

# Chap9. 코드를 재사용하고 일반화 할 수 있도록 하라

## 이번 장의 목표

- 안전하게 재사용할 수 있는 코드 작성 방법
- 다양한 문제를 해결하기 위해 일반화된 코드를 작성하는 방법

## 다시 읽어보기

### 조코나코 2장

이번 장은 간결한 추상화 계층을 만들고, 코드를 모듈화 하는 것과 매우 깊은 관련이 있습니다.

앞서 2장에서 상위 문제를 해결할 때, 일련의 하위 문제(추상화 계층)로 세분화 해서 해결하는 방법을 다루었습니다. 기억나시나요?

다른 개발자가 이미 주어진 하위 문제를 해결했다면, 해당 문제에 대한 해결책을 재사용할 수 있겠죠?

이번 장에서 안전하게 재사용할 수 있는 코드를 작성하는 방법을 알아봅시다!

### 9.1 가정을 주의하라

우리는 코드 작성 시, "~는 ~할 것이다" 라는 가정을 바탕으로 작업을 합니다.

하지만, 이 책에서는 이러한 가정으로 인해 코드가 더 취약해지고 활용도가 낮아져 재사용하기에 안전하지 않을 수 있다고 경고를 합니다.

그래서 그 가정으로 초래될 비용과 이점을 생각해봐야 한다고 하는데요!

명백한 이득이 미미하다면 가정을 하지 않는 것이 최선일 수도 있다고 합니다. 한 번 자세히 살펴봅시다!

#### 9.1.1 가정은 코드 재사용 시 버그를 초래할 수 있다.

## 예제 9.1 가정을 포함하는 코드

```
class Article {  
    private List<Section> sections;  
  
    List<Image> getAllImages() {  
        for (Section section in sections) {  
            if (section.containsImages()) {  
                // 기사 내에 이미지를 포함하는 섹션은 최대  
                // 하나만 있다.  
                return section.getImages();  
            }  
        }  
        return [];  
    }  
}
```

코드 내에서 주석문으로 설명된 가정  
이미지를 가지고 있는 첫 번째 섹션의 이미지만 반환한다.

이 코드는 사용자가 읽을 수 있는 뉴스 웹 사이트의 기사를 Article 클래스로 나타낸 것입니다.

getAllImages() 함수는 기사에 포함된 모든 이미지를 반환하는 함수로, 이미지가 포함된 섹션을 찾을 대가지 문서 내의 섹션을 반복해서 확인하고 해당 섹션의 이미지를 반환합니다.

이 코드는 기사에 이미지가 포함된 섹션이 하나만 있을 것이라고 가정을 하는데요!

이미지를 포함하는 부분이 발견되자 마자 for-loop를 종료하기 때문에 성능은 소폭 향상되겠지만,

getAllImages() 함수가 이미지가 있는 섹션이 두 개 이상인 기사에 사용된다면 치명적인 버그를 초래할 것입니다.

"소폭으로 향상되는 성능을 위해, 재사용성을 포기하고 안전하지 못한 코드를 작성한다"는 것은 옳지 않은 접근 방법이겠죠?

위에서 말했던, 명백한 이득이 미미하다면 가정을 하지 않는 것이 최선이다라는 말이 해당 사항에 적합한 말이겠네요!

### Note

명백한 이득이 미미하다? -> 가정을 하지 않고 코드를 짜야한다는 결론이 나옵니다.

## 9.1.2 해결책: 불필요한 가정을 피하라

## 예제 9.2 가정을 제거한 코드

```
class Article {  
    private List<Section> sections;  
    ...  
  
    List<Image> getAllImages() {  
        List<Image> images = [];  
        for (Section section in sections) {  
            images.addAll(section.getImages());  
        }  
        return images;  
    }  
}
```

모든 섹션에서  
이미지를 모아서 반환한다.

앞서 했던 가정을 없애고 `getAllImages()` 함수가 이미지가 포함된 첫번째 섹션이 아닌 모든 섹션의 이미지를 반환하도록 변경된 코드입니다. `for` 루프가 끝까지 돌아야 하겠지만, 성능에 영향을 미칠 가능성은 적습니다.

### 섣부른 최적화

섣부른 최적화를 피하려는 열망은 소프트웨어 공학과 컴퓨터 과학 내에서 잘 정립된 개념이다. 코드 최적화는 일 반적으로 비용이 든다. 즉, 최적화된 해결책을 구현하는 데 더 많은 시간과 노력이 필요하며 그 결과 코드는 종종 가독성이 떨어지고, 유지 관리가 더 어려워지며, 가정을 하게 되면 견고함이 떨어질 가능성이 있다. 게다가 최적화는 보통 프로그램 내에서 수천 번 혹은 수백만 번 실행되는 코드 부분에 대해 이루어질 때 상당한 이점이 있다 따라서 대부분의 경우에는 큰 효과 없는 코드 최적화를 하느라고 애쓰기보다는 코드를 읽을 수 있고, 유지보수 가능하며, 견고하게 만드는 데 집중하는 것이 좋다. 코드의 어떤 부분이 여러 번 실행되고 그 부분을 최적화하는 것이 성능 향상에 큰 효과를 볼 수 있다는 점이 명백해질 때에라야 최적화 작업을 해도 무방하다.

### 9.1.3 해결책: 가정이 필요하면 강제적으로 하라

위에서 "명백한 이득이 미미하다면 가정을 하지 말자"고 했었습니다.

그러면 이 문장의 대우는 가정을 하려면 명백한 이득이 있어야 한다가 되겠네요?

맞습니다. 때로는 가정이 필요하거나 가정으로 얻는 이득이 훨씬 더 큰 비용을 가져올 경우가 있습니다.

하지만, 코드에 가정이 있을 때, 다른 개발자들은 여전히 이 사실을 모를 수 있습니다.

다른 개발자들이 혼란을 겪지 않도록 하면서 가정을 해서 코드를 작성하고 싶다면 어떻게 해야할까요?

그 방법은 다음과 같습니다.

## Note

1. "가정이 깨지지 않게 만들라" : 가정이 깨지면 컴파일 되지 않는 방식으로 코드를 작성한다면 가정이 항상 유지될 수 있다. (3장과 7장에서도 언급)
2. "오류 전달 기술을 사용하라" : 가정을 깨는 것이 불가능하게 만들 수 없는 경우, 오류를 감지하고 오류 신호 전달 기술을 사용하여 신속하게 실패하도록 코드를 작성할 수 있다. (3장 끝부분, 4장에서도 언급)

### 예제 9.3 가정을 포함하는 코드

```
class Article {  
    private List<Section> sections;  
    ...  
  
    Section? getImageSection() {  
        // 기사 내에 이미지를 포함하는 섹션은 최대  
        // 하나만 있다.  
        return sections  
            .filter(section -> section.containsImages())  
            .first();  
    }  
}
```

이미지를 갖는 첫 번째 섹션을 반환하거나  
이미지를 갖는 섹션이 없는 경우  
널을 반환한다.

예제 9.1은 for문을 사용했지만, 이번 예제는 stream을 사용했네요!

둘 다 동일하게 "기사에서 한 섹션에만 이미지가 있다" 가정을 포함하는 코드입니다.

또한 가정을 사용하려면, "가정이 필요하면 강제적으로 하라" 는 조건을 충족하지 못한 코드이기도 합니다.

### 예제 9.4 가정에 의존하는 호출자

```
class ArticleRenderer {  
    ...  
  
    void render(Article article) {  
        ...  
        Section? imageSection = article.getImageSection();  
        if (imageSection != null) {  
            templateData.setImageSection(imageSection);  
        }  
        ...  
    }  
}
```

기사 템플릿은 이미지를 갖는 섹션을  
최대 하나만 처리할 수 있다.

여기서는 ArticleRenderer 클래스의 render 메서드에서 getImageSection을 호출하여 이미지 섹션을 가져오고, 그 결과가 null이 아닌 경우에만 setImageSection을 호출하도록 되어 있습니다. 이 코드가 정상적으로 작동하려면 문서에는 이미지 섹션이 하나만 포함되어 있어야 한다는 암묵적인 가정이 필요합니다. (왜냐하면 getImageSection 함수는 이미지를 갖는 첫번째 섹션을 반환하거나 없으면 null을 반환하기 때문)

즉, 이 코드는 문서에 이미지 섹션이 하나만 있다고 가정하고 있지만, 실제로 문서 작성자가 여러 이미지 섹션을 추가한다면 코드가 예상과 다른 방식으로 작동할 수 있다는 것입니다!

### 예제 9.5 가정의 강제적 확인

```
class Article {  
    private List<Section> sections;  
    ...  
    Section? getOnlyImageSection() {  
        List<Section> imageSections = sections  
            .filter(section -> section.containsImages());  
  
        assert(imageSections.size() <= 1,  
              "기사가 여러 개의 이미지 섹션을 갖는다");  
  
        return imageSections.first();  
    }  
}
```

함수의 이름은  
호출자가 할 가정을 나타낸다.

어서선은 이 가정을  
강제로 확인한다.

ImageSections 리스트에서 첫 번째 항목을 반환하거나  
리스트가 비어 있으면 널을 반환한다.

여러 이미지 섹션으로 된 문서를 나타내는 것이 지원되지 않는 상황이라는 것을 가정해봅시다.

그럴 경우, 2번 : 오류 전달 기술을 사용하는 방법으로 기사에 이미지 섹션이 하나 이하임을 강제 할 수 있겠네요!

위 코드는 오류 전달 기법을 사용하여 가정을 강제로 인지하게 함으로서 신속하게 실패하기 위한 코드입니다. assert 문을 사용하여 imageSections 리스트의 크기가 1 이하임을 확인합니다. 만약 imageSections의 크기가 2 이상이면 assert 문이 실패하여 예외가 발생하고, “기사가 여러 개의 이미지 섹션을 갖는다”는 메시지가 표시됩니다.

## 오류 전달 기법

4장에서는 여러 가지 오류 전달 기법에 대해 자세히 논의했는데, 특히 호출하는 쪽에서 오류로부터 복구하기를 원하는지의 여부에 따라 어떻게 기법의 선택이 달라지는지 살펴봤다.

예제 9.5는 어서션을 사용하는데 호출하는 쪽에서 오류로부터 복구하기를 원하지 않는 경우에 적합하다. 기사가 프로그램 내에서 내부적으로 생성된다면 가정을 깨는 것은 프로그래밍 오류를 의미하며, 이는 어서션이 적절하다는 것을 의미한다. 하지만 외부 시스템이나 사용자에 의해 기사가 제공된다면, 호출하는 쪽에서 더 매끄러운 방식으로 오류를 처리하기를 원할 수도 있다. 이 경우에는 명시적인 오류 전달 기법이 더 적절할 수 있다.

## 느낀점

이렇게, 가정을 강제적으로 하는 방법도 살펴보았습니다.

저는 이 글을 통해 “명백한 성능 이점을 가져오지 않는 이상 가정에 의존하지 않는 코드를 작성하자“는 결론을 내렸습니다. 예를 들어, 페이지 처리처럼 모든 데이터를 가져오지 않고 특정 페이지만 가져와서 렌더링 속도를 향상시킬 수 있는 경우에는 가정을 사용할 수 있겠지만, 이때는 오류 전달 기술을 사용하거나 가정이 깨지지 않도록 코드를 작성해야 한다고 생각합니다. 즉, 가정을 강제해야 하는 상황에서는 이러한 방식을 도입해 코드를 작성할 것 같습니다.

## 9.2 전역 상태를 주의하라

전역 상태(global state) 또는 전역 변수(global variable)는 실행되는 프로그램 내의 모든 컨텍스트 사이에서 공유됩니다. 전역 변수를 정의하는 일반적인 방법은 다음과 같습니다.

- 자바나 C# 같은 언어에서 변수를 static으로 표시하는 방법 (이 책의 의사 코드에서도 사용되는 방식입니다).
- C++와 같은 언어에서는 클래스나 함수의 외부, 즉 파일 수준에서 변수를 정의합니다.
- 자바스크립트 기반 언어에서는 전역 window 객체의 속성으로 변수를 정의합니다.

## 예제 9.6 전역변수를 갖는 클래스

```
class MyClass {  
    private Int a = 3;           ← 인스턴스 변수  
    private static Int b = 4;     ← static으로 표시되어 있기 때문에  
                                전역변수다.  
  
    void setA(Int value) { a = value; }  
    Int getA() { return a; }  
  
    void setB(Int value) { b = value; }  
    Int getB() { return b; }  
  
    static Int getBStatically() { return b; } ← 정적 함수  
}  
}
```

전역 변수에 대한 설명을 위해 예제 9.6 코드를 살펴보겠습니다. 코드에 대해 다음 사항을 주의해야 합니다.

- a는 인스턴스 변수입니다. MyClass 의 각 인스턴스는 자체의 a 변수를 갖습니다. 한 인스턴스에서 이 변수를 수정할 때, 다른 인스턴스에는 영향을 미치지 않습니다.
- b는 정적 변수이며, 이는 전역 변수를 의미합니다. 따라서 MyClass 의 모든 인스턴스 간에 공유되며, MyClass 의 인스턴스를 통해 접근하지 않아도 됩니다.
- getBStatically() 함수는 정적 함수로 표시되어 있으며, 이는 MyClass.getBStatically() 와 같은 구문으로 호출되기 때문에 클래스의 인스턴스가 필요 없습니다. 이와 같은 정적 함수는 클래스에 정의된 정적 변수에는 접근할 수 있지만, 인스턴스 변수에는 접근할 수 없습니다.

```
MyClass instance1 = new MyClass();
MyClass instance2 = new MyClass();
```

```
instance1.setA(5);
instance2.setA(7);
print(instance1.getA()) // 출력: 5
print(instance2.getA()) // 출력: 7
instance1.setB(6);
instance2.setB(8);
print(instance1.getB()) // 출력: 8
print(instance2.getB()) // 출력: 8
print(MyClass.getBStatically()) // 출력: 8
```

MyClass의 각 인스턴스는  
자신만의 'a' 변수를 별도로 갖는다.

전역변수 'b'는 MyClass의  
모든 인스턴스 사이에 공유된다.

← 'b'는 MyClass의 인스턴스를 통하지 않고 정적으로 접근할 수 있다.

#### **NOTE** 전역성과 가시성을 혼동하지 말라

변수가 전역인지의 여부를 가시성과 혼동해서는 안 된다. 변수의 가시성은 변수가 퍼블릭인지 혹은 프라이빗인지를 나타내며, 코드의 다른 부분이 해당 변수에 접근할 수 있는지를 지시한다. 변수는 전역 여부와 관계없이 퍼블릭이나 프라이빗일 수 있다. 요점은 전역변수가 클래스의 인스턴스나 함수의 자체 버전 대신 프로그램의 모든 콘텍스트 간에 공유된다는 것이다.

다음 코드는 전역변수 b가 정적 콘텍스트 뿐만 아니라 클래스의 모든 인스턴스 간에 공유되는 동안 인슬턴스 변수 a가 개별 인스턴스에 어떻게 적용되는지를 보여줍니다.

전역 변수는 프로그램 내의 모든 콘텍스트에 영향을 미치기 때문에 전역변수를 사용할 때는 누구도 해당 코드를 다른 목적으로 재사용하지 않을 것이라는 암묵적인 가정을 전제합니다. 이전 9.1장에서 보았듯이 가정에는 비용이 수반됩니다. 전역 상태는 코드를 매우 취약하게 만들고 재사용하기도 안전하지 않기 때문에 일반적으로 이점보다 비용이 큽니다.

### 9.2.1 전역 상태를 갖는 코드는 재사용하기에 안전하지 않을 수 있다.

#### 전역 변수의 사용 유혹과 문제점

프로그램에서 여러 부분이 어떤 특정 상태를 공유하고 접근할 필요가 있을 때, 전역 변수로 정의하고 싶을 수 있습니다. 이렇게 하면 코드의 어느 부분에서든 해당 상태에 쉽게 접근할 수 있기 때문입니다. 그러나 이렇게 하면 코드의 재사용이 안전하지 않을 수 있습니다.

### 예제 9.7 ShoppingBasket 클래스

```
class ShoppingBasket {  
    private static List<Item> items = [];  
  
    static void addItem(Item item) {  
        items.add(item);  
    }  
  
    static List<Item> getItems() {  
        return List.copyOf(items);  
    }  
}
```

static으로 표시되어 전역변수다.

static으로 표시된 함수

### 예제 9.8 ShoppingBasket을 사용하는 클래스

```
class ViewItemWidget {  
    private final Item item;  
  
    ViewItemWidget(Item item) {  
        this.item = item;  
    }  
    ...  
  
    void addItemToBasket() {  
        ShoppingBasket.addItem(item);  
    }  
}  
  
class ViewBasketWidget {  
    ...  
    void displayItems() {  
        List<Item> items = ShoppingBasket.getItems();  
        ...  
    }  
}
```

전역 상태를 수정한다.

전역 상태를 읽는다

## 예시: 온라인 쇼핑 애플리케이션

온라인 쇼핑 애플리케이션을 만들고 있다고 가정해봅시다. 이 애플리케이션에서 사용자는 상품을 탐색하고, 장바구니에 추가하며, 마지막에 결제(체크아웃)할 수 있습니다. 이때 사용자 장바구니의 항목 목록은 애플리케이션의 여러 부분(예: 상품 추가 기능, 장바구니 검토 화면, 결제 기능 등)에서 접근해야 할 공유 상태입니다. 이러한 상황에서 사용자 장바구니 내용을 전역 변수로 저장하고 싶어질 수 있습니다. 예제 9.7은 장바구니에 전역 상태를 사용했을 때의 코드를 보여줍니다. 이 코드에서 주의해야 할 사항은 다음과 같습니다:

- items 변수는 static으로 표시됩니다. 이는 특정 ShoppingBasket 인스턴스에 연결되지 않고 전역 변수로 사용된다는 의미입니다.
- addItem()과 getItems() 함수도 static으로 표시됩니다. 따라서 코드 내 어디서든 ShoppingBasket 인스턴스 없이 ShoppingBasket.addItem(...)이나 ShoppingBasket.getItems()와 같은 방식으로 호출할 수 있으며, 전역 변수 items에 접근하게 됩니다.

### 예제 9.8 ShoppingBasket을 사용하는 클래스

```
class ViewItemWidget {  
    private final Item item;  
  
    ViewItemWidget(Item item) {  
        this.item = item;  
    }  
    ...  
  
    void addItemToBasket() {  
        ShoppingBasket.addItem(item);  
    }  
}  
  
class ViewBasketWidget {  
    ...  
    void displayItems() {  
        List<Item> items = ShoppingBasket.getItems();  
        ...  
    }  
}
```

전역 상태를 수정한다.

전역 상태를 읽는다

ViewItemWidget 은 사용자가 장바구니의 내용을 볼 수 있습니다. 이는 ShoppingBasket.getItems() 를 호출하면 됩니다.

또한, 사용자가 본 항목을 자신의 장바구니에 추가할 수 있는데 이는 `ShoppingBasket.addItem()` 을 호출하면 됩니다.

누군가 이 코드를 재사용하려고 하면 어떻게 되는가?



그림 9.1 전역 상태를 사용하면 코드 재사용이 안전하지 않을 수 있다.

### 1. 단일 인스턴스 가정의 취약성

- 이 코드에서는 암묵적으로 프로그램을 실행하는 인스턴스당 하나의 장바구니만 필요하다는 가정을 하고 있습니다. 이 가정이 유지되면 코드가 정상적으로 동작하겠지만, 실제로는 이 가정이 쉽게 깨질 수 있습니다.

### 2. 가정이 깨질 가능성

- 장바구니 내용을 서버에 저장하도록 변경하여 서버가 여러 사용자 요청을 처리하게 되면, 서버 인스턴스 당 여러 장바구니가 필요해집니다.
- 사용자가 나중을 위해 장바구니 항목을 저장할 수 있는 기능을 추가하면, 클라이언트는 현재 장바구니 외에도 저장된 장바구니까지 처리해야 합니다.
- 신선 농산물 등 별도의 공급자와 배송 메커니즘이 필요한 다른 품목이 추가되면, 별도의 장바구니로 관리해야 할 수도 있습니다.

### 3. 문제 발생의 가능성

- 두 개의 코드가 동시에 ShoppingBasket 클래스를 사용하고 있다면, 이들은 서로 간섭할 수 있습니다. 전역 상태를 사용하게 되면, ShoppingBasket 클래스 같은 공유 객체를 여러 코드에서 동시에 접근할 수 있게 됩니다. 이 경우 하나의 코드가 ShoppingBasket에 항목을 추가하거나 제거할 때, 이를 참조하는 모든 코드에 영향을 미치게 됩니다.
- 예를 들어, 하나의 코드가 ShoppingBasket.items라는 정적 필드에 항목을 추가하면, 다른 모든 코드에서도 이 변경 사항이 반영됩니다. 따라서 클래스의 모든 인스턴스가 동일한 items 필드를 공유하게 되어, 단일 애플리케이션 인스턴스에서 여러 사용자가 장바구니를 동시에 사용할 경우 심각한 오류가 발생할 수 있습니다.

## 9.2.2 해결책: 공유 상태에 의존성을 주입하라

### 예제 9.9 수정된 ShoppingBasket 클래스

```
class ShoppingBasket {
    private final List<Item> items = [];
    ← 정적이 아닌
    ← 인스턴스 변수

    void addItem(Item item) {
        items.add(item);
    }

    void List<Item> getItems() {
        ← 정적이 아닌 멤버 함수
        return List.copyOf(items);
    }
}
```

앞 장에서는 의존성 주입(DI, Dependency Injection) 기술에 대해 논의했습니다. 의존성 주입이란, 클래스가 생성자에서 다른 클래스의 인스턴스를 직접 생성해 사용하는 대신, 필요한 인스턴스를 외부에서 ‘주입’받아 사용하는 방식입니다. 이를 통해 클래스 간의 결합도를 낮추고 유연성을 높일 수 있습니다. 의존성 주입은 전역 상태를 사용하는 것보다 더 통제된 방식으로 클래스 간의 상태를 공유하는 좋은 방법이기도 합니다.

이전의 ShoppingBasket 클래스는 정적 변수와 정적 메서드를 사용하여 전역 상태를 가지고 있었습니다. 첫 번째 개선 단계는 ShoppingBasket 클래스를 인스턴스화할 수 있는 형태로 변경하여 각 인스턴스가 고유한 상태를 갖도록 만드는 것입니다. 예제 9.9에서는 이렇게 변경된 ShoppingBasket 클래스를 보여줍니다. 이 코드에서 주목해야 할 사항은 다음과 같습니다.

- `items`는 더 이상 정적 변수가 아닙니다. 이제 인스턴스 변수로 변경되어, `ShoppingBasket` 클래스의 특정 인스턴스와 연결됩니다. 따라서 `ShoppingBasket` 클래스의 두 인스턴스를 생성하면, 각각 다른 항목을 가질 수 있습니다.
- `addItem()` 및 `getItems()` 메서드는 더 이상 정적 메서드가 아닙니다. 이제 `ShoppingBasket` 클래스의 인스턴스를 통해서만 접근할 수 있으며, `ShoppingBasket.addItem(...)`이나 `ShoppingBasket.getItems()`와 같은 호출은 더 이상 불가능합니다.

이러한 개선을 통해 `ShoppingBasket` 클래스는 고유한 인스턴스 상태를 유지할 수 있으며, 전역 상태로 인한 예상치 못한 오류를 방지할 수 있습니다.

### 예제 9.10 의존성 주입된 ShoppingBasket

```
class ViewItemWidget {  
    private final Item item;  
    private final ShoppingBasket basket;  
  
    ViewItemWidget(Item item, ShoppingBasket basket) { ←  
        this.item = item;  
        this.basket = basket;  
    }  
    ...  
  
    void addItemToBasket() { ←  
        basket.addItem(item);  
    }  
}  
  
class ViewBasketWidget {  
    private final ShoppingBasket basket;  
  
    ViewBasketWidget(ShoppingBasket basket) { ←  
        this.basket = basket;  
    }  
  
    void displayItems() { ←  
        List<Item> items = basket.getItems();  
        ...  
    }  
}
```

의존성이 주입된 ShoppingBasket

주입된 ShoppingBasket 인스턴스에 대해 호출된다.

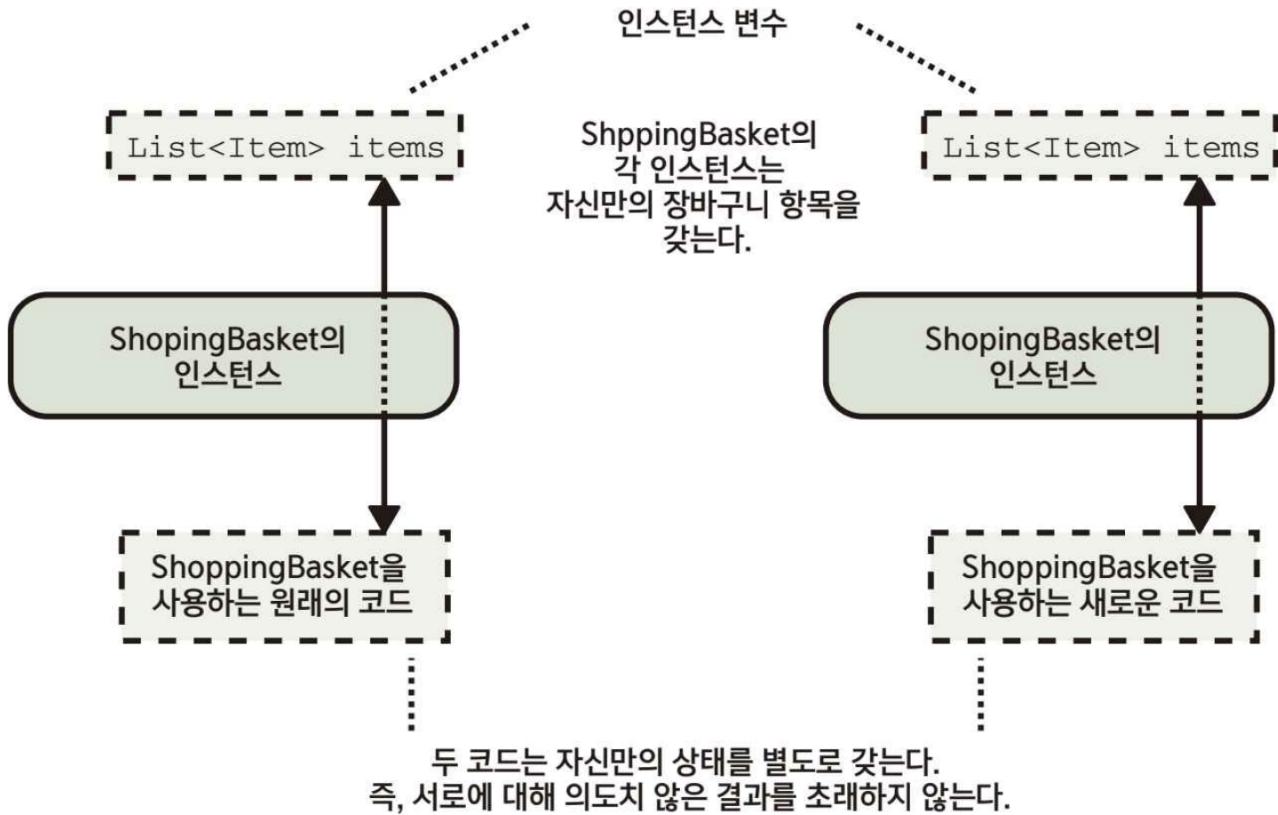


그림 9.2 상태를 클래스 인스턴스 내에 캡슐화함으로써 코드 재사용이 안전해진다.

## 의존성 주입을 통한 상태 제어

- `ShoppingBasket`을 인스턴스로 만들고, 필요한 클래스에 주입하여 사용할 수 있도록 변경합니다.
- 이렇게 하면 코드 A와 코드 B에 서로 다른 `ShoppingBasket` 인스턴스를 주입하여 각자의 상태를 독립적으로 유지할 수 있습니다.
- 예제: 일반 상품용 장바구니와 신선 상품용 장바구니를 각각 따로 생성하고, 각 장바구니에는 독립적인 `ViewBasketWidget`을 연결하여 서로 간섭하지 않도록 합니다.

## 느낀점

이번 내용을 통해 전역 상태가 코드의 유지보수성과 재사용성에 큰 영향을 미친다는 점을 다시금 실감했습니다. 특히, 정적 필드나 메서드를 통해 상태를 공유하는 것은 편리해 보이지만, 실제로는 예기치 못한 문제를 초래할 수 있어 웬만하면 피해야 한다는 생각이 들었습니다.

## 9.3 기본 반환값을 적절하게 사용하라

합리적인 기본 값을 설정하는 것은 사용자 친화적인 소프트웨어를 만드는 좋은 방법입니다. 프로그램이 실행될 때마다 매번 설정을 지정해야 한다면, 사용자에게 불편을 초래할 수 있으며 사용성 측면에서 좋지 않습니다.



워드 프로세서를 개발한다고 가정해 봅시다. 사용자가 워드 작업을 바로 시작할 수 있도록 텍스트 스타일링에 기본 값을 설정해 두었습니다. 사용자가 기본 설정을 원하지 않는 경우, 이를 재설정할 수도 있습니다.

UserDocumentSettings 클래스는 특정 문서에 대한 사용자 환경 설정을 저장하는 역할을 하며, 여기에서 기본 글꼴을 선택할 수 있습니다. 사용자가 글꼴을 지정하지 않으면 getPreferredFont() 메서드는 기본 값으로 Font.ARIAL을 반환합니다.

이 방식은 초기 설정 요건을 충족하지만, 사용자가 기본 글꼴로 Arial을 원하지 않을 경우

UserDocumentSettings 클래스를 다시 사용할 때 문제가 발생할 수 있습니다. 사용자가 명시적으로 Arial을 선택한 것인지, 기본 값으로 설정된 것인지 구분할 수 없기 때문입니다.

## 재사용성과 적응성의 문제

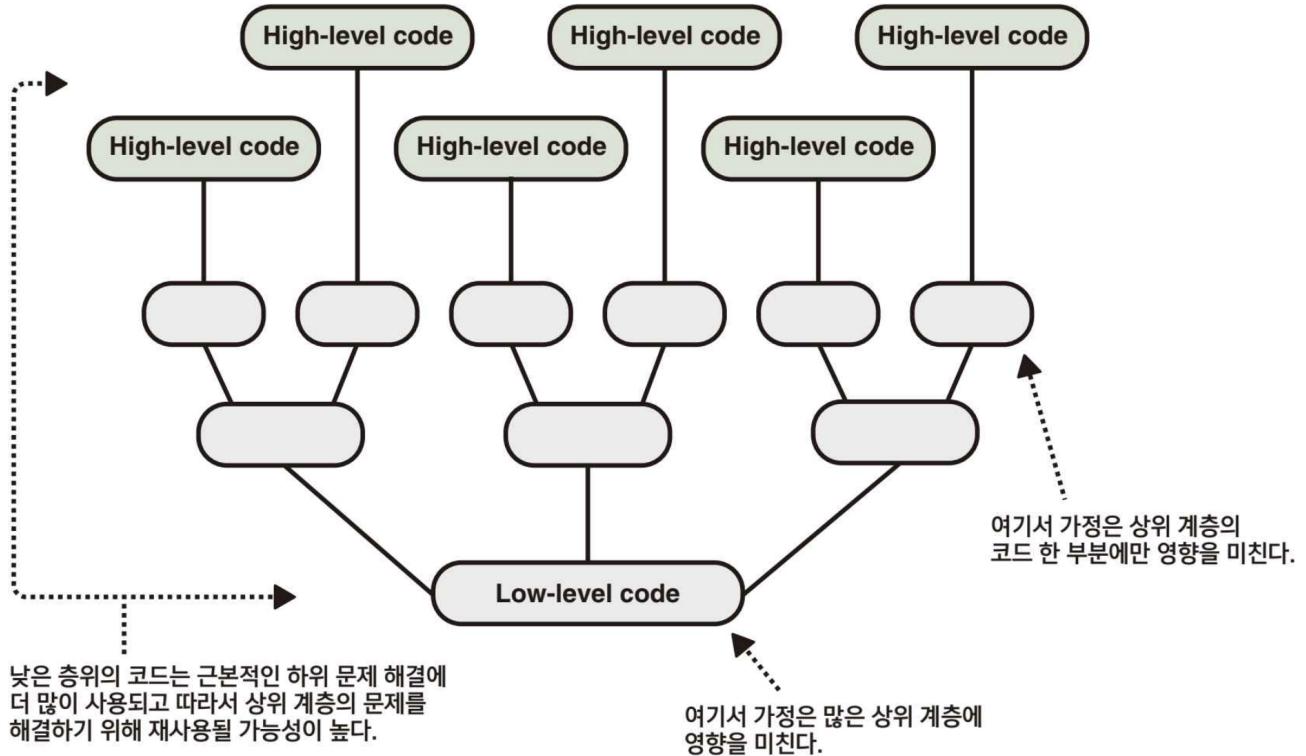
이러한 방식은 프로그램의 적응성을 저해할 수 있습니다. 예를 들어, 이 워드 프로세서를 대기업에 판매했는데, 이 회사가 기본 글꼴을 전역적으로 지정하고자 하는 경우 문제가 됩니다. UserDocumentSettings 클래스에서 회사 전체 기본 값이 적용되어야 하는 경우와 사용자가 별도로 설정한 경우를 구분할 수 없기 때문입니다.

결론적으로, 기본 반환 값을 UserDocumentSettings 클래스에 결합해 두면, 모든 상위 계층 코드에 Arial이 기본 글꼴로 설정된다는 가정을 하게 됩니다. 처음에는 문제가 없을 수 있지만, 다른 개발자가 코드를 재사용하거나 요건이 변경되면 이러한 가정은 쉽게 문제가 될 수 있습니다.

2장은 간결한 추상화 계층의 이점을 강조했습니다. 이를 위한 주요 방법 중 하나는 서로 다른 하위 문제들을 여러 코드로 분리하는 것입니다. 그러나 UserDocumentSettings 클래스는 이러한 원칙을 위반하고 있습니다.

사용자 환경 설정을 검색하는 것과 프로그램에 적합한 기본 값을 정의하는 것은 별개의 하위 문제입니다. 하지만 UserDocumentSettings 클래스는 이 두 기능을 분리할 수 없는 방식으로 함께 묶고 있습니다.

이로 인해 UserDocumentSettings 클래스를 사용하는 모든 사용자는 기본 값 구현을 사용할 수밖에 없습니다. 더 나은 방법은 이 두 가지를 별개의 하위 문제로 분리하여, 상위 계층의 코드가 자신에게 적합한 방식으로 기본 값을 처리할 수 있도록 하는 것입니다.



**그림 9.3** 가정은 상위 계층에 영향을 미친다. 낮은 층위의 코드에서 기본값을 반환하면 높은 층위의 코드에 영향을 미치는 가정을 하는 것이 된다.

#### 낮은 계층의 코드에서 기본 값을 설정할 때 재사용성 문제가 생기는 이유

낮은 계층의 코드에서 기본 값을 설정하면, 해당 설정이 상위 계층 코드 전체에 적용됩니다. 이로 인해 다음과 같은 문제가 발생합니다.

1. **상위 계층의 재사용성 제한:** 상위 계층이 특정 기본 값에 종속되면서 유연성이 떨어집니다. 새로운 상황이나 요구사항에 맞춰 기본 값을 변경해야 할 때 어려움이 발생할 수 있습니다.
2. **변화에 대한 민감성:** 낮은 계층에서 기본 값을 설정한 경우, 기본 값을 수정해야 하는 요구사항이 생길 때 모든 계층에 영향을 미칩니다. 이를 위해 코드 전반을 수정해야 할 가능성이 높아지고, 이는 코드 유지보수의 복잡성을 증가시킵니다.

#### 9.3.2 해결책: 상위 수준의 코드에서 기본값을 제공하라

### 예제 9.13 널값 반환

```
class UserDocumentSettings {  
    private final Font? font;  
    ...  
  
    Font? getPreferredFont() {  
        return font;  
    }  
}
```

사용자가 선호하는 폰트가 없는 경우  
널값을 반환한다.

기본 값을 UserDocumentSettings 클래스에서 직접 결정하지 않도록 하는 간단한 방법은, 사용자가 제공한 값이 없을 때 `null`을 반환하는 것입니다. 이렇게 하면 기본 값 제공과 사용자 설정 처리는 별개의 하위 문제로 분리됩니다.

### 예제 9.14 기본값을 캡슐화하기 위한 클래스

```
class DefaultDocumentSettings {  
    ...  
  
    Font getDefaultFont() {  
        return Font.ARIAL;  
    }  
}
```

### 예제 9.15 설정에 대한 추상화 계층 나열

```
class DocumentSettings {  
    private final UserDocumentSettings userSettings;  
    private final DefaultDocumentSettings defaultSettings;  
  
    DocumentSettings(  
        UserDocumentSettings userSettings,  
        DefaultDocumentSettings defaultSettings) {  
        this.userSettings = userSettings;  
        this.defaultSettings = defaultSettings;  
    }  
    ...  
  
    Font getFont() {  
  
        Font? userFont = userSettings.getPreferredFont();  
        if (userFont != null) {  
            return userFont;  
        }  
        return defaultSettings.getFont();  
    }  
}
```

] 사용자 설정과 기본값은  
의존성 주입으로 제공된다.

상위 계층 코드에서는 기본 값을 제공하는 별도의 전용 클래스를 정의하여, 기본 값 설정에 집중하고 여러 위치에서 필요에 따라 적절한 기본 값을 손쉽게 설정할 수 있습니다.

DocumentSettings 클래스는 기본 값과 사용자 설정 값을 구체적으로 구현 및 관리하되, 외부에서 설정을 주입받아 필요한 부분만 변경할 수 있게 설계되었습니다. 이러한 설계를 통해 호출 코드에서는 원하는 방식으로 기본 값을 유연하게 설정할 수 있어 재사용성이 더욱 높아집니다!

널 병합 연산자는 자바에 없어요...

## 기본 반환값 매개변수

어떤 기본값을 사용할지에 대한 결정을 호출하는 쪽에서 하도록 하면 코드의 재사용성이 향상된다. 그러나 널 병합 연산자를 지원하지 않는 언어에서 널을 반환하면 호출하는 쪽에서는 널 처리를 위한 반복적인 코드를 작성해야 한다.

코드의 일부에서 사용하는 한 가지 방법은 기본 반환값 매개변수를 사용하는 것이다. 이에 대한 예로 자바의 `Map.getOrDefault()` 함수가 있다. 맵에 키값이 있으면 그 값이 반환되지만, 키값이 없는 경우 지정된 기본값이 반환된다. 이 함수에 대한 호출은 다음과 같이 할 수 있다.

```
String value = map.getOrDefault(key, "default value");
```

이렇게 하면 호출하는 쪽에서는 널값을 처리할 필요 없이 적절한 기본값을 결정할 수 있다.

예제 9.15에서는 `if`문을 이용해서 널값을 처리하고 있다. 자바도 널 병합 연산자를 제공하지 않기 때문에 `if`문을 이용해야 하는데, 이럴 경우 호출하는 쪽에서 `null` 처리를 해야합니다. 이때 "기본 반환값 매개변수"를 사용하면 호출하는 쪽에서 널값을 처리할 필요 없이 적절한 기본값을 결정을 할 수 있습니다!!

하지만, 기본 값을 반환하는 코드를 작성할 때는 그 값을 어디서 사용할지 주의하는 것이 좋습니다.

기본 값을 반환하면, 상위 계층의 코드에서 해당 값을 사용할 것이라는 가정을 하게 되어 코드의 재사용성과 적응성을 제한할 수 있습니다.

특히, 낮은 계층의 코드에서 기본 값을 반환하는 것은 문제가 될 수 있습니다. 이런 경우에는 단순히 `null`을 반환하고, 상위 계층에서 기본 값을 정의하는 것이 더 나을 수 있는데, 이렇게 하면 상위 계층에서 설정한 가정이 더 적합할 가능성이 높습니다.

### 느낀점

기본 값을 설정할 때 낮은 계층에서 이를 정의하는 것보다, 상위 계층에서 설정할 수 있도록 하는 것이 코드의 유연성과 재사용성에 훨씬 유리하다는 점을 배웠습니다.

## 9.4 함수의 매개변수를 주목하라

## 예제 9.16 TextOptions 클래스 나열

```
class TextOptions {  
    private final Font font;  
    private final Double fontSize;  
    private final Double lineHeight;  
    private final Color textColor;
```

여러 가지 스타일링 옵션을  
하나로 묶어 캡슐화한다.

```
TextOptions(  
    Font font,  
    Double fontSize,  
    Double lineHeight,  
    Color textColor) {  
    this.font = font;  
    this.fontSize = fontSize;  
    this.lineHeight = lineHeight;  
    this.textColor = textColor;  
}
```

```
Font getFont() { return font; }  
Double getFontSize() { return fontSize; }  
Double getLineHeight() { return lineHeight; }  
Color getTextColor() { return textColor; }  
}
```

함수가 객체나 클래스에 포함된 모든 정보를 필요로 하는 경우에는, 해당 객체나 클래스의 인스턴스를 매개변수로 받는 것이 바람직합니다. 이렇게 하면 매개변수의 수가 줄어들고, 캡슐화된 데이터의 세부사항을 일일이 처리하지 않아도 되기 때문입니다.

하지만 함수가 필요한 정보가 한두 가지에 불과한 경우, 객체나 클래스의 인스턴스를 통째로 매개변수로 사용하는 것은 코드의 재사용성을 떨어뜨릴 수 있습니다.

### 9.4.1 필요 이상으로 매개변수를 받는 함수는 재사용하기 어려울 수 있다.

### 예제 9.17 필요 이상의 매개변수를 받는 함수

```
class TextBox {  
    private final Element textContainer;  
    ...  
  
    void setTextStyle(TextOptions options) {  
        setFont(...);  
        setFontSize(...);  
        setLineHeight(...);  
        setTextColor(options); // setTextColor() 함수를  
        // 호출한다.  
    }  
  
    void setTextColor(TextOptions options) { // TextOptions 클래스의 인스턴스를  
        // 매개변수로 받는다.  
        textContainer.setStyleProperty(  
            "color", options.getTextColor().asHexRgb()); // 텍스트 색상만 사용한다.  
    }  
}
```

## 예제 시나리오

다음은 텍스트 상자 위젯(TextBox)에서 텍스트 스타일을 설정하는 코드입니다. 이 코드는 TextBox 클래스에 `setTextStyle()` 과 `setTextColor()` 라는 두 개의 public 메서드를 포함하고 있으며, 이 메서드들은 TextOptions 인스턴스를 매개변수로 받습니다.

- `setTextStyle()` : TextOptions에 포함된 모든 정보를 필요로 하므로, TextOptions를 매개변수로 사용하는 것이 적절합니다.
- `setTextColor()` : TextOptions에서 텍스트 색상 정보만 필요로 합니다. 이 메서드는 텍스트 색상만을 사용하기 때문에 필요 이상의 데이터를 포함한 TextOptions 전체를 매개변수로 받고 있습니다.

현재 `setTextColor()` 는 `setTextStyle()` 내에서만 호출되므로 큰 문제가 되지 않지만, 다른 곳에서 재사용하려고 하면 불필요한 데이터가 포함되어 어려움이 발생할 수 있습니다.

## 문제 상황 예시

## 예제 9.18 너무 많은 매개변수를 받는 함수

```
void styleAsWarning(TextBox textBox) {  
    TextOptions style = new TextOptions(  
        Font.ARIAL,  
        12.0,  
        14.0  
        Color.RED);  
    textBox.setTextColor(style);  
}
```

관련 없고  
일부러 만들어낸 값

예를 들어, TextBox에 경고 메시지 스타일을 적용하는 함수를 구현한다고 가정해 보겠습니다. 이 함수는 텍스트 색상만 빨간색으로 설정하고 다른 모든 스타일 정보는 그대로 유지해야 합니다. setTextColor()를 사용하려고 할 때, 텍스트 색상만 필요함에도 불구하고 TextOptions 전체 인스턴스를 생성해야 합니다. 이렇게 하면 코드가 혼란스러워지고, 텍스트 색상만 설정하는 의도와는 달리 다른 스타일 속성(예: 글꼴, 크기, 줄 높이 등)도 포함되는 것처럼 보일 수 있습니다.

### 9.4.2 해결책: 함수는 필요한 것만 매개변수로 받도록 하라

### 예제 9.19 필요한 것만 받는 함수

```
class TextBox {  
    private final Element textElement;  
    ...  
  
    void setTextStyle(TextOptions options) {  
        setFont(...);  
        setFontSize(...);  
        setLineHeight(...);  
        setTextColor(options.getTextColor());  
    }  
  
    void setTextColor(Color color) {  
        textElement.setStyleProperty("color", color.asHexRgb());  
    }  
}
```

setTextColor() 함수는 텍스트 색상만으로 호출한다.

매개변수로 Color 인스턴스를 받는다.

TextBox.setTextColor() 메서드는 TextOptions 객체 전체가 아닌, Color 객체 하나만 매개변수로 받도록 개선한 코드입니다.

```
void styleAsWarning(TextBox textBox) {  
    textBox.setTextColor(Color.RED);  
}
```

이렇게 하면 함수가 단순해지고 코드가 더 명확해지며, 관련 없는 값을 생성하여 객체를 만드는 과정을 피할 수 있습니다.

일반적으로 함수가 필요한 정보만 받도록 설계하면, 함수 재사용성이 높아지고 이해하기 쉬워집니다. 불필요한 데이터가 포함되지 않아 혼란을 줄일 수 있습니다.

하지만 상황에 따라 캡슐화된 전체 객체를 전달하는 것도 합리적일 수 있습니다. 예를 들어, 캡슐화된 객체에 10개의 속성이 있지만 그 중 8개를 필요로 하는 함수가 있다면, 객체 전체를 전달하는 편이 오히려 모듈성을 유지하는데 더 도움이 될 수 있습니다. 이는 모듈성을 손상시키지 않고, 필요한 정보를 유지할 수 있는 방법입니다.

## 느낀점

가장 많이 도움이 된 챕터입니다. 항상 매개변수로 객체를 줄지, 객체 내에서 사용중인 필드를 줄지 고민을 하곤 했었는데 이번 챕터를 읽고 궁금증이 해결이 되었습니다.

앞으로는 절반 이상의 필드를 필요로 하는 경우에는 객체를 인자로, 절반 미만의 필드를 필요로 할 때는 개별 필드를 인자로 넘기는 방식을 적용하고자 합니다!

## 9.5 제네릭의 사용을 고려해라

클래스는 종종 다른 유형의 인스턴스나 참조를 포함합니다. 그 대표적인 예가 **리스트(List)** 클래스입니다. 예를 들어, 문자열을 저장하는 리스트가 필요한 경우, 리스트 클래스는 문자열 객체를 저장하며, 특정 경우에는 문자열 리스트가 필요하고 다른 경우에는 정수 리스트가 필요할 수 있습니다. 만약 문자열과 정수를 저장하기 위해 완전히 별개의 리스트 클래스를 작성해야 한다면 이는 매우 번거롭습니다. (ex.

```
IntegerList<Integer>, StringList<String> )
```

다행히도 많은 프로그래밍 언어가 **제네릭(generic)**이나 **템플릿(template)**을 지원하여, 참조하는 모든 타입을 구체적으로 명시하지 않고도 클래스를 작성할 수 있습니다. 이를 통해 리스트 클래스 하나로 다양한 타입의 요소를 저장할 수 있으며, 예시는 다음과 같습니다.

```
List<String> stringList = Arrays.asList("hello", "world");
List<Integer> intList = Arrays.asList(1, 2, 3);
```

### Note

**Tip:** 다른 클래스에 의존하지만 그 클래스가 무엇인지 신경 쓰지 않는다면 제네릭 사용을 고려해야 합니다. 제네릭을 사용하면 코드가 일반화되어 재사용성이 크게 향상됩니다.

### 9.5.1 특정 유형에 의존하면 일반화를 제한한다.

단어 맞히기 게임: 이 게임은 참가자들이 단어를 제시하면, 다른 참가자들이 그 단어를 맞히는 방식입니다. 예를 들어, 한 참가자가 “사과”라는 단어를 제시하면, 나머지 참가자들이 그 단어를 설명을 듣고 맞히려고 하는 상황입니다.

이때 해결해야 할 하위 문제 중 하나는 단어 모음을 저장하고 관리하는 것\*\*입니다. 또한 각 단어를 무작위로 선택할 수 있어야 하고, 시간을 초과하면 그 단어는 다시 모음으로 돌아가도록 설정해야 합니다.

### 예제 9.20 문자열 유형을 하드 코딩해서 사용

```
class RandomizedQueue {  
    private final List<String> values = [];  
  
    void add(String value) {  
        values.add(value);  
    }  
  
    /**  
     * 큐로부터 무작위로 한 항목을 삭제하고 그 항목을 반환한다.  
     */  
    String? getNext() {  
        if (values.isEmpty()) {  
            return null;  
        }  
        Int randomIndex = Math.randomInt(0, values.size());  
        values.swap(randomIndex, values.size() - 1);  
        return values.removeLast();  
    }  
}
```

하드 코드로  
String을 사용

RandomizedQueue 클래스: 게임에서 필요한 기능 중 하나는 여러 단어를 저장하고, 그 중 무작위로 하나를 꺼내는 기능입니다. 이 때 단어를 모아 두는 클래스로 RandomizedQueue 가 있다고 가정해봅시다.

이 클래스는 add() 메서드로 새로운 단어를 추가하고, getNext() 메서드로 무작위 단어를 꺼내며, 꺼낸 단어는 목록에서 제거합니다.

그런데 RandomizedQueue 가 문자열(**String**)에만 의존하도록 설계되었다면, 이 클래스는 오직 문자열로만 된 단어들만 저장할 수 있습니다. 예를 들어, 사진을 사용해서 설명하는 게임이 필요할 경우, 단어 대신 사진 객체(이미지 파일 등)를 RandomizedQueue 에 추가할 수 없어 문제가 생깁니다.

### 9.5.2 해결책: 제네릭을 사용하라

## 예제 9.21 제네릭 사용

```
class RandomizedQueue<T> {           ← T는 제네릭의 유형에 대한  
    private final List<T> values = [];  
  
    void add(T value) {  
        values.add(value);  
    }  
  
    /**  
     * 큐에서 무작위로 한 항목을 삭제한 후에 그 항목을 반환한다.  
     */  
    T? getNext() {  
        if (values.isEmpty()) {  
            return null;  
        }  
        Int randomIndex = Math.randomInt(0, values.size());  
        values.swap(randomIndex, values.size() - 1);  
        return values.removeLast();  
    }  
}
```

T는 제네릭의 유형에 대한  
자리 표시자다.

유형 자리 표시자는  
클래스 내에서  
사용할 수 있다.

제네릭(Generic)을 사용하면 클래스가 특정 데이터 타입에 종속되지 않고, 여러 유형의 데이터를 유연하게 처리할 수 있습니다!

예를 들어, RandomizedQueue라는 클래스가 문자열만 저장하도록 설계되었다면, 이 클래스를 재사용하여 다른 유형의 데이터를 저장하려면 수정이 필요하게 됩니다. 하지만 제네릭을 사용하면 이러한 문제를 해결할 수 있습니다.

RandomizedQueue<T>와 같이 제네릭 타입 T를 설정하면, 이 클래스는 문자열뿐 아니라 사진, 정수, 사용자 정의 객체 등 어떤 데이터 유형도 저장할 수 있는 범용적인 클래스가 됩니다. 이때 컴파일러는 T가 사용할 데이터 유형이라는 것을 알고 처리하므로, 특정 유형에 종속되지 않는 유연한 코드가 완성됩니다.

예를 들어, 문자열을 저장하고 싶다면 RandomizedQueue<String> words = new RandomizedQueue<String>()로 선언할 수 있고, 사진을 저장하고 싶다면 RandomizedQueue<Picture> pictures = new RandomizedQueue<Picture>()와 같이 선언할 수 있습니다. 이 방식으로 RandomizedQueue는 타입에 제한되지 않고 여러 곳에서 재사용될 수 있습니다.

## 제네릭 및 널 형식

예제 9.21 코드에서 `getNext()` 함수는 큐가 비어 있으면 널을 반환한다. 널값을 큐에 저장하지 않는 한 이 방법은 문제가 없고 이렇게 가정하는 것은 합리적이다(3장에서 논의한 것처럼 확인(check)이나 어서션(assertion)을 사용해 이 가정을 강제로 적용하는 것을 고려할 수 있다).

`RandomizedQueue<String?>`과 같이 정의하고 널값을 대기열에 저장하려 한다면 문제가 될 수 있다. `getNext()`가 널값을 반환할 때 이것이 큐 내에 존재하는 널값인지 아니면 큐가 비어 있다는 것인지 구별할 수 없기 때문이다. 이런 경우를 지원하고자 한다면 `getNext()`를 호출하기 전에 큐가 비어 있지 않은지 확인할 수 있는 `hasNext()` 함수를 제공할 수 있다.

## 느낀점

이 챕터는 제네릭 사용의 중요성을 이해하는 데 큰 도움이 되었습니다.

```
package com.beat.global.common.dto;

import com.beat.global.common.exception.base.BaseSuccessCode;

public record SuccessResponse<T>(
    int status,
    String message,
    T data
) {
    public static <T> SuccessResponse<T> of(final BaseSuccessCode
baseSuccessCode, final T data) {
        return new SuccessResponse<>(baseSuccessCode.getStatus(),
baseSuccessCode.getMessage(), data);
    }

    public static <T> SuccessResponse<T> from(final BaseSuccessCode
baseSuccessCode) {
        return new SuccessResponse<>(baseSuccessCode.getStatus(),
baseSuccessCode.getMessage(), null);
    }
}
```

저는 보통 세분화 된 에러나 성공 Response에 대한 데이터를 DTO로 주곤 합니다.

그때, 제네릭 타입을 사용하는데, 만약 제네릭 타입을 사용하지 않을 경우 각 DTO 별로 `SuccessResponse`를 새롭게 구축해야 합니다. 하지만 이렇게 하면 하나의 `SuccessResponse` 클래스 내에서 여러 DTO를 응답값으로 줄 수 있게 됩니다!

## 요약

1. 코드 재사용의 이점: 동일한 하위 문제가 자주 발생하기 때문에, 코드를 재사용하면 미래의 자신과 팀원의 시간과 노력을 절약할 수 있다.
2. 근본적인 하위 문제 해결: 다른 개발자가 해결하려는 문제와는 상위 수준이 다르더라도, 특정 하위 문제에 대해 작성한 해결책을 재사용할 수 있도록 기본적인 하위 문제를 식별하고, 재사용 가능한 코드로 구성하는 데 집중해야 한다.
3. 추상화 계층과 모듈화: 간결한 추상화 계층을 만들고 코드를 모듈화하면, 코드 재사용과 일반화가 훨씬 쉬워지고 안전해진다.
4. 가정의 위험성: 코드를 작성할 때 가정을 하게 되면, 코드는 더 취약해지고 재사용이 어려워질 수 있다.
  - 가정의 이점이 비용보다 큰지 판단해야 한다.
  - 가정을 강제적으로 적용할 때는, 그 가정이 적절한 계층에서 이루어졌는지 확인하라.
5. 전역 상태 사용의 위험성: 전역 상태는 많은 비용을 수반하는 가정을 필요로 하고, 재사용에 매우 위험한 코드를 만든다. 대부분의 경우 전역 상태를 피하는 것이 최선이다.