모던자바인액션 12장, 13장

디폴트 메서드

전통적인 자바에서 인터페이스 인터페이스를 구현하는 클래스는 인터페이스에서 정의하는 모든 메서드 구현을 제공하거나 아니면 슈퍼클래스의 구현을 상속받아야 한다. 하지만 자바 8에서는 기본 구현을 포함하는 인터페이스를 정의하는 두 가지 방법을 제공한다.

- 첫 번째는 인터페이스 내부에 정적 메서드를 사용하는 것이고
- 두 번째는 인터페이스의 기본 구현을 제공할 수 있도록 디폴트 메서드 기능을 사용하는 것이다.

자바 8에서는 메서드 구현을 포함하는 인터페이스를 정의할 수 있다 그러면 결과적으로 기존 인터페이스를 구현하는 클래스는 자동으로 인터페이스에 추가된 새로운 메서드의 디폴트 메서드를 상속받게 된다.

예로 List 인터페이스의 sort 메서드를 살펴보자.

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

이 디폴트 메서드 덕분에 리스트에 직접 sort를 호출할 수 있게 되었다.

```
List<Integer> numbers = Arrays.asList( 3, 5, 1, 2, 1, 5, 2);
numbers.sort(Comparator.naturalOrder());
```

변화하는 API

API를 바꾸는 것이 왜 어려운지 예제를 통해 살펴보자. 우리가 인기 있는 자바 그리기 라이브러리 설계자가 되었다고 가정하자. 릴리스한 지 몇 개월이 지나면서 Resizable에 몇가지 기능이 부족하다는 사실을 알게 되었다, 그래서 Resizable에 setRelativeSize를 추가한 다음에 Square와 Rectangle 구현도 고쳤다. 하지만 기존에 Resizable 인터페이스를 구현한 사용자는 우리가 어떻게 할 수가 없다.

Resizable을 고치면 몇 가지 문제가 발생한다. 첫 번째로 Resizable을 구현하는 모든 클래스는 setRelativeSize 메서드를 구현해야 한다. 하지만 라이브러리 사용자가 직접 구현한 Ellipse는 setRelativeSize 메서드를 구현하지 않는다. 인터페이스에 새로운 메서드를 추가하면 **바이너리 호환성**은 유지된다.

하지만 언젠가는 누군가가 Resizable을 인수로 받는 Utils.paint에서 setRelativeSize를 사용하도록 코드를 바꿀 수 있다. 이때 Ellipse 객체가 인수로 전달되면 다음과 같은 런타임 에러가 발생 할 것이다.

```
Exception in thread "main" java.lang.AbstractMethodError: lambdasinaction.chap9.Ellipse.setRelativeSize(II)V
```

또한 사용자가 Ellipse를 포함하는 전체 애플리케이션을 재빌드할 때 다음과 같은 컴파일 에러가 발생한다.

```
... Ellipse is not abstract and does not override abstract method setRelativeSize(int,int) in Resizable
```

바이너리 호환성, 소스 호환성, 동작 호환성

- **바이너리 호환성** - 뭔가를 바꾼 이후에도 에러 없이 기존 바이너리가 실행될 수 있는 상황. 예를 들어 인터페이스에 메서드를 추가했을 때 추가된 메서드를 호출하지 않는 한 문제가 일어나지 않는다.
- **소스 호환성** - 코드를 고쳐도 기존 프로그램을 성공적으로 재컴파일할 수 있음을 의미한다. 예를 들어 인터페이스에 메서드를 추가하면 소스 호환성이 아니다 (추가한 메서드를 구현하도록 클래스를 고쳐야 하기 때문이다).
- **동작 호환성** - 코드를 바꾼 다음에도 같은 입력값이 주어지면 프로그램이 같은 동작을 실행한다는 의미다. 예를 들어 인터페이스에 메서드를 추가하더라도 프로그램에서 추가된 메서드를 호출할 일은 없으므로 동작 호환성은 유지된다.

디폴트 메서드란?

default 메서드는 default라는 키워드로 시작하며 다른 클래스에 선언된 메서드처럼 메서드 바디를 포함한다. 아래 코드를 예로 들자:

```
public inteface Sized {
    int size();
    default boolean isEmpty() {
      return size() == 0;
    }
}
```

이제 Sized 인터페이스를 구현하는 모든 클래스는 isEmpty의 구현도 상속받는다. 즉, 인터페이스에 디폴트 메서드를 추가하면 소스 호환성이 유지된다.

추상 클래스와 자바 8의 인터페이스

추상 클래스와 인터페이스는 뭐가 다를까? 둘 다 추상 메서드와 바디를 포함하는 메서드를 정의할 수 있다.

1) 클래스는 하나의 추상 클래스만 상속받을 수 있지만 인터페이스를 여러 개 구현할 수 있다.

2) 추상 클래스는 인스턴스 변수로 공통 상태를 가질 수 있다. 하지만 인터페이스는 인스턴스 변수를 가질 수 없다.

동작 다중 상속

자바에서 클래스는 한 개의 다른 클래스만 상속할 수 있지만 인터페이스는 여러 개 구현할 수 있다.

```
public class ArrayList<E> extends AbstractList<E> //한 개의 클래스를 상속
    implements List<E>, RandomAccess, Cloneable, Serializable {   //네 개의 인터페이스를 구현
  }
```

이렇듯 자바 8에서는 인터페이스가 구현을 포함할 수 있으므로 클래스는 여러 인터페이스에서 동작을 상속받을 수 있다. 다중 동작 상속이 어떤 장점을 제공하는지 예제로 살펴보자.

우리가 만드는 게임에 다양한 특성을 갖는 여러 모양을 정의한다고 가정하자.

```
public interface Rotatable {
    void setRotationAngle(int angleInDegrees);
    int getRotationAngle();
    default void rotateBy(int angleInDegrees) {
      setRotationAngle((getRotationAngle() + angleInDegrees) % 360);
    }
}
```

Rotatable을 구현하는 모든 클래스는 SetRotationAngle과 getRotationAngle의 구현을 제공해야 하지만 rotateBy는 기본 구현이 제공되므로 따로 구현을 제공하지 않아도 된다.

다음은 Moveable과 Resizable 코드다.

```
public interface Moveable {
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);

    default void moveHorizontally(int distance) {
      setX(getX() + distance);
    }

    default void moveVertically(int distance) {
      setY(getY() + distance);
    }
}

public interface Resizable {
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);

    default void setRelativeSize(int wFactor, int hFactor) {
      setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
    }
}
```

이제 이들 인터페이스를 조합해서 게임에 필요한 다양한 클래스를 구현할 수 있다. 다음처럼 움직일수 있고, 회전할수 있으며 ,크기를 조절할 수 있는 Monster 클래스를 구현할 수 있다.

```
public class Monster implements Rotatable, Moveable, Resizable {
    //모든 추상 메서드의 구현은 제공해야 한다.
    //디폴트 메서드의 구현은 제공하지 않아도 된다.
}
```

Monster 클래스는 다양한 디폴트 메서드를 자동으로 상속받았으며 이 메서드를 직접 호출할 수 있다.

```
Monster m = new Monster();
m.moveVertically(10);
```

이번에는 움직일 수 있으며 회전할 수 있지만 크기는 조절할 수 없느 Sun 클래스를 정의한다고 해보자.

```
public class Sun implements Moveable, Rotatable {}
```

옳지 못한 상속

상속으로 코드 재사용 문제를 모두 해결할 수 있는 것은 아니다, 예를 들어 한 개의 메서드를 재사용하려고 100개의 메서드와 필드가 정의되어 있는 클래스를 상속받는 것은 좋지 않다. 이럴 때는 **델리게이션(delegation)**, 즉 멤버 변수를 이용해서 클래스에서 필요한 메서드를 직접 호출하는 메서드를 작성하는 것이 좋다. 상속을 사용해서 클래스를 정의한 경우 부모 클래스와 자식 클래스 사이에 강한 연관관계가 생기게 된다.

위임 (delegation)

- 위임은 has a 관계로 클래스 내에서 위임 관계에 있는 클래스의 인스턴스를 가지고 있는 상태이다.
  - 스프링 → 의존관계 주입 (dependency injection)
- 상속이 클래스 사이의 관계라면 위임은 인스턴스 사이의 관계이다.
- 상속 관계는 정적인 관계로 컴파일 시간에 모든 관계가 정해지고 위임 관계는 동적인 관계로 런타임 시간 동안 관계가 변경될 수 있다.

알아야 할 세 가지 해결 규칙

1) 클래스가 항상 이긴다. 클래스나 슈퍼클래스에서 정의한 메서드가 디폴트 메서드보다 우선권을 갖는다.

2) 1번 규칙 이외의 상황에서는 서브인터페이스가 이긴다. 상속관계를 갖는 인터페이스에서 같은 시그니처를 갖는 메서드를 정의할 때는 서브인터페이스가 이긴다. 즉, B가 A를 상속받는다면 B가 A를 이긴다.

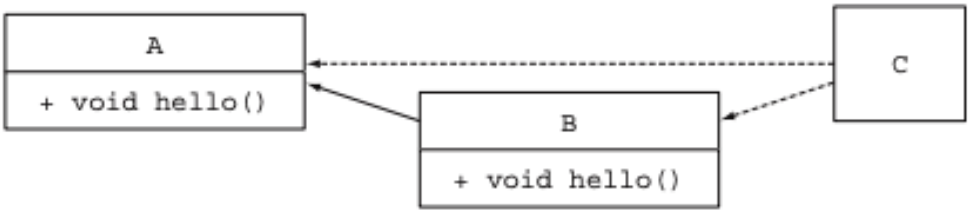
3) 여전히 디폴트 메서드의 우선순위가 결정되지 않았다면 여러 인터페이스를 상속받는 클래스가 명시적으로 디폴트 메서드를 오버라이드하고 호출해야 한다.

아래 예제를 살펴보자

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

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

```
    }
}
public class C implements B, A {
    public static void main(String... args) {
      new C().hello();      //무엇이 출력될까?
    }
}
```

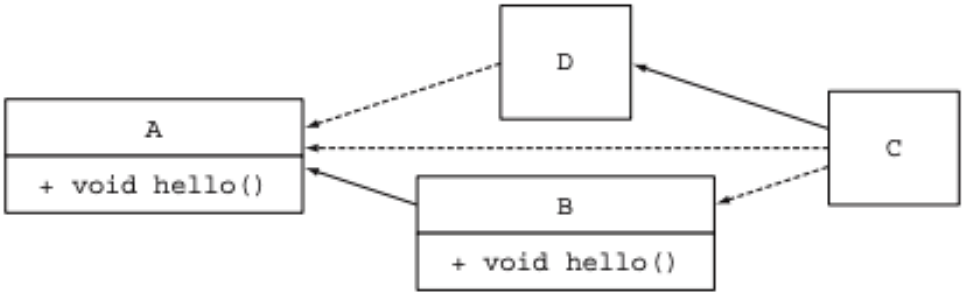


컴파일러는 2번 규칙에 의해서 B의 hello를 선택한다.

다른 예제

```
public class D implements A{}

public class C extends D implements B, A {
    public static void main(String ... args) {
      new C().hello();      //무엇이 출력될까?
    }
}
```



D는 hello를 오버라이드하지 않았고 단순히 인터페이스 A를 구현했다. 따라서 D는 인터페이스 A의 디폴트 메서드 구현을 상속받는다. 2번 규칙에서는 클래스나 슈퍼클래스에 메서드 정의가 없을 때는 디폴트 메서드를 정의하는 서브 인터페이스가 선택되므로 컴파일러는 인터페이스 B의 hello를 출력한다.

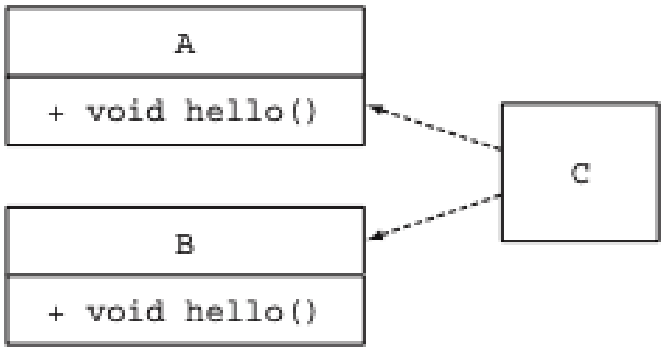
충돌 그리고 명시적인 문제 해결

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

public interface B {
```

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

public class C implements B, A {}
```



이번에는 인터페이스 간에 상속관계가 없으므로 2번 규칙을 적요할 수 없다. 그러므로 A와 B의 hello 메서드를 구별할 기준이 없기 때문에 자바 컴파일러는 Error: class C inherits unrelated defaults for hello() from types B and A 에러를 발생시킨다.

이를 해결하기 위해서는 클래스 C에서 직접 사용하려는 메서드를 명시적으로 선택해야한다.

```
public class C implements B, A {
    void hello() {
      B.super.hello();
    }
}
```

다이아몬드 문제

```
public class Diamond {

  public static void main(String... args) {
    new D().hello();
  }

  static interface A {

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

  }

  static interface B extends A {}

  static interface C extends A {}

  static class D implements B, C {}
```

```
}
```

D는 B와 C중 누구의 디폴트 메서드 정의를 상속받을까? 실제로 선택할 수 있는 메서드 선언은 하나뿐이다. A만 디폴트 메서드를 정의하고 있으므로 **프로그램 출력 결과는 Hello from A** 가 된다.

만약에 B에도 같은 시그니처의 디폴트 메서드 hello가 있다면 어떻게 될까? B는 A를 상속받으므로 B가 선택된다. 만약 B와 C가 모두 디폴트 메서드 hello를 정의하고 있다면 이전에 에제처럼 충돌이 발생하므로 명시적으로 호출해야 한다.

하지만 만약에 인터페이스 C에 디폴트 메서드가 아닌 추상 메서드를 추가하면 어떤 일이 벌어질까?

```
public interface C extends A {
    void hello();
}
```

C는 A를 상속받으므로 C의 추상 메서드 hello가 A의 디폴트 메서드 hello보다 우선권을 갖는다. 따라서 컴파일 에러가 발생하며, 클래스 D가 어떤 hello를 사용할지 명시적으로 선택해서 에러를 해결해야 한다.