11. 디폴트 메서드

#1.인강/0.자바/7.자바-고급3편

디폴트 메서드가 등장한 이유

자바 8에서 **디폴트 메서드(default method)** 가 등장하기 전에는 **인터페이스에 메서드를 새로 추가**하는 순간, 이미 배포된 기존 구현 클래스들이 해당 메서드를 구현하지 않았기 때문에 전부 컴파일 에러를 일으키게 되는 문제가 있었다. 이 때문에 특정 인터페이스를 이미 많은 클래스에서 구현하고 있는 상황에서, 인터페이스에 새 기능을 추가하려면 기존 코드를 일일이 모두 수정해야 했다.
디폴트 메서드는 이러한 문제를 해결하기 위해 등장했다. 자바 8부터는 **인터페이스에서 메서드 본문을 가질 수 있도록** 허용해 주어, **기존 코드를 깨뜨리지 않고 새 기능을 추가**할 수 있게 되었다.

다음 예제들을 통해 왜 이런 기능이 필요했고, 어떻게 해결되는지 단계별로 살펴보자.

예제1

인터페이스와 구현 클래스

먼저 기존 코드에서 알림 기능을 처리하는 `Notifier` 인터페이스와 세 가지 구현체(`EmailNotifier`, `SMSNotifier`, `AppPushNotifier`)가 있다고 하자. `Notifier`는 단순히 메시지를 알리는 `notify()` 메서드 한 가지만 정의하고 있고, 각 구현체는 해당 기능을 구현한다.

```
package defaultmethod.ex1;

public interface Notifier {
    // 알림을 보내는 기본 기능
    void notify(String message);
}
```

```
package defaultmethod.ex1;
```

```
public class EmailNotifier implements Notifier {
    @Override
    public void notify(String message) {
        System.out.println("[EMAIL] " + message);
    }
}
```

```
package defaultmethod.ex1;

public class SMSNotifier implements Notifier {
    @Override
    public void notify(String message) {
        System.out.println("[SMS] " + message);
    }
}
```

```
package defaultmethod.ex1;

public class AppPushNotifier implements Notifier{
    @Override
    public void notify(String message) {
        System.out.println("[APP] " + message);
    }
}
```

다음은 이 기능을 사용하는 코드이다.

```
package defaultmethod.ex1;

import java.util.List;

public class NotifierMainV1 {
    public static void main(String[] args) {
        List<Notifier> notifiers = List.of(new EmailNotifier(), new
SMSNotifier(), new AppPushNotifier());
        notifiers.forEach(n -> n.notify("서비스 가입을 환영합니다!"));
```

```
    }
}
```

실행 결과

```
[EMAIL] 서비스 가입을 환영합니다!
[SMS] 서비스 가입을 환영합니다!
[APP] 서비스 가입을 환영합니다!
```

지금까지는 문제없이 잘 작동한다. 이제 요구사항을 추가해보자.

예제2

예제2를 시작하기 전에 예제1에 있는 `defaultmethod.ex1` 코드를 모두 복사해서 `defaultmethod.ex2`에 넣어두자.

복사한 후에 `ex2` 패키지에 있는 다음 클래스의 이름을 변경하자.

- `NotifierMainV1` ➜ `NotifierMainV2`
- `ex1` 패키지 대신에 `ex2` 패키지의 내용을 사용하도록 **import를 주의해야 한다!**

인터페이스에 새로운 메서드 추가 시 발생하는 문제

요구사항이 추가되었다. 알림을 **미래의 특정 시점**에 자동으로 발송하는 스케줄링 기능을 추가해야 한다고 해보자. 그래서 `Notifier` 인터페이스에 `scheduleNotification()` 메서드를 추가하자.

```
package defaultmethod.ex2;

import java.time.LocalDateTime;

public interface Notifier {
    // 알림을 보내는 기본 기능
    void notify(String message);

    // 신규 기능 추가
    void scheduleNotification(String message, LocalDateTime scheduleTime);

}
```

- `scheduleNotification()` 기능이 추가되었다.

이제 기존의 `EmailNotifier`, `SMSNotifier`, `AppPushNotifier` 클래스를 살펴보자.

```
package defaultmethod.ex2;

import java.time.LocalDateTime;

public class EmailNotifier implements Notifier {
    @Override
    public void notify(String message) {
        System.out.println("[EMAIL] " + message);
    }

    @Override
    public void scheduleNotification(String message, LocalDateTime
scheduleTime) {
        System.out.println("[EMAIL 전용 스케줄링] message: " + message + ", time:
" + scheduleTime);
    }
}
```

- 이렇게 `EmailNotifier`는 새로운 메서드를 구현했다.

```
package defaultmethod.ex2;

public class SMSNotifier implements Notifier {
    @Override
    public void notify(String message) {
        System.out.println("[SMS] " + message);
    }
}
```

```
package defaultmethod.ex2;

public class AppPushNotifier implements Notifier {
    @Override
    public void notify(String message) {
        System.out.println("[APP] " + message);
    }
```

```
}
```

하지만 `SMSNotifier`, `AppPushNotifier` 는 아직 `scheduleNotification()` 을 구현하지 않았기 때문에 컴파일 오류가 발생한다.

```
package defaultmethod.ex2;

import java.time.LocalDateTime;
import java.util.List;

public class NotifierMainV2 {
    public static void main(String[] args) {
        List<Notifier> notifiers = List.of(new EmailNotifier(), new
SMSNotifier(), new AppPushNotifier());
        notifiers.forEach(n -> n.notify("서비스 가입을 환영합니다!"));

        // 스케줄 기능 추가
        LocalDateTime plus1Days = LocalDateTime.now().plusDays(1);
        notifiers.forEach(n -> {
            n.scheduleNotification("hello!", plus1Days);
        });
    }
}
```

- 참고: `Collection` 도 `forEach()` 메서드를 통한 내부 반복을 제공한다.

실행 결과 - 컴파일 오류

```
java: defaultmethod.ex2.AppPushNotifier is not abstract and does not override
abstract method scheduleNotification(java.lang.String,java.time.LocalDateTime)
in defaultmethod.ex2.Notifier

java: defaultmethod.ex2.SMSNotifier is not abstract and does not override
abstract method scheduleNotification(java.lang.String,java.time.LocalDateTime)
in defaultmethod.ex2.Notifier
```

`scheduleNotification()` 메서드가 `Notifier` 인터페이스에 새로 추가됨에 따라, 기존에 존재하던 `SMSNotifier`, `AppPushNotifier` 구현 클래스들이 **강제로** 이 메서드를 구현하도록 요구된다.

- 규모가 작은 예제에서는 그럼 나머지 두 클래스도 재정의하면 된다고 생각할 수 있다.
- 하지만 실무 환경에서 **해당 인터페이스를 구현한 클래스가 수십~수백 개**라고 한다면, 이 전부를 수정해서 새 메서드를 재정의해야 한다.
- 심지어 우리가 만들지 않은, **외부 라이브러리**에서 `Notifier`를 구현한 클래스가 있다면 그것까지 전부 깨질 수 있어, 호환성을 깨뜨리는 매우 심각한 문제가 된다.

예제3 - Notifier 수정

디폴트 메서드로 문제 해결

자바 8부터 이러한 **하위 호환성** 문제를 해결하기 위해 **디폴트 메서드**가 추가되었다. 인터페이스에 메서드를 새로 추가하면서, **기본 구현**을 제공할 수 있는 기능이다. 예를 들어, `Notifier` 인터페이스에 `scheduleNotification()` 메서드를 **default** 키워드로 작성하고 기본 구현을 넣어두면, 구현 클래스들은 이 메서드를 굳이 재정의하지 않아도 된다.

```
package defaultmethod.ex2;

import java.time.LocalDateTime;

public interface Notifier {
    // 알림을 보내는 기본 기능
    void notify(String message);

    // 신규 기능 추가
    //void scheduleNotification(String message, LocalDateTime
scheduleTime); // 주석

    // default 키워드를 사용해 기본 구현 제공
    default void scheduleNotification(String message, LocalDateTime
scheduleTime) {
        System.out.println("[기본 스케줄링] message: " + message + ", time: " +
scheduleTime);
    }

}
```

이제 `EmailNotifier`처럼 재정의한 특별한 로직을 쓰고 싶다면 여전히 재정의하면 되고, `SMSNotifier`, `AppPushNotifier` 처럼 재정의하지 않으면 인터페이스에 작성된 기본 구현을 사용하게 된다.

```
package defaultmethod.ex2;

import java.time.LocalDateTime;

public class EmailNotifier implements Notifier {
    @Override
    public void notify(String message) {
        System.out.println("[EMAIL] " + message);
    }

    @Override
    public void scheduleNotification(String message, LocalDateTime
scheduleTime) {
        System.out.println("[EMAIL 전용 스케줄링] message: " + message + ", time:
" + scheduleTime);
    }
}
```

- `scheduleNotification()` : 직접 재정의

```
package defaultmethod.ex2;

public class SMSNotifier implements Notifier {
    @Override
    public void notify(String message) {
        System.out.println("[SMS] " + message);
    }
}
```

- `scheduleNotification()` : 인터페이스의 기본 구현 사용

```
package defaultmethod.ex2;

public class AppPushNotifier implements Notifier {
    @Override
    public void notify(String message) {
        System.out.println("[APP] " + message);
    }
```

```
}
```

- `scheduleNotification()`: 인터페이스의 기본 구현 사용

NotifierMainV2 - 실행 결과

```
[EMAIL] 서비스 가입을 환영합니다!
[SMS] 서비스 가입을 환영합니다!
[APP] 서비스 가입을 환영합니다!
[EMAIL 전용 스케줄링] message: hello!, time: (실행 시점 + 1일)
[기본 스케줄링] message: hello!, time: (실행 시점 + 1일)
[기본 스케줄링] message: hello!, time: (실행 시점 + 1일)
```

- `EmailNotifier`는 `scheduleNotification()` 을 재정의했기 때문에 **[EMAIL 전용 스케줄링] ...** 이 출력
- `SMSNotifier`, `AppPushNotifier` 는 재정의하지 않았기 때문에 **인터페이스의 기본 구현**인 **[기본 스케줄링] ...** 이 출력

결과적으로 **새 메서드가 추가되었음에도 불구하고** 해당 인터페이스를 구현하는 기존 클래스들이 **큰 수정 없이도** (또는 전혀 수정 없이도) 동작을 계속 유지할 수 있게 된다.

디폴트 메서드 소개

자바는 처음부터 인터페이스와 구현을 명확하게 분리한 언어였다.
자바가 처음 등장했을 때부터 인터페이스는 **구현 없이 메서드의 시그니처만을 정의**하는 용도로 사용되었다.

- **인터페이스 목적**: 코드의 계약(Contract)을 정의하고, 클래스가 어떤 메서드를 반드시 구현하도록 강제하여 **명세와 구현을 분리**하는 것이 주된 목적이었다.
- **엄격한 규칙**: 인터페이스에 선언되는 메서드는 기본적으로 모두 추상 메서드(abstract method)였으며, 인터페이스 내에서 구현 내용을 포함할 수 없었다. 오직 `static final` 필드와 `abstract` 메서드 선언만 가능했다.
- **결과**: 이렇게 인터페이스가 엄격하게 구분됨으로써, 클래스는 여러 인터페이스를 구현(implements)할 수 있게 되고, 각각의 메서드는 클래스 내부에서 구체적으로 어떻게 동작할지를 자유롭게 정의할 수 있었다. 이를 통해 객체지향적인 설계와 다형성을 극대화할 수 있었다.

자바 8 이전까지는 인터페이스에 새로운 메서드를 추가하면, 해당 인터페이스를 구현한 모든 클래스에서 그 메서드를 구현해야 했다.

자바가 기본으로 제공하는 수많은 인터페이스를 생각해보자. 예를 들어 `Collection`, `List` 같은 인터페이스는 이미 많은 개발자들이 해당 인터페이스를 구현해서 사용한다. 또는 많은 라이브러리들도 해당 인터페이스를 구현한 구현체를 제공한다. 예를 들어 특정 상황에 최적화된 컬렉션이 있을 수 있다.
이런 상황에서 만약 자바가 버전 업을 하면서 해당 인터페이스에 새로운 기능을 추가한다면 어떻게 될까?
새로운 자바 버전으로 업데이트 하는 순간 전 세계에서 컴파일 오류들이 발생할 것이다.
이런 문제를 방지하기 위해 자바는 하위호환성을 그 무엇보다 큰 우선순위에 둔다.
결국 인터페이스의 이런 엄격한 규칙 때문에, 그 동안 자바 인터페이스에 새로운 기능을 추가하지 못하는 일이 발생하게 되었다.

이런 문제를 해결하기 위해 자바 8에서 디폴트 메서드가 도입되었다. 결과적으로 인터페이스의 엄격함은 약간은 유연하게 변경되었다.

디폴트 메서드의 도입 이유

- **하위 호환성(Backward Compatibility) 보장**: 인터페이스에 새로운 메서드를 추가하더라도, 기존 코드가 깨지지 않도록 하기 위한 목적으로 디폴트 메서드가 도입되었다. 인터페이스에 디폴트 구현(기본 구현)을 제공하면, 기존에 해당 인터페이스를 구현하던 클래스들은 추가로 재정의하지 않아도 정상 동작하게 된다.
- **라이브러리 확장성**: 자바가 제공하는 표준 라이브러리에 정의된 인터페이스(예: `Collection`, `List`)에 새 메서드를 추가하면서, 사용자들이나 서드파티 라이브러리 구현체가 일일이 수정하지 않아도 되도록 만들었다. 이를 통해 자바 표준 라이브러리 자체도 적극적으로 개선할 수 있게 되었다.
  - 예: `List` 인터페이스에 `sort(...)` 메서드가 추가되었지만, 기존의 모든 `List` 구현체를 수정하지 않아도 된다.
- **람다와 스트림 API 연계**: 자바 8에서 함께 도입된 람다(Lambda)와 스트림(Stream) API를 보다 편리하게 활용하기 위해 인터페이스에서 구현 로직을 제공할 필요가 있었다.
  - `Collection` 인터페이스에 `stream()` 디폴트 메서드 추가
  - `Iterable` 인터페이스에 `forEach` 디폴트 메서드 추가
- **설계 유연성 향상**: 디폴트 메서드를 통해 인터페이스에서도 일부 공통 동작 방식을 정의할 수 있게 되었다. 이는 추상 클래스와의 경계를 어느 정도 유연하게 만들지만, 동시에 지나치게 복잡한 기능을 인터페이스에 넣는 것은 오히려 설계를 혼란스럽게 만들 수 있으므로 주의해야 한다.

디폴트 메서드 정리

- 자바 8에 처음 등장한 기능
- 인터페이스 내에서 선언되는 메서드이지만, 몸통(구현부)을 가질 수 있음
- `default` 키워드를 사용
- 추가된 이유: 이미 배포된 인터페이스에 새 메서드를 추가할 때 발생하는 **하위 호환성 문제**를 해결하기 위해

사용 예시

```
```

```java
public interface MyInterface {
    void existingMethod();

    default void newMethod() {
        System.out.println("새로 추가된 디폴트 메서드입니다.");
    }
}
```

- `default void newMethod() { ... }` 형태로 작성
- 구현 클래스에서는 필요한 경우에만 재정의하면 됨
- 재정의하지 않으면 **인터페이스**에 작성한 기본 구현이 그대로 사용됨

디폴트 메서드의 올바른 사용법

디폴트 메서드는 강력한 기능이지만, 잘못 사용하면 오히려 코드가 복잡해지고 유지보수하기 어려워질 수 있다. 다음은 디폴트 메서드를 사용할 때 고려해야 할 주요 사항이다.

1. 하위 호환성을 위해 최소한으로 사용

- 디폴트 메서드는 주로 **이미 배포된 인터페이스**에 새로운 메서드를 추가하면서 기존 구현체 코드를 깨뜨리지 않기 위한 목적으로 만들어졌다.
- 새 메서드가 필요한 상황이고, 기존 구현 클래스가 많은 상황이 아니라면, 원칙적으로는 각각 구현하거나, 또는 추상 메서드를 추가하는 것을 고려하자.
- 불필요한 디폴트 메서드 남용은 코드 복잡도를 높일 수 있다.

2. 인터페이스는 여전히 추상화의 역할

- 디폴트 메서드를 통해 인터페이스에 로직을 넣을 수 있다 하더라도, 가능한 한 로직은 구현 클래스나 별도 클래스에 두고, 인터페이스는 **계약(Contract)의 역할**에 충실한 것이 좋다.
- 디폴트 메서드는 어디까지나 **하위 호환을 위한 기능**이나, **공통으로 쓰기 쉬운 간단한 로직**을 제공하는 정도가 이상적이다.

3. 다중 상속(충돌) 문제

- 하나의 클래스가 여러 인터페이스를 동시에 구현하는 상황에서, **서로 다른 인터페이스에 동일한 시그니처의 디폴트 메서드**가 존재하면 충돌이 일어난다.
- 이 경우 **구현 클래스**에서 반드시 메서드를 재정의해야 한다. 그리고 직접 구현 로직을 작성하거나 또는 어떤 인터페이스의 디폴트 메서드를 쓸 것인지 명시해 주어야 한다.

- 예:

```
interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}

interface B {
    default void hello() {
        System.out.println("Hello from B");
    }
}

public class MyClass implements A, B {
    @Override
    public void hello() {
        // 반드시 충돌을 해결해야 함
        // 1. 직접 구현
        // 2. A.super.hello();
        // 3. B.super.hello();
    }
}
```

4. 디폴트 메서드에 상태(state)를 두지 않기

- 인터페이스는 일반적으로 상태 없이 동작만 정의하는 추상화 계층이다.
- 인터페이스에 정의하는 디폴트 메서드도 "구현"을 일부 제공할 뿐, 인스턴스 변수를 활용하거나, 여러 차례 호출 시 상태에 따라 동작이 달라지는 등의 동작은 지양해야 한다.
- 이런 로직이 필요하다면 클래스(추상 클래스 등)로 옮기는 것이 더 적절하다.

정리

자바 8에서 새롭게 추가된 **디폴트 메서드(default method)** 는 하위 호환성 문제를 해결하기 위해 인터페이스에 메서드 구현부를 둘 수 있도록 한 기능이다. 기존에는 인터페이스에 메서드를 추가할 경우, 이를 구현하는 모든 클래스에서 강제로 메서드를 재정의해야 했으나, 디폴트 메서드로 **기본 구현**을 제공하면 기존 구현 클래스가 바로 깨지지 않고 계속 동작할 수 있다.

- **등장 배경**
  - 기존 인터페이스에 새로운 메서드를 추가하면 이미 배포된 구현체들이 전부 컴파일 에러가 나는 문제가 발생
  - 표준 라이브러리(예: `Collection`, `List`) 역시 이 문제 때문에 쉽게 확장할 수 없었음
  - 디폴트 메서드를 통해 인터페이스에 기본 구현을 제공함으로써, 하위 호환성을 깨뜨리지 않고 기능을 확장 가능

- **사용 예시**

```
public interface MyInterface {
    void existingMethod();
    default void newMethod() {
        System.out.println("새로 추가된 디폴트 메서드입니다.");
    }
}
```

  - `default` 키워드를 통해 메서드 몸통을 인터페이스에 직접 정의
  - 필요한 경우 구현 클래스에서 재정의하면 되고, 재정의하지 않으면 인터페이스의 기본 구현 사용

- **장점**
  - 기존 코드를 수정하지 않고도 인터페이스에 새 기능을 추가 가능
  - 자바 표준 라이브러리를 비롯한 여러 인터페이스가 적극적으로 확장될 수 있음

- **주의사항**
  1. **하위 호환성을 위한 최소한의 활용** – 불필요하게 디폴트 메서드를 남발하면 인터페이스 역할이 애매해지고 복잡해질 수 있음
  2. **인터페이스 본연의 추상화 목적 유지** – 인터페이스는 여전히 계약(Contract) 역할에 충실해야 하며, 복잡한 로직은 클래스 쪽에 두는 것이 바람직
  3. **다중 상속(충돌) 문제** – 여러 인터페이스에서 동일 시그니처의 디폴트 메서드가 있을 경우, 구현 클래스에서 반드시 충돌을 해결해야 함
  4. **상태(state)를 두지 말 것** – 디폴트 메서드는 메서드 구현부만 제공하는 것이며, 인터페이스에 인스턴스 상태를 가지는 것은 지양

결국 디폴트 메서드는 **인터페이스의 확장성을 높이면서도, 기존 코드와의 하위 호환성을 보장**하기 위한 자바의 주요한 언어적 지원이다. 다만 이를 활용할 때는 **인터페이스의 추상화 역할을 흐리지 않도록** 주의 깊은 설계가 필요하다.
따라서 무분별한 사용은 지양하고, 가급적 간단한 공통 기본 동작이나 이미 사용 중인 인터페이스를 확장할 때만 제한적으로 사용하는 것이 좋다.