4장. 추상클래스, 인터페이스

abstract class, interface

추상 클래스(abstract class)

❖ 추상 클래스

객체를 직접 생성할 수 있는 클래스를 실체 클래스라고 한다면 이 클래스들의 **공통적인 특성을 추출**해서 선언한 클래스를 ***추상 클래스***라 한다.

추상클래스와 실체 클래스는 **상속 관계**를 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

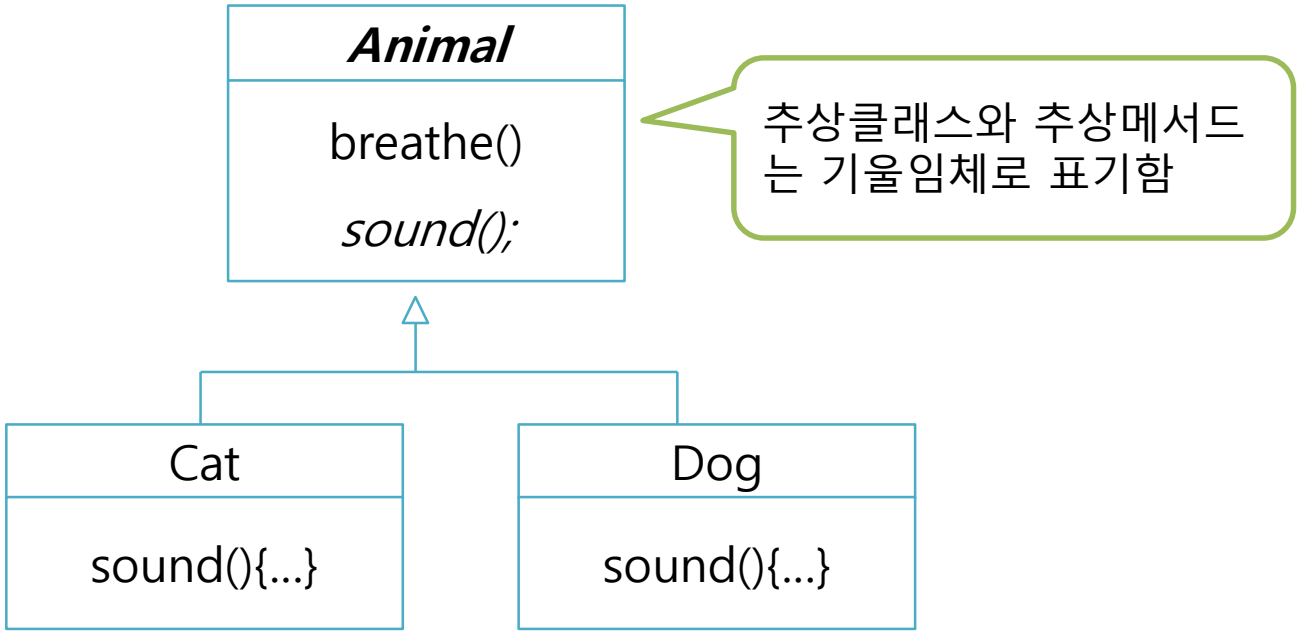
```
public abstract class 클래스이름{
    //필드, 생성자, 메서드
}
```

▪ 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

```
public abstract class Animal {

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

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

}
```

추상 메서드

- 동물의 소리를 구현한 추상클래스 상속

```
public class Cat extends Animal{

}
```

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{

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

추상 메서드

- 동물의 소리를 구현한 추상클래스 상속

```
public class Dog extends Animal{

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

추상 메서드

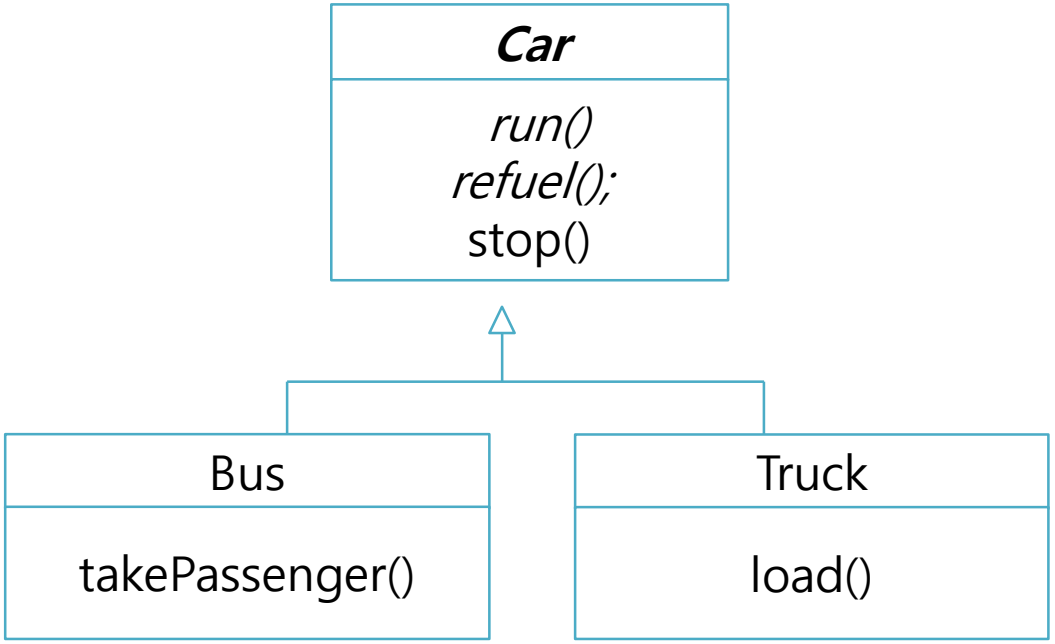
- 동물의 소리를 구현한 추상클래스 상속

```
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

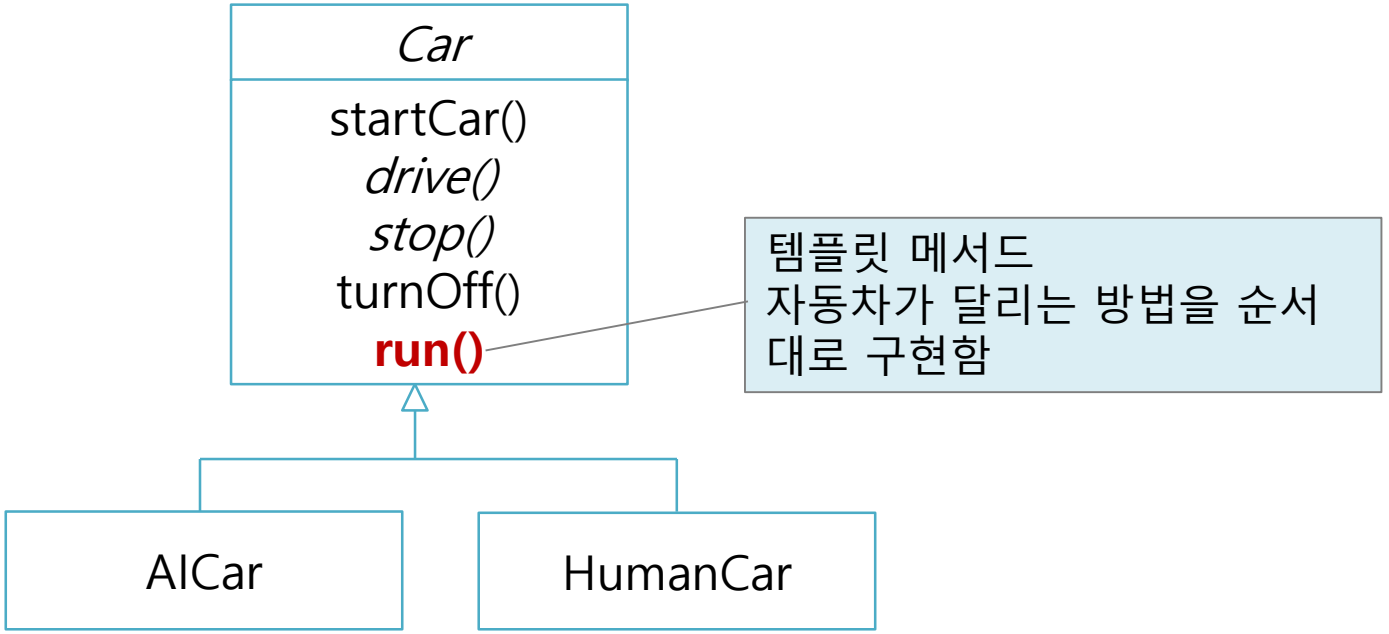
- 자동차를 구현한 추상 클래스 상속 예

```
//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.
```

템플릿 메서드

- **템플릿 메서드란?**
  - 템플릿 메서드 : 추상 메서드나 구현된 메서드를 활용하여 전체 기능의 흐름(시나리오)를 정의하는 메서드.
  - **final**로 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.

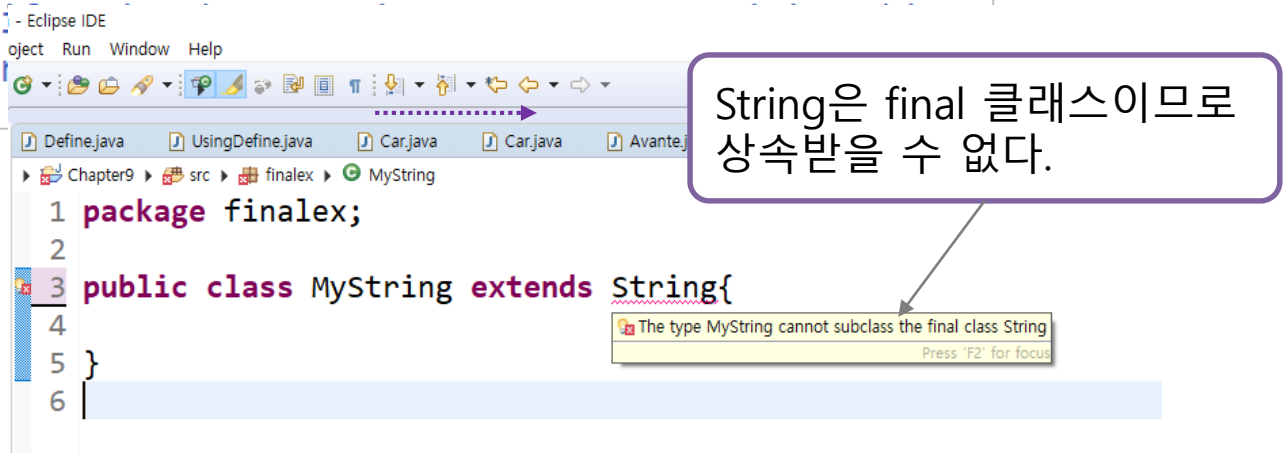
```
public class CarTest {

    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.
```

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
     *
```



인터페이스(Interface)

▪ 인터페이스란?

- 설계도(계약서) 역할을 하는 일종의 추상(abstract) 타입이다.
- 클래스가 "이 기능들을 반드시 구현해야 한다" 는 약속을 정의한다.
- 인터페이스는 메서드의 선언(시그니처) 만 포함하고, 구현(몸체) 은 없다.

  -> 실제 동작은 인터페이스를 implements 하는 클래스가 구현한다.

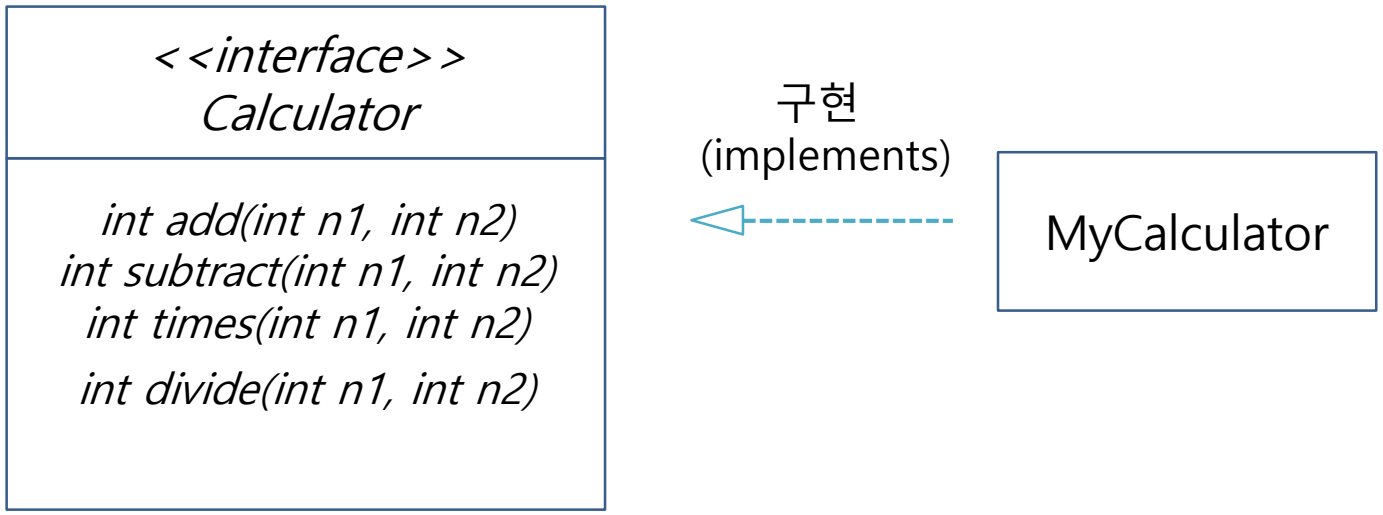
▪ 인터페이스 선언과 구현

```
interface 인터페이스 이름{
        //추상메서드
        메서드 이름(매개변수, ...)
        메서드 이름(매개변수, ...)
}
```

```
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 선언하면 상수로 변환됨(public static final)
  - **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)

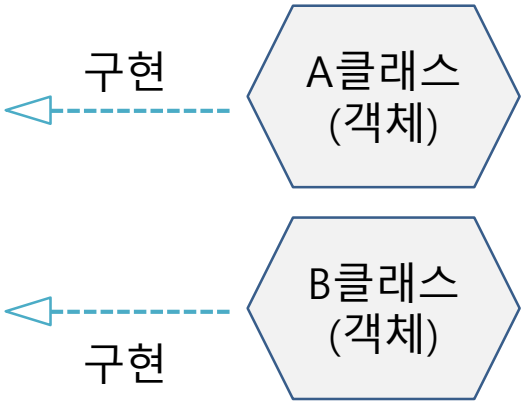
다중구현과 다형성

▪ 다중구현과 다형성

1. **다중 구현** : 클래스는 하나의 클래스만 상속할 수 있지만, 인터페이스는 여러 개를 동시에 구현할 수 있다.

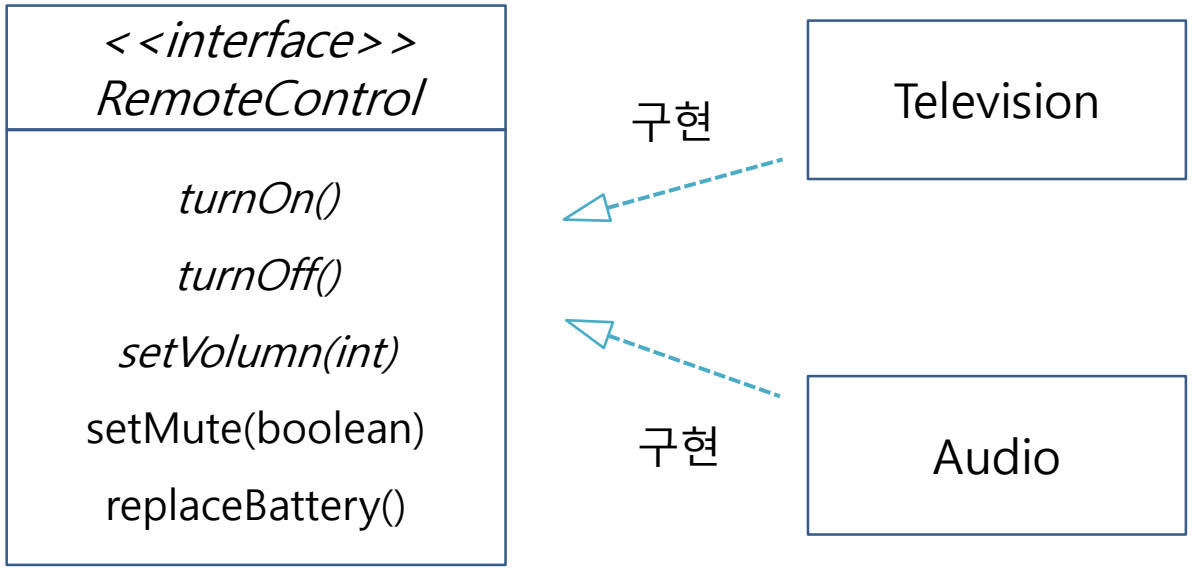
2. 인터페이스도 **다형성**을 구현하는 기술이 사용된다.
   부모 타입에 어떤 지식 객체를 대입하느냐에 따라 실행 결과가 달라지듯이, 인터페이스 타입에 어떤 구현 객체를 대입하느냐에 따라 실행 결과가 달라진다.

```
interface IFace{
    void mehtod1();
    void method2();
}
```



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**

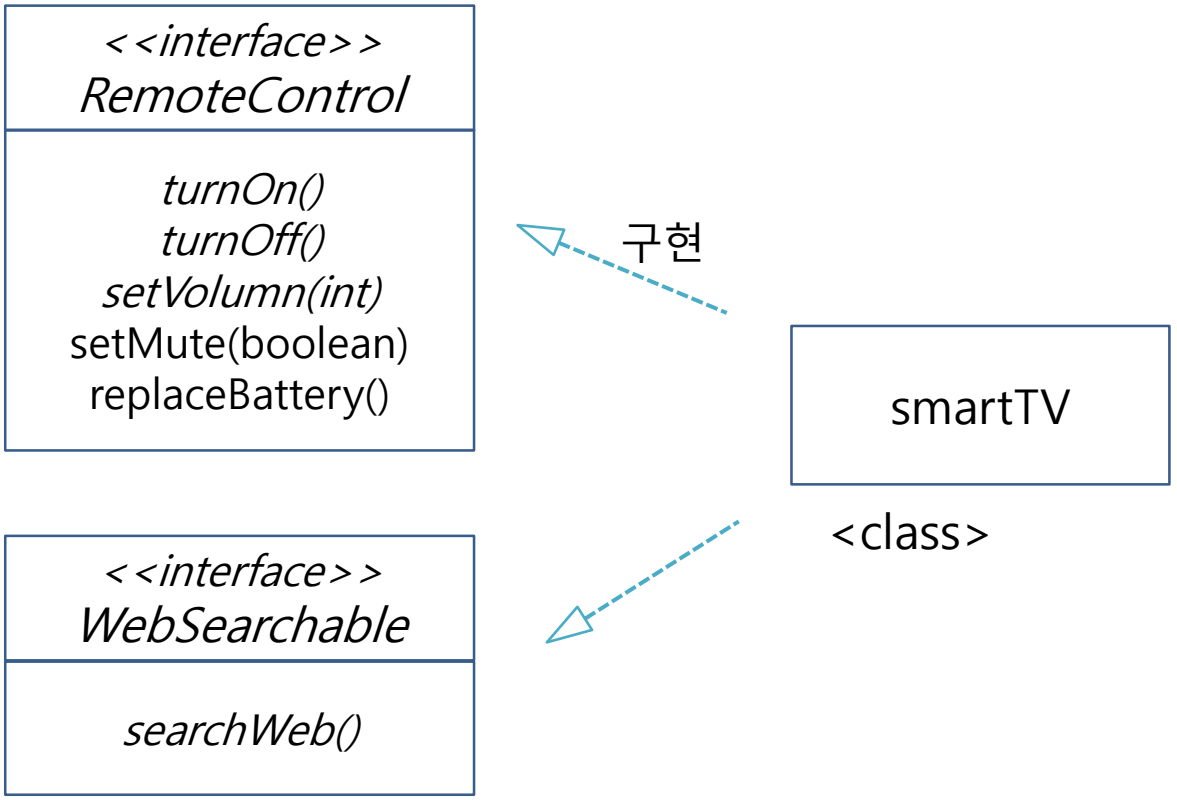
```
public class RemoteControlTest {

    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 = 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);   //무음 활성화
remocon.setMute(false); //무음 해제
remocon.turnOff();

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

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

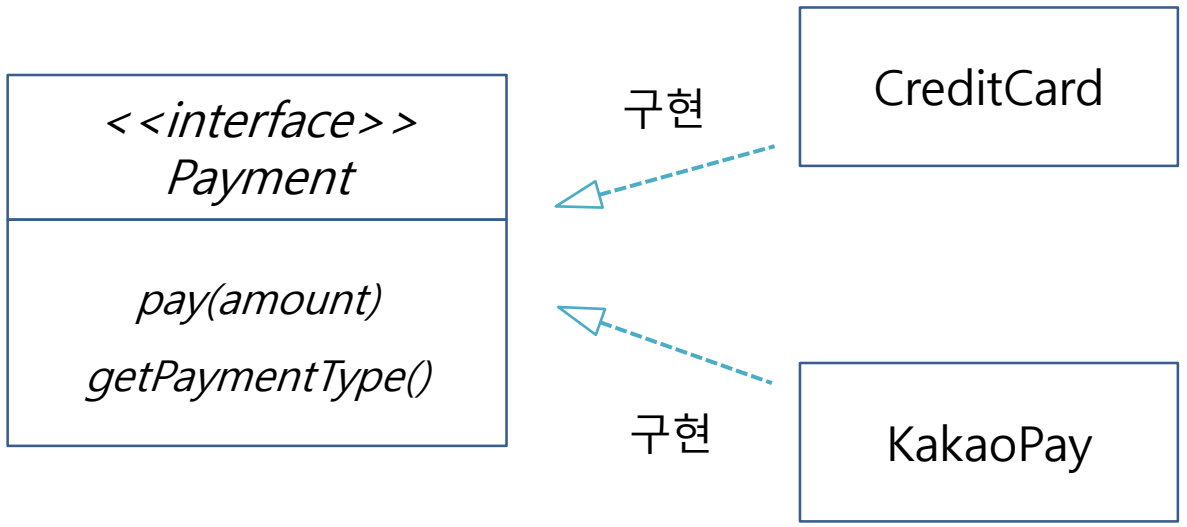
```
TV를 켭니다. 현재 상태: ON
현재 TV 볼륨: 7
현재 TV 볼륨: 0
현재 TV 볼륨: 10
무음 모드 활성화
무음 모드 해제
TV를 끕니다. 현재 상태: OFF
검색 중: www.youtube.com
검색 중: www.naver.com
배터리를 교환합니다.
```

결제 시스템

- **결제 시스템(Payment System)**

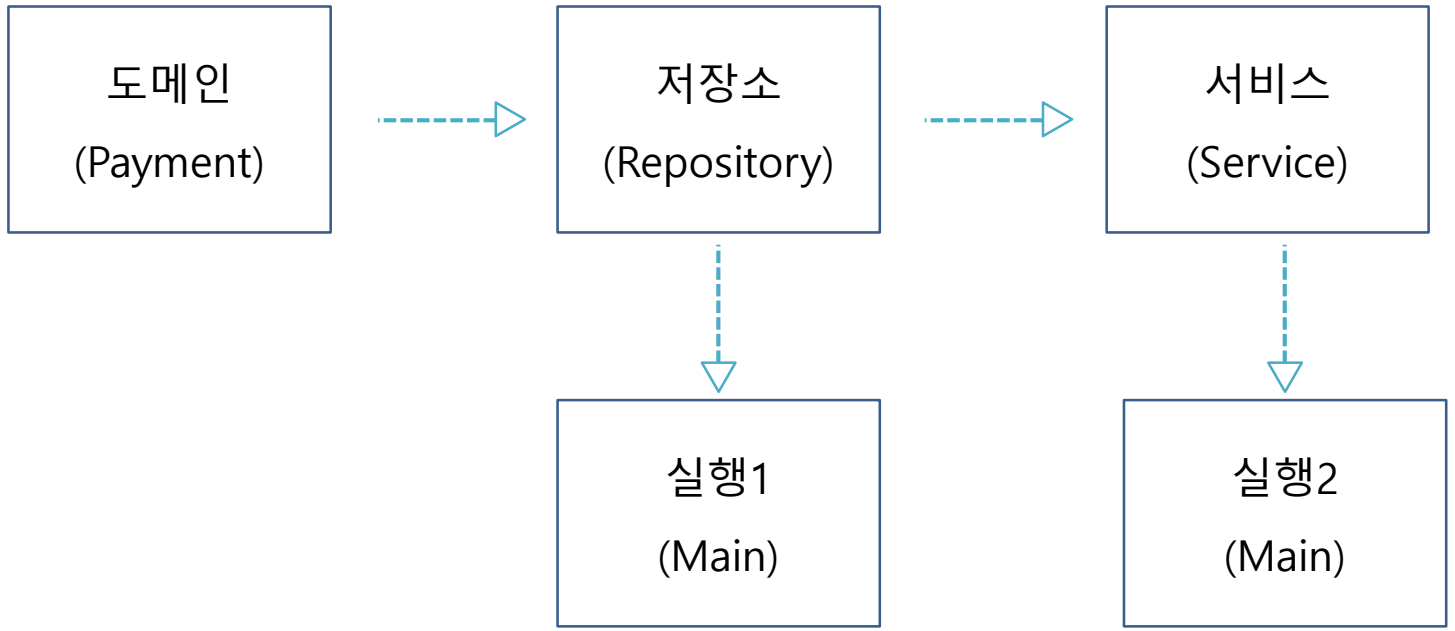
결제 시스템 예제 → 다양한 결제 수단을 Payment 인터페이스로 통합

인터페이스가 유연성, 확장성, 의존성 주입(DI) 을 가능하게 해서 유지보수성과 테스트 용이성을 크게 높여준다



결제 시스템

- **결제 시스템(Payment System) 계층**



결제 시스템

- **결제 시스템(Payment System)**

```
//결제 인터페이스
interface Payment {
 void pay(int amount);     //결제 기능
 String getPaymentType(); // 어떤 결제 수단인지 반환
}

//신용카드 결제
class CreditCardPayment implements Payment {

 @Override
 public void pay(int amount) {
     System.out.println(amount + "원을 신용카드로 결제했습니다.");
 }

 @Override
 public String getPaymentType() {
     return "CreditCard";
 }
}
```

결제 시스템

- **결제 시스템(Payment System)**

```
//카카오페이 결제
class KakaoPayPayment implements Payment {

 @Override
 public void pay(int amount) {
     System.out.println(amount + "원을 카카오페이로 결제했습니다.");
 }

 @Override
 public String getPaymentType() {
     return "KakaoPay";
 }
}
```

결제 시스템

- **결제 시스템(Payment System) – 저장소(리포지터리) 계층**

```java
//결제 내역 저장 인터페이스
interface PaymentRepository {
 void save(String paymentType, int amount);
}

//DB 저장 구현체 (여기서는 JDBC 기반 예시)
class JdbcPaymentRepository implements PaymentRepository {
 @Override
 public void save(String paymentType, int amount) {
     // 실제 DB 저장 로직 (단순 예시)
     System.out.println("DB에 저장됨: " + paymentType + " - " + amount + "원");

     /*try (Connection conn = DriverManager.getConnection(
             "jdbc:mysql://localhost:3306/javadb", "root", "1234")) {
         String sql = "INSERT INTO payment (payment_type, amount) VALUES (?, ?)";
         PreparedStatement pstmt = conn.prepareStatement(sql);
         pstmt.setString(1, paymentType);
         pstmt.setInt(2, amount);
         pstmt.executeUpdate();
     } catch (SQLException e) {
         e.printStackTrace();
     }*/
 }
}
```

결제 시스템

- 결제 시스템(Payment System)

```
public class PaymentApp {

    public static void main(String[] args) {
         // 저장소 생성 (DB 대신 콘솔 출력)
        PaymentRepository repo = new JdbcPaymentRepository();

        // 결제 수단 생성
        Payment card = new CreditCardPayment();
        Payment kakao = new KakaoPayPayment();

        // 결제 실행 및 저장
        int amount1 = 10000;
        card.pay(amount1);
        repo.save(card.getPaymentType(), amount1);

        int amount2 = 5000;
        kakao.pay(amount2);
        repo.save(kakao.getPaymentType(), amount2);
    }
}
```

결제 시스템

- 결제 시스템(Payment System) – Service 계층 만들기

```java
public class PaymentService {

    private final PaymentRepository paymentRepository;

    // 생성자 주입 (DI)
    public PaymentService(PaymentRepository paymentRepository) {
        this.paymentRepository = paymentRepository;
    }

    // 결제 실행 및 DB 저장
    public void processPayment(Payment payment, int amount) {
        payment.pay(amount);  // 결제 진행
        paymentRepository.save(payment.getPaymentType(), amount); // DB 저장
    }
}
```

결제 시스템

- **결제 시스템(Payment System)**

```
public class PaymentApp {
    public static void main(String[] args) {
        // DB 저장소 준비
        PaymentRepository repo = new JdbcPaymentRepository();
        PaymentService service = new PaymentService(repo);

        // 결제 수단 준비
        Payment card = new CreditCardPayment();
        Payment kakao = new KakaoPayPayment();

        // 결제 실행
        service.processPayment(card, 10000);
        service.processPayment(kakao, 5000);
    }
}
```

```
10000원을 신용카드로 결제했습니다.
DB에 저장됨: CreditCard - 10000원
5000원을 카카오페이로 결제했습니다.
DB에 저장됨: KakaoPay - 5000원
```

결제 시스템

- 결제 시스템(Payment System) - MySQL

```
use javadb;

-- payment table
create table payment(
    payment_type    varchar(20) primary key,
    amount  int not null
);

select * from payment;
```

payment_type	amount
CreditCard	10000
KakaoPay	5000
NULL	NULL

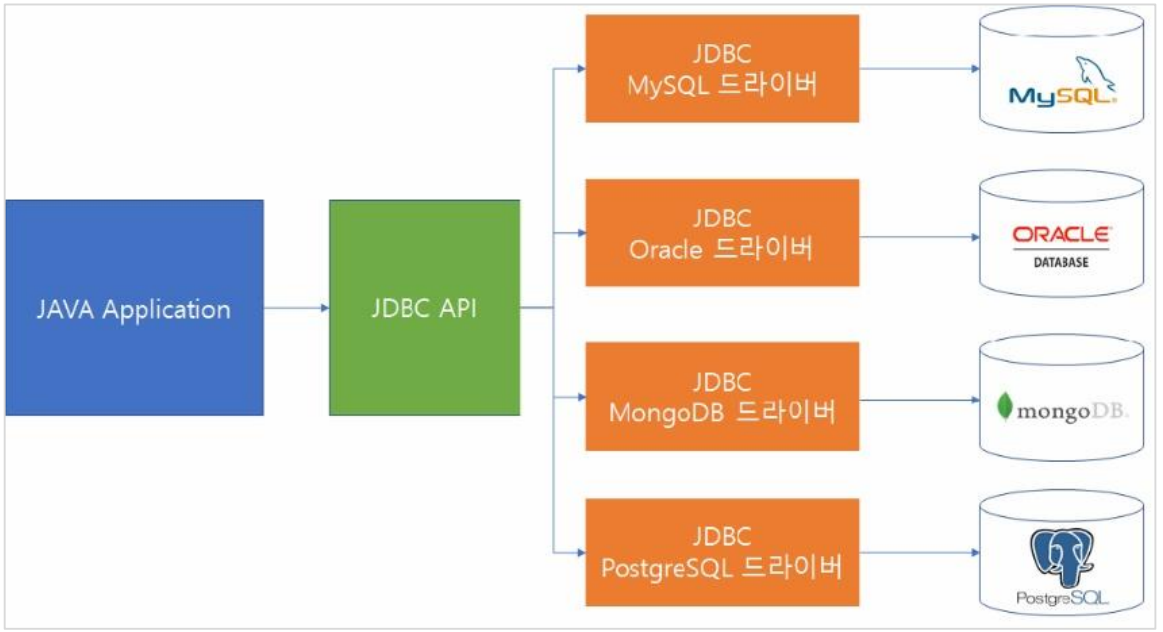
jDBC 드라이버

- **JDBC 드라이버 예제(DB 연결)**

Java에서 데이터베이스에 접근할 때 쓰는 JDBC도 인터페이스 기반이다.

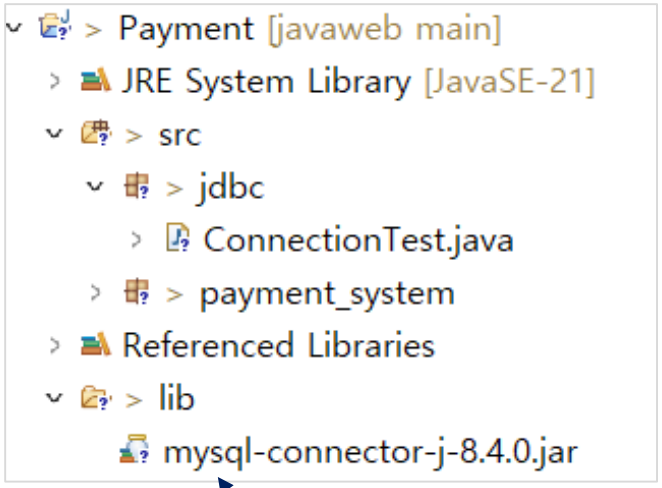
Connection, Statement 등은 모두 인터페이스이다.

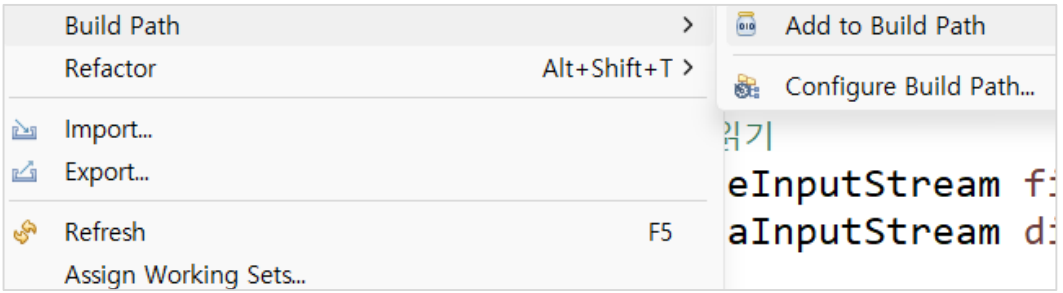
MySQL, Oracle 등은 각각 해당 인터페이스를 구현한 클래스를 제공한다.



jDBC 드라이버

- **JDBC 드라이버 예제(DB 연결)**





1. lib폴더 만들기
2. mysql 드라이버 .jar파일 복사
3. 클래스 패스 설정

jar 파일 > 우측마우스 > Build Path > Add to Build Path

jDBC 드라이버

- **JDBC 드라이버 예제(mysql - DB 연결)**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionTest {
    public static void main(String[] args) {

        Connection conn = null; //Connection 인스턴스 생성

        try {
            //mysql driver 클래스 이름
            Class.forName("com.mysql.cj.jdbc.Driver");

            //접속
            conn = DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/javadb", //db의 url
                    "root",  //username 계정
                    "1234"     //password
                    );

            System.out.println(conn + "DB 접속 성공!!");
```

jDBC 드라이버

- **JDBC 드라이버 예제(mysql - DB 연결)**

```
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
com.mysql.cj.jdbc.ConnectionImpl@7c9d8e2DB 접속 성공!!
```

jDBC 드라이버

- **JDBC 드라이버 예제(mysql - DB 연결)**

close()를 사용하지 않는 try ~ with ~ resource 문

```
public class ConnectionTest2 {
    public static void main(String[] args) {
        //try ~ with ~ resource문
        try(Connection conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/javadb",
                "root",
                "1234")){

            System.out.println(conn + "DB 접속 성공!!");
        }catch(SQLException e) {
            e.printStackTrace();
        }
    }
}
```

람다식(lambda expression)

함수형 프로그래밍과 람다식

- 자바는 객체를 기반으로 프로그램을 구현하며, 어떤 기능이 필요하다면 클래스를 만들고 그 안에 기능을 구현한 메서드(함수)를 만든 후 사용한다.
- 그러므로, 클래스가 없다면 메서드를 사용할 수 없다.
- **자바 8이후부터 사용할 수 있는 람다식은 객체 없이 인터페이스의 구현만으로 메서드를 호출할 수있다.**

람다식 구현하기

- 함수 이름이 없는 **익명 함수**를 만드는 것으로 익명 객체 구현.
- 표현식 : **(매개변수) -> { 실행문; }**

```
int add(int x, int y){
  return x + y;
}
```

일반 메서드(함수)

```
(x, y) ->  x + y
```

람다식

람다식(lambda expression)

람다식 문법 살펴보기

- **매개변수 자료형과 괄호 생략하기** : 매개변수가 하나인 경우 괄호 생략

```
(str) -> { System.out.println(str); }
```

- **중괄호 생략하기** : 중괄호 안의 구현 부분이 한 문장인 경우 중괄호 생략

```
str -> System.out.println(str);
```

- **매개변수가 없다면 괄호 생략할 수 없음**

```
( ) -> { 실행문; }
```

- **return 생략**

```
(x, y) -> {return x + y};

(x, y) -> x + y;
```

람다식(lambda expression)

함수형 인터페이스

람다식을 구현하기 위해 **함수형 인터페이스**를 만들고, 인터페이스에 람다식으로 구현할 메서드를 선언한다.

람다식은 이름이 없는 익명 함수로 구현하기 때문에 메서드에 **오직 하나의 추상 메서드**만 선언할 수 있다. (여러 개는 구현이 모호해지므로)

- **객체지향언어**는 객체를 기반으로 구현하는 방식
- **함수형 프로그램**은 함수를 기반으로 하고 자료를 입력받아 구현하는 방식

람다식(lambda expression)

함수형 인터페이스

```
package lambda;

@FunctionalInterface
public interface MyFunctionalInterface {

    public void method();
    //public void method2(); //메서드는 1개만 사용 가능
}
```

람다식(lambda expression)

람다식 표현법

```
public class MyFunctionalInterfaceTest {

    public static void main(String[] args) {
        //인터페이스의 객체 생성
        MyFunctionalInterface fi;

        fi = () -> { //매개변수가 없는 람다식 표현
            String str = "Hello~ lambda";
            System.out.println(str);
        };
        fi.method();

        //{} 생략 가능
        fi = () -> System.out.println("Hello~ lambda");
    }
}
```

람다식(lambda expression)

객체 지향 방식으로 구현하기

```
package interface_impl;


public interface MyMath {

    public int myAbs(int n);
}
```

```
public class MyMathImpl implements MyMath{

    @Override
    public int myAbs(int n) {
        int value = (n < 0 ? -n : n); //절대값 연산
        return value;
    }

}
```

람다식(lambda expression)

객체 지향 방식으로 구현하기

```
public class MyMathTest {

    public static void main(String[] args) {

        MyMathImpl math = new MyMathImpl();
        System.out.println("절대값: " + math.myAbs(-4));
    }

}
```

람다식(lambda expression)

람다식으로 구현하기

```
package lambda;

@FunctionalInterface
public interface MyMath {
    //1개의 추상메서드만 사용 가능
    public int myAbs(int n);

    //public int mySquare(int n); //오류
}
```

애너테이션을 명시해서 실행전에 오류 체크

```
public class MyAbsTest {

    public static void main(String[] args) {
        //인터페이스형으로 객체 생성
        MyMath math;

        //절대값
        math = (x) -> (x < 0) ? -x : x;
        System.out.println("절대값: " + math.myAbs(-4));
    }
}
```

람다식(lambda expression)

객체 지향 프로그래밍 방식과 람다식 비교

1. 객체 지향 프로그래밍 방식

```
package interface_impl;

public interface StringConcat {
    public void makeString(String s1, String s2);
}
```

```
public class StringConcatImpl implements StringConcat{

    @Override
    public void makeString(String s1, String s2) {
        System.out.println(s1 + ",  " + s2);
    }

}
```

람다식(lambda expression)

객체 지향 프로그래밍 방식과 람다식 비교

1. 객체 지향 프로그래밍 방식

```
public class StringConctTest {

    public static void main(String[] args) {

        StringConcatImpl concat = new StringConcatImpl();
        concat.makeString("Hill", "State");
    }

}
```

람다식(lambda expression)

객체 지향 프로그래밍 방식과 람다식 비교

2 람다식 방식

```
package lambda.concat;

@FunctionalInterface
public interface StringConcat {
    public void makeString(String s1, String s2);
}
```

```
public class StringConcatTest {
    public static void main(String[] args) {
        String str1 = "Hill";
        String str2 = "State";

        StringConcat concat;

        concat = (s1, s2) -> System.out.println(s1 + ", " + s2);
        concat.makeString(str1, str2);
    }
}
```

람다식(lambda expression)

람다식 활용 예제

```
@FunctionalInterface
public interface Workable {
    void work();
}
```

```
public class Person {
    public void action(Workable workable) {
        workable.work();
    }
}
```

람다식(lambda expression)

람다식 활용 예제

```
public class LambdaPersonTest {

    public static void main(String[] args) {
        Person person = new Person();

        person.action(() -> {
            System.out.println("출근을 합니다");
            System.out.println("프로그래밍을 합니다.");
        });

        person.action(() -> System.out.println("퇴근을 합니다."));
    }
}
```