10장. 추상클래스, 인터페이스

abstract class, interface

추상 클래스(abstract class)

❖ 추상 클래스

객체를 직접 생성할 수 있는 클래스를 실체 클래스라고 한다면 이 클래스들의 공통적인 특성을 추출해서 선언한 클래스를 추상 클래스라 한다.

추상클래스와 실체 클래스는 상속 관계를 구성한다.
왜 추상클래스를 사용하는가?
실체 클래스의 필드와 메서드의 이름을 통일할 목적으로 사용한다.
(전화와 스마트폰 클래스 멤버의 이름이 달라 복잡하고 유지보수 등 작업이 느려질 수 있다)

TelePhone 클래스 – owner(소유자), powerOn() - 전원을 켜다
SmartPhone 클래스 – user(소유자), trunOn() – 전원을 켜다

추상 클래스(abstract class)

❖ 추상 클래스

▪ 추상 클래스의 선언

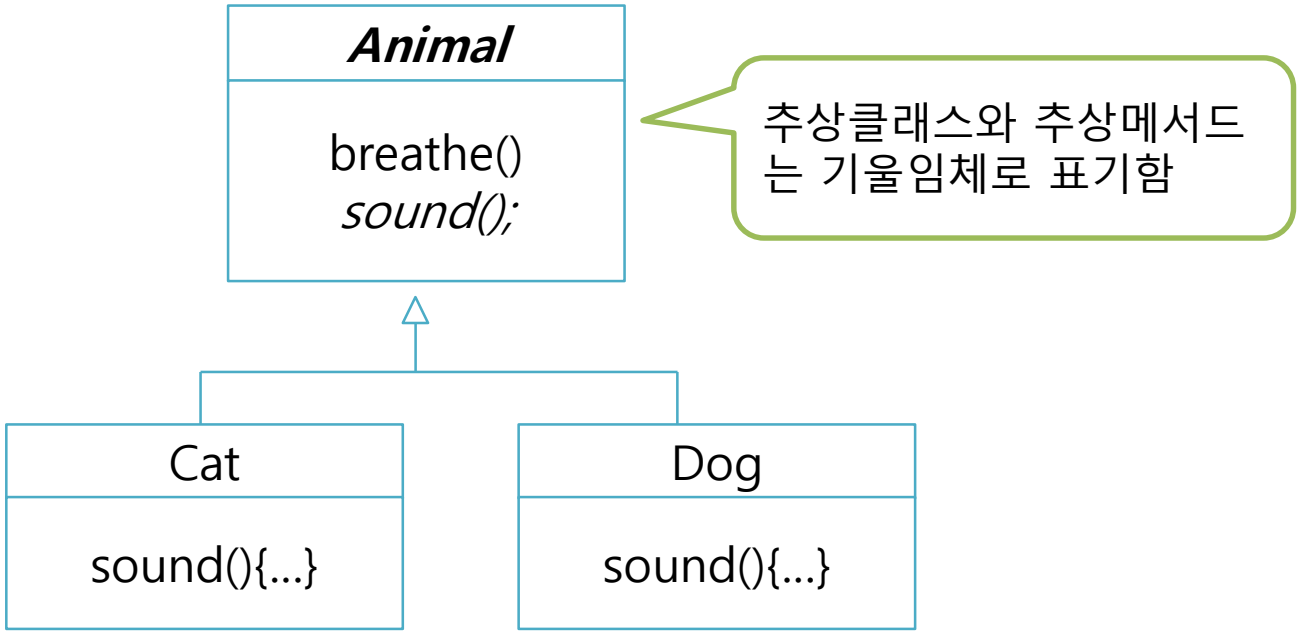
```
public abstract class 클래스이름{
    //필드, 생성자, 메서드
}
```

▪ 추상 메서드

- 추상 메서드도 abstract 예약어를 사용한다.
- 메서드를 구현하지 않고 선언만 한다. { } 구현부가 없다.
- 상속받는 실체 클래스는 추상메서드를 필수적으로 구현해야 한다.

추상 클래스(abstract class)

- 동물의 소리를 구현한 추상클래스 상속



추상 메서드

- 동물의 소리를 구현한 추상클래스 상속

```
package abstract_class.animal;

public abstract class Animal {

    String kind; //동물의 종

    void breathe() {
        System.out.println("동물이 숨을 쉽니다.");
    }

    //추상 메서드 선언
    public abstract void cry();

}
```

추상 메서드

- 동물의 소리를 구현한 추상클래스 상속

```
package abstract_class.animal;

public class Cat extends Animal{

    public Ca
        this.
    }


}
```

The type Cat must implement the inherited abstract method Animal.cry()

2 quick fixes available:

- Add unimplemented methods
- Make type 'Cat' abstract

Press 'F2' for focus

추상메서드는 반드시 구현해야 함

```
public class Cat extends Animal{

    public Cat() {
        this.kind = "포유류";
    }

    @Override
    public void cry() {
        System.out.println("야~ 옹!");
    }
}
```

추상 메서드

- 동물의 소리를 구현한 추상클래스 상속

```
public class Dog extends Animal{

    public Dog() {
        this.kind = "포유류";
    }

    @Override
    public void cry() {
        System.out.println("멍멍!");
    }
}
```

추상 메서드

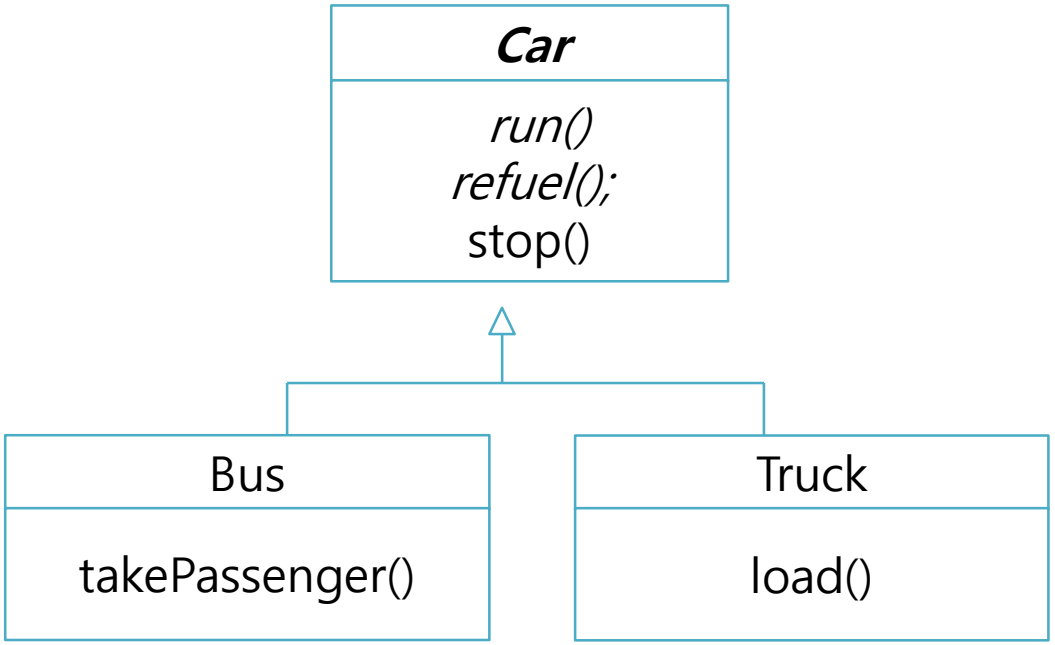
- 동물의 소리를 구현한 추상클래스 상속

```
public class AnimalTest {

    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.breathe();
        cat.cry();

        Dog dog = new Dog();
        dog.breathe();
        dog.cry();

        //메서드의 다형성
        animalCry(new Cat());
        animalCry(new Dog());

    }

    //동물의 울음소리 메서드 정의
    public static void animalCry(Animal animal) {
        animal.cry();
    }
}
```

```
동물이 숨을 쉽니다.
야~ 옹!
동물이 숨을 쉽니다.
멍멍!
야~ 옹!
멍멍!
```

추상 클래스(abstract class)

- **자동차를 구현한 추상 클래스 상속 예.**



추상 클래스 실습

- 자동차를 구현한 추상 클래스 상속 예

```
public abstract class Car {
    public abstract void run();

    public abstract void refuel();

    public void stop() {
        System.out.println("차가 멈춥니다.");
    }
}
```

추상 클래스 실습

- 자동차를 구현한 추상 클래스 상속 예

```
public class Bus extends Car{

    public void takePassenger() {
        System.out.println("승객을 버스에 태웁니다.");
    }

    @Override
    public void run() {
        System.out.println("버스가 달립니다.");
    }

    @Override
    public void refuel() {
        System.out.println("천연 가스를 충전합니다.");
    }
}
```

추상 클래스 실습

- 자동차를 구현한 추상 클래스 상속 예

```
public class Truck extends Car{

    @Override
    public void run() {
        System.out.println("트럭이 달립니다.");
    }

    @Override
    public void refuel() {
        System.out.println("휘발유를 주유합니다.");
    }

    public void load() {
        System.out.println("짐을 싣습니다.");
    }
}
```

추상 클래스 실습

- 자동차를 구현한 추상 클래스 상속 예

```
//Bus 객체 생성
Bus bus = new Bus();

bus.run();
bus.refuel();
bus.takePassenger();

//Truck 객체 생성
Truck truck = new Truck();

truck.run();
truck.refuel();
truck.load();
```

```
버스가 달립니다.
천연 가스를 충전합니다.
버스에 승객을 태웁니다.
트럭이 달립니다.
휘발유를 주유합니다.
트럭에 짐을 싣습니다.
```

final 예약어

- **상수를 의미하는 final 변수**
  - 해당 선언이 최종 상태이고, 결코 수정될 수 없음을 뜻한다.

```
package constant;

public class Constant {
    static int num = 10;    //전역 변수
    static final int NUM = 100; //상수 선언

    public static void main(String[] args) {

        num = 20;
        //NUM = 1000; //수정 불가

        System.out.println(num);
        System.out.println(NUM);
    }
}
```

final 상수

- 여러 파일에서 공유하는 상수

```
public class Define {
    public static final int MIN = 1;
    public static final int MAX = 99999;
    public static final int ENG = 1001;
    public static final int MATH = 2001;
    public static final double PI = 3.14;
    public static final String GOOD_MORNING = "Good Morning!";
}
```

final 상수

- 여러 파일에서 공유하는 상수

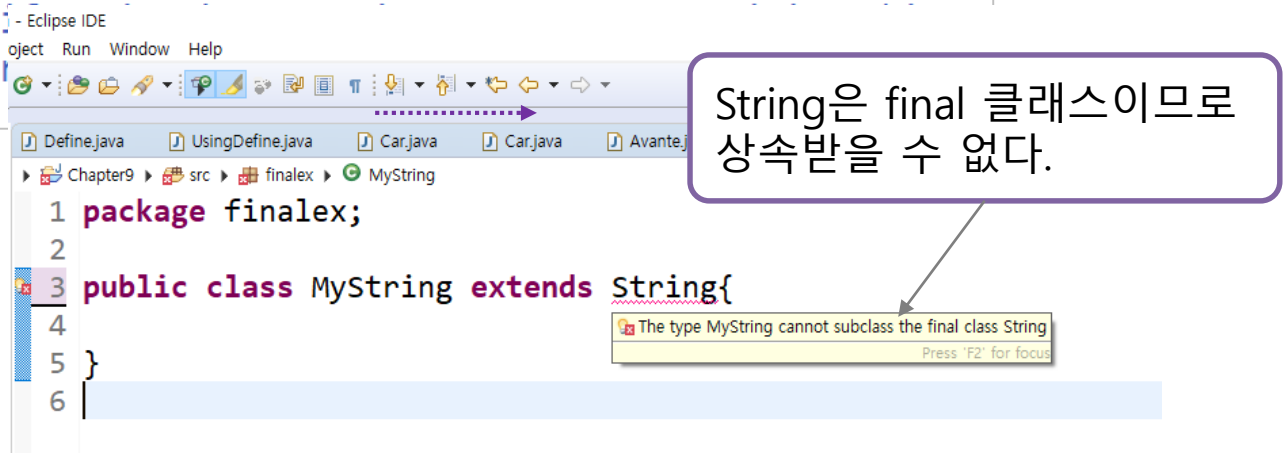
```
public class UsingDefine {

    public static void main(String[] args) {
        System.out.println(Define.GOOD_MORNING);
        System.out.println("최솟값은 " + Define.MIN + "입니다.");
        System.out.println("최대값은 " + Define.MAX + "입니다.");
        System.out.println("수학 과목 코드값은 " + Define.MATH + "입니다.");
        System.out.println("영어 과목 코드값은 " + Define.ENG + "입니다.");
    }
}
```

```
Good Morning!
최솟값은 1입니다.
최대값은 99999입니다.
수학 과목 코드값은 2001입니다.
영어 과목 코드값은 1001입니다.
```

final 클래스

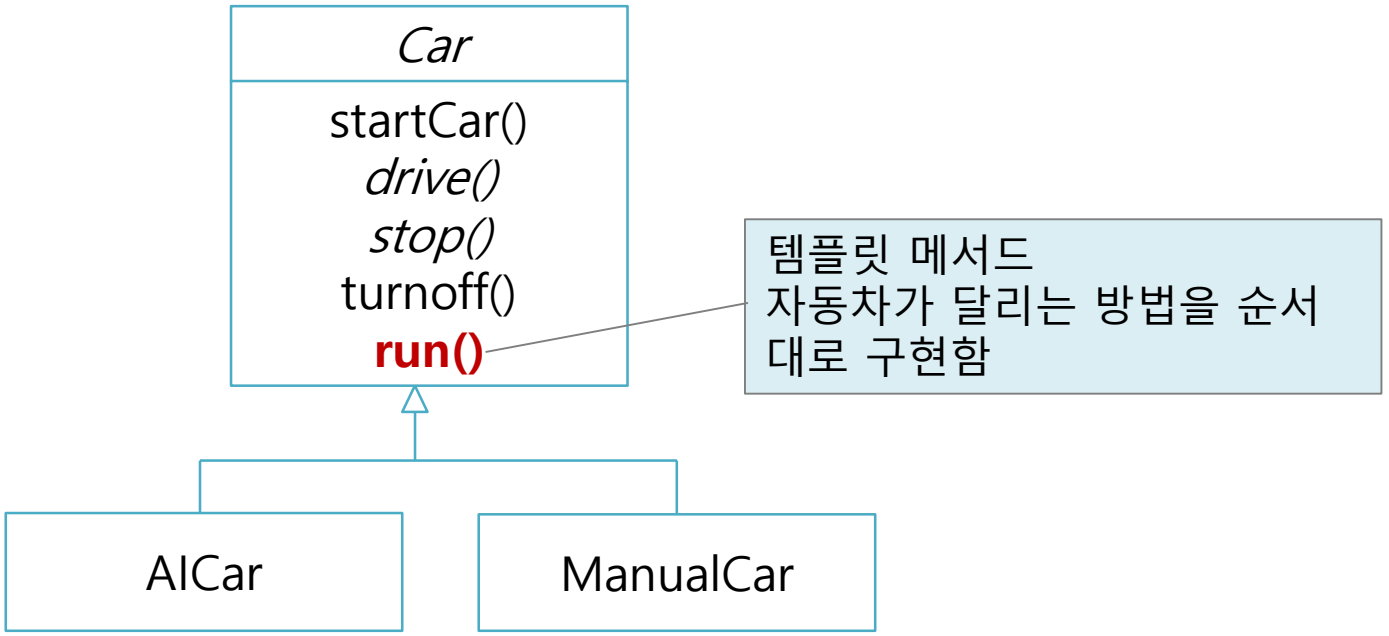
- **보안과 관련되어 있거나 기반클래스가 변하면 안 되는 경우**
  - ✓ String이나 Integer 클래스 등.

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    /**
     * The value is used for character storage.
     *
     * @implNote This field is trusted by the VM, and is a subject to
     * constant folding
     * field after const
     *
```



템플릿 메서드

- **템플릿 메서드란?**
  - 템플릿 메서드 : 추상 메서드나 구현된 메서드를 활용하여 전체 기능의 흐름(시나리오)를 정의하는 메서드.
  - final로 선언하면 하위 클래스에서 재정의 할 수 없음



템플릿 메서드

- 템플릿 메서드를 사용한 추상클래스 상속 예.

```
public abstract class Car {
    public abstract void drive();
    public abstract void stop();

    public void startCar() {
        System.out.println("시동을 켭니다.");
    }

    public void turnOff() {
        System.out.println("시동을 끕니다.");
    }

    public final void run() {
        startCar();
        drive();
        stop();
        turnOff();
    }
}
```

final로 선언
상속받은 하위 클래스가 메서드를 재정의 할 수 없다.

템플릿 메서드

- 템플릿 메서드를 사용한 추상클래스 상속 예.

```
public class HumanCar extends Car{

    @Override
    public void drive() {
        System.out.println("사람이 차를 운전합니다.");
    }

    @Override
    public void stop() {
        System.out.println("사람이 브레이크를 밟아 정지합니다.");
    }
}
```

템플릿 메서드

- 템플릿 메서드를 사용한 추상클래스 상속 예.

```
public class AICar extends Car{

    @Override
    public void drive() {
        System.out.println("자동차가 자율 주행합니다.");
    }

    @Override
    public void stop() {
        System.out.println("자동차가 스스로 멈춥니다.");
    }
}
```

템플릿 메서드

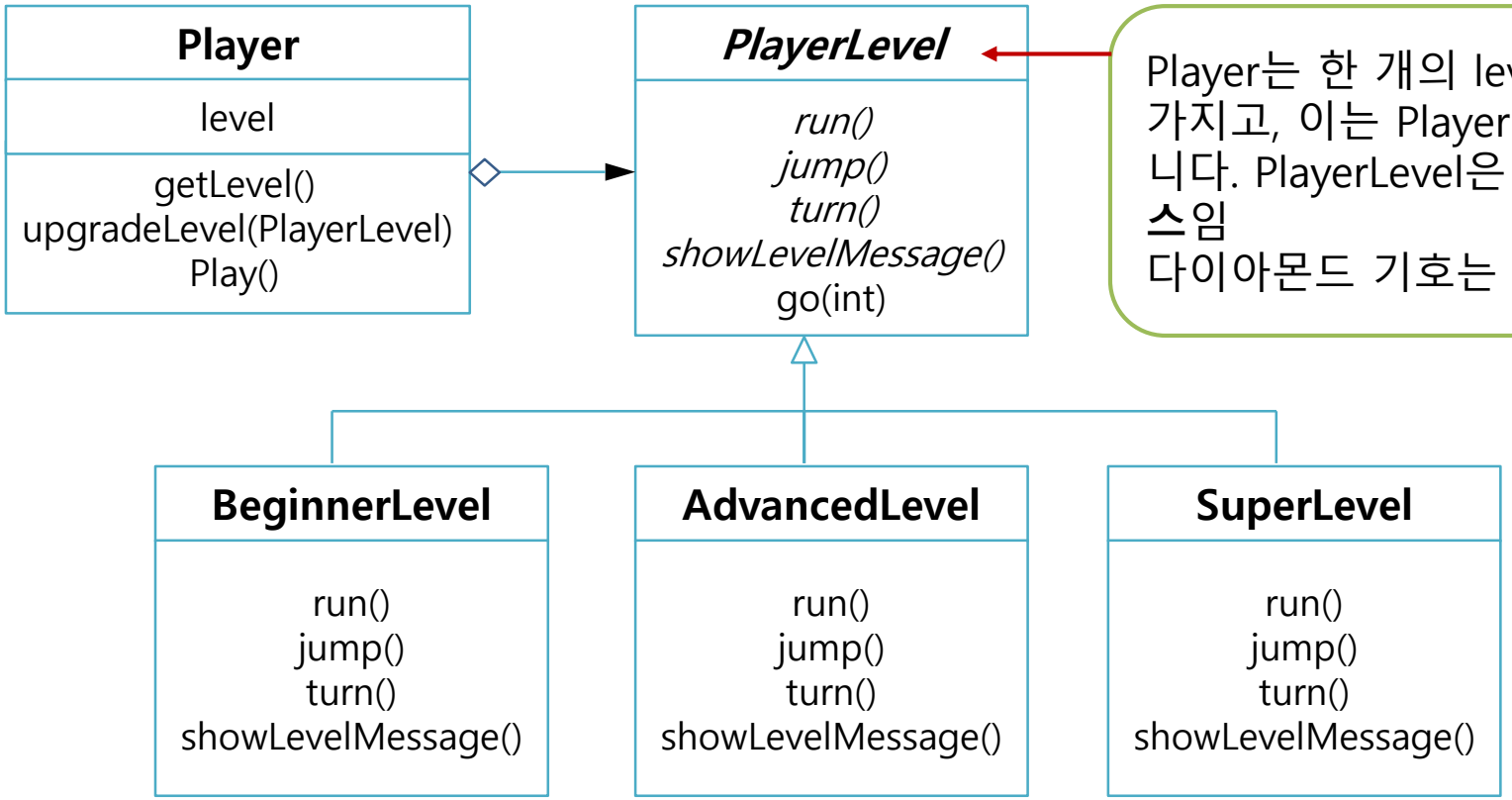
- 템플릿 메서드를 사용한 추상클래스 상속 예.

```
public class CarTest {

    public static void main(String[] args) {

        System.out.println("===== 사람이 운전하는 자동차 =====");
        Car hisCar = new HumanCar();
        hisCar.run();

        System.out.println("===== 자율 주행하는 자동차 =====");
        Car myCar = new AICar();
        myCar.run();
    }
}
```

```
===== 사람이 운전하는 자동차 =====
시동을 켭니다.
사람이 차를 운전합니다.
사람이 브레이크를 밟아 정지합니다.
시동을 끕니다.
===== 자율 주행하는 자동차 =====
시동을 켭니다.
자동차가 자율 주행합니다.
자동차가 스스로 멈춥니다.
시동을 끕니다.
```

Game Level App

예제 시나리오
Player가 있고 이 플레이어가 게임을 합니다. 게임에서 Player가 가지는 레벨에 따라 세가지 기능이 있는데 run(), jump(), turn()입니다.



Game Level App

PlayerLevel 클래스

각 레벨에서 수행할 공통 기능은 PlayLevel 추상 클래스에서 선언한다.

```
public abstract class PlayerLevel {
    public abstract void run();
    public abstract void jump();
    public abstract void turn();
    public abstract void showLevelMessage();

    final public void go(int count) {
        run();
        for(int i=0; i<count; i++) {
            jump();
        }
        turn();
    }
}
```

한번 run하고, count만큼 jump하고, 한번 turn한다.

Game Level App

Beginner 클래스

초보자 레벨에서는 천천히 달리 수만 있다. 점프나 턴을 할 수 없다.

```
public class Beginner extends PlayerLevel {

    @Override
    public void run() {
        System.out.println("천천히 달립니다.");
    }

    @Override
    public void jump() {
        System.out.println("jump할 줄 모르지롱.");
    }

    @Override
    public void turn() {
        System.out.println("Turn할 줄 모르지롱.");
    }

    @Override
    public void showLevelMessage() {
        System.out.println("*****초보자 레벨입니다.*****");
    }
}
```

Game Level App

Advanced 클래스

중급자 레벨에서는 빠르게 달릴 수 있고, 높이 점프할 수 있다. 턴을 할 수 없다.

```
public class AdvancedLevel extends PlayerLevel{

    @Override
    public void run() {
        System.out.println("빨리 달립니다.");
    }

    @Override
    public void jump() {
        System.out.println("높이 jump합니다.");
    }

    @Override
    public void turn() {
        System.out.println("Turn 할 줄 모르지롱.");
    }

    @Override
    public void showLevelMessage() {
        System.out.println("*****중급자 레벨입니다.*****");
    }
}
```

Game Level App

SuperLevel 클래스

고급자 레벨에서는 매우 빠르게 달릴 수 있고, 매우 높이 점프할 수 있다. 턴하는 기술도 사용할 수 있다.

```
public class SuperLevel extends PlayerLevel{

    @Override
    public void run() {
        System.out.println("매우 빨리 달립니다.");
    }

    @Override
    public void jump() {
        System.out.println("매우 높이 jump합니다.");
    }

    @Override
    public void turn() {
        System.out.println("한 바퀴 돕니다.");
    }

    @Override
    public void showLevelMessage() {
        System.out.println("*****고급자 레벨입니다.*****");
    }
}
```

Game Level App

Player 클래스

```
public class Player {
    //PlayerLevel 클래스 참조
    private PlayerLevel level;

    public Player() {
        level = new Beginner();
        level.showLevelMessage();
    }

    public void upgradeLevel(PlayerLevel level) {//매개변수의 다형성
        this.level = level;
        level.showLevelMessage();
    }

    public void play(int count) { //템플릿 메서드 호출
        level.go(count);
    }
}
```

부모 타입으로 객체 생성(다형성)

Game Level App

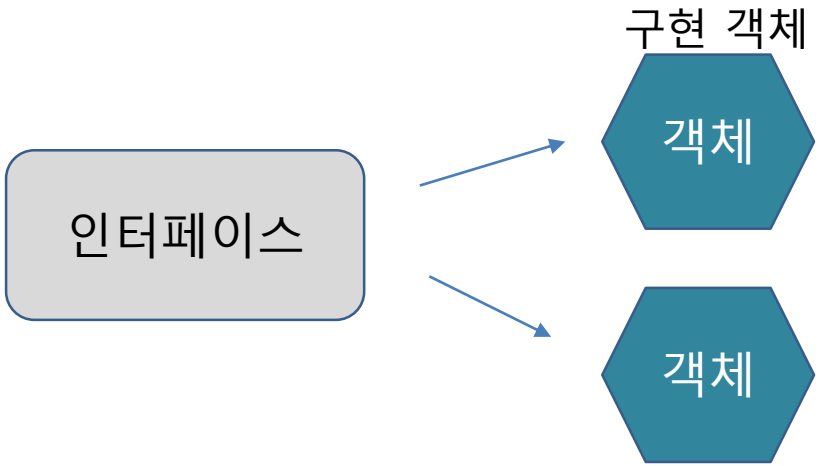
테스트 프로그램 실행

```
Player player = new Player();
//처음 생성시 BeginnerLevel
player.play(1);

//중급자 레벨
AdvancedLevel aLevel = new AdvancedLevel();
player.upgradeLevel(aLevel);
player.play(2);

//고급자 레벨
SuperLevel sLevel = new SuperLevel();
player.upgradeLevel(sLevel);
player.play(3);
```

```
*****초보자 레벨입니다.*****
천천히 달립니다.
jump할 줄 모르지롱.
Turn할 줄 모르지롱.
*****중급자 레벨입니다.*****
빨리 달립니다.
높이 jump합니다.
높이 jump합니다.
Turn 할 줄 모르지롱.
*****고급자 레벨입니다.*****
매우 빨리 달립니다.
매우 높이 jump합니다.
매우 높이 jump합니다.
매우 높이 jump합니다.
한 바퀴 돕니다.
```

인터페이스(Interface)

- **인터페이스란?**
  - 모든 메서드가 추상메서드(abstract method)로 이루어진 클래스이다.
  - **형식적인 선언만 있고 구현은 없다.** ***추상클래스처럼 상속 관계는 아님***
  - **인터페이스의 역할** : 클래스 혹은 프로그램이 제공하는 기능을 명시적으로 선언하는 역할을 한다. 즉 인터페이스만 봐도 어떤 매개변수가 사용되는지 또는 어떤 자료형이 반환되는지 알 수 있다.



인터페이스(Interface)

- **인터페이스 선언**

```
interface 인터페이스 이름{

    //추상메서드
    메서드 이름(매개변수, ...)

}
```

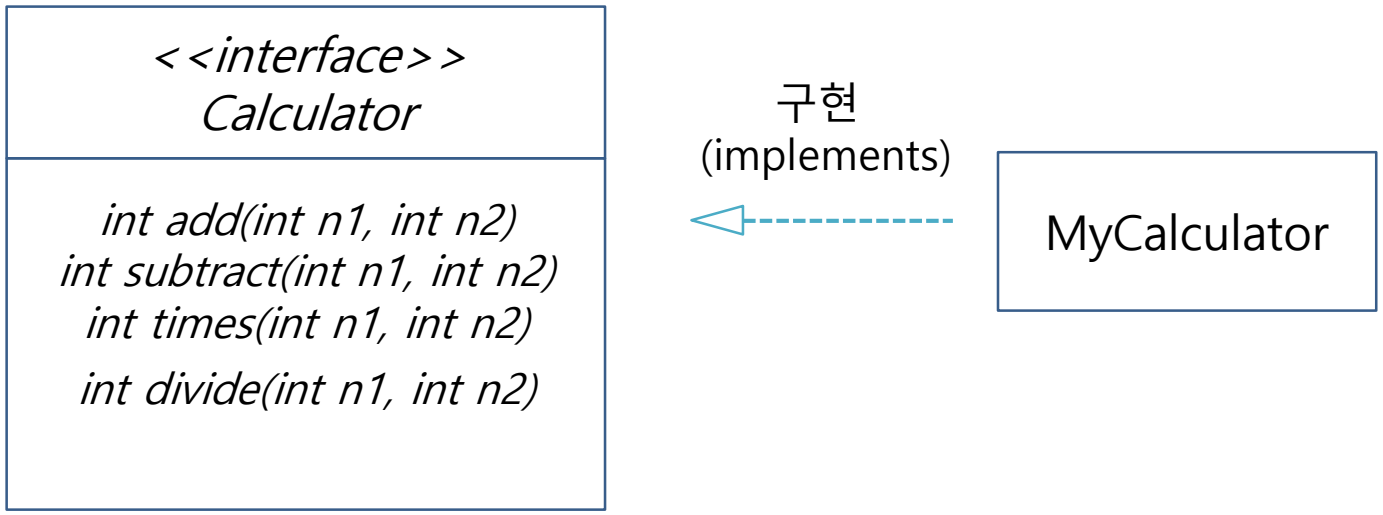
- **구현 클래스**

Implements 키워드 사용

```
class 구현클래스 이름 implements 인터페이스 이름{

    실체 메서드 구현
}
```

인터페이스(Interface)

- **정수형 계산기를 인터페이스로 구현하기**



인터페이스(Interface)

- Calculator 인터페이스

```
package interfaces.calculator;

public interface Calculator {

    int add(int n1, int n2);
    int subtract(int n1, int n2);
    int times(int n1, int n2);
    int divide(int n1, int n2);

}
```

인터페이스(Interface)

- **MyCalculator 클래스**

```
public class MyCalculator implements Calculator{

    @Override
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    @Override
    public int subtract(int n1, int n2) {
        return n1 - n2;
    }

    @Override
    public int times(int n1, int n2) {
        return n1 * n2;
    }

    @Override
    public int divide(int n1, int n2) {
        if(n2 == 0)
            throw new ArithmeticException("0으로 나눌 수 없습니다.");
        return n1 / n2;
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Chapter9/interfaces.calculator.MyCalculator.divide(MyCalculator.java:22)
        at Chapter9/interfaces.calculator.CalculatorTest.main(CalculatorTest.java:12)
```

인터페이스(Interface)

- **CalculatorTest 클래스**

```
public class CalculatorTest {

    public static void main(String[] args) {

        MyCalculator calc = new MyCalculator();

        int num1 = 10, num2 = 0;

        try {
            System.out.println(calc.add(num1, num2));
            System.out.println(calc.subtract(num1, num2));
            System.out.println(calc.times(num1, num2));
            System.out.println(calc.divide(num1, num2));
        }catch(ArithmeticException e) {
            System.out.println("오류: " + e.getMessage());
        }
    }
}
```

인터페이스 구성 요소

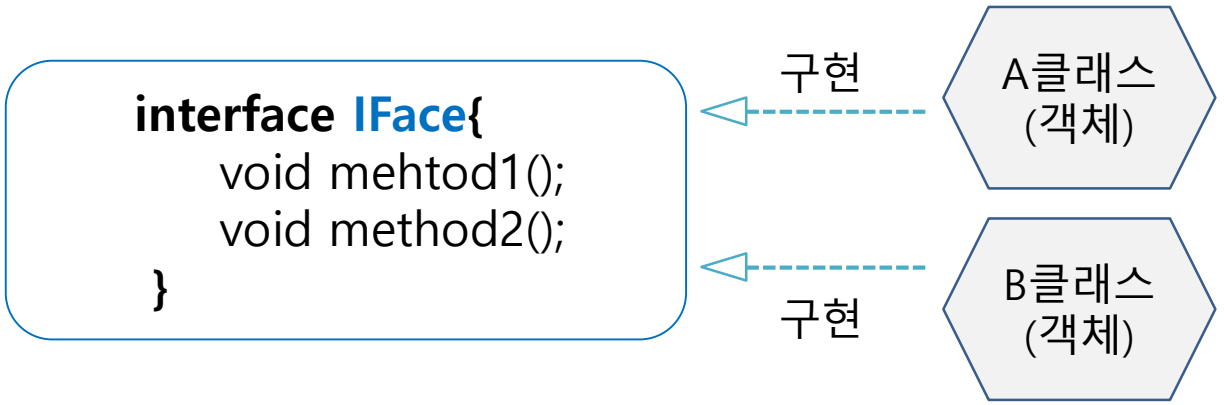
- **인터페이스의 요소**
  - **인터페이스 상수** : 인스턴스를 생성할 수 없으며 멤버 변수도 사용할 수 없다. -> 변수를 선언해도 오류가 나지 않는 이유는 상수로 변환됨
  - **추상메서드** : 구현부가 없는 추상메서드로 구성
  - **디폴트 메서드** : 기본 구현을 가지는 메서드, 구현 클래스에서 재정의 할수 있음 (자바 8부터 가능)
  - **정적 메서드** : 인스턴스 생성과 상관없이 인터페이스 타입으로 사용할 수 있는 메서드(자바 8부터 가능)

인터페이스와 다형성

▪ 타입 변환과 다형성

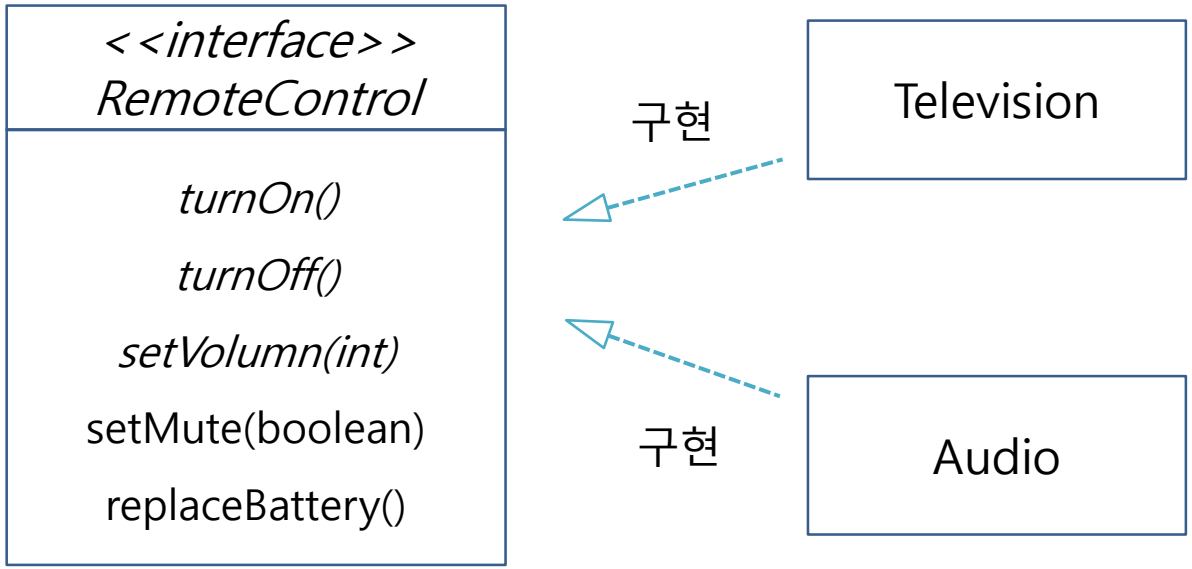
인터페이스도 다형성을 구현하는 기술이 사용된다.

다형성은 하나의 타입에 대입되는 객체에 따라서 실행 결과가 다양한 형태로 나오는 성질을 말한다. 부모 타입에 어떤 지식 객체를 대입하느냐에 따라 실행 결과가 달라지듯이, 인터페이스 타입에 어떤 구현 객체를 대입하느냐에 따라 실행 결과가 달라진다.



인터페이스

- 리모컨으로 TV와 오디오 구현하기



인터페이스

- **RemoteControl 인터페이스**

```java
package interfaces.remotecontrol;

public interface RemoteControl {
    //인터페이스 상수
    public int MAX_VOLUMNE = 10;
    public int MIN_VOLUMNE = 0;

    //추상 메서드
    public void turnOn();
    public void turnOff();
    public void setVolume(int volume);

    //디폴트 메서드
    default void setMute(boolean mute) {
        System.out.println(mute ? "무음 모드 활성화" : "무음 모드 해제");
    }

    //정적 메서드
    static void replaceBattery() {
        System.out.println("배터리를 교환합니다.");
    }
}
```

인터페이스

- **Television 클래스**

```
public class Television implements RemoteControl{
    private int volume;
    private boolean isPoweredOn;

    @Override
    public void turnOn() {
        if(!isPoweredOn) { //전원이 꺼져 있어 전원을 켬
            isPoweredOn = true;
            System.out.println("TV를 켭니다. 현재 상태: ON");
        }
    }

    @Override
    public void turnOff() {
        if(isPoweredOn) { //전원이 켜 있어 전원을 끔
            isPoweredOn = false;
            System.out.println("TV를 끕니다. 현재 상태: OFF");
        }
    }
```

인터페이스

- **Television 클래스**

```
    @Override
    public void setVolume(int volume) {
        if(volume > RemoteControl.MAX_VOLUMNE) {
            this.volume = RemoteControl.MAX_VOLUMNE; //최대 볼륨 설정
        }else if(volume < RemoteControl.MIN_VOLUMNE) {
            this.volume = RemoteControl.MIN_VOLUMNE; //최소 볼륨 설정
        }else {
            this.volume = volume;
        }
        System.out.println("현재 TV 볼륨: " + this.volume);
    }
}
```

인터페이스

- **RemoteControl 테스트**

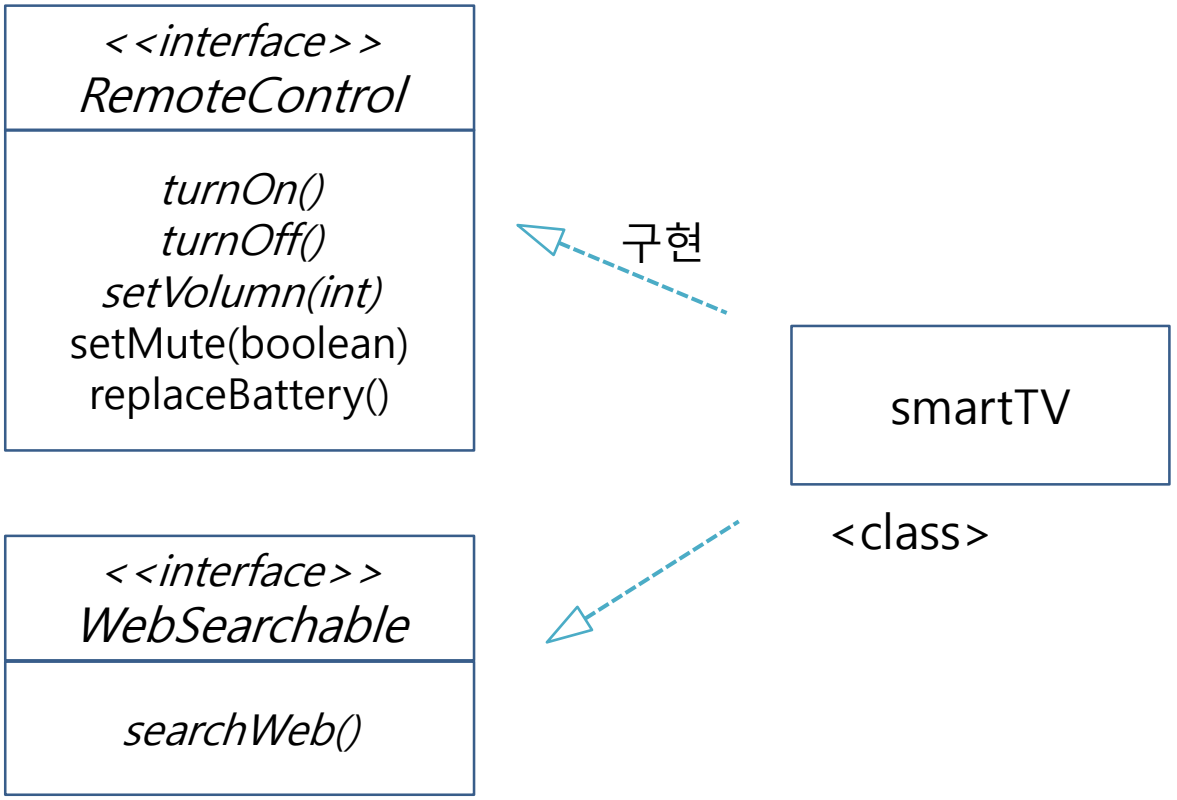
```
public class RemoteControlTest {

    public static void main(String[] args) {
        //인터페이스(부모) 형으로 객체 생성(자동 형변환)
        RemoteControl recomon = new Television();

        //기능 테스트
        recomon.turnOn();
        recomon.setVolume(7);
        recomon.setVolume(-1);   //최소 볼륨으로 제한
        recomon.setVolume(12);   //최대 볼륨으로 제한
        recomon.setMute(true);
        recomon.setMute(false);
        recomon.turnOff();

        RemoteControl.replaceBattery();
    }
}
```

```
TV를 켭니다. 현재 상태: ON
현재 TV 볼륨: 7
현재 TV 볼륨: 0
현재 TV 볼륨: 10
무음 모드 활성화
무음 모드 해제
TV를 끕니다. 현재 상태: OFF
배터리를 교환합니다.
```

다중 인터페이스 구현

- **리모컨, 검색 인터페이스를 구현한 스마트TV**

  인터페이스는 한 클래스가 여러 인터페이스를 다중 구현할 수 있다.



다중 인터페이스

- WebSearchable 인터페이스

```
package interfaces.smart_tv;

public interface WebSearchable {

    void searchWeb(String url);   //추상메서드
}
```

다중 인터페이스

- SmartTV 클래스

```
public class SmartTV implements RemoteControl, WebSearchable{

    private int volume;
    private boolean isPoweredOn;

    @Override
    public void searchWeb(String url) {
        System.out.println("검색 중: " + url);
    }

    @Override
    public void turnOn() {
        if(!isPoweredOn) { //전원이 꺼져 있어 전원을 켬
            isPoweredOn = true;
            System.out.println("TV를 켭니다. 현재 상태: ON");
        }
    }
```

다중 인터페이스

- **SmartTV 테스트**

```
//인터페이스 타입으로 객체 생성
RemoteControl remocon = new SmartTV();
WebSearchable searcher = (WebSearchable)remocon;

//리모컨 테스트
remocon.turnOn();
remocon.setVolume(7);
remocon.setVolume(-1);   //최소 볼륨으로 제한
remocon.setVolume(12);   //최대 볼륨으로 제한
remocon.setMute(true);
remocon.setMute(false);
remocon.turnOff();

//검색 기능 테스트
searcher.searchWeb("www.youtube.com");
searcher.searchWeb("www.naver.com");

//배터리 교환
RemoteControl.replaceBattery();
```

인터페이스와 다형성

- **고객 상담 전화 배분 프로그램**

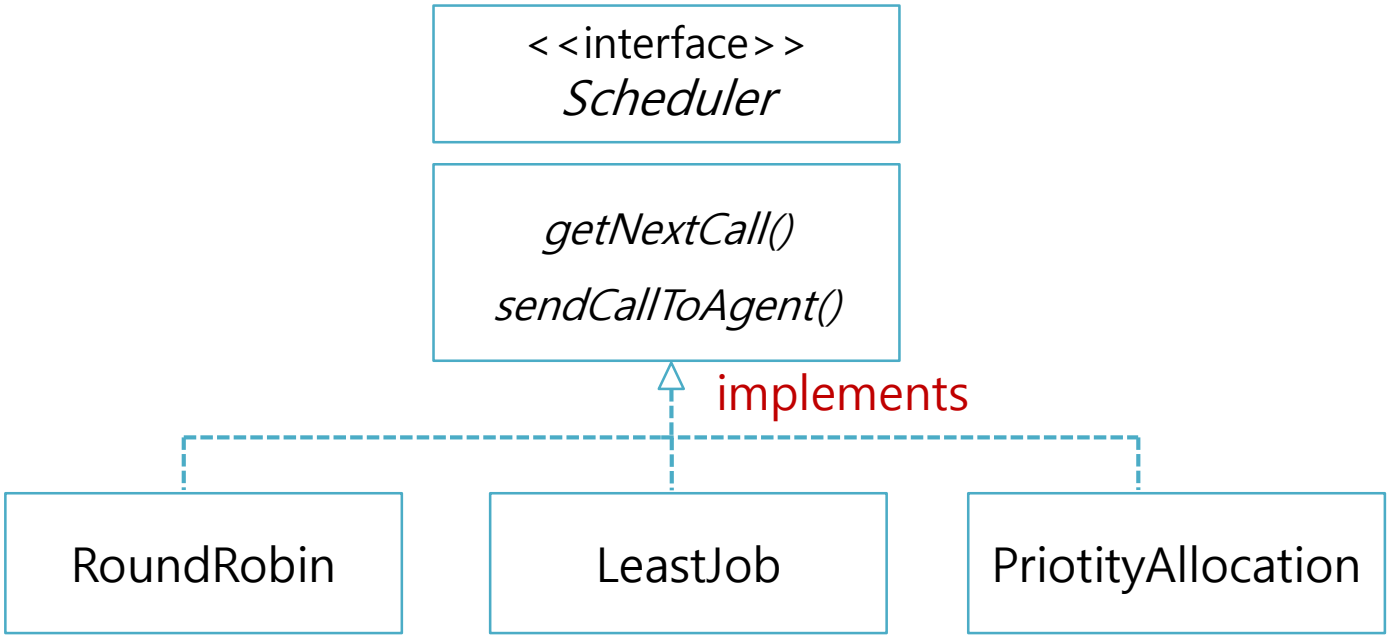
예제 시나리오

고객 센터에는 전화 상담을 하는 상담원들이 있습니다. 일단 고객센터로 전화가 오면 대기열에 저장됩니다. 상담원이 지정되기 전까지는 대기 상태가 됩니다.

1. 순서대로 배분하기 - RoundRobin
2. 짧은 대기열 찾아 배분하기 - LeastJob
3. 우선순위에 따라 배분하기 - PriorityAllocation

인터페이스와 다형성

- **고객 상담 전화 배분 프로그램**



인터페이스와 다형성

- **Scheduler 인터페이스**

```java
public interface Scheduler {
    //다음 전화를 가져오기
    public void getNextCall();

    //상담원에게 전화를 배분하기
    public void sendCallToAgent();
}
```

구현

Scheduler 인터페이스를 구현한 LeastJob 클래스

```java
public class LeastJob implements Scheduler{
    @Override
    public void getNextCall() {
        System.out.println("상담 전화를 순서대로 대기열에서 가져오기");
    }

    @Override
    public void sendCallToAgent() {
        System.out.println("현재 상담 업무가 없거나 대기가 가장 적은 상담원에게 할당합니다.");
    }
}
```

인터페이스와 다형성

- RoundRobin 클래스

```
public class RoundRobin implements Scheduler{
    @Override
    public void getNextCall() {
        System.out.println("상담 전화를 순서대로 대기열에서 가져오기");
    }

    @Override
    public void sendCallToAgent() {
        System.out.println("다음 순서 상담원에게 배분합니다.");
    }
}
```

- PriorityAllocation 클래스

```
public class PriorityAllocation implements Scheduler{
    @Override
    public void getNextCall() {
        System.out.println("고객 등급이 높은 고객의 전화를 먼저 가져옵니다.");
    }

    @Override
    public void sendCallToAgent() {
        System.out.println("업무 skill이 높은 상담원에게 우선 배분합니다.");
    }
}
```

인터페이스와 다형성

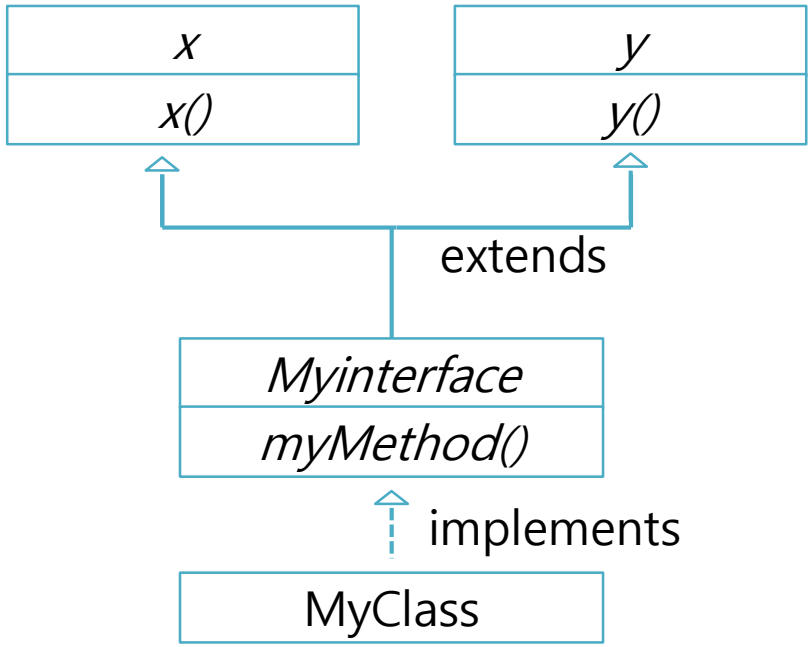
```
public class SchedulerTest {
    public static void main(String[] args) throws IOException { //예외 처리
        System.out.println("전화 상담 배분 방식을 선택하세요.");
        System.out.println("R : 한명씩 차례로 배분");
        System.out.println("L : 쉬고 있거나 대기가 가장 적은 상담원에게 배분");
        System.out.println("P : 우선 순위가 높은 고객 먼저 할당");

        int ch = System.in.read();  //할당 방식을 입력받아 ch에 대입
        Scheduler scheduler = null;

        if(ch=='R' || ch=='r') {  //입력받은 값이 'R' 이나 'r'이면
            scheduler = new RoundRobin();  //다형성으로 생성
        }
        else if(ch=='L' || ch=='l') {
            scheduler = new LeastJob();
        }
        else if(ch=='P' || ch=='p') {
            scheduler = new PriorityAllocation();
        }
        else {
            System.out.println("지원되지 않는 기능입니다.");
            return;
        }

        scheduler.getNextCall();       //입력 받은 정책의 메서드 호출
        scheduler.sendCallToAgent();
    }
}
```

```
전화 상담 배분 방식을 선택하세요.
R : 한명씩 차례로 배분
L : 쉬고 있거나 대기가 가장 적은 상담원에게 배분
P : 우선 순위가 높은 고객 먼저 할당
L
상담 전화를 차례대로 대기열에서 가져옵니다.
현재 상담업무가 없거나 대기가 가장 적은 상담원에게 배분합니다.
```

인터페이스 상속

- **인터페이스 상속하기**

  인터페이스간에도 상속이 가능하다. 구현 코드를 통해 기능을 상속하는 것이 아니므로 **형 상속(type inheritance)**라고 한다. 클래스의 경우는 단일 상속이지만, 인터페이스는 다중 상속이 가능하다.



인터페이스 상속

X 인터페이스

```
package interfaces.inheritance;

public interface X {

    void x();
}
```

Y 인터페이스

```
public interface Y {

    void y();

}
```

MyInterface 인터페이스

```
//인터페이스 X, Y를 다중 상속한 MyInterface
public interface MyInterface extends X, Y{

    void myMethod();
}
```

인터페이스 상속

- MyClass 구현 클래스

```
public class MyClass implements MyInterface{

    @Override
    public void x() {
        System.out.println("x()");
    }

    @Override
    public void y() {
        System.out.println("y()");
    }

    @Override
    public void myMethod() {
        System.out.println("myMethod()");
    }
```

인터페이스 상속

- MyClass 테스트

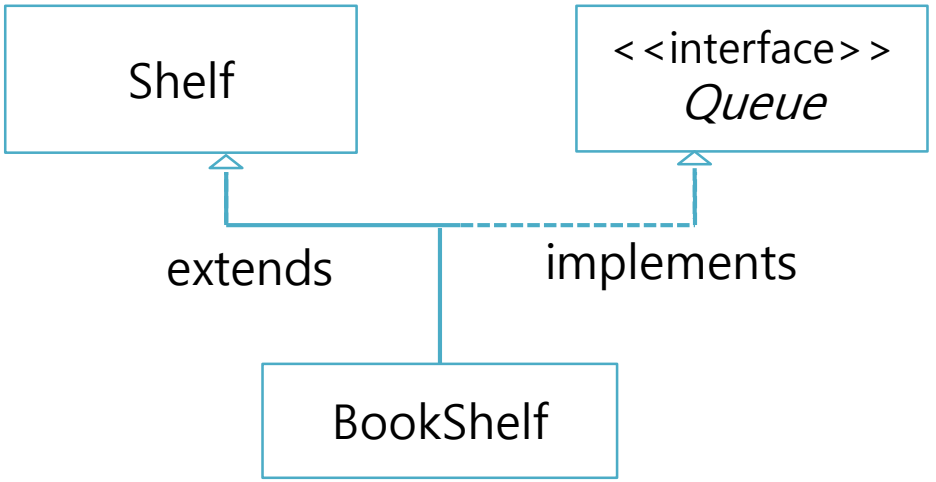
```
public static void main(String[] args) {
    //자동 타입 변환
    MyClass myClass = new MyClass();
    X x = myClass;
    x.x();

    Y y = myClass;
    y.y();

    //X와 Y를 상속한 iClass 객체 생성
    System.out.println("** 다중 상속한 iClass 출력 **");
    MyInterface iClass = myClass;
    iClass.myMethod();
    iClass.x();
    iClass.y();
}
```

```
x()
y()
myMethod()
x()
y()
```

인터페이스 활용

- **인터페이스 구현과 클래스 상속 함께 사용하기**

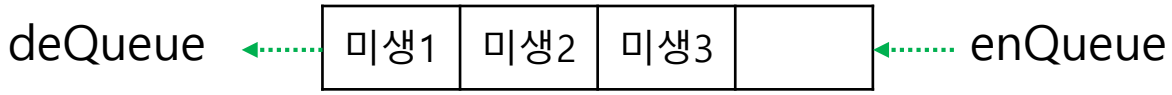
  Queue 인터페이스를 구현하고 shelf 클래스를 상속받는 BookShelf 클래스



인터페이스 활용

- **큐(Queue) 자료 구조**

  Queue : 선입선출(먼저 들어온 자료가 먼저 나옴) -> 선착순, 지하철 타기




```
package bookshelf;

public interface Queue {

    void enQuee(String title); //리스트의 맨 마지막에 추가

    String deQueue();    //리스트의 맨 처음 항목 반환

    int getSize();       //현재 Queue에 있는 개수 반환
}
```

인터페이스 활용

- **Shelf 클래스**

```
public class Shelf {
    //문자열을 저장할 리스트 생성
    protected ArrayList<String> shelf;

    public Shelf() {
        shelf = new ArrayList<>();
    }

    public ArrayList<String> getShelf(){
        return shelf;
    }
}
```

인터페이스 활용

- **BookShelf 클래스**

```
public class BookShelf extends Shelf implements Queue{

    @Override
    public void enQueue(String title) {
        shelf.add(title);  //ArrayList의 객체인 shelf를 상속받음
    }

    @Override
    public String deQueue() {
        return shelf.remove(0);
    }

    @Override
    public int getSize() {
        return shelf.size();
    }
}
```

상속

구현

인터페이스 활용

- **BookShelfTest 클래스**

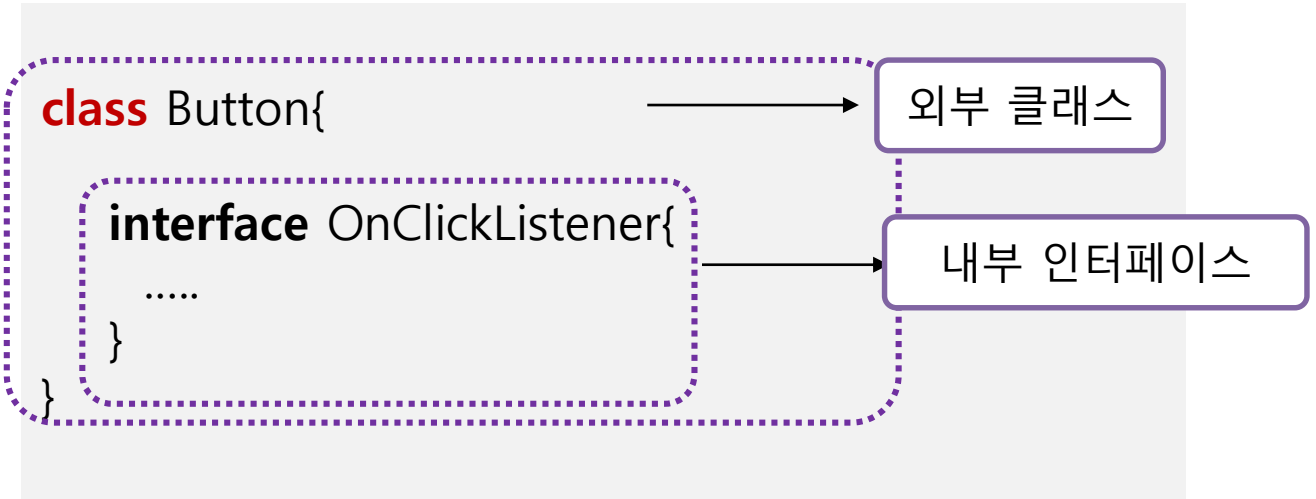
```
public class BookShelfTest {
    public static void main(String[] args) {
        //인터페이스 타입으로 객체 생성
        Queue shelfQueue = new BookShelf();

        //자료 삽입
        shelfQueue.enQueue("반응형 웹");
        shelfQueue.enQueue("혼공 Java");
        shelfQueue.enQueue("스프링부트");

        //자료의 개수
        System.out.println("현재 리스트의 개수 " + shelfQueue.getSize());

        //자료 출력
        System.out.println(shelfQueue.deQueue());
        System.out.println(shelfQueue.deQueue());
        System.out.println(shelfQueue.deQueue());
    }
}
```

내부(중첩) 인터페이스

- **내부 인터페이스**

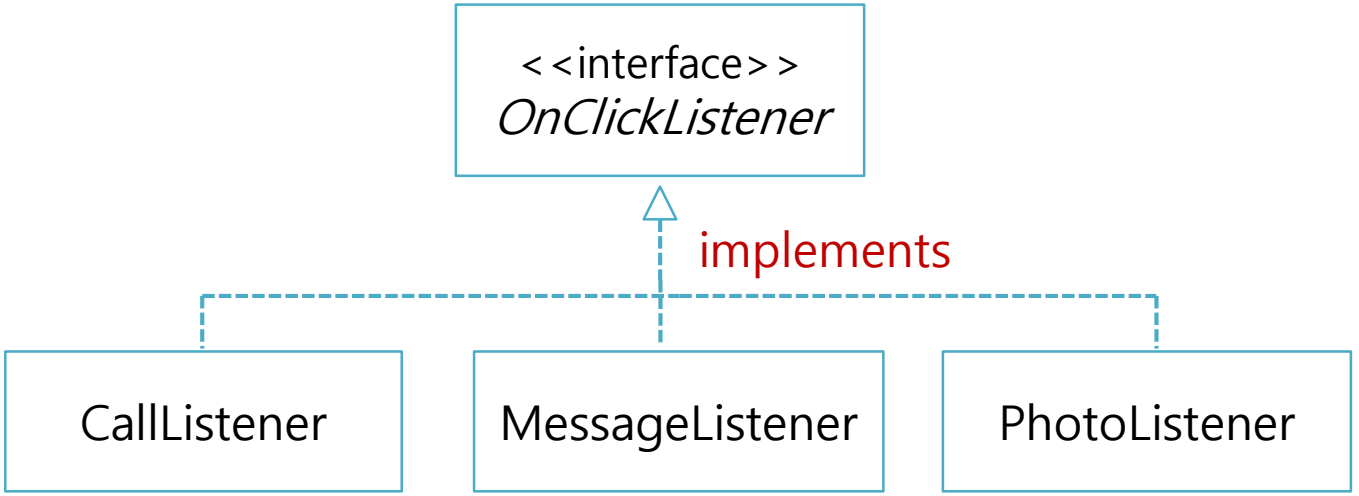
클래스의 멤버로 선언된 인터페이스를 중첩 인터페이스라 한다.

인터페이스를 클래스 내부에 선언하는 이유는 해당 클래스와 긴밀한 관계를 맺는 구현 클래스를 만들기 위함이다.



내부(중첩) 인터페이스

- 버튼을 클릭했을때 이벤트를 처리하는 객체 만들기



내부(중첩) 인터페이스

- **내부 인터페이스 사용 예제**

```
package innerinterface;

public class Button {

    private OnClickListener listener; //인터페이스형 멤버 변수(필드)

    interface OnClickListener{ //내부 인터페이스
        public void onClick();
    }

    public void setListener(OnClickListener listener) {
        //OnClickListener 객체를 매개변수로 전달 받음
        this.listener = listener;
    }

    public void touch() {
        listener.onClick();
    }
}
```

내부(중첩) 인터페이스

- 내부 인터페이스 – 구현 클래스 만들기

```
public class CallListener implements Button.OnClickListener{
    //Button 클래스의 OnClickListener에 접근 -> 구현 클래스 만들기
    @Override
    public void onClick() {
        System.out.println("전화를 겁니다.");
    }

}
```

```
public class MessageListener implements Button.OnClickListener{
    //Button 클래스의 OnClickListener에 접근
    @Override
    public void onClick() {
        System.out.println("문자를 보냅니다.");
    }
}
```

내부(중첩) 인터페이스

- 내부 인터페이스 테스트

```
public class ButtonTest {

    public static void main(String[] args) {

        Button button = new Button();

        //CallListener 객체 생성
        button.setListener(new CallListener());
        button.touch();

        //MessageListener 객체 생성
        button.setListener(new MessageListener());
        button.touch();
    }
}
```

```
전화를 겁니다.
문자를 보냅니다.
```