아이템 20) 추상 클래스보다는 인터페이스를 우선하라

자바가 제공하는 다중 구현 메커니즘 2가지

- 추상클래스
  - 추상클래스가 정의한 타입을 구현한 클래스는 반드시 **추상클래스의 하위클래스가 되어야 함**
  - 자바는 단일 상속만 지원하니까, 이 방식은 새로운 타입을 정의하는 데 큰 제약이 됨
- 인터페이스
  - 선언한 메소드를 모두 정의하고, 일반 규약을 잘 지킨 클래스라면 다른 어떤 클래스를 상속했든 **같은 타입으로 취급함**

인터페이스 장점 활용 1) 믹스인 정의

- `믹스인` : 클래스가 구현할 수 있는 타입으로, 믹스인을 구현한 클래스에 **원래 "주된 타입" 외에도 특정 선택적 행위를 제공한다고 선언하는 효과**를 줌
  - ex) Comparable 은 자신을 구현한 클래스의 인스턴스끼리는 순서를 정할 수 있다고 선언하는 믹스인 인터페이스 / Iterable, Serializable, AutoCloseable…
- 하지만 클래스는 2가지를 상속받을 수 없고 기존 클래스에 덧씌울 수가 없기에 클래스 계층구조에서는 믹스인을 삽입하기 어려움

인터페이스 장점 활용 2) 계층구조가 없는 타입 프레임워크를 만들 수 있다.

```
public class Item20Test {
    interface Singer {
        void Sing();
    }

    interface Songwriter {
        void compose(int chartPosition);
    }

    interface SingerSongWriter extends Singer, Songwriter {
        void strum();
    }
}
```

- 같은 구조를 클래스로 만들려면 가능한 조합 전부를 각각 클래스로 정의한 계층구조가 만들어짐 → 속성이 n 개라면 지원할 조합수는 2^n 개로, `조합 폭발` 이 일어나게 됨

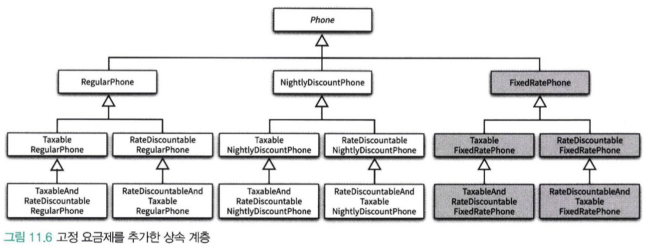

그림 11.6 고정 요금제를 추가한 상속 계층

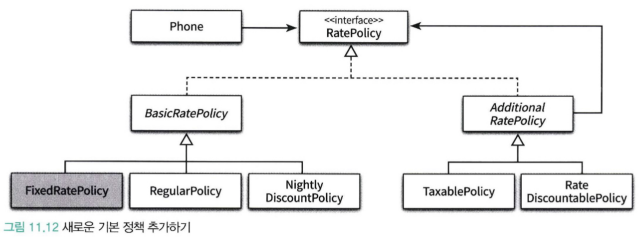

그림 11.12 새로운 기본 정책 추가하기

약정 할인 정책이라는 새로운 부가 정책이 필요한가? 역시 클래스 '하나'만 추가하면 된다.

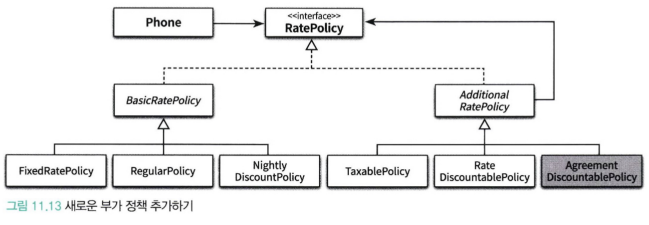

그림 11.13 새로운 부가 정책 추가하기

인터페이스 장점 활용 3) Wrapper 클래스

- 래퍼 클래스 : 기존에 인터페이스를 구현한 클래스를 주입받아 **기존 구현체에 부가기능을 손쉽게 더할 수 있는 클래스**

  → `데코레이터 패턴(Decorator Pattern)`
- 인터페이스 메소드 중 구현 방법이 명확한 경우에는, 디폴트 메소드로 제공하는 것도 가능

인터페이스와 추상 클래스의 장점을 모두 취하는 방법 : 템플릿 메소드 패턴

- 인터페이스와 추상 골격 구현 클래스를 함께 제공하자
  - `인터페이스` : 타입을 정의 (필요한 경우 디폴트 메소드도 제공)
  - `골격 구현 클래스` : 나머지 메서드들까지 구현

  → 템플릿 메소드 패턴
- 명명 관례 : 인터페이스 (Interface) / 골격 구현 클래스 (AbstractInterface)
- 골격 구현 클래스로 구현을 하게 되면, 추상 클래스처럼 구현을 도와주지만 추상클래스로 타입 정의 시 따라오는 제약에서는 자유롭다

[골격 구현 작성 단계]

1. 인터페이스를 살펴, **다른 메서드들의 구현에 사용되는 기반 메소드들을 선정**
   - 기반 메소드들은 골격 구현에서는 `추상 메서드` 가 된다

```
public interface Interface {
  public boolean equals();
  public int getSize();
  public boolean isEmpty();
}
```

2. 기반 메소드들을 사용하여 직접 구현 가능한 메소드를 모두 `디폴트 메소드` 로 제공 (하지만 equals 와 hashCode 같은 Object 의 메소드 제외)
   - ***이 때, 인터페이스 메서드 모두가 기반 메서드와 디폴트 메서드가 된다면 골격 구현 클래스를 별도로 만들 이유가 없음***

```
public interface Interface {
  public boolean equals();
  public int getSize(); // 기반 메서드
  public default boolean isEmpty(){  // 기반 메서드를 통해 만든 default 메서드
    return getSize() > 0;
  }
}
```

3. 기반 메서드나 디폴트 메서드를 통해 만들지 못한 메서드가 있다면, 이 인터페이스를 구현하는 골격 클래스 생성하고 남은 메서드를 작성하여 넣는다.
   - 이 때 구현 클래스가 필요하면 public이 아닌 필드와 메서드를 추가해도 된다.

```
public abstract class AbstractAInterface implements Interface{

  @Override
  public boolean equals(Object obj) {

    ...

    return ..;
  }
}
```

- 구조상 골격 구현을 확장하지 못하는 경우라면, 인터페이스를 직접 구현해야 한다
  - 인터페이스가 직접 제공하는 디폴트 메소드의 이점은 여전히 누릴 수 있음
- 골격 구현 클래스 우회적으로 이용하는 방법

> 인터페이스를 구현한 클래스에서 해당 골격 구현을 확장한 private 내부 클래스를 정의하고, 각 메소드 호출을 내부 클래스 인스턴스에 전달한다

```
// 1. 인터페이스를 구현한 클래스에서 해당 골격 구현을 확장한 private 내부 클래스 정의
class MyExtendedClass implements MyInterface {
    private SkeletonImplementation skeleton;
    public MyExtendedClass() {
        skeleton = new SkeletonImplementation();
    }

    @Override
    public void doSomething() {
        System.out.println("Extended Implementation");

        // 2. 각 메소드 호출을 내부 클래스 인스턴스에 전달
        skeleton.doSomething();
    }
}
```

```
// 인터페이스 정의
interface MyInterface {
    void doSomething();
}

// 골격 구현 클래스 정의
class SkeletonImplementation implements MyInterface {
    @Override
    public void doSomething() {
        System.out.println("Skeleton Implementation");
    }
}
```

정리

- 다중 구현용 타입으로는 `인터페이스` 가 적합
- 골격구현 : “가능한 한” 인터페이스의 디폴트 메소드로 제공하여, 그 인터페이스를 구현한 모든 곳에서 활용하도록 하는 것이 좋음
    - 가능한 한, 이라는 이유는 인터페이스에 걸려 있는 구현상의 제약 (추상 메소드 강제 구현, 상태의 부재) 때문에 골격 구현을 추상 클래스로 제공하는 경우가 더 흔함

1) 인터페이스와 골격 구현

- 인터페이스는 추상 메서드의 집합으로 구현 클래스에 구현을 강제함
- 골격 구현 클래스는 인터페이스를 구현하면서 일부 메서드를 기본 구현으로 제공할 수 있음
- 이 방식은 **여러 클래스가 다른 기본 구현을 공유하거나, 다중 상속을 지원해야 하는 경우에 유용** → 그러나 Java에서는 다중 상속을 클래스 수준에서 지원하지 않으므로, 인터페이스와 골격 구현 클래스를 함께 사용하는 것은 다소 제한적일 수 있음

```
interface MyInterface {
    void doSomething();
}

// 골격 구현 클래스 정의
class SkeletonImplementation implements MyInterface {
    @Override // 인터페이스를 구현
    public void doSomething() {
        System.out.println("Skeleton Implementation");
    }

    public void doSomething2(){ // 일부 메소드 기본 구현으로 제공
        System.out.println("Skeleton Implementation 2");
    }
}

// 인터페이스와 골격 구현 클래스를 사용하는 클래스
class MyUsingClass implements MyInterface {
    private MyInterface skeleton;

    public MyUsingClass() {
        skeleton = new SkeletonImplementation();
    }

    @Override
    public void doSomething() {
```

```
        System.out.println("My Using Class");
        skeleton.doSomething();
    }
}
```

2) 추상클래스와 골격 구현

- 인터페이스와 달리 **일반 메서드와 필드를 포함**할 수 있으며, **구현 클래스에서 반드시 구현해야 하는 메서드를 추상 메서드로 선언 가능**
- 따라서 골격 구현을 추상 클래스로 제공하면 **인터페이스 메서드 외에도 필요한 보조 메서드나 속성을 함께 제공 가능**
    - 이 방식은 인터페이스의 제약을 피하고 구체적인 메서드와 상태를 추가할 필요가 있는 경우 유용

```
interface MyInterface {
    void doSomething();
}

// 추상 클래스를 이용한 골격 구현
abstract class AbstractSkeleton implements MyInterface {
    @Override
    public void doSomething() {
        System.out.println("Abstract Skeleton Implementation");
    }

    // 추상 메서드로 추가적인 동작을 선언
    abstract void additionalOperation();
}

// 구체적인 클래스에서 추상 클래스를 확장
class MyConcreteClass extends AbstractSkeleton {
    @Override
    void additionalOperation() {
        System.out.println("Additional Operation");
    }
}
```