



13장 타입스크립트와 객체 지향

1. TS의 객체 지향

프론트엔드에 객체 지향을 어떻게 적용하지? → 사실 우리는 이미 객체 지향을 적용하고 있다.

컴포넌트 **객체의 한 형태** 는 스스로 책임을 져야하는 역할을 수행하면서 다른 컴포넌트 객체와 협력하는 독립적인 객체이다. → 컴포넌트를 조합하는 것도 객체 지향을 활용하는 것이다.

- JS

JS는 프로토타입 기반의 객체 지향 언어로 분류된다. 그러나 전통적인 객체 지향 프로그래밍 언어에서 기대할 수 있는 일부 기능을 지원하지 않아 객체지향을 온전히 구현하기 어렵다.

→ TS가 접근제어자 **private** 나 추상 클래스, 추상 메서드 같은 기능을 지원하면서 해결한다.

TS : 객체지향을 구현할 수 있도록 도와주는 JS의 슈퍼셋

- TS : 점진적, 구조적, 덕 타이핑이 결합한 언어

노미널 타이핑 언어 **JAVA** - 인터페이스와 클래스가 1:1 대응

구조적 타이핑 언어 **TS** - 하나의 클래스에 여러 인터페이스가 연결될 수 있으며, 하나의 인터페이스에 여러 클래스가 연결될 수도 있다.

점진적 타이핑 : 프로그램 전체가 아닌 개발자가 명시한 일부분만 정적 타입 검사를 거치게 하고, 나머지 부분은 그대로 동적 타입 검사가 이루어지게 하여 점진적 개선을 할 수 있도록 해준다.

덕 타이핑 : 객체의 변수와 메서드 집합의 타입을 결정하게 해준다.

구조적 타이핑 : 객체의 속성에 해당하는 특정 타입의 속성을 갖는지를 검사하여 타입 호환성을 결정한다.

ex) TS

노미널 타이핑 : 명시적인 선언이나 이름에 의존하여 명확한 상속 관계를 지향한다. (↔ 구조적 타이핑)

ex) 자바, C#

✓ 객체 지향의 관점에서 TS가 프론트엔드 주는 이점

1. prop : 컴포넌트 간의 협력 관계 표현

의존성 주입 **DI** 패턴을 더욱 명확하게 표현해준다.

DI 패턴

A 클래스가 B 클래스에 의존하더라도,
A가 B의 구체 클래스가 아닌 B의 인터페이스에 의존하도록 설계한다.

- DI 패턴을 따르면 객체 간의 결합도를 낮출 수 있다.

2. TS는 객체 지향의 폭을 넓혀준다.



웹 개발을 하면서 제대로 된 객체 지향을 구현하기 어려운 이유

- 과연 우리는 컴포넌트를 만들 때 컴포넌트의 역할과 컴포넌트 간의 협력에 초점을 맞추고 있었을까

웹개발에서는 **선언적인 언어, 문법** `JSX` 을 사용할 때가 있기 때문이다.

객체지향 일반적으로 객체 지향을 구현하려면 객체 간의 협력 관계를 먼저 고려하고, 메시지를 정의하여 해당 메시지를 수신할 알맞은 객체를 결정하는 절차를 따르게 된다.

프론트엔드 그러나 HTML 마크업을 선언적으로 작성할 때는 컴포넌트 간의 관계를 먼저 떠올리기 어렵다. → 컴포넌트 간의 협력 관계를 먼저 고려하고 메시지를 정하는 것은 현실적으로 힘들다.

객체지향 구현 이유 : 변경이 용이하고 유지보수성이 높은 설계를 위해

프론트엔드 그러나 사전에 레이아웃의 변화를 예측할 수 없다.

- 레이아웃의 변동에 대응 위한 다양한 패턴 등장 `MVP` `MVC` , `MVVM`

레이아웃은 예상치 못한 변동 사항이 생길 가능성이 높기 때문에 미확정 영역으로 두고, 공통 컴포넌트와 비즈니스 영역에서 객체 지향 원칙을 적용하여 설계하면 좋은 구조를 개발할 수 있다.



2. 우아한형제들의 활용 방식

✓ 우아한형제들 팀의 설계 방식

- 온전히 **레이아웃만 담당** 하는 컴포넌트 영역
- 컴포넌트 영역 위에서 레이아웃과 비즈니스 로직을 연결해주는 **커스텀** **훅** 영역
- 훅 영역 위에서 **객체**로서 상호 협력하는 **모델** 영역
- 모델 영역 위에서 API를 해석하여 모델로 전달하는 **API 레이어** 영역

1. 컴포넌트 영역

온전히 레이아웃 영역만 담당한다.

➡ 비즈니스 로직은 `useCartStore` 내부 어딘가에 존재할 것이다.

▼ 장바구니 관련 다이얼로그 컴포넌트 코드

```
// components/CartCloseoutDialog.tsx

import { useCartStore } from "store/modules/cart";

const CartCloseoutDialog: React.VFC = () => {
  const cartStore = useCartStore();

  return (
    <Dialog
      opened={cartStore.PresentationTracker.isDialogOpen("")}
      title="마감 세일이란?"
      onRequestClose={cartStore.PresentationTracker.closeDialog}
    >
      <div
        css={css`
          margin-top: 8px;
        `}
      >
        지점별 한정 수량으로 제공되는 할인 상품입니다. 재고 소진 시 가
        달라질 수 있습니다. 유통기한이 다소 짧으나 좋은 품질의 상품입니
      </div>
    </Dialog>
  );
};

export default CartCloseoutDialog;
```

2. 커스텀 훅

장바구니에 상품을 담는 비즈니스 로직을 레이아웃과 연결해주기 위한 커스텀 훅 영역

- 해당 스토어 객체에서 최종적으로 사용되는 `setupContext` 는 컨텍스트와 관련된 혹은 다루는 유틸리티 함수이므로 ,해당 스토어를 혹은 영역의 로직으로 볼 수 있다.

➡ `addToCart` 는 분명 API를 호출하는 함수일 것이다. 내부에서 `addToCartRequest` 시리얼라이저 함수를 호출하고 있다.

시리얼라이저 함수 : 데이터 구조나 객체를 저장하거나 전송하기 쉬운 형식으로 변환하는 함수

▼ 전역 상태를 관리하는 스토어 내의 useCartStore

```
// store/cart.ts

class CartStore {
  public async add(target: RecommendProduct): Promise<void> {
    const response = await addToCart(
      addToCartRequest({
        auths: this.requestInfo.AuthHeaders,
        cartProducts: this.productsTracker.PurchasableProd
        shopID: this.shopID,
        target,
      })
    );

    return response.fork(
      (error, _, statusCode) => {
        switch (statusCode) {
          case ResponseStatus.FAILURE:
            this.presentationTracker.pushToast(error);
            break;
          case ResponseStatus.CLIENT_ERROR:
            this.presentationTracker.pushToast(
              "네트워크가 연결되지 않았습니다."
            );
            break;
          default:
            this.presentationTracker.pushToast(
              "연결 상태가 일시적으로 불안정합니다."
            );
        }
      }
    );
  }
}
```

```

        }
      },
      (message) => this.applyAddedProduct(target, message)
    );
  }
}

const [CartStoreProvider, useCartStore] = setupContext<Car
export { CartStore, CartStoreProvider, useCartStore };

```

반환값 : AddToCartReuquest 타입의 객체

매개변수 : RecommendProduct 타입의 target

➡ 해당 타입에 대한 정의는 **모델** 영역에서 확인할 수 있다.

▼ addToCartRequest 시리얼라이저 함수

```

// serializers/cart/addToCartRequest.ts

import { AddToCartRequest } from "models/externals/Cart/Re
import { IRequestHeader } from "models/externals/lib";
import {
  RecommendProduct,
  RecommendProductItem,
} from "models/internals/Cart/RecommendProduct";
import { Product } from "models/internals/Stuff/Product";

interface Params {
  auths: IRequestHeader;
  cartProducts: Product[];
  shopID: number;
  target: RecommendProduct;
}

function addToCartRequest({
  auths,
  cartProducts,
  shopID,
  target,

```

```

}: Params): AddToCartRequest {
  const productAlreadyInCart = cartProducts.find(
    (product) => product.getId() === target.getId()
  );

  return {
    body: {
      items: target.getItems().map((item) => ({
        itemId: item.id,
        quantity: getItemQuantityFor(productAlreadyInCart,
          salePrice: item.price,
        })),
      productId: target.getId(),
      shopId: shopID,
    },
    headers: auths,
  };
}

export { addToCartRequest };

```

3. 모델 영역

RecommendProduct : 클래스로 표현된 객체로 추천 상품을 나타낸다. 이 객체는 다른 컴포넌트 및 모델 객체와 함께 협력하게 된다.

```

// models/Cart.ts

export interface AddToCartRequest {
  body: {
    shopId: number;
    items: { itemId: number; quantity: number; salePrice: num
    productId: number;
  };
  headers: IRequestHeader;
}

```

```

/**
 * 추천 상품 관련 class
 */

export class RecommendProduct {
  public getId(): number {
    return this.id;
  }

  public getName(): string {
    return this.name;
  }

  public getThumbnail(): string {
    return this.thumbnailImageUrl;
  }

  public getPrice(): RecommendProductPrice {
    return this.price;
  }

  public getCalculatedPrice(): number {
    const price = this.getPrice();
    return price.sale?.price ?? price.origin;
  }

  public.getItems(): RecommendProductItem[] {
    return this.items;
  }

  public getType(): string {
    return this.type;
  }

  public getRef(): string {
    return this.ref;
  }
}

```



```

constructor(init: any) {
  this.id = init.id;
  this.name = init.displayName;
  this.thumbnailImageUrl = init.thumbnailImageUrl;
  this.price = {
    sale: init.displayDiscounted
      ? {
          price: Math.floor(init.salePrice),
          percent: init.discountPercent,
        }
      : null,
    origin: Math.floor(init.retailPrice),
  };
  this.type = init.saleUnit;
  this.items = init.items.map((item) => {
    return {
      id: item.id,
      minQuantity: item.minCount,
      price: Math.floor(item.salePrice),
    };
  });
  this.ref = init.productRef;
}

private id: number;
private name: string;
private thumbnailImageUrl: string;
private price: RecommendProductPrice;
private items: RecommendProductItem[];
private type: string;
private ref: string;
}

```

4. API 레이어 영역

혹에서 실제로 실행되는 `addToCart` 함수를 확인하자.

```
// apis/Cart.ts

// APIResponse는 데이터 로드 성공한 상태와 실패한 상태의 반환 값을 제네릭으로 정의
// (APIResponse<OK, Error>)
interface APIResponse<OK, Error> {
  // API 응답에 성공한 경우의 데이터 형식
  ok: OK;
  // API 응답에 실패한 경우의 에러 형식
  error: Error;
}

export const addToCart = async (
  param: AddToCartRequest
): Promise<APIResponse<string, string>> => {
  return (await GatewayAPI.post<IAddCartResponse>("/v3/cart",
    (data) => data.message
  ));
};
```

객체 지향 구현 자체를 클래스라고 생각하는 착각을 많이 한다.

→ 전혀 그렇지 X다. 클래스는 객체를 표현하는 방법의 도구일 뿐이다. 컴포넌트를 함수형으로 선언하든 클래스형으로 선언하든 모두 객체를 나타낸다.

✓ 함수 vs. 클래스 컴포넌트

리액트 혹은 나오며 함수 컴포넌트의 사용률이 높아졌고, 리액트 공식 문서에서도 함수 컴포넌트를 권장한다.

- 틀에서 찍어내듯 일관된 템플릿에 맞춘 컴포넌트를 많이 생성해야 할 경우 → 클래스 컴포넌트 방식

ex) 페이지 템플릿을 클래스 컴포넌트로 만들어서 공통으로 정의되어야 할 행동 **내비게이션의 뒤로가기 버튼 눌렀을 때의 동작** 을 **abstract** 메서드로 만들어 사용하기도 한다.
(
템플릿 강제화 기능)

애플리케이션 설계는 **트레이드오프**의 결과물이다.

트레이드오프 : 하나의 것을 얻기 위해 다른 것을 포기해야 하는 상황

3. 캡슐화와 추상화

추상화 : 객체들을 모델링하는 과정 자체. 이 객체들을 좀 더 사람이 인지할 수 있도록 적합한

Cart, **Product**, **Seller** 설계를 하는 것

▼ 추상화 상세 설명

추상클래스 **abstract**

```
// 추상 class
// 추상화 : 프로퍼티나 메소드에 이름만 선언해주고, 구체적인 기능은 상속
// 추상 class를 상속받아 만든 수많은 객체들이 동일한 메소드를 가지고 5
```

캡슐화 : 다른 객체 내부의 데이터를 꺼내와서 직접 다루지 않고, 해당 객체에서 처리할 행위를 따로 요청함으로써 협력하는 것

ex) 컴포넌트 **객체**의 내부 데이터인 상태가 바로 캡슐화의 대상이다. → 컴포넌트 내의 상태와 prop을 잘 다루는 것

▼ 캡슐화 상세 설명

1. 정보은닉 : 외부 코드가 알 필요가 없는 정보는 알려주지 않고, 그냥 호출하여 쓰면 됨.

a. ex) plusHp 함수

2. 접근 제어


a. private, public 접근 허용 불허 구분

3. set, get 통제


a. 필드를 접근, 변경 제한을 두거나

허용하는 방법을 서버코드가 마련해줘요.

! Prop drilling이 심할수록 컴포넌트 간의 결합도가 높아지고, 내부 처리 로직이 외부로 드러난다. → 캡슐화를 저해한다.

 **옵저버 패턴** 등장, 상태관리 라이브러리 **Redux** 등장

적절하게 캡슐화되고 추상화된 컴포넌트를 활용하면 애플리케이션을 더 유기적으로 구성할 수 있다.

 최종적인 우리의 지향점 : 객체들이 유기적으로 협력하게끔 해서 올바르게 도메인을 분리하는 것



객체 지향 패러다임에 매몰되기보다는 어떻게 하면 더 **유기적인 협력 관계**를 만들어낼 수 있을지 **명확하게 도메인을 분리**할 수 있을지에 집중해보자.

→ 그러면 객체 지향이 추구하고자 하는 방향에 가깝게 프론트엔드 개발을 할 수 있을 것이다.

4. 정리

우리는 이미 객체지향을 구현하고 있다.

함수, 클래스, 모듈을 분리하는 것도 객체 지향 프로그래밍의 일부다.

객체 그 자체보다는 객체의 책임을 먼저 생각하는 것이 객체 지향의 핵심이다. 즉, 화면을 구성하기 전에 개별적인 컴포넌트를 먼저 생각하자.

객체지향은 유지보수를 위한 개념이므로, 유지보수가 필요 X다면 객체지향을 꼭 따를 필요가 없을 수도 있다.