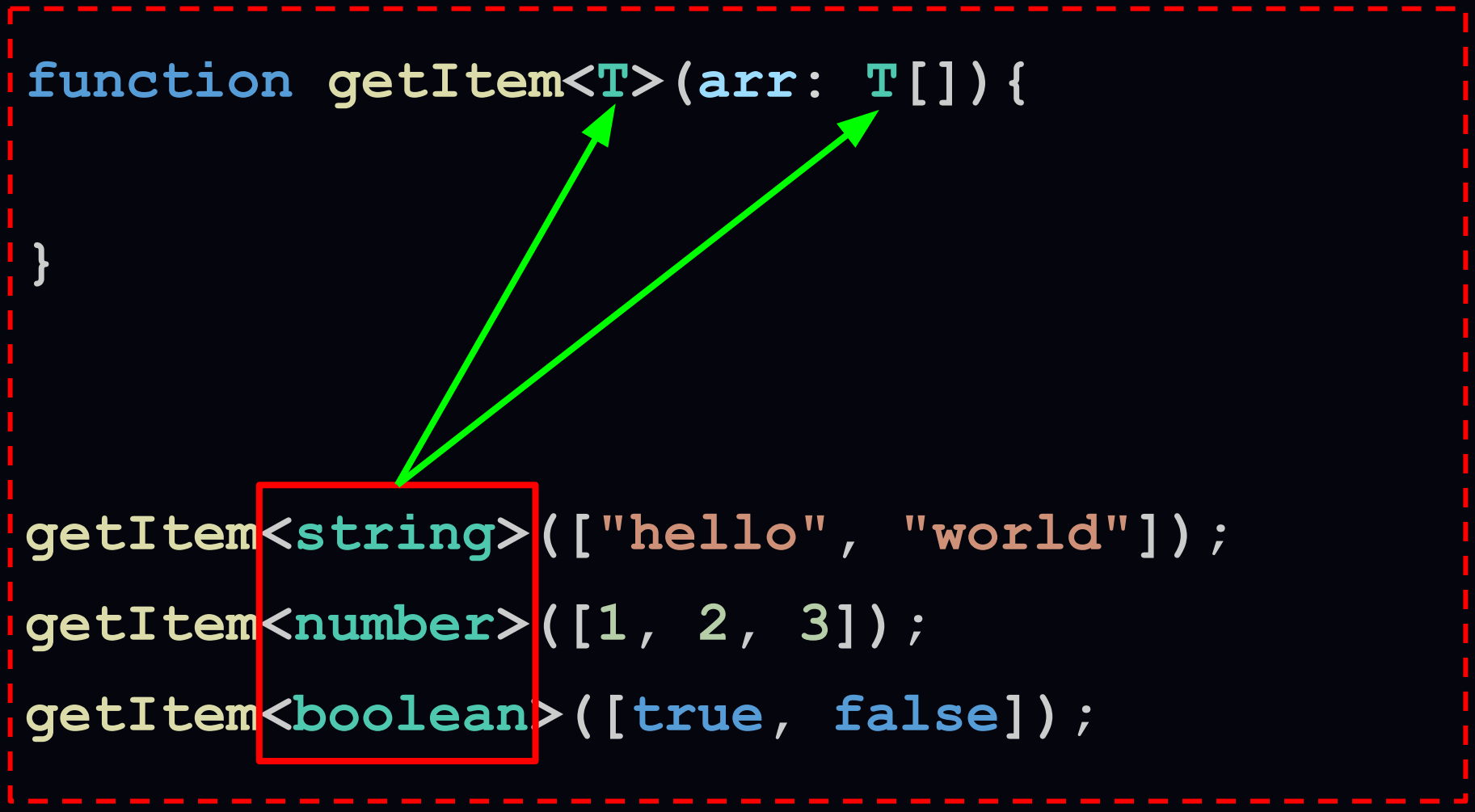
ใช้ Generic

```
function getItem<T>(arr: T[]){
    return arr[0];
}

//เรียกใช้งาน
getItem<string>(["hello", "world"]);
getItem<number>([1, 2, 3]);
getItem<boolean>([true, false]);
```

```ts
function getItem<T>(arr: T[]){
    return arr[0];
}

//เรียกใช้งาน
getItem<string>(["hello", "world"]);
getItem<number>([1, 2, 3]);
getItem<boolean>([true, false]);
```

1

เขียนโค้ดแค่จุดเดียว
แต่ใช้งานได้หลากหลาย


function getItem<T>(arr: T[]){
}
getItem<string>(["hello", "world"]);
getItem<number>([1, 2, 3]);
getItem<boolean>([true, false]);

```
getItem<string>(["hello", "world"]);

getItem<number>([1, 2, 3]);

getItem<boolean>([true, false]);
```

เมื่อส่งข้อมูลเข้าไปทำงานในฟังก์ชั่น ต้องอ้างอิงตามชนิดข้อมูลที่ระบุใน <>

```
getItem<string>(["hello", "world"]);

getItem<number>([1, 2,"สาม"]);

getItem<boolean>([true, false]);
```

เมื่อส่งข้อมูลเข้าไปทำงานในฟังก์ชั่น ต้องอ้างอิงตามชนิดข้อมูลที่ระบุใน <>

สร้างโปรเจกต์
React & TypeScript

รู้จักกับ Vite

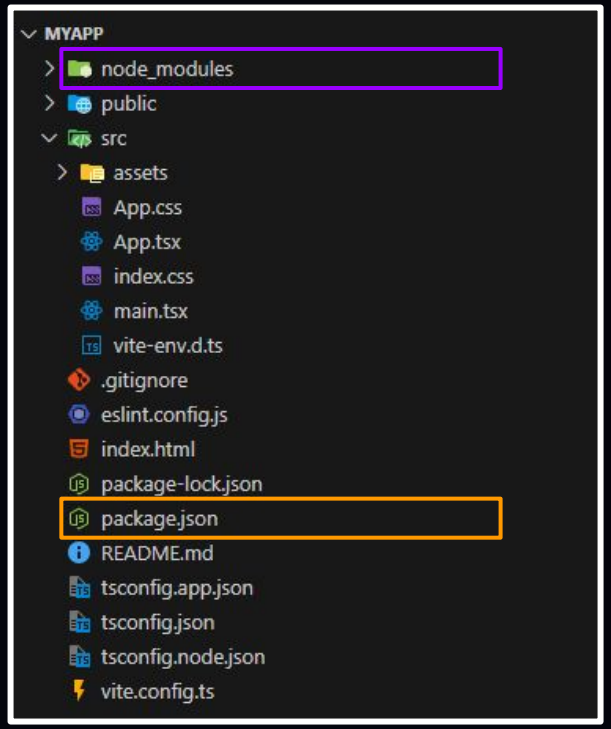
- Vite คือ เครื่องมือสำหรับการพัฒนาเว็บไซต์สมัยใหม่มีจุดเด่นในเรื่องการ Run & Build แอพพลิเคชั่นได้อย่างรวดเร็ว อีกทั้งยังสามารถปรับแต่งได้ง่าย เหมาะสำหรับนำมาใช้ในการพัฒนาเว็บด้วย React , Vue และอื่นๆ

สร้างโปรเจกต์ React & TypeScript

- npm create vite@latest <ชื่อโปรเจกต์>
- เลือก react -> typescript
- cd <ชื่อโปรเจกต์>
- npm install & npm run dev

โครงสร้างโปรเจกต์

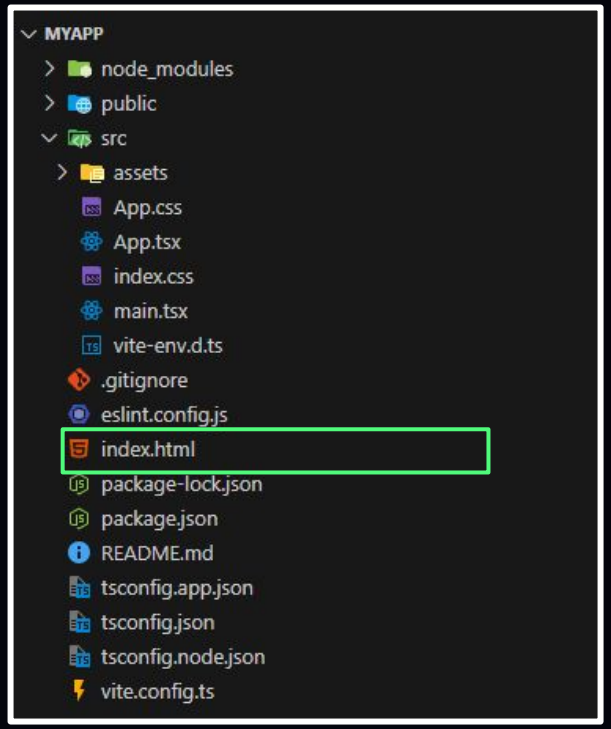
โครงสร้างโปรเจกต์



- node_modules คือ โฟลเดอร์สำหรับจัดเก็บโมดูลหรือไลบรารี่ที่จะนำมาใช้งานในโปรเจกต์

- package.json คือ ไฟล์ที่เก็บข้อมูลพื้นฐานต่างๆ เกี่ยวกับโปรเจกต์ (ชื่อ , เวอร์ชั่น) รวมถึง package หรือไลบราลี่ที่จะนำมาใช้งานในโปรเจกต์

โครงสร้างโปรเจกต์



- ไฟล์ index.html เป็นไฟล์ HTML สำหรับแสดงผลลัพธ์ใน Browser โดยเนื้อหาที่นำมาแสดงผลนั้นมาจากคอมโพเนนต์ (Components)

- สิ่งที่เราสนใจในไฟล์ index.html คือคำสั่งที่อยู่ใน <body> พื้นที่ และ <script>...</script> เป็นการนำไฟล์ main.tsx เข้ามาใช้งานเพื่อดึงเนื้อหาที่อยู่ใน App.tsx *(คอมโพเนนต์เริ่มต้น)* มาแสดงผลในพื้นที่ดังกล่าว

```html
<body>
  <div id="root"></div>
  <script type="module" src="/src/main.tsx"></script>
</body>
```

index.html

src > main.tsx

```tsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.tsx'

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

main.tsx ทำการเชื่อม App Component (App.tsx) เข้ากับ index.html

- สิ่งที่เราสนใจในไฟล์ index.html คือคำสั่งที่อยู่ใน <body> พื้นที่ และ <script>...</script> เป็นการนำไฟล์ main.tsx เข้ามาใช้งานเพื่อดึงเนื้อหาที่อยู่ใน App.tsx *(คอมโพเนนต์เริ่มต้น)* มาแสดงผลในพื้นที่ดังกล่าว

```html
<body>
  <div id="root"></div>
  <script type="module" src="/src/main.tsx"></script>
</body>
```

1

index.html

src > main.tsx

```tsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.tsx'

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>
)
```

2

3

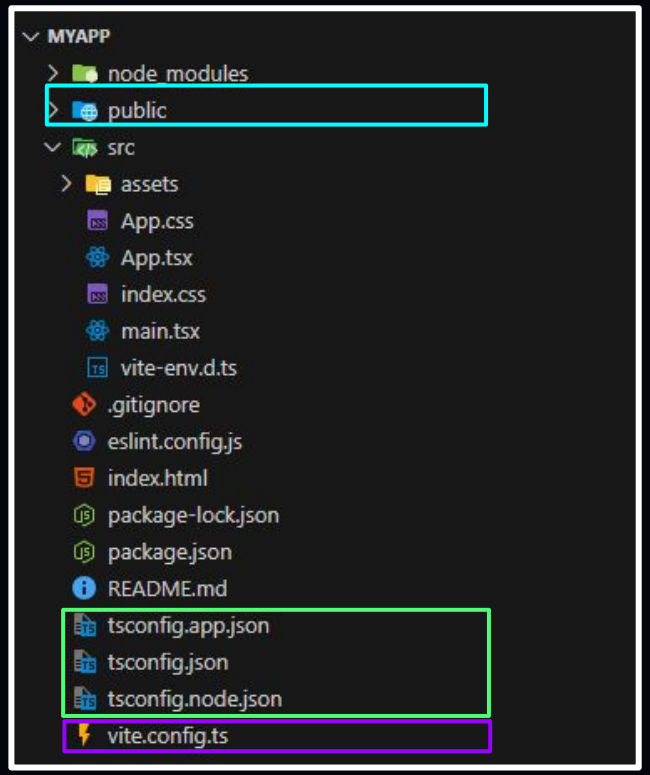
main.tsx ทำการเชื่อม App Component (App.tsx) เข้ากับ index.html

โครงสร้างโปรเจกต์

src คือ โฟลเดอร์ที่เก็บไฟล์คอมโพเนนต์หรือโครงสร้างหลักของแอพพลิเคชั่น ซึ่งประกอบด้วยไฟล์และโฟลเดอร์สำคัญ ดังนี้

- App.css เป็นไฟล์ที่เก็บโค้ด CSS สำหรับนำไปใช้งานในคอมโพเนนต์เริ่มต้น
- App.tsx เป็นไฟล์คอมโพเนนต์เริ่มต้นของแอพพลิเคชั่น
- main.tsx คือไฟล์หน้าแรกของแอพพลิเคชั่น *(เชื่อมโยงกับไฟล์ index.html)*
- index.css คือไฟล์ CSS ที่ใช้งานในโปรเจกต์ *(ใช้คู่กับไฟล์ main.tsx)*
- assets คือ โฟลเดอร์ที่จัดเก็บไฟล์ต่างๆสำหรับนำมาใช้งานในคอมโพเนนต์

โครงสร้างโปรเจกต์



- public คือ โฟลเดอร์ที่จัดเก็บไฟล์ต่างๆที่จะนำใช้งานในโปรเจกต์ เช่น รูปภาพ เสียง วิดีโอ
- tsconfig.json คือ ไฟล์สำหรับเก็บการตั้งค่าการใช้งาน TypeScript ในโปรเจกต์ โดยเชื่อมโยงระหว่างไฟล์ tsconfig.app.json และไฟล์ tsconfig.node.json
- vite.config.ts คือ ไฟล์สำหรับตั้งค่าการทำงานเพิ่มเติมของ Vite เช่น Plugin ต่างๆ เป็นต้น

tsconfig.json

```
{
  "files": [],
  "references": [
    { "path": "./tsconfig.app.json" },
    { "path": "./tsconfig.node.json" }
  ]
}
```

tsconfig.app.json (ใช้กับ React)

```
{
  "compilerOptions": {
    "tsBuildInfoFile": "./node_modules/.tmp/tsconfig.app.tsbuildinfo",
    "target": "ES2022",
    "useDefineForClassFields": true,
    "lib": ["ES2022", "DOM", "DOM.Iterable"],
    "module": "ESNext",
    "jsx": "react-jsx",
    …
  },
  "include": ["src"]
}
```

tsconfig.app.json (ใช้กับ React)

```
{
  "compilerOptions": {
    "tsBuildInfoFile": "./node_modules/.tmp/tsconfig.app.tsbuildinfo",
    "target": "ES2022",
    "useDefineForClassFields": true,
    "lib": ["ES2022", "DOM", "DOM.Iterable"],
    "module": "ESNext",
    "jsx": "react-jsx",
    …
  },
  "include": ["src"]
}
```

นำทุกไฟล์ในโฟลเดอร์ src มาประมวลผล
(ตรวจสอบและคอมไพล์)

หน้าที่ของ tsconfig.app.json

- ไฟล์ที่เก็บการตั้งค่า TypeScript เพื่อนำไปใช้งานร่วมกับโค้ดฝั่ง Client (React Application) หรือโค้ดที่จะนำไปรันบน Browser

- โดยส่วนใหญ่เป็นการตั้งค่าให้ TypeScript เข้าใจโค้ดของ React (โฟลเดอร์ src) และคอมไพล์ไปเป็นโค้ดที่ Browser เข้าใจ เพื่อให้สามารถรันแอพพลิเคชั่นได้

tsconfig.node.json (ใช้กับ Node.js)

```jsonc
{
  "compilerOptions": {
    "tsBuildInfoFile": "./node_modules/.tmp/tsconfig.node.tsbuildinfo",
    "target": "ES2023",
    "lib": ["ES2023"],
    "module": "ESNext",
    "skipLibCheck": true,
    …
  },
  "include": ["vite.config.ts"]
}
```

หน้าที่ของ tsconfig.node.json

- ไฟล์ที่เก็บการตั้งค่า TypeScript เพื่อนำไปใช้งานร่วมกับ Node.js เป็นหลัก เช่น การรันคำสั่งหรือ Script ที่ทำงานอยู่บน Node.js Environment (Build , Deploy, Config ต่างๆ) รวมถึงการตั้งค่าเกี่ยวกับ Build Tools เช่น ไฟล์ตั้งค่าของ Vite (vite.config.ts)

State & UseState

รู้จักกับ State

State คือ *ข้อมูลที่ถูกเก็บไว้ในคอมโพเนนต์* สามารถเปลี่ยนแปลงได้ตามการกระทำของผู้ใช้งาน เช่น สถานะการล็อกอิน ข้อมูลที่กรอกในฟอร์ม จำนวนครั้งที่กดปุ่ม เป็นต้น

เมื่อมีการเปลี่ยนค่าข้อมูลใน State ก็จะส่งผลให้ คอมโพเนนต์ที่เป็นเจ้าของ State นั้นอัปเดตหรือแสดงผลหน้าเว็บใหม่ทันที *(re-render)*

โครงสร้างคำสั่ง

import {useState} from ‘react’

- [ชื่อ State , ฟังก์ชั่นเปลี่ยนแปลงข้อมูลใน State] = useState(ค่าเริ่มต้น)
- [ชื่อ State , ฟังก์ชั่นเปลี่ยนแปลงข้อมูลใน State] = useState<type>(ค่าเริ่มต้น)

จะได้ Array ที่ Destructuring จาก useState

ตัวอย่างการสร้าง State

```
const [name, setName] = useState("ก้องรักสยาม ");
```

- *useState* มีการส่งค่ากลับมาเป็นอาร์เรย์ที่มีข้อมูล 2 จำนวนและใช้วิธีแยกข้อมูลดังกล่าวด้วย Arrays Destructing ประกอบด้วย
  - name คือ ตัวแปรที่เก็บค่าสถานะปัจจุบัน (ค่าเริ่มต้น คือ ก้องรักสยาม)
  - setName ฟังก์ชั่นที่ใช้สำหรับอัปเดตหรือเปลี่ยนแปลงข้อมูลในตัวแปร name

ตัวอย่างการสร้าง State

```
const [name, setName] = useState("ก้องรักสยาม ");

const [salary, setSalary] = useState(30000);

const [isVisible, setIsvisible] = useState(true);
```

ตัวอย่างการสร้าง State

```tsx
const [name, setName] = useState<string>("ก้องรักสยาม ");

const [salary, setSalary] = useState<number>(30000);

const [isVisible, setIsvisible] = useState<boolean>(true);
```

ระบุ Generic เพื่อบังคับให้เก็บข้อมูลลงใน State โดยอ้างอิงตามชนิดข้อมูลที่กำหนดไว้ใน <>

ตัวอย่างการสร้าง State

```
const [name, setName] = useState<string>("ก้องรักสยาม ");
const [salary, setSalary] = useState<number>("สามหมื่น ");
const [isVisible, setIsvisible] = useState<boolean>(true);
```

ระบุ Generic เพื่อบังคับให้เก็บข้อมูลลงใน State โดยอ้างอิงตามชนิดข้อมูลที่กำหนดไว้ใน <>

การสร้าง State (Object)

ออกแบบโครงสร้าง Object

```
interface Employee {
   id: number
   name: string
   salary: number
}
```

ตัวอย่างการสร้าง State (Object)

```
const [person, setPerson] = useState<Employee>({
    id: 1,
    name: "kong",
    salary: 30000,
});
```

ตัวอย่างการสร้าง State (Object)

```
const [person, setPerson] = useState<Employee>({
    id: 1,
    name: "kong",
    salary: 30000,
});
```

นำโครงสร้างที่ระบุใน Interface Employee มาใช้งาน

การสร้าง State (Array)

การสร้าง State (Array)

```
const [data, setData] = useState<Employee[]>([
  { id: 1, name: "ก้องรักสยาม ", salary: 50000 },
  { id: 2, name: "ลูกน้ำ", salary: 25000 },
  { id: 3, name: "ชาลี", salary: 30000 },
  …
]);
```

การสร้าง State (Array)

```
const [data, setData] = useState<Employee[]>([
  { id: 1, name: "ก้องรักสยาม ", salary: 50000 },
  { id: 2, name: "ลูกน้ำ", salary: 25000 },
  { id: 3, name: "ชาลี", salary: 30000 },
  ...
]);
```

สร้างอาร์เรย์สำหรับเก็บกลุ่ม
Object ของ Employee

การสร้าง State (Array)

```
<ul>
      {data.map((item) => (
         <li key={item.id}>
           ชื่อ {item.name} , เงินเดือน {item.salary} บาท
         </li>
      ))}
</ul>
```

การสร้าง State (Array)

```
<ul>
     {data.map((item) => (
        <li key={item.id}>
          ชื่อ {item.name} , เงินเดือน {item.salary} บาท
        </li>
     ))}
</ul>
```

รู้จักกับคีย์ (Key)

- Keys หมายถึง Property พิเศษที่ทำงานอยู่ใน React *โดย Keys จะมีค่าที่ไม่ซ้ำกัน* ถูกกำหนดขึ้นมาเพื่อให้ React สามารถแยกแยะและตรวจสอบได้ว่ามีคอมโพเนนต์ใดบ้างที่มีการเปลี่ยนแปลงการทำงาน เช่น เพิ่ม ลบ แก้ไขหรือสลับตำแหน่ง ทำให้สามารถจัดการการอัปเดต DOM ได้อย่างมีประสิทธิภาพ *(ส่วนใหญ่นำมาใช้งานกับ List ของ Component)*

รู้จักกับคีย์ (Key)

- React จะทำการ Update และ Render เฉพาะคอมโพเนนต์ที่เปลี่ยนแปลงเท่านั้น *(ไม่ได้ Render ใหม่ทั้งหมด)* ส่งผลให้ *สถานะ (State)* ของคอมโพเนนต์แต่ละตัวที่อยู่ในรายการนั้นๆ ไม่สูญหายในระหว่างที่มีการเปลี่ยนแปลงการทำงาน

การสร้าง Component

Functional Component

- สร้างคอมโพเนนต์ในรูปแบบฟังก์ชั่น ซึ่งรองรับการเขียนทั้งในรูปแบบฟังก์ชั่นแบบปกติ (Declaration Function) และ Arrow Function โดยจะต้องกำหนดให้ตัวอักษรตัวแรกของชื่อฟังก์ชั่นเป็นตัวพิมพ์ใหญ่เสมอ

Functional Component

ฟังก์ชั่นแบบปกติ (1)

```
function Header(){
    return(
        <>
            <p>Hello React & TypeScript</p>
        </>
    );
}
export default Header
```

Functional Component

ฟังก์ชั่นแบบปกติ (2)

```
export default function Header(){
    return(
        <>
            <p>Hello React & TypeScript</p>
        </>
    );
}
```

Functional Component

Arrow Function

```
const Header=()=>{
    return(
        <>
            <p>Hello React & TypeScript</p>
        </>
    );
}
export default Header;
```

External Component

- การสร้างคอมโพเนนต์ในรูปแบบแยกเป็นไฟล์ด้านนอก โดยมีนามสกุล .tsx แล้ว export ออกมาเพื่อให้สามารถนำโครงสร้างคอมโพเนนต์ในไฟล์ดังกล่าวไปใช้งานในคอมโพเนนต์อื่นๆได้

External Component

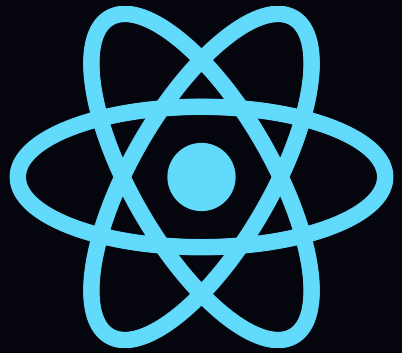
ตัวอย่าง : src/components/Header.tsx

```
export default function Header(){

. . .

}
```

เรียกใช้งานด้วยคำสั่ง import (src/App.tsx)

```
import Header from "./components/Header";
```

ขั้นตอนการสร้าง Component



- สร้างไฟล์คอมโพเนนต์ (.tsx) และจัดเก็บลงในโฟลเดอร์ components เพื่อให้ง่ายต่อการจัดการ
- ออกแบบโครงสร้างการทำงานของ Component และ export เพื่อให้ไฟล์อื่นสามารถนำไปใช้งานได้
- เรียกใช้งาน Component ผ่านคำสั่ง import

Props

รู้จักกับ Props (Properties)

- Props คือ การส่งข้อมูลเข้าไปทำงานในคอมโพเนนต์ โดยข้อมูลดังกล่าวจะถูกส่งจากคอมโพเนนต์ที่อยู่สูงกว่า (Parent Component) ไปยังคอมโพเนนต์ที่อยู่ต่ำกว่า (Child Component)

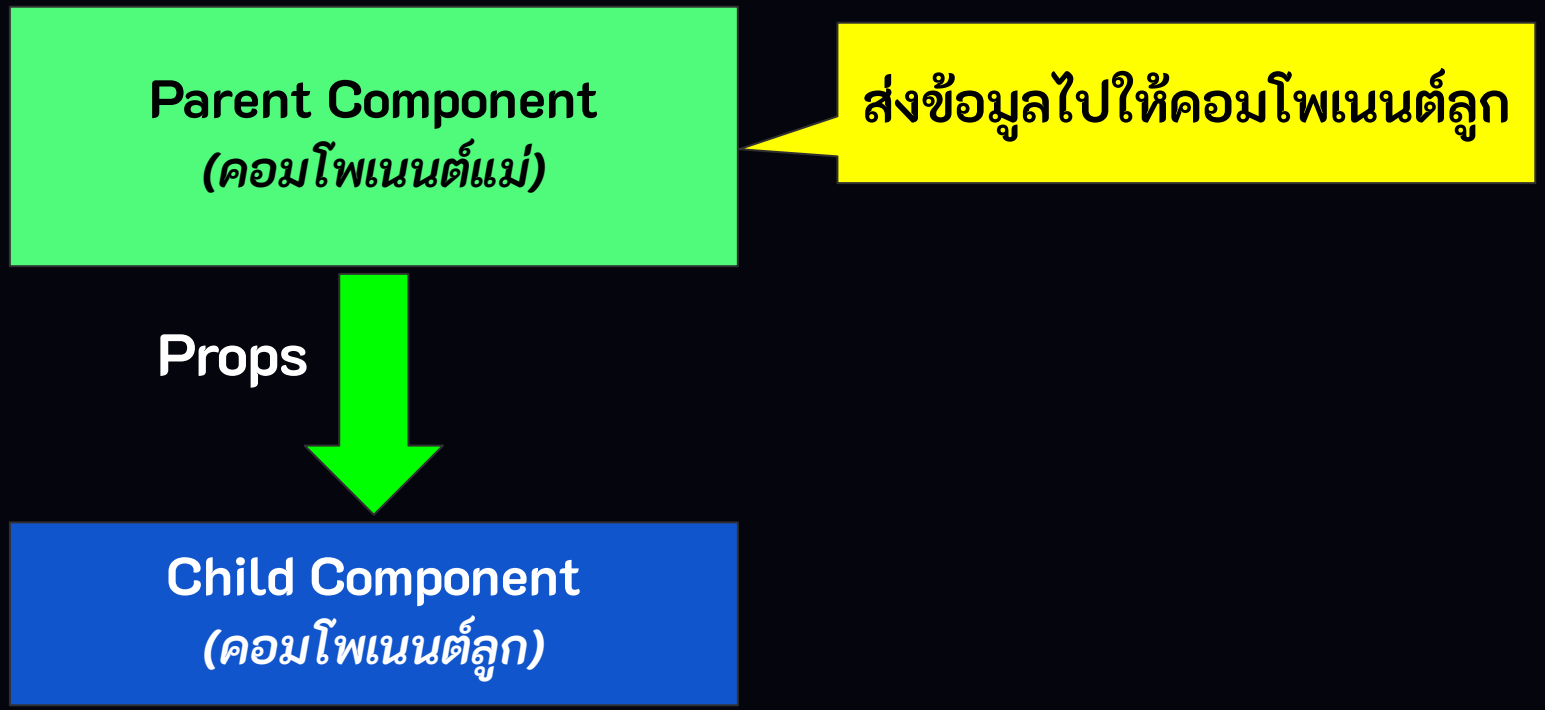
การทำงานของ Props

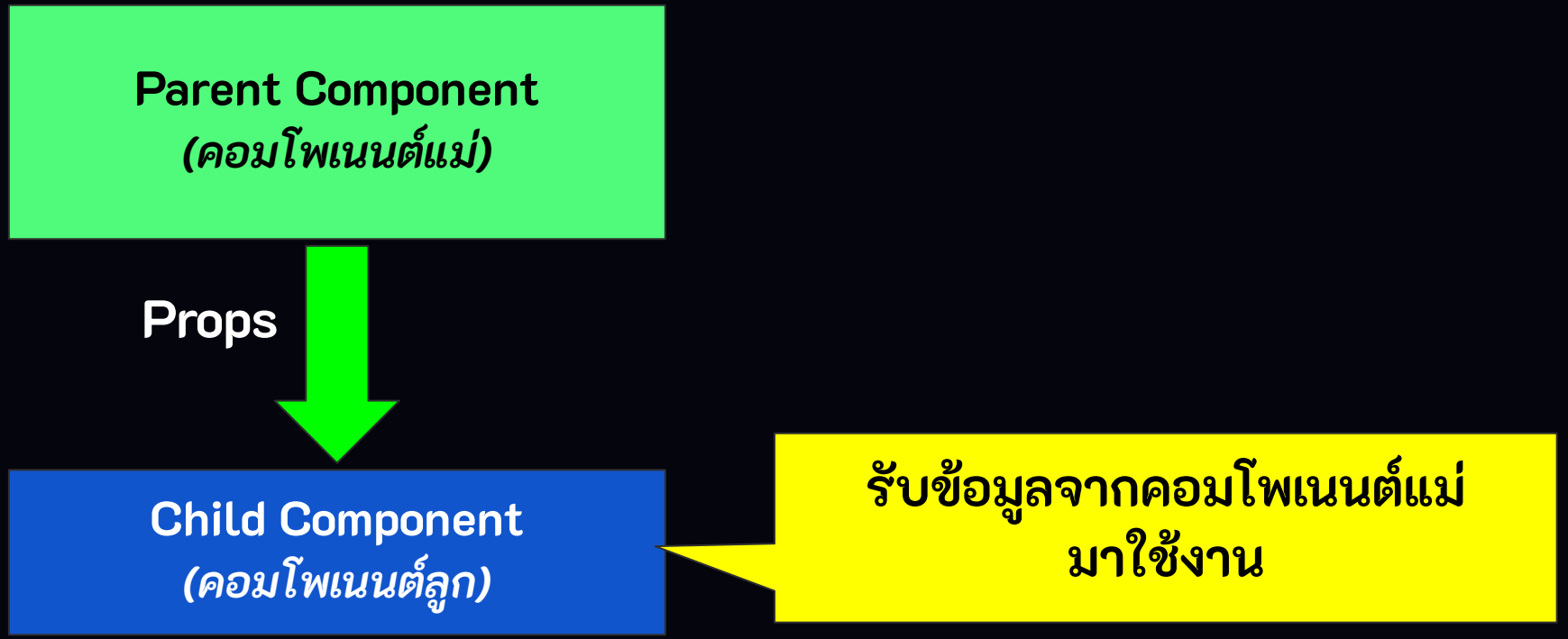
การส่งค่า Props จาก Parent Component ไปยัง Child Component

มีวัตถุประสงค์ดังนี้

- กำหนดข้อมูลสำหรับแสดงเนื้อหาใน Child Component
- กำหนดความสามารถบางอย่างให้กับ Child Component

ในการติดต่อสื่อสารแลกเปลี่ยนข้อมูลกันระหว่างคอมโพเนนต์


Parent Component
(คอมโพเนนต์แม่)
ส่งข้อมูลไปให้คอมโพเนนต์ลูก
Props
Child Component
(คอมโพเนนต์ลูก)


Parent Component
(คอมโพเนนต์แม่)
Props
Child Component
(คอมโพเนนต์ลูก)
รับข้อมูลจากคอมโพเนนต์แม่
มาใช้งาน

การสร้าง Props

Props แบบค่าเดียว

<ชื่อคอมโพเนนต์ *ชื่อพร็อพ =ค่าที่กำหนดให้พร็อพ*/>

Props แบบหลายค่า

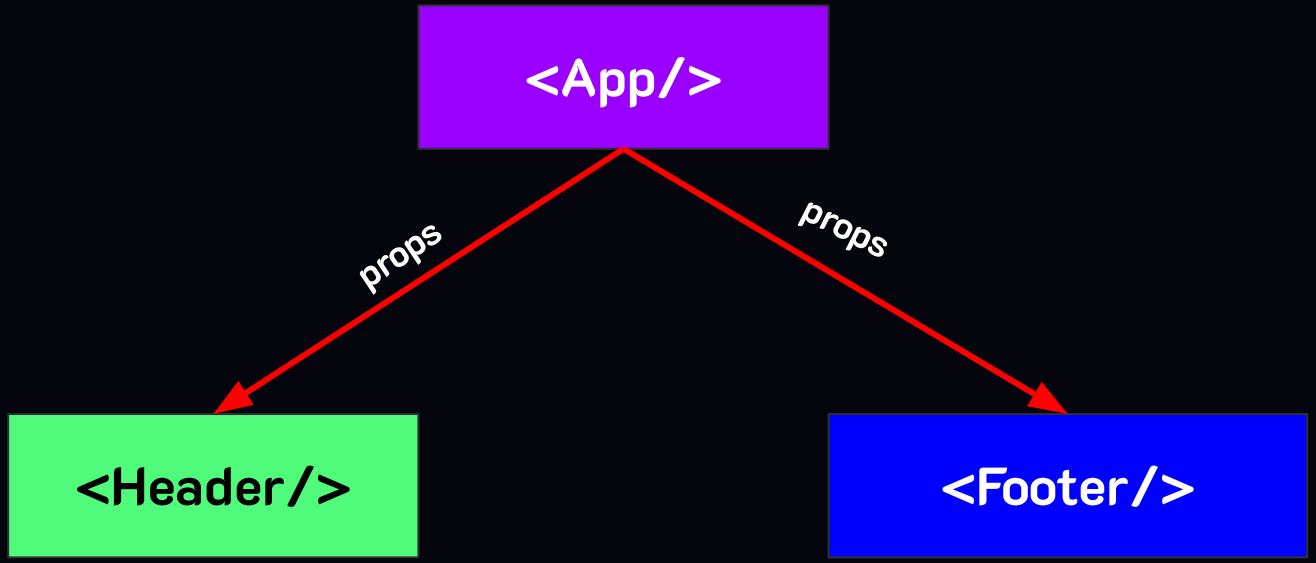
<ชื่อคอมโพเนนต์ *ชื่อพร็อพ =ค่าที่1 ชื่อพร็อพ =ค่าที่2* />

Props สามารถกำหนดค่าได้หลากหลายชนิด

- string : <Component props="หน้าแรก" />
- number : <Component props={5} />
- boolean : <Component props={true} />
- array : <Component props={['item1', 'item2']} />
- object : <Component props={{name: 'Kong', age: 25}} />
- function : <Component props={() => console.log('คลิก')} />

Props สามารถกำหนดค่าได้หลากหลายชนิด

- string : <Component title="หน้าแรก" />
- number : <Component count={5} />
- boolean : <Component isAdmin={true} />
- array : <Component items={['item1', 'item2']} />
- object : <Component user={{name: 'Kong', age: 25}} />
- function : <Component onClick={() => console.log('คลิก')} />



App.tsx (คอมโพเนนต์แม่)

```
<Header title="แอพจัดการข้อมูลพนักงาน "/>
<Footer company="ก้องรักสยาม " year={2025}/>
```

Header.tsx (คอมโพเนนต์ลูก)

```
const Header=(props)=>{
    return(
        <>
            <h1>{props.title}</h1>
        </>
    );
}
```

Header.tsx (คอมโพเนนต์ลูก)

```
const Header=(props)=>{
    return(
        <>
            <h1>{props.title}</h1>
        </>
    );
}
```

Props มีชนิดข้อมูลเป็น Object

Header.tsx (คอมโพเนนต์ลูก)

```tsx
interface HeaderProps{
    title:string
}
const Header=(props:HeaderProps)=>{
    return(
        <>
            <h1>{props.title}</h1>
        </>
    );
}
```

Header.tsx (คอมโพเนนต์ลูก)

```tsx
interface HeaderProps{
    title:string
}
const Header=(props:HeaderProps)=>{
    return(
        <>
            <h1>{props.title}</h1>
        </>
    );
}
```

Header.tsx (คอมโพเนนต์ลูก)

แบบไม่ใช้ Destructuring

```tsx
interface HeaderProps{
    title:string
}
const Header=(props:HeaderProps)=>{
    return(
        <>
            <h1>{props.title}</h1>
        </>
    );
}
```

Header.tsx (คอมโพเนนต์ลูก)

แบบใช้ Destructuring

```tsx
interface HeaderProps{
    title:string
}
const Header=({title}:HeaderProps)=>{
    return(
        <>
            <h1>{title}</h1>
        </>
    );
}
```

Footer.tsx (คอมโพเนนต์ลูก)

แบบไม่ใช้ Destructuring

```tsx
interface FooterProps{
    company:string
    year:number
}
function Footer(props:FooterProps){
    return(
        <>
            บริษัท {props.company} จำกัด | ปีที่ก่อตั้ง : {props.year}
        </>
    );
}
```

Footer.tsx (คอมโพเนนต์ลูก)

แบบใช้ Destructuring

```tsx
interface FooterProps{
    company:string
    year:number
}
function Footer({company,year}:FooterProps){
    return(
        <>
            บริษัท {company} จำกัด | ปีที่ก่อตั้ง : {year}
        </>
    );
}
```

Complete Course