**리액트와 타입스크립트**

**자바스크립트 동적 타입의 문제점**

**런타임 에러 발생 가능성**

실행하는 동안에만 타입 오류를 발견

실행 중에 예상치 못한 타입 관련 오류가 발생

**타입 불일치 문제**

함수에 전달되는 인자의 타입이 예상과 다를 경우, 예기치 않은 동작 발생

디버깅을 어렵게 만들고, 프로그램의 예측 가능성을 저하 시킴

**유지보수 어려움**

프로젝트 커지면서 함수나 변수 사용처 파악 어려움

타입이 명시적이지 않을 경우, 프로젝트를 이해하고 변경하는 것이 어려움

**타입스크립트 개요**

자바스크립트의 상위 집합 언어로, 정적 타입기능 제공하는 언어

Microsoft에서 개발, 타입을 명시적으로 지정

**주요 특징**

**자바스크립트 호환성:** js 코드를 그대로 사용할 수 있고 점진적으로 타입스크립트를 도입

**정적 타입 사용:** 변수, 매개변수, 반환 값 등에 명확한 타입을 지정하여 오류를 사전에 방지

**타입 추론 기능:** 타입 스크립트는 초기화된 변수의 타입을 자동으로 추론하는 기능

**타입스크립트 장점**

**코드 안정성과 신뢰성**

코드 실행전 타입 검사를 수행하여 런타임 중 발생할 수 있는 오류 사전에 방지

타입 불일치로 인한 오류 줄이고 코드의 예측 가능성을 높임

**가독성과 유지 보수성**

명시적인 타입 선언은 코드의 가독성을 높이고, 코드 이해하기 쉬움

**타입 확장성**

* ﻿﻿타입스크립트는 자체 타입 시스템을 활용하여 복잡한 데이터 구조나 비즈니스 로직을 효율적으로 모델링할 수 있습니다.

﻿﻿인터페이스, 제네릭 등의 기능을 통해 타입을 추상화하고 재사용 가능한 코드를 작성할 수 있습니다.

**타입스크립트 단점**

**트랜스파일링 단계 추가**

* ﻿﻿타입스크립트는 자바스크립트로 변환하는 트랜스파일링 단계가 필요합니다.
* ﻿﻿이로 인해 빌드 시간이 길어질 수 있으며, 오류가 발생할 경우 이를 수정해야 합니다.

**복잡성 증가**

* ﻿﻿타입스크립트를 도입하면 초기 학습 곡선이 존재하며, 추가적인 설정과 구성 파일이 필요합니다.
* ﻿﻿타입 정의와 같은 추가적인 코드를 작성해야 하므로, 개발 초기에는 시간이 더 소요될 수 있습니다.

**타입 선언 관리**

* ﻿﻿외부 라이브러리나 모듈을 사용할 때, 해당 라이브러리의 타입 정의 파일이 필요할 수 있습니다.

﻿﻿타입 선언이 잘못되거나 누락된 경우, 이를 별도로 작성하거나 관리해야 하는 부담이 생길 수 있습니다

타입 스크립트 기본 타입

Boolean: 참 또는 거짓

Number: 모든 종류의 숫자

String: 텍스트 데이터

Array: 같은 타입의 값들이 순서대로 저장된 리스트

Tuple: 고정된 수의 요소 각각이 명시된 타입을 가질 수 있는 배열

Enum: 이름이 있는 상수 집합을 정의

Any: 모든 타입의 값을 가짐

Void: 주로 함수에서 반환 값이 업음을 의미

Null/undefined: 기본적으로 모든 타입의 하위 타입

**함수에 타입 사용하기**

const increment = (value: number, step: number = 1): number => {

return value + step;

}

// ( ? ) 선택적 매개변수

const print = (message: string, prefix?: string): void => {

if (prefix) {

console.log(`$(prefix): $(message}`);

}else {

console.log(message);

}

}

**타입 추론**

* ﻿﻿TypeScript는 코드에서 명시적으로 타입을 지정하지 않아도, 변수나 함수의 타입을 자동으로 추론합니다.
* ﻿﻿초기값을 통해 타입을 추론하므로, 명시적인 타입 선언을 줄일 수 있습니다.
* ﻿﻿명확한 타입이 필요한 경우 명시적인 타입 선언을 권장합니다.

﻿﻿복잡한 타입 구조나 외부 라이브러리 사용 시 타입 추론이 어려울 수 있으므로, 필요에 따라 타입을 명시적으로 선언하는 것이 좋습니다

타입 별칭 만들기

* ﻿﻿타입 별칭은 기존 타입에 새로운 이름을 지어주는기능입니다.
* ﻿﻿복잡한 타입이나 반복 사용되는 타입을 간결하게 표현할 수 있습니다.

﻿﻿타입 별칭을 사용함으로써 코드의 가독성을 높이고 중복을 줄일 수 있습니다.

// 타입 별칭 생성 (Type Aliases)

type ID = number;

type Name = string;

type User ={

id: ID;

｝

let userld: ID = 123;

let userName: Name = "Alice":

let user: User = {

id: userld, username: userName,

｝

**유니언 타입 만들기**

* ﻿﻿유니온 타입은 두 개 이상의 타입을 허용하는 타입입니다.

﻿﻿값이 여러 타입 중 하나일 수 있을 때 유용하게 사용됩니다.

// 유니온 타입 (Union Types)

type Status = "active" | "inactive";

let userStatus: Status = "active";

userStatus="blocked": // Error: 'blocked'은 'active'| 'inactive' 중 하나가 아닙니

**인터페이스 만들기**

* ﻿﻿인터페이스는 타입을 정의하는 또 다른 방법입니다.
* ﻿﻿클래스나 객체가 반드시 구현해야 하는 구조를 명시할 수 있습니다.
* ﻿﻿인터페이스는 확장할 수 있습니다. 따라서 한 인터페이스를 여러 곳에서 확장하여 사용할 수 있습니다.

﻿﻿클래스에서 인터페이스를 implements 키워드를 사용하여 구현할 수 있습니다. 이는 객체 지향 프로그래밍에서 인터페이스를 사용하여 다형성을 지원하는 데 유용합니다.

**Types vs Interfaces**

**주요 차이점**

* ﻿﻿**확장성:** 인터페이스는 나중에 새 속성을 추가할 수 있지만, 타입은 생성 후 변경할 수 없습니다.
* ﻿﻿**선언 병합:** 인터페이스는 선언 병합이 가능하지만, 타입 별칭은 불가능합니다.
* ﻿﻿**사용 범위:** 인터페이스는 객체의 형태를 선언하는 데만 사용할 수 있지만, 타입은 원시 타입도 이름을 붙일 수 있습니다.

**공통점**

* ﻿﻿둘 다 객체의 형태를 정의하는 데 사용할 수 있습니다.

﻿﻿대부분의 경우 서로 대체 가능합니다.

**사용 시기**

* **﻿﻿인터페이스 사용**
* ﻿﻿객체의 구조를 정의할 때
* ﻿﻿코드의 다른 부분에서 해당 구조를 확장해야 할 가능성이 있을 때
* ﻿﻿선언 병합이 필요한 경우
* ﻿﻿컴파일 성능을 고려할 때 (extends를 사용한 인터페이스가 일반적으로 더 성능이 좋음)
* ﻿﻿**타입 별칭 사용**
* ﻿﻿유니온 타입, 교차 타입 등 복잡한 타입을 정의할 때
* ﻿﻿튜플이나 함수 타입을 정의할 때
* ﻿﻿원시 타입에 이름을 붙이고 싶을 때
* ﻿﻿인터페이스로 표현할 수 없는 형태를 정의해야 할 때

개인의 선호도에 따라 선택할 수 있으며, 타입스크립트가 필요에 따라 다른 종류의 선언이 필요하다고 알려줄 것입니다. 일반적으로 interface를 사용하다가 type의 기능이 필요할 때 전환하는 것이 좋습니다.

**타입스크립트 고급 활용**

**제네릭 (Generics) 사용하기**

제네릭은 타입스크립트에서 코드의 재사용성을 높이고, 타입 안전성을 유지할 수 있습니다. 제네릭은 함수, 클래스, 인터페이스, 타입 별칭 등에 적용할 수 있으며, 트랜스파일링 타임에 타입이 결정됩니다.

**제네릭 함수 선언**

함수명 뒤에 꺾쇠 괄호(<>)를 사용하여 타입 매개변수를 지정합니다

function identity<T>(arg: T): T {

return arg

}

// 사용 예시

let output1 = identity<string>("Hello");

let output2 = identity<number>(42)

**제네릭 인터페이스 선언**

제네릭 인터페이스는 인터페이스 이름 뒤에 꺾쇠 괄호(<>)를 사용하여 타입 매개변수를 지정합니다.

interface GenericldentityFn<T>{

(arg: T): T;

}

function identitys<T>(arg: T): T{

return arg;

}

let myldentity: GenericldentityFn<number> = identity:

**제네릭 (Generics) 사용하기**

**제네릭 제한 조건**

제네릭 타입 매개변수에 특정 타입을 확장하도록 제약을 둘 수 있습니다

interface Lengthwise {

length: number;

}

function loggingldentity<T extends Lengthwise>(arg: T): T {

console.log(arg.length);

return arg;

}

loggingldentity({ length: 10, value: 3 });// OK

// loggingldentity(3): // Error: number 타입에는 length 프로퍼티가 없음

T 는 Lengthwise 인터페이스를 확장하므로, T는 length 프로퍼티를 가져야 합니다

**타입 가드 (Type Guards)**

타입 가드는 런타임에서 타입을 좁히기 위해 사용됩니다. 조건문을 통해 변수의 타입을 더욱 자세하게 추론할 수 있게 됩니다.

**1. 사용자 정의 타입 가드**

* ﻿﻿개발자가 직접 정의하는 함수를 통해 타입을 좁힙니다.

﻿﻿is 키워드를 사용한 타입 서술어(type predicate)를 반환 타입으로 사용합니다

function isFish(pet: Fish | Bird): pet is Fish {

return (pet as Fish).swim !== undefined;

}

if(isFish(pet)){

pet.swim() // 여기서 pet은 Fish 타입으로 처리됩니다.

}

**타입 가드 (Type Guards)**

**2. in 연산자 타입 가드**

• 객체에 특정 프로퍼티가 존재하는지 확인하여 타입을 좁힙니다.

function move(pet: Fish | Bird) {

if ("swim" in pet) {

return pet.swim0() //기서 pet은 Fish 타입으로 추론됩니다.

}

return pet.fly()// 여기서 pet은 Bird 타입으로 추론됩니다.

}

**3. typeof 타입 가드**

• 자바스크립트의 typeof 연산자를 사용하여 기본 타입을 체크합니다

function padLeft(value: string, padding: string | number) {

if (typeof padding === "number") {

return Array(padding + 1).join("") + value;

}

return padding + value; // 여기서 padding은 string 타입으로 추론됩니다.

}

**4. instanceof 타입 가드**

• 클래스의 인스턴스를 체크하여 타입을 좁힙니다

if (padder instanceof SpaceRepeatingPadder){

padder.numspaces;// padder는 SpaceRepeatingPadder 타입으로 추론됩니다.

}

**타입 가드의 장점과 주의사항**

**타입 가드의 장점**

* ﻿﻿코드의 타입 안전성 향상
* ﻿﻿런타임 오류 감소
* ﻿﻿IDE의 자동 완성 기능 개선
* ﻿﻿코드 가독성 및 유지보수성 향상

**주의사항**

* ﻿﻿과도한 사용은 코드 복잡성을 증가시킬 수 있습니다.
* ﻿﻿런타임 체크로 인한 성능 영향을 고려해야 합니다.

> 타입 가드를 효과적으로 사용하면 타입스크립트의 타입 시스템을 최대한 활용하여 더 안전하고 명확한 코드를 작성할 수 있습니다. 특히 유니온 타입을 다룰 때 매우 유용합니다.

**타입 유틸리티**

타입스크립트는 코드의 가독성과 유지보수성을 높이기 위해 여러 가지 유틸리티 타입을 제공합니다. 이 유틸리티 타입들을 사용하면 기존 타입을 바탕으로 새로운 타입을 쉽게 생성할 수 있습니다.

• **Partial**: 모든 프로퍼티를 선택적으로 만듭니다

interface Todo {

title: string;

description: string;

}

let partialTodo: Partial<Todo> = {

title: "Learn TypeScript",

}

**Record:** 특정 키 타입과 값 타입을 가진 객체 타입을 만듭니다

type Page = "home" | "about" | "contact";

const nav: Record<Page, Pagelnfo> = {

home: { title: "Home" }, about: { title: "About Us" }, contact: { title: "Contact" },

}

**컴포넌트에 타입스크립트 적용하기**

**• 함수형 컴포넌트:** 함수형 컴포넌트를 작성할 때, React.Fc (Functioncomponent) 타입을 사용합니다.

interface AppProps {

message: string;

}

const App = ({ message }: AppProps) => {

return < div>(message)</div>;

};

**이벤트 타입 지정**

// 마우스 이벤트

const handleClick = (event: React.MouseEvent<HTMLButtonElement>) => {

console.log(event.currentTarget);

｝

<button onClick={handleClick)>Click me</button>;

// 폼 이벤트

const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {

event.preventDefault();

console.log("Form submitted");

｝

<form onSubmit=(handleSubmit)>

<button type="submit">Submit</button>

</form>;

// 입력 이벤트

const handleChange = (event: React.ChangeEvent<HTMLInputElement>)=>{

console.log(event.target.value);

};

**훅에 타입 적용**

// useState

const [count, setCount] = React.useState<number>(0);

// useContext

interface UserContextType {

name: string;

age: number;

}

const UserContext = React.createContext<UserContextType | null>(null);

const MyComponent: FC = () => {

const user = React.useContext(UserContext);

if (user) {

return <div>{user.name}</div>;

}

return <div>No user</div>;

}

**타입 에러 해결 및 디버깅**

* 명시적 타입 캐스팅은 타입스크립트에서 변수나 표현식의 타입을 개발자가 명시적으로 지정하는 것을 말합니다.
* ﻿﻿명시적 타입 캐스팅은 일반적으로 타입스크립트가 자동으로 타입을 추론할 수 없는 복잡한 경우나 다양한 타입 간 변환 시에 사용됩니다.

﻿﻿그러나 과사용은 타입 안정성을 저하시킬 수 있으므로 필요한 경우에만 사용하는 것이 좋습니다.

let someValue: any = "this is a string";

let strlength: number = (<string>someValue).length;

let someValue: any = "this is a string";

let strength: number = (someValue as string).length;

**디버깅**

**타입 추론 오류 확인**

* ﻿﻿IDE나 터미널에서 타입 추론이 잘못된 부분을 확인할 수 있습니다.
* ﻿﻿IDE는 보통 타입스크립트의 타입 추론 오류를 강조 표시합니다.

**타입스크립트 설정 검토**

* ﻿﻿tsconfigjson 파일을 통해 strict 모드를 활성화하고, 다양한 타입 체크 옵션을 설정하여 타입 에러를 미리 방지할 수 있습니다.

**타입 에러 메시지 이해**

* ﻿﻿타입 에러 메시지를 정확히 이해하고, 해당 라인의 코드와 관련된 부분을 검토하여 해결하는 것이 중요합니다.

**타입스크립트 디버거 사용**

﻿﻿vsCode와 같은 IDE에서 제공하는 타입스크립트 디버거를 활용하여 변수의 값과 타입을 실시간으로 확인할 수 있습니다.

**명확한 타입 정의:** 변수, 매개변수, 함수 등에 명확한 타입을 지정해야 합니다. 타입을 명시함으로써 코드의 의도를 명확히 하고, 버그를 사전에 방지할 수 있습니다.

* ﻿﻿**타입 추론 이용: 타입** 추론을 적절히 활용하여 코드를 간결하게 유지할 수 있습니다. 필요한 경우에는 명시적인 타입 지정을 추가로 해주어야 합니다.
* ﻿﻿**타입 에러 처리:** 컴파일 시간에 발생하는 타입 에러를 주의 깊게 살펴야 합니다. 타입 에러 메시지를 이해하고, 해당 부분을 수정하여야 합니다.
* ﻿﻿**타입스크립트 최신 기능 활용:** 새로운 타입스크립트 기능과 문법을 적극 활용하여 코드를 더욱 안전하고 효율적으로 작성할 수 있습니다.

﻿﻿**any 타입 사용 금지:** 타입스크립트에서 any 타입 사용은 타입 안정성을 저하시킬 수 있으므로, 명시적인 타입 지정을 통해 코드의 안정성과 가독성을 높이는 것을 권장합니다.